University of Edinburgh

School of Informatics

Smart Contract Programming

Blockchains & Distributed Ledgers
Fall 2023

Abstract

This project aims to develop a smart contract based on the game of *Odds and evens* and deploy it on Ethereum's testnet, Sepolia. The different problems to be discussed are the implementation of a secure, gas-efficient and fair smart contract.

Miquel Muñoz García-Ramos

Contents

1 Game modelling

1.1 Introduction

The smart contract consists of modelling the classic game of evens and nones. In this game, two players A and B interact and choose a number from 1 to 100. If the sum of both numbers is even, A wins, otherwise, if the sum is odd, B is the winner. The amount to be won is the sum of the value chosen by both players in *weis*.

1.2 Game high-level modelling decisions

In this case, it has been chosen to model the game as a zero-sum game. That is, the amount that one player wins is the same amount that the opposing player loses (without taking into account gas costs). The possibility of any other third party paying for the winner's prize has been ruled out as they would have to have an incentive to do so.

A far-fetched option could be for the losing player to gamble the losses with the contract creator being the intermediary and beneficiary. This could be implemented by setting up a scenario in which with a certain probability the losing player could lose less and the intermediary would take the losses. However, it would have to be modelled in such a way that the intermediary would make a profit in the long run and that the game would incentivise the losing player as attractively as possible to bet his losses and lose even more. However, it has been chosen to maintain some simplicity and discard this option.

In the *updateBalances* function, it can be seen how the player balances are updated as a zero sum game. The data structure *playerBalances* has been used to associate each of the addresses of each player with its corresponding balance. However, each player's balance can only be withdrawn once the game is over using the *withdrawBalance* function which transfers the funds using the transfer function and updates the balance to 0. The rationale for choosing this type of funds transfer is further developed in point 3.

The game uses a commit-revelation scheme to prevent cheating. In the sending phase, players send the hashed move and in the revealing phase they reveal the chosen move with its nonce. The contract verifies that the hash entered in the commit phase matches the hash of the value and nonce entered later in the revealing phase. When both players have revealed then the result of the game is computed. In this case, it has been chosen that the player who revealed last is the one who pays for the gas costs of computing the due result (more extensively developed in point 2).

It should also be noted that each player must have deposited at least 200 weis in order to start the game. This is the maximum amount that a player could lose given that the maximum betting range of the two players is 100. In this way, it is prevented that one of the two players may not reveal when the opposing player's move is revealed. Concretely, the contract establishes a countdown in which if one of the two players decides to act adversely, and not to reveal (seeing that has lost after the first player reveals), given that 200 weis has already been deposited, the player who has not revealed within the given time pays the maximum that could have lost: the value revealed by the opposing player plus 100 (the highest value in his range of possible choices).

The choice of the following data types/structures aligns with the requirements of the contract:

- uint8, uint16: These have been used to represent integers such as the maximum move range, the revealed moves, the timestamp of the first player to reveal (used to calculate the timeout) or the revealed moves of each of the players. Int8 allows to store those values that its range is [1,100] in a less costly way.
- address payable: Used to store Ethereum addresses of both players. payable is used for addresses that can receive Ether.
- bytes32: Used for hashing and storing hashed moves.
- mapping (address => uint): Used to store player balances, associating each player's address with their balance.
- event: Used to log significant events in the contract providing transparency such as: LogRegistration, LogReveal, LogTimeoutClaim or LogWithdrawal.

It should be noted that the reset function resets to the predefined value (0) the addresses of the players of the previous game, the hashed moves, the timestamp of the first player to reveal and the revealed moves. In this way, the code is organized in such a way that to determine in which phase of the game the player is, these values are used to see if they are equal to 0 or not. For example, knowing if the hashed move of both players is different from 0, it can allow to execute functions of the revealing phase since it indicates that the sending phase has been finished.

The only global value that is not modified by the reset function (apart from the contract constants) is the map of the associated contract balances with their respective addresses. This is because players who have played in completed games must be able to withdraw the funds once they have finished the game.

2 Gas evaluation

The following analysis has been carried out with regard to gas costs:

	Player A	Player B	A - B
Just after deployment	91649 gas	96013 gas	-4364
Next matches	74549 gas	78913 gas	-4364

Table 1: *commitMove(bytes32 hashedMove)*

	Player A	Player B	A - B
A reveals first	58000 gas	52438 gas	5562
B reveals first	54501 gas	60221 gas	-5720

Table 2: *reveal(uint8 move, string memory nonce)*

Deployment cost: 1999637 gas. *ClaimTimeout()* cost: 49180 gas. *Withdraw-Balance()* cost: 34792 gas.

An approach has been made to make the contract as fair as possible so that both players have to call the same methods the same number of times. However, these methods do not execute exactly the same lines of code depending on whether the player is A or B, or depending on which player executes it first, so there are slight differences explained below.

To begin with, the function *obtainHash* allows both players to obtain the hash that will be input to the *commitMove* function with no gas costs as it is executed as a pure function externally. Then, both players call commitMove.

The player who calls commitMove first (player A) will pay for less costs since when committing player B will have to make extra checks such as checking that player B's address is not the same as A's or that the hash entered by player B is not the same as the one entered by A (explained why in point 3). So there is a saving of 4364 gas for the player who commits first in this first phase of the game. It should be noted that the first time the game is played after deployment it is 17100 gas more expensive and in the following games this cost decreases for both players.

After the commit phase, we can observe that the player who reveals earlier is the one who pays for more gas. What generates this difference most significantly are the following lines of code.

```
// Timer starts after the first move is revealed.
if (timeStampFirstReveal == 0) {
    //First player to reveal
    timeStampFirstReveal = uint16(block.timestamp);
}
else {
    //Last player to reveal
    updateBalances((movePlayerA + movePlayerB)%2 == 0,
    movePlayerA + movePlayerB);
    reset();
}
```

It can be noticed that the first player to act has to store the timestamp so that the timeout can be correctly calculated later. This generates a cost difference that has been tried to balance by giving the gas costs of computing the result to the second player to reveal. Although it does not balance completely, we see that the difference is very small between both costs: 5562 of benefit for player B in case A reveals first and otherwise, 5720 of benefit for player A. One could come to think looking at the tables that the almost perfect way to balance the costs would be to force player A to reveal first. In that case the difference between the players' costs would be only 1198, obtaining an almost optimal gas fairness. However, the user experience would suffer as player B would have to wait for player A to reveal. As the difference is very small, it has been decided not to implement this option.

On the other hand, using the necessary bits for each type of integer has been a key factor in optimizing the contract. In the case of the timeout, setting it even higher could increase security but both efficiency (bit storage) and user experience would be worsened. The contract could be optimized even more, by not sending logs or returning values for each function call but the user and development experience would be greatly deteriorated.

3 Potential hazards and vulnerabilities

3.1 Denial-of-service attacks

To protect from denial-of-service attacks, a pull over push strategy has been developed.

In case a sequential transfer would have been implemented, after a transfer the contract could get stuck. This could be done by transferring the funds to another opponent who failed on purpose, whereby one of the two players might not receive his prize.

However, with the strategy of letting users withdraw the funds (pull) instead of transferring them at the end of the game (push) allows each transfer to be isolated in its own transaction. It also reduces gas cap issues and increases gas fairness. However, it creates a trade-off between security and user experience.

It is necessary that the withdrawal of funds can only be executed at the end of the game since the updated balance is not known until the result has been computed.

3.2 Reentrancy attacks

Nevertheless, the withdraw function mentioned in the previous point has been implemented in order to avoid reentrancy attacks. All internal work is first finished (updating the balance to 0) after saving the balance to be sent in an auxiliary variable. In addition, the function *transfer* has been chosen instead of *call* because *transfer* prevents reentrancy attacks. However, there is an adverse effect since the transaction would fail if the gas costs increase since the transfer function has a limit of 2300 gas. Therefore, the fallback function must be kept simple, otherwise the transfer could fail.

3.3 Solidity/Ethereum hazards

No strict equality controls have been used with reference to the balance of the contract since the contract has been defined to receive ethereum through the *commitMove* method when a player decides to play, but it also has fallback and receive functions defined although they are empty since their only purpose is that the contract can receive funds to profit (although it does not interact directly with the game). If there were such strict comparisons based on the balance of the contract it would be possible to send funds to attack the contract by executing these functions or not even that, for example calculating the address of the contract before being created by the formula *Contract's address = hash(sender address, nonce)* would allow to send funds to the contract without executing these functions.

No logic has been used taking into account the *tx.origin* either. The potential vulnerability arises from the fact that *tx.origin* returns the address of the original sender of the transaction, which might not be the same as the immediate caller.

3.4 Front-Running

Front-running in the context of Ethereum and Solidity refers to the act of a malicious actor exploiting the predictability of transaction execution order to gain an advantage. Given that miners have the ability to alter the order of transactions, this is an important factor to consider. It should also be noted that all contract variables (even if defined as private) are public on the blockchain, so the following measures should be taken into account.

It has been decided to prevent the second player to commit from entering the same hash as the previous player. The vulnerability arises from the mathematical property that the sum of two even numbers or two odd numbers is even. The second player to commit could advance his transaction after seeing the hashed move of the first player to act. Then, would appear as player A with the same hashed move as the opposite player. In that case, it would only be enough to wait for the opposite player to reveal and afterwards, reveal the same move. On the other hand, since each player must enter the nonce, the probability that they commit with the same value and the same nonce (in a non-adversarial way) is negligible, and in case it happens they could rectify and enter another nonce with the same value.

A commitment scheme strategy has also been used to prevent these vulnerabilities. Both players must first submit their hashed move, which they can obtain externally through the *obtainHash* function. This function uses *keccak256*, a hash function that takes an arbitrary-length input and produces a fixed-size (256-bit) hash value. The inputs to this function (the nonce and the chosen value) are parsed with *abi.encode* which is a function provided by Solidity that is used to encode the given arguments into a byte array.

Once the two players have committed the game will not allow any more players to register. Then, it proceeds to the revelation phase, where both players enter the move to reveal and the nonce chosen. If the hash(<move, nonce>) matches the previously entered hash, then the revelation phase is considered completed and the result of the game is computed. A countdown has also been implemented that starts when the first player decides to reveal. However, it has had to be set high enough to try to protect as much as possible from a potential malicious miner who might try to alter the value of the *block.timestamp* (since the countdown is computed with this value). Either player can claim to end the game when the countdown has expired and one of the two players has not revealed, benefiting the one player who has revealed.

3.5 Other security mechanisms

It is worth mentioning that the contract requires to be compiled with a version higher or equal to version 0.8.2. This way, it is protected from possible overflow or underflow attacks since solidity protects them natively from version 0.8+.

Also before starting the game both players must deposit 200 weis. This is the maximum amount they could lose in a game. Players cannot send the exact amount of weis they have wagered as it is the same as the number they have chosen to decide the winner. Therefore, the contract could be vulnerable as the chosen move would be revealed prematurely. Consequently, after they both transfer 200 weis, once they have revealed and the result has been computed the players' balances are duly updated.

4 Tradeoffs

Many of these tradeoffs have already been mentioned, but to recap, the use of transfer instead of call protects against reentrancy attacks but carries a risk should gas costs exceed the 2300 limit.

On the other hand, tradeoffs between user experience and security include requiring a withdrawal function to implement the pull strategy. Another such tradeoff could be the high timeout that has been chosen so that the other player has to disclose. This is inconvenient in the sense that the player who has acted first has to wait 10 minutes for the other player to finally not reveal (in case the player decides not to do so), but it increases security against a possible attempt of manipulation of the block timestamp by a malicious miner.

It should be noted, that emitting events or returning values in each of the functions increases gas costs but provides a better user and development experience. The contract also provides helper functions that allow querying the state of the game quickly and efficiently.

An obvious tradeoff is the commit-revelation scheme used to develop the game. However, it is necessary at the security level to prevent vulnerabilities such as front-running already explained above, even if they significantly worsen the user experience.

5 Analysis of the partner's contract

Partner's contract address: 0x9eDCe6554A00665F18b44A3926252ae7A33EeDEa

In terms of security, no vulnerabilities have been found that either player could use to their advantage and win the game. This is because the partner has also used a commitment scheme that protects against all the vulnerabilities explained above. In addition, he has also developed a pull strategy for the withdrawal of funds to protect against possible denial-ofservice or reentrancy attacks and has taken into account the particularities of the Solidity/Ethereum hazards.

Gas costs: Deployment cost: 2326907 gas. *callTimeout()* cost: 55852 gas. *withdraw()* cost: 32860 gas.

	Player A	Player B
joinGame	120582 gas	105870 gas

Table 3: *joinGame* partner

	First player to reveal	Last player to reveal
verifyCommit	41315 gas	59470 gas

Table 4: *verifyCommit* partner

In general, it can be clearly seen how in all phases of the game, deployment, *callTimeout*, *withdraw*, *joinGame* the partner contract is more expensive. Minus, *verifyCommit* that it is slightly more efficient (see tables 1 and 2 for more detailed comparison). If we add up the costs of each contract to see the total difference of the interaction of the two players, the partner contract is 361147 units of gas more expensive (being the deployment the main factor). One of the causes is the logic that has been used to update the game state:

```
enum State {AwaitingPlayerACommit, AwaitingPlayerBCommit,
    AwaitingVerify, AwaitingPlayerAVerify,
    AwaitingPlayerBVerify, DecidingWinner}
}
```

This logic requires constant updates depending on each of the players' actions and game status checks in order to move on to the next phase. However, in the contract I have developed, these checks are done by checking the default values of the variables (in case it is 0 it has not yet been updated). Here is an example of this handling of the partner state:

```
if (currentState == State.AwaitingPlayerACommit) {
    //Player A commits value. [...]
    currentState = State.AwaitingPlayerBCommit;
    [...] }
```

It should also be noted that the partner's contract is more unfair. In total, the maximum cost difference between player A and B that can be achieved in my contract is 10084 gas units compared to the 32867 that can be reached. The difference is greater, but the most important reason is that the contract is more expensive in general. The partner has tried to reduce this difference, for example by trying to divide the calculation of the result between the two players. However, this division of the calculation to try to improve gas fairness is achieved through a trade-off of sacrificing efficiency as more information is stored and more calculations are performed:

```
function verifyPlayerA(uint8 value, bytes32 password) private {
    [...]
    currentGame.modSum += (value % 2);}

function verifyPlayerB(uint8 value, bytes32 password) private {
    [...]
    currentGame.modSum += (value % 2);}
```

Another difference to comment, is that the partner has decided to penalize the player who does not reveal with 200 weis. In my case, the penalty is the sum of the player who has revealed plus 100 weis(the maximum that could have been lost if had revealed). The reason for my approach is that it is fairer in the sense that there is no need to increase this penalty that much, as it is considered a worst case scenario.

6 Code of the contract and address

Contract address on Sepolia: 0xaf75167623162E82B0C1999a907162754a5ebd4A

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity >=0.8.2;

contract OddEven {
    uint8 constant public BET_MIN         = 1;
    // The minimum bet (1 wei)
    uint8 constant public BET_MAX         = 100;
    // The maximum bet (100 wei)
    uint16 constant public REVEAL_TIMEOUT = 10 minutes;
    // Timeout of the revelation phase
    uint16 private timeStampFirstReveal;
    // TimeStamp of the first reveal

    address payable addressPlayerA;
    address payable addressPlayerB;

    bytes32 private hashedMovePlayerA;
    bytes32 private hashedMovePlayerB;

    uint8 private movePlayerA;
    uint8 private movePlayerB;


    mapping (address => uint) private playerBalances;

    // Events
    event LogRegistration(address indexed playerAddress, uint8 playerIndex);
    event LogReveal(address indexed playerAddress, uint8 move);
    event LogTimeoutClaim(address indexed playerAddress, uint amount);
    event LogWithdrawal(address indexed playerAddress, uint amount);


    /*--------------------------------------------------------*/
    /*_____________________ COMMIT PHASE _____________________*/
    /*--------------------------------------------------------*/

    function obtainHash(uint8 move, string memory nonce) public pure returns (bytes32) {
        require(move >= BET_MIN && move <= BET_MAX,
        "Move is not within the allowed range [1,100].");
        return keccak256(abi.encode(move, nonce));
    }

    // Returns the player's index if the commit is successful.
    // If not, reverts throwing an error.
```

```
function commitMove(bytes32 hashedMove) external payable returns (uint8) {
    require(playerBalances[msg.sender] + msg.value >= 200, "You must have more than
    200 wei in your balance to register.");
    if (addressPlayerA == address(0x0)) {
        addressPlayerA = payable(msg.sender);
        hashedMovePlayerA = hashedMove;
        playerBalances[addressPlayerA] += msg.value;
        emit LogRegistration(addressPlayerA, 1);
        return 1;
    } else if (addressPlayerB == address(0x0)) {
        require(msg.sender != addressPlayerA, "PlayerA has already registered.");
        require(hashedMove != hashedMovePlayerA,
        "The hashes of the two players must be different.");
        addressPlayerB = payable(msg.sender);
        hashedMovePlayerB = hashedMove;
        playerBalances[addressPlayerB] += msg.value;
        emit LogRegistration(addressPlayerB, 2);
        return 2;
    }
    else revert("Both players have already registered.");
}

/*--------------------------------------------------------*/
/*____________________ REVEAL PHASE ____________________*/
/*--------------------------------------------------------*/

// Compares hash(<move, nonce>) with the stored hashed move.
I
f these match, returns the move.
function reveal(uint8 move, string memory nonce) public returns (uint8) {
    require(hashedMovePlayerA != 0x0 && hashedMovePlayerB != 0x0,
    "The commit phase has not ended.");
    require(timeStampFirstReveal == 0 || uint16(block.timestamp)
    < timeStampFirstReveal + REVEAL_TIMEOUT, "The time to reveal has run out.");
    // If hashes match, the given move is saved.
    bytes32 hashedMove = obtainHash(move, nonce);
    if (msg.sender == addressPlayerA) {
        require(movePlayerA == 0, "The player has already revealed.");
        require(hashedMove == hashedMovePlayerA, "The hashes do not match.
        Check the entered move and nonce.");
        movePlayerA = move;
        emit LogReveal(addressPlayerA, move);
    } else if (msg.sender == addressPlayerB) {
        require(movePlayerB == 0, "The player has already revealed.");
        require(hashedMove == hashedMovePlayerB, "The hashes do not match.
        Check the entered move and nonce.");
        movePlayerB = move;
        emit LogReveal(addressPlayerB, move);
    } else {
        revert("The player has not been registered.");
    }
    // Timer starts after the first move is revealed.
    if (timeStampFirstReveal == 0) {
        //First player to reveal
        timeStampFirstReveal = uint16(block.timestamp);
    }
    else {
        //Last player to reveal
        updateBalances((movePlayerA + movePlayerB)%2 == 0, movePlayerA + movePlayerB);
        reset();
    }
    return move;
}

/*-------------------------------------------------------*/
/*_____________________ RESULT PHASE ____________________*/
/*-------------------------------------------------------*/

function claimTimeout() external returns(uint8){
    require(timeStampFirstReveal != 0, "None of the players have yet revealed.");
    require(uint16(block.timestamp) > timeStampFirstReveal + REVEAL_TIMEOUT,
    "There is still time to reveal. Check revealTimeLeft().");
    // Due to the requirement one of the two moves being 0 (initialised value),
    // the player who has not shown is penalised with the maximum he could have lost:
    // the value of the opposing player's bet + 100.
    uint8 amount = movePlayerA + movePlayerB + 100;
    updateBalances(movePlayerB == 0, amount);
    reset();
    emit LogTimeoutClaim(msg.sender, amount);
    return amount;
}

// If winnerPlayerA is true, addressPlayerA pays addressPlayerB the given amount.
// Otherwise, addressPlayerB pays addressPlayerA the given amount.
function updateBalances(bool winnerPlayerA, uint8 amount) private {
    if (winnerPlayerA) {
        playerBalances[addressPlayerA] += amount;
        playerBalances[addressPlayerB] -= amount;
    } else {
        playerBalances[addressPlayerA] -= amount;
        playerBalances[addressPlayerB] += amount;
    }
}

// Withdraw funds
function withdrawBalance() external returns(uint) {
    require(msg.sender != addressPlayerA && msg.sender != addressPlayerB,
    "Before withdrawing funds, the game must end.");
    uint amountToWithdraw = playerBalances[msg.sender];
    playerBalances[msg.sender] = 0;
```

```
        payable(msg.sender).transfer(amountToWithdraw);
        emit LogWithdrawal(msg.sender, amountToWithdraw);
        return amountToWithdraw;
    }


    // Reset the game.
    function reset() private {
        timeStampFirstReveal     = 0;
        addressPlayerA         = payable(address(0x0));
        addressPlayerB         = payable(address(0x0));
        hashedMovePlayerA = 0x0;
        hashedMovePlayerB = 0x0;
        movePlayerA     = 0;
        movePlayerB     = 0;
    }

    // receive() and fallback() functions.
    receive () external payable {}
    fallback () external payable {}

    /*--------------------------------------------------------*/
    /*___________________ HELPER FUNCTIONS____________________*/
    /*--------------------------------------------------------*/

    // Returns the player (caller) index.
    // If the player does not exist returns 0.
    function getPlayerIndex() public view returns (uint8) {
        if (msg.sender == addressPlayerA) {
            return 1;
        } else if (msg.sender == addressPlayerB) {
            return 2;
        } else {
            return 0;
        }
    }

    // Returns true if both players have successfully commited a valid move.
    // Otherwise returns false.
    function bothCommited() public view returns (bool) {
        return (hashedMovePlayerA != 0x0 && hashedMovePlayerB != 0x0);
    }

    // Returns how much time (seconds) is left for the second player to reveal the move.
    // If none of the players have revealed, returns the maximum timeout.
    function revealTimeLeft() public view returns (uint16) {
        if (timeStampFirstReveal != 0) {
            if (uint16(block.timestamp) < timeStampFirstReveal + REVEAL_TIMEOUT)
                return timeStampFirstReveal + REVEAL_TIMEOUT - uint16(block.timestamp);
            return 0;
        }
        return REVEAL_TIMEOUT;
    }

    // Returns the contract balance.
    function getContractBalance() public view returns (uint) {
        return address(this).balance;
    }

    // Returns the player's balance.
    function getMyBalance() public view returns (uint) {
        return playerBalances[msg.sender];
    }
}
```

References

[1] Dimitris Karakostas Slide credits: DK, Aggelos Kiayias, Aydin Abadi, Christos Nasikas, Dionysis Zindros, *Blockchains Distributed Ledgers Lectures 03 and 04, The University of Edinburgh*

[2] *Solidity — Solidity 0.8.23 documentation, https://docs.soliditylang.org/en/v0.8.23/*

[3] *The Solidity Contract-Oriented Programming Language, 2023. https://github.com/ethereum/solidity*