HIT AND SUNK PROJECT

Peer to Peer and Blockchain Course

Università di Pisa

UNIVERSITÀ DI PISA

Professor:

Maria Laura Ricci

Student:

Francesco Rubino

04/09/23

SUMMARY

DESCRIPTION OF THE PROJECT

The aim of the HitAndSunk project is to implement a dApp of a battleship game based on the use of a blockchain for the management of the data structures needed for the implementation of the application itself. The reason behind this decision is to enjoy the advantages obtained using a blockchain, that are:

- The absence of a centralized entity for the management of the communication and data structures necessary for the functioning of the application. The blockchain assures a decentralized approach;
- The transparency of the code used for the functioning of the application. The player know a priori how the information given in input is processed;

These characteristics are necessary for the player to trust the correct and fair play of the application, extremely important in a game that provide a reward.

On the other side, considering the case of the battleship game, it's important for the player to provide not in advance information about the positions of their ship on the battlefield. If this condition is not met it would be easy for the adversary to obtain these information on the blockchain. Merkle tree technology is used to guarantee this property. The merkle tree can be implemented using the hash of the leaves combined with a seed (to avoid the predictability). The leaves encode the position of the battlefield. Once the merkle tree is implemented, the merkle root can be saved on the blockchain at the beginning of the game and the merkle proofs can be generated step by step during the game, used as a certificate of the correctness of the responses about the tries.

To obtain this, a smart contract to deploy on the blockchain and a front end code to interact with it have been implemented. The tools used are:

- Node.js: Next.js provides building blocks to create web applications. It handles the tooling and configuration needed for React and it also provides automatic compilation, building and fast refresh;
- React: a Javascript library used to build web applications with interactive user interface;
- Bootstrap: a free, open-source development framework for the creation of web sites and web apps;
- Bulma: a CSS open-source framework that provides ready-to-use front-end component to create websites and web apps;
- Ganache: a private Ethereum blockchain environment that allows to emulate the Ethereum blockchain so that is possible to interact with smart contracts in your own private blockchain;
- Truffle: a development environment framework and asset pipeline for blockchains using the Ethereum Virtual Machine (EVM). It has been used for smart contract compilation and deploying;
- Metamask: cryptocurrency wallet that enables users to access the Web3 ecosystem of the dApp;
- Web3.js: collection of libraries that allow to interact with a local ethereum node;
- MerkleTree: library that provides function for the implementation and verification of merkle trees.

FUNCTIONAL DEVELOPMENT

As we previously mentioned, the application implements a battleship game between two players. The battlefields consist of a matrix 8x8 and the ships are 5, of 1,2,3,4 and 5 length.

At the beginning the players should connect to their Metamask accounts and request to start a game to the smart contract. It's important to notice that there is the possibility for the players to play with a friend, asking it directly to the smart contract. In this case, the second player should indicate the game ID using a specific function.

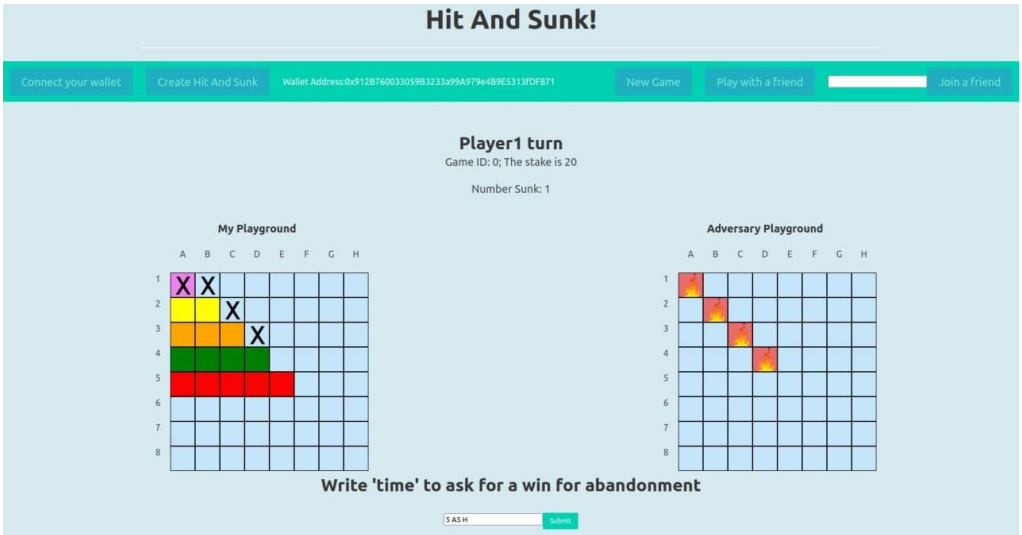
Once the smart contract matches two players, they start proposing a bet. Once the proposes of the players are the same, the smart contract emits an event indicating to the players the amount to pay to continue with the configuration steps. Once both the player have payed the bet, the disposition step starts.

The players decide the disposition of their ship on the battlefield, build the relative merkle tree and send the merkle root to the smart contract. This is necessary because they cannot send the clear position of the ship to the smart contract, otherwise it would be easy for the adversary to check the blockchain and obtain this information. If both the players have sent their merkle root, the game can begin.

Once a player tries a hit, the adversary sends the result of the try, appending the merkle proof demonstrating the correctness of the declaration. If the smart contract verifies correctly the proof, the player turn changes and so on until one of the two players sunks all the adversary ships.

At this point the winner cannot withdraw the reward yet because it should demonstrate the correctness of its initial disposition, sending the merkle proof of all the positions of its battlefield. Once the smart contract has verified the correctness of the merkle proofs, it definitively sets the winner and it can withdraw the reward.

To avoid the abandonment of a player, that would lead to the loss of the bet to the adversary too, a punishment mechanism have been implemented. Anytime during the adversary turn, a player can ask to the contract to punish the adversary. If a certain number of blocks is deployed on the blockchain before the adverary response, the player can ask again for punishment and automatically win the game. This is necessary to avoid that the money get stuck in the contract.



IMPLEMENTATION

Both a smart contract and a front end have been implemented. The smart contract have been defined to support 10 games to play. The language used for the smart contract is Solidity. The libraries used are MerkleProof.sol and SafeMath.sol, both belonging to the library set @openzeppelin. The first is used for the verification of the merkle proofs provided by the players and the second is used for the bound checking of the operations on the smart contract uint8 variables. A variety of constants have been defined to indicate the various states that the contract can meet, the values that the position of the battlefields can have and the direction of the ship on the battlefield. This is necessary to keep the code more readable.

```
23          uint constant verification_player1_win= 2;
24          uint constant verification_player2_win = 3;
25
26          uint constant match_over = 4;
27          uint constant wait_player1_outcome = 5;
28          uint constant wait_player2_outcome = 6;
29
30
31          uint constant wait_for_player = 8;
32          uint constant wait_for_friend = 9;
33
34          uint constant decide_reward = 10;
35
36          uint constant wait_both_transactions = 11;
37          uint constant wait_player1_transaction = 12;
38          uint constant wait_player2_transaction = 13;
39
40          uint constant wait_both_dispositions = 14;
41          uint constant wait_player1_disposition = 15;
42          uint constant wait_player2_disposition = 16;
43
44          uint constant match_payed = 17;
```

Then a set of variables used for the management of the game is defined. In particular the contract keeps track of:

- Players and winners;
- Merkle root of the players;
- Battlefield of the games;
- Information about the sunk ship of the games;
- Rewards set by the players on the the games;
- State of the game;
- Information about the last punishment request made by one of the two player of a game;
- Balance of the contract.

```
73          address [num_games] public player1;
74          address [num_games] public player2;
75          address payable [num_games] public winner;
76
77          bytes32[num_games] public merkle_root1;
78          bytes32[num_games] public merkle_root2;
79
80          uint16[2*num_cells*num_games] public game_matrix;
81
82          /*************** SCORE VARIABLE *****************/
83          // Here we save whether or not the player has sunk the ships of different dimension
84
85          bool[2*num_games] public sunk_1;
86          bool[2*num_games] public sunk_2;
87          bool[2*num_games] public sunk_3;
88          bool[2*num_games] public sunk_4;
89          bool[2*num_games] public sunk_5;
90
91
92          uint[num_games*2] public reward;
93          uint[num_games] public state;
94          uint[num_games] public last_inactivity_request;
95          uint public balance = 0;
```

For an efficient comunication between the front end and the smart contract I used the solidity event mechanism. A set of events have been defined and called into the smart contract functions, so that the front-end code - implementing listeners of events - can immediately know when something important has changed.

```
 98          event newGameInfo(uint game_ID, uint state_ID, address player);
 99          event changeReward(uint player, uint new_reward, uint gaID);
100          event amountToPay(uint amount, uint gaID);
101          event changeState(uint new_state, uint gaID);
102          event matchWinner(uint winner_player, uint gaID);
103          event hitTried(uint player, uint cell, uint gaID);
104          event gotWater(uint player, uint cell, uint gaID);
105          event hitShip(uint player, uint cell, uint gaID);
106          event sunkShip(uint player, uint cell, uint direction, uint dimension, uint gaID);
107          event cellVerified(uint player, uint cell, uint gaID);
108          event sunkVerified(uint player, uint number_sunk, uint gaID);
109          event abandonment_request(uint gaID, uint player);
```

In particular:

- newGameInfo is emitted when a new player has joined a game. This is necessary for the front-end code to understand when to go on to the reward set phase;
- changeReward is emitted when a player changes its propose. It is helpful for the players to coordinate each other, so that they know which is the propose of the adversary;
- amountToPay is emitted to advise the players about the sum to pay to go on to the disposition phase;

- changeState is emitted when an intermediate state is met, as for example when player 1 has payed the bet but not the player 2, or during the final verification phase;
- matchWinner is emitted when the final verification phase has ended correctly and the winning player can withdraw the reward, or when a punishment happens;
- hitTried is emitted when a player tries a position and the other player should respond about that position;
- gotWater, hitShip and sunkShip are emitted when the response is respectively water, hit or sunk. It is emitted only after the merkle proof has been verified;
- cellVerified is emitted during the final verification phase to indicate that the proof about a position has been correctly verified;
- sunkVerified is emitted during the final verification phase to indicate that the ship, that is not been found by the adversary during the game, has been correctly verified;
- abandonmentRequest is emitted when a player asks for the adversary punishment.

Later in the code, functions have been defined for the creation of new games, the setting of the reward, the setting of the merkle root of the disposition and the management of the game and the final verification stage, as described on the functional development part.

A particular focus should be set on the code implementing the verification of the merkle proofs. When a position should be verified, the player provides the merkle proof and the seed as input of the called functions. The seed is summed with the encoding of the position and given in input to the keccak256 function to obtain the hash. Later the hash and the proof are used as input of the verify function, that permit to continue the execution if the result is true.

An example of code is the one below, consisting of the verify_cell function used in the final verification phase. The keccak256 function is part of the MerkleProof.sol library while the add function is part of the SafeMath.sol function.

```
function verify_cell(bytes32[] calldata proof, uint8 seed, uint player, uint cell, uint game_ID) public returns(bool result)
{
    require (game_ID < num_games);
    require (cell < num_cells);
    require ((player == 1 && state[game_ID] == verification_player1_win && msg.sender == player1[game_ID]) ||
             (player == 2 && state[game_ID] == verification_player2_win && msg.sender == player2[game_ID]));

    require (game_matrix[2*game_ID*num_cells + num_cells*(player-1) + cell] == hit ||
             game_matrix[2*game_ID*num_cells + num_cells*(player-1) + cell] == not_asked);
    /*
        VERIFY THE CORRECTNESS OF THE CELL
    */
    bytes32 merkleLeaf = keccak256(abi.encodePacked(ship.add(seed)));
    if (player == 1) require (MerkleProof.verify(proof, merkle_root1[game_ID], merkleLeaf));
    else             require (MerkleProof.verify(proof, merkle_root2[game_ID], merkleLeaf));

    if  (game_matrix[2*game_ID*num_cells + num_cells*(player-1) + cell] == not_asked) game_matrix[2*game_ID*num_cells + num_cells*(play
    else game_matrix[2*game_ID*num_cells + num_cells*(player-1) + cell] = proven_asked;
    emit cellVerified(player, cell, game_ID);
    return true;
}
```

For the correct functioning of the dApp the smart contract should be deployed on the Ganache blockchain, so that the front-end code can interact with it.

For this purpose, the main framework used is web3.js, that provides a set of functions to communicate with the Metamask wallet and the Ganache blockchain. These functions are called by the frontend code to link to a Metamask account, so that the interaction with the smart contract can start.

```
const initWeb3 = async () =>
{
    console.log("Starting the initWeb3 function");
    if (typeof window != "undefined" && typeof window.ethereum !== "undefined")
    {
        try
        {
            await window.ethereum.request({method: "eth_requestAccounts"});

            const web3 = new Web3(window.ethereum);
            setWeb3(web3);

            const accounts = await web3.eth.getAccounts()
            setMyAddress(accounts[0]);
        }
        catch(err)
        {
            console.log(err.message);
        }
    }
    else alert("Metamask not installed");
    document.getElementById("Web3-button").disabled = true;
}
```

```
const initContract = async () =>
{
    if (!HContract && web3_obj)
    {
        const new_contract = HitAndSunkContract(web3_obj);
        if (!new_contract) console.log("New contract not created during the initContract phase");
        setContract(new_contract);
        document.getElementById("Contract_Button").disabled = true;
    }
    else
    {
        console.log("Contract already binded");
        return;
    }
}
```

The functions initWeb3 and initContract are linked to two buttons set on the user interface, so that the player can decide when to link to the Metamask wallet and setup the variables. Anyway these are necessary before calling the smart contract for a new game. When the variable HContract is modified, a React useEffect function is defined to automatically setup the smart contract events listeners, so that the front-end code can notice when some event is emitted by the smart contract. Of course an event listener is defined for each of the events defined on the smart contract. The function bindEvents() is used to abilitate the buttons whose are associated to the new game functions.

```
//Hook triggered by the setting of the contract that set the contract event listeners
useEffect(() =>
{
    if (HContract)
    {
        console.log("Setting contract event listeners");
        setContractEventListeners();
        bindEvents();
        init();
    }
}, [HContract]);
```

Once the player clicks on the new game button a new game request is sent to the smart contract, that matches it with another player that asked for a new game. When this happens, an input field on the user interface can be used by the player to write proposes about the reward of the game. A button can be used to send the propose to the smart contract. When the proposes of the players are the same, an amountToPay event occurs and a code for the payment is executed. Once both the payments are sent, the input field can be used by the players to decide the disposition of the ships on the battlefield. When a player has chosen all the ship dispositions, a code generates a seed array and uses it in conjuction with the battlefield data structure to generate the merkle tree. The merkle root of the relative merkle tree is then sent to the smart contract.

```
//PREPARATION OF THE MERKLE TREE

console.log(my_playground);
var seededGround = my_playground.map((elem, index) => elem + seeds[index]);
console.log(seededGround);
hashedGround = seededGround.map((selem) => Web3.utils.keccak256(Web3.utils.encodePacked(selem)));
myMerkleTree = new MerkleTree(
    hashedGround,
    Web3.utils.keccak256,
    {
        sortPairs: true,
    }
);
myMerkleRoot = myMerkleTree.getHexRoot();
console.log(myMerkleTree.toString());
console.log(myMerkleRoot);

HContract.methods.commit_disposition(player, GaID, myMerkleRoot).send({from: my_address});
console.log("Disposition Sent");
```

When both the players send their merkle root, the game can begin. At this point a piece of code associates with each of the buttons corresponding to the adversary playground of the user interface a function to try a hit, called tryHit, using as input the offset of the position on the battlefield. Each time a player has a try, the smart contract saves the information about the try and emits an event, listened by the other player so that it can respond. When the adversary provides the response about the guess, it executes a code generating the relative merkle proof and sends it with the seed found at the same offset. An example is shown below, consisting of the code about a try that got a water.

```
switch(my_playground[cell_tried])
{
    case water:
        console.log("The cell tried is water");
```

```
        var proof = myMerkleTree.getHexProof(hashedGround[cell_tried]);
        var leafSeed = seeds[cell_tried];

        console.log("The value used are: player, cell, game_ID, equal to " + player + " " + cell_tried + " " + GaID);

        HContract.methods.outcome_water(proof, leafSeed, player, cell_tried, GaID).send({from: my_address});
    break;
```

When a player sunk all the adversary ships, the smart contract emits a changeState event indicating that the final verification step can begin. Now the winning player provides the merkle proof of all the position in its disposition. This is important because if the player have cheated it would not be able to provide the proofs for all the positions and would go stuck on this phase. In this situation the adversary has the time to ask for a punishment toward the adversary. If the final verification step ends correctly the smart contract emits a matchWinner and the player automatically calls a function to withdraw the reward.

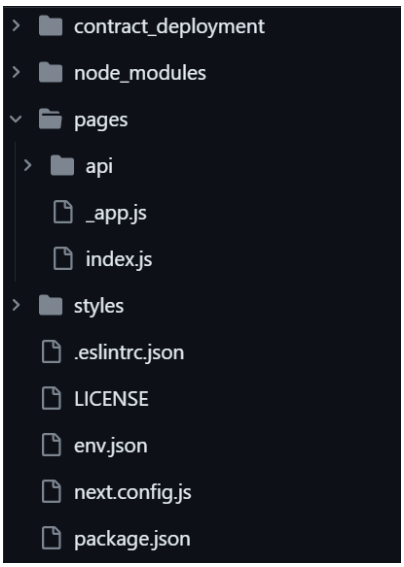
```
HContract.events.matchWinner().on("data", (event) =>
{
    if (last_event_num >= event.blockNumber)
    {
        console.log("We are receiving an already received event. Discard.");
        return;
    }
    last_event_num = event.blockNumber;
    console.log(event);
    if (event.returnValues["gaID"] != GaID) return;
    var event_winner = event.returnValues["winner_player"];
    if (player == event_winner)
    {
        var amount = 0;
        if ((player == 1 && state == wait_player2_transaction)||(player == 2 && state == wait_player1_transaction)) amount = reward;
        else amount = reward*2;

        HContract.methods.withdraw_reward(GaID, amount).send({from: my_address});
        alert("Congratulation! You won the game and withdraw the reward!");
    }
});
```

PROJECT DECISIONS

1. **Decision about manually checking the contract using view function or using the solidity events mechanism.**
   I found that the first approach would be so demanding and would waste time and energy checking the contract even when it doesn't change its internal values. The event mechanism is so useful in these cases where there are more entities that frequently interact with the smart contract because the front-end can act just when something has been updated, so I decided for this option;
2. **Decision about the implementation of the battleship datastructure as a vector or as a matrix**.
   The matrix implementation I tried didn't work correctly, so I decided to implement the data structure as a vector, doing the opportune checking each time the data structure must be handled;
3. **Decision about the number of positions to verify during the final verification step.**
   To have a correct functioning the positions should be checked all at the final verification step but for simplicity and to reduce the time needed to demonstrate the functioning of the game during the exam I decided to check only the ship position on the code. Anyway I am aware that some trick can be implemented and it should be changed in a real deployment;
4. **Decision about the division of the new game request phase, the reward set phase and the disposition phase.**
   I decided to divide them to obtain a more realistic behavior of the application. I thougth that a player would not necessarily know the amount of the bet at the beginning of the game and this implementation permits to reach an agreement with the adversary;
5. **Decision about the number of block that should be deployed to punish the adversary**.
   For easily demonstrate the functioning during the exam I set it at a low number but it should be augmented in a real deployment.

FILE SYSTEM

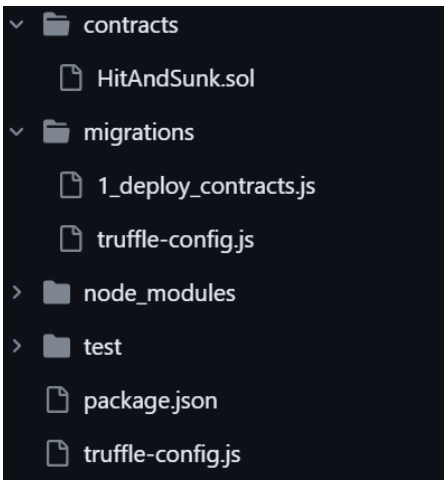
The file system corresponds to the one shown into the next image. The front-end code is uploaded in the pages folder. The style folder contains css code and images used for the implementation of the user interface.



The env.json file contains information about the address of the smart contract deployed and about the position of the smart contract abi in the file system.

The contract_development folder contains a set of folder needed for the correct deployment of the smart contract. The folder build/contracts have the abi of the smart contract (it is a json file), obtained once it has been compiled and deployed correctly on the blockchain. It is necessary for the front-end code to obtain a contract instance and understand how to properly interact with the smart contract.

The contracts folder contains the solidity files corresponding to the smart contract code. The smart contract can be deployed using truffle, based on the deployment file contained into the migrations folder (in the project it is 1_deploy_contracts.js). The truffle-config.js file is used by truffle to deploy the smart contract on the Ganache private blockchain.



MANUAL FOR THE SETUP

1. The first thing to do is to create a new blockchain with Ganache. It can be done opening it and selecting “new workspace”. Now we can add a project tapping on the relative button and select the truffle-config.js file contained in the “contract_deployment” folder of the project. Once it has been selected, click on “start” and the blockchain is created;
2. Now we should deploy the smart contract on the blockchain. To do this we should open a new terminal and navigate to the “contract_deployment” folder of the project. To compile and deploy the contract the command used is:
    - Truffle migrate –reset;

   If all is correct the smart contract now is deployed on the blockchain. Be sure to eventually delete the “build/contract” folder into “contract_deployment” if it is already present before this step;
3. Now that the contract is deployed we should copy the smart contract address (it can be found both on the terminal after the deployment or in Ganache on the information of the contract) into the “contractAddress” voice of the env.json file. This is important for the front-end code to communicate with the contract;
4. It is important to link (at least) two blockchain addresses generated by Ganache to the Metamask wallet. It can be done this way:
    a. tap on the key image of the Ganache address and copy the private key;
    b. open Metamask and tap “import account” on the account tab;
    c. paste the value on the field requested and confirm;
5. open (at least) two terminals on the main folder of the project and execute the command “npm run dev” on both. Two instances of the front-end code are now deployed on the ports 3000 and 3001;
6. We can now open two web pages and browse to “http://localhost:3000” and “http://localhost:3001”;
7. We are ready to start the game.

INSTRUCTIONS FOR THE DEMO

1. The first things to do are to tap on the “Connect your wallet” button first and the “Create Hit and Sunk” button later. The first one is used to link the application instance to one of the metamask addresses belonging to the Ganache blockchain. Of course it is important to choose two different accounts for the two players. The second button is used to create the playgrounds and to obtain a contract instance, used by the front-end code to interact with the smart contract;
2. Now we can choose one of the three buttons on the right, that are “New Game”, “Play with Friend” and “Join Friend”. The last one takes as input the value of the text input on its left. This one should contain the game ID of the game started by a friend. Once two players are matched to play, the application switches to a reward set phase;
3. During this phase the input text and the button on the bottom of the user interface can be used by the players to propose a bet. Once both the bets of the players are the same, the next step is to perform the payments, that should be confirmed using the Metamask interface. Now that both the payments are sent, the disposition phase can begin;
4. During the disposition phase the input on the bottom of the user interface can be used to set the positions of the ships on the battlefield, using the format “dimension position direction”. The dimension should be between 1 and 5 and the direction can be “H” to indicate horiziontal direction or “V” to indicate vertical direction. An example of ship setting is “3 B2 H” to indicate that the ship with length equal to 3 is set horizontally starting from the cell equal to B2. Once both the players have set all the ships and sent the disposition information to the smart contract, the game can begin;
5. During the game phase the input on the bottom of the user interface can be used to ask for adversary punishment to the smart contract. It can be done writing “time” on the text and pressing the button on the right. To have a try the player can simply click on the corresponding cell on the adversary battlefield. The adversary automatically sends the result of the try, appending the merkle proof. The game can go on until one of the players has sunk all the adversary ships. This event causes the beginning of the final verification phase;
6. During the final verification phase the player automatically sends all the proofs relative to the ships that have not been found by the adversary during the game. Once all the cells have been verified, the smart contract emits an event that advice the player about its winning. Now an automatic function call is requested by the front end code to withdraw the reward by the smart contract.

Of course it’s important to confirm all the transactions that Metamask tries to deploy on the blockchain during the application functioning.

POTENTIAL VULNERABILITIES

- Reentrancy attack: it is an attack that can leverage a smart contract code that update the balance after the calling of the payment function. This can be dangerous because an attacker can exploit this delayed update to execute sequencial and successful withdraws before the balance is actually modified. To avoid this kind of problems, the smart contract implemented update the balance before the calling of the transfer functions. Anyway deeper analysis on the smart contract code should be done in case of a real deployment;
- The final verification implemented on the smart contract is partial for simplicity reasons (only the ships are proved and not the positions of the water). Anyway this can lead to malicious behaviours of the players. A possible solution is to count the number of hit for each player and compare it with the actual number of positions that can contain a ship. A similar mechanism should be implemented before a real deployment to avoid malicious behaviors of the players;
- DoS: if the miners decide to not include a player transaction into the blocks to deploy, the adversary can ask for punishment and win the game;
- At the moment, the front end code doesn't provide any mechanism to recover informations about a previously started game. This means that if the player closes the user interface or crashes, it would not be able to continue the game and it would lose it. Before the deployment in a real environment a front end code should be implemented to face this kind of situations;
- At the moment the smart contract can support 10 games simultaneously, but there is not a mechanism to reuse data structures of old finshed games. Moreover two malicious users can decide to "occupy" all the game slots, resulting into a DoS attack. To reduce the impact of these malicious behaviours, some mechanism to remove the malicious occupations of game slots should be implemented. Moreover, a blacklist mechanism should be implemented to mark the players that abandon the games or that occupy the game slots for a long time;
- Over/underflow attack: the smart contract code implement either "manual" bound checking for the input and the results of operations or automatic checks using the SafeMath library. Anyway deeper analysis should be done in case of real deployment.

COSTS EVALUATION

The costs for the execution of the transaction have been calculated during a test on the functioning of the game. Some samples of prizes payed have been taken in account and the average costs for each function are shown below:

- New_game: 0.00024742 ETH, 90297 gas;
- Set_reward: 0.000021436 ETH, 81726 gas;
- Send_transaction: 0.00021003 ETH, 83994 gas;
- Send_disposition: 0.000212 ETH, 84780 gas;
- Try_hit: 0.00015754 ETH, 63000 gas;
- Outcome_sunk: 0.00033338 ETH, 123691 gas;
- Outcome_water: 0.0001923 ETH, 77214 gas;
- Outcome_hit: 0.00019358 ETH, 77214 gas;
- Adversary_quit: 0.00019065 ETH, 75577 gas;
- Verify_cell: 0.0001845 ETH, 71403 gas;
- Verify_sunk: 0.000449 ETH, 178207 gas;
- Withdraw_reward: 0.00017626 ETH, 70486 gas.

The max base fee and the max priority fee used by Metamask are 2.500000684 and 2.5.

Moreover a sample of a game consisting of both player missing all the possible tries has been taken in account. In this case a reward of 20 ETH in total was set (10 ETH for each player) and the final balance of the account were 109.9835 ETH for the winner and 89.9837 ETH for the loser.

The cost of the deployment of the contract is of 5043063 gas. The cost in terms of Ether for the deployment during the testing phase was 0.017020337625 ETH.