P2PBC: Hit and Sunk! Playing battleship on Ethereum

Andrea Franceschi (mat.: 665958)

February 11, 2024

Abstract

This project aims to **implement** a real-time **battleship game** for two players on the **Ethereum blockchain**, featuring a user interface inspired by the popular animated series "Rick and Morty".

Contents

1 Introduction

The battleship game is *deployed* on the **Ganache Ethereum blockchain**, where players can play *space battles* through **smart contracts** and *Web3-enabled* **front-end interfaces** on 10x10 grid.
The **back-end** logic is reflected by the **smart contract** written in *Solidity*.
The **front-end** is built using *JavaScript*, *HTML*, and *CSS*, showing an GUI inspired by "Rick and Morty" toon, with the **Web3** library to **interact** wtih **Ganache**, allowing players to interact with the game directly from the web browsers.

2 Project folder organization

- **contract**: contains the *Battleship.sol* file contract
- **build**: JSON file containing information about compiled and *deployed contract*
- **migrations**: migration script responsible for the *contract distribution* (truffle)
- **node_modules**: *third-party libraries* and *dependencies* that the project relies on
- **src**: contains the HTML, CSS, JavaScript code, and images: *frontend*
- **test**: includes the code to evaluate the *gas cost*

3 Front-end

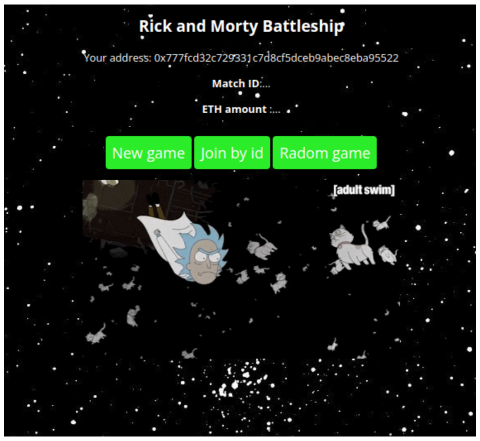
3.1 GUI

The front-end provides a graphical user interface (GUI) for players to engage a Battleship game on a 10x10 board with the following spaceships to be positioned on the board:

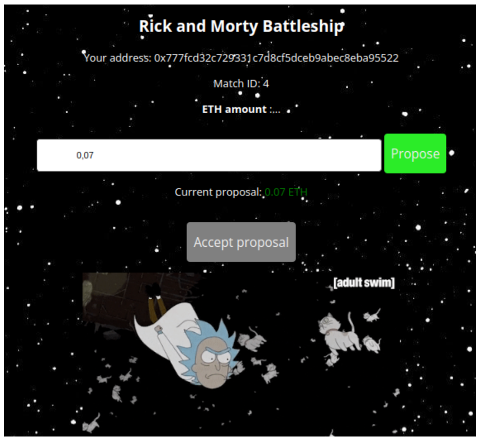
- Asimov's ship *(3x1)*
- Wormhole craft *(3x1)*
- Destroyer *(4x1)*
- Deathstar *(5x1)*
- War ship *(2x1)*

The GUI shows the following steps in order to set up a match:

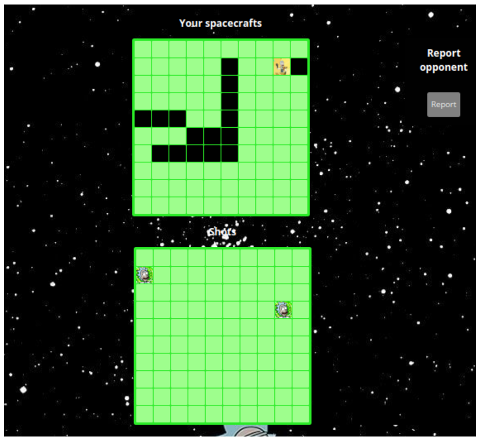
- a user must first create a new match and then wait for an opponent. The opponent has the option to join either by specifying the ID of the game or by joining a random match.
- once both players have joined the match, they are required to set a stake in ETH as an agreement to start the game.
- after setting the stakes and positioning their spacecrafts on the game board, the players must submit the Merkle Root of their respective boards to ensure fairness and prevent cheating.
- now the game begins and the players can finally enjoy battling against each other!


Rick and Morty Battleship
Your address: 0x777fcd32c729331c7d8cf5dceb9abec8eba95522
Match ID:...
ETH amount :...
New game
Join by id
Radom game
[adult swim]

(a) Welcome page


Rick and Morty Battleship
Your address: 0x777fcd32c729331c7d8cf5dceb9abec8eba95522
Match ID: 4
ETH amount :...
0,07
Propose
Current proposal: 0.07 ETH
Accept proposal
[adult swim]

(b) Created match



(c) Play

Figure 1: GUI flow

3.2 Interaction

Since the project requires *user interactions* to fire sequence of *actions* that must be processed by the smart contract, the main logic has *event-driven* designed using JQuery for DOM manipulation and event handling.
This allows the interaction with HTML elements, so users can **trigger actions** by clicking buttons. Subsequently, these user actions *prompt the execution* of corresponding *functions*, orchestrating interactions with the smart contract. These functions encompass essential operations like creating a **new match**, joining a **random game**, set and confirm **stake proposals**, send an **attack** or accuse the opponent of **cheating**.
For *seamless integration* with the **smart contract**, **Web3**.js emerges as choose, facilitating the execution of game-related transactions and **state updates**.
To encapsulate the front-end **logic** and **interactions** with the smart contract, all relevant code resides within the "**game.js**" file.
In order to check the **fairness** of the player, the first thing to check is that the players have **submitted** correctly their **board**, this was achieved thanks to the **Merkle Tree**, a data structure that verify the **integrity** of data in distributed systems, provides a compact and **secure representation of the game board**'s state for *verification.* Since a plyar's board is a matrix 10x10 that has to be flatten in order to compute the Merkle Tree, and the cost to buil it is O(n), roughly each single construction roughly costs 100xC (where C is a constant to compute the hashing operation - **Keccack256**)
The **first choice** was to exploit the **Lanyard** organization APIs to address this objective, but after have included Lanyard libraries, the functions were **not visibile** inside the Web3 *application.*
To **address this problem**, the function responsible to generate the Merkle Tree has been imple-

mented converting the pseudocode in JavaScript(see 8), in particular takes a playerGrid as input, which represents the game board of a player and:

1. flattens the grid cells of the `playerGrid` to create a one-dimensional array,
2. generates leaf nodes by mapping each cell state to a hashed value. The hash computed by concatenating the cell state with a randomly generated salt and then hashing the result using the `keccak256` function provided by `window.web3Utils`.
3. initializes an array with the leaf nodes.
4. the Merkle tree is given by repeatedly combining neighboring leaf nodes to compute parent nodes until only one root node remains.
5. in each iteration of building the MT, it pairs leaf nodes pairwise, computes the hash of their combination using the XOR function, and pushes the resulting hash to the next level (`lastLevel`) of the MT.
6. updates the leaf nodes with the parent nodes obtained from the previous step.
7. stores the current level of nodes in the `merkleTreeLevels` array.
8. finally, it returns the root of the Merkle tree, which is the first element of the `leafNodes` array.

4 Smart contract

The smart contract has been implemented in Solidity.
In order to represent *game-related data*, a `struct` called `Battle` has been defined.
By declaring a *publicly accessible* array named `gamesArray` that holds elements of type `Battle`, the contract enables the retrieval of information about each match. The **position** of each match's data in the array corresponds to its unique match ID.

4.1 Execution flow

To manage the game **execution flow**, the contract has been implemented in the following way:

1. **Creating a Match**:
   a) a player **initiates** the game by creating a match using the `createMatch` function, providing the *board size* and the *number* of *ships* (the contract allows to pass custom board size and number of ship, modifyng the parameters in the file game.js) initializing its parameters and adding it to the list of active matches.
   b) it increments the counter `currentGames`, indicating the number of active matches.
   c) emits an event `newMatchCreated` to notify the creation of a new match, providing details such as the address of the match proposer *(msg.sender)* and match ID.
2. **Joining a Match**:
   another player joins the match either by explicitly selecting a joinable match or by joining a random joinable match:
   a) `JoinMatch` allows a player to join an existing match identified by match ID, verifying the match is joinable, the adresses of the players and the creator of the match is not in another game
      - it emits a `playersJoined` event to notify that both players have joined the match
   b) `randomGame` verifies the current player is able to join the match (not in other games)
      - it calls a private function (`findJoinableMatch`) that uses a *random value* to select a match, ensuring that each joinable match has an equal chance of being selected:

    – the `randomValue` function hashes together the current block's: hash, timestamp and the base fee, using the `keccak256` hashing algorithm

  - it emits an event `playersJoined` indicating that the players have joined the match

3. **Proposing and Accepting Stake**:
   once both players have joined the match, one of them proposes a stake for the match using the `proposeStake` function:

   a) verifies that transaction's sender (*msg.sender*) is one of the players involved in the match

   b) the opponent can rise up the stake, forwarding the desired amount of ETH to the player (handled by the front-end)

   c) it emits an event `stakeProposal` to notify that a stake proposal has been made

   d) the other player then accepts the proposed stake using the `stakeConfirm` function.

      - second step in confirming and accepting the proposed stake of that match.
      - it ensures that the sender is not the one who proposed the stake for this match (otherwise aborts, can't confirm their own stake)
      - it sets the stake amount for the match to be equal to the previously proposed stake amount `matchInstance.stakeProposal`
      - emits an event `stakeAccepted` to notify that the stake proposal has been accepted for the specified match

4. **Sending ETH as Stake**:

   a) both players send their stakes to the contract using the `sendEth` public and payable function so can be called from outside the contract and can receive ETH along with the call

   b) ensures that only valid players can participate, and the amount of Ether sent is greater than 0

   c) it updates the stake of the correponding player in the array of matches

   d) if the sender matches the *playerX* for the specified match, the value of ETH sent is added to *stakeX*

   e) if the sender is **not** *playerX*, it implies the sender is *playerY*, so the value of Ether sent is added to stakeY.

5. **Starting the Match**:

   a) the game can start iff both players have **submitted** their **Merkle Roots**, this is achived by the function `sendMerkleRoot`:

   b) if the sender is *playerX*, it sets the Merkle root for that player, otherwise, it sets the Merkle root for playerY.

   c) if both players have provided their Merkle roots (and are not zero), it sets the `startedMatch` flag to true, indicating that the match has started emitting the event `matchStarted`

6. **Performing Attacks**:

   a) players take turns attacking each other's board using the `shot` function.

   b) the attacker specifies the row and column of the opponent's board to attack

   c) it checks whether it's the sender's turn to attack by comparing the current player turn stored in the match instance with the sender's address

   d) it determines the opponent based on the sender's address. If the sender is *playerX*, the opponent is *playerY*, and vice versa.

   e) it emits an `attackPerformed` event to notify that an attack has been performed

7. **Submitting Attack Proof**:

   a) after performing an attack, the attacker submits an attack proof using the `submitAttackProof` function

b) the proof includes the result of the attack and a Merkle Proof to validate the attack.

c) it calculates the computed Merkle root using the provided Merkle proof

d) retrieves the player's Merkle root and the number of their remaining ships, comparing the computed Merkle root with the player's Merkle root to validate the attack:

- if the Merkle Roots match, the attack is valid
- otherwise, the match finishes

e) also checks if either player has **no ships left**. If so, it determines the **winner** based on the **remaining ships**, sets the match status as finished, and emits a *matchFinished* event.

8. **Verifying the Board**:
it verifies the correctness of the opponent's game board submission and determine the winner or loser based on the game

a) when the match is finished the function `verifyBoard` is invoked

b) it determines the winner and loser based on the remaining ships of each player.

c) if there's no winner (indicating cheating), the function reverts.

d) if no cheating is detected, the reward is transferred to the winner through `transferReward`.

9. **Accusing Opponent**:

a) if a player suspects the opponent of cheating, they can accuse them using the `accuseOpponent` function.

b) checks if a timeout for accusation has been set. If so, it verifies if more than 5 blocks have passed since then.

c) if the timeout is exceeded, it declares a winner based on who was accused and ends the match due to an AFK (Away From Keyboard) timeout

- the stake is transferred to the winner's account

10. **Ending the Match**:

a) the match ends when one player has sunk all the opponent's ships or when a player has been accused (and is confirmed) to be a cheater.

b) The winner receives the stakes from both players as rewards to its address (the double of the stake)

5 Vulnerabilities

1. **Reentrancy:**

   - to mitigate this potential vulnerability, as we seen during the course, **instead** of using `call.value()`, the contract uses the function `transfer()` for Ether transfers, which do not allow reentrancy.
   - **methods** have been implemented ensuring that **no internal state updates** happen after an ETH **transfer** or an external function call inside a method
   - **balances** of users are **updated** prior to the transfer to prevent it.

2. **Aritmetic Overflow/Underflow:**

   – the contract does **not** directly **utilize** SafeMath library for arithmetic operations, which would ensure that aritmetic overflow and underflow vulnerabilities arise, but this kind of attacks are affected by versions of Solidity smaller that 0.8.*. Using **greater versions** (Solidity - 0.8.19 solc-js), the compiler automatically performs the **checks**.

   – have been included *modifiers* and *function parameters* to **validate inputs** and ensure that **values passed to arithmetic operations** are within **expected ranges**. For example, the `validSize` modifier **ensures** that the **board size** provided when creating a match is **greater than 0**, preventing potential **underflow** vulnerabilities.

3. **Front-Running:**

   – to mitigate this vulerability, the **commit-reveal** scheme was adopted

   – for instance in the `sendMerkleRoot` function, players initially send a transaction with their Merkle root data, which is the commitment part (*commit*). This data is stored in the contract but not immediately revealed to the other player.

   – after both players have **committed** their Merkle root data, the match is started by both players revealing (*reveal*) their committed data. This is done through the `sendMerkleRoot` function again.

4. **Phishing:**

   – by relying only on *msg.sender* for **authentication** and avoiding the use of `tx.origin`, the contract effectively **prevents** phishing attacks that attempt to exploit the *tx.origin* global variable

   – this ensures that only the immediate caller of a function is authenticated, mitigating the risk of impersonation by malicious DApps.

5. **Unprotected Ether Withdrawal:**

   – ether withdrawal is handled securely within the contract functions, ensuring that **only authorized actions** trigger fund **transfers**. The `transferRewards` function, for example, checks for *cheating detection* before transferring rewards, ensuring that funds are only transferred when the game outcome is legitimate.

6. **Uninitialized Storage Variables:**

   – the contract **initializes** all **storage** variables properly, reducing the risk of *uninitialized storage variables* leading to unexpected behavior. So the contract ensures **predictable behavior** and prevents unexpected state changes.

7. **Unchecked External Calls:**

   – there is *no involve to any external calls* to **untrusted contracts** or interfaces, minimizing the risk of **unchecked external calls** leading to **unexpected behavior**. By avoiding external dependencies and interactions with untrusted contracts, the contract reduces the **attack surface** and ensures the security of *its operations.*

6 Gas Cost

Gas evaluation was conducted for a game played on an **8x8 board**, considering scenarios where *each player misses all shots*, with 3 *warships* positioned on each player's grid. Consequently, each function consumes the *following amounts of gas*:

- `createMatch`: 243370
- `joinMatch`: 40734
- `randomGame`: 43938
- `proposeStake`: 72227
- `stakeConfirm`: 56501
- `sendEth`: 48726
- `sendMerkleRoot`: 51042
- `shot`: 40421
- `accuseOpponent`: 44865
- `verityBoard`: 52106
- **total**: **693930**

7 User guide

In order to set up and run the application:

- *install* and *run* Ganache
- i*nstall* Metamask extension on two browsers and *configure* two different wallet addresses (from your Ganache account)
- proceed to the main project folder
  - `cd BattleshipR-M`
  - *install npm* with your package manager
- run the application
  - `npm start`
  - if your browser doesn't fire up a window, join to http://localhost:3000/ (browserA)
  - reach the same address in the browserB
- in the browserA click on *play!* button
- in the browserA click on *New game* button (match ID is returned, say x)
- in the browserB click on *Join by id* specifying the id x (or join randomly)
- with one of the browser *propose a stake*
- *accept* or *rise up* the stake
- *place the ships* in both browsers and *play*!

8 Contribution

8.1 Merkle Tree

This function constructs a Merkle tree from the player grid data

```
 merkleTree: async function (playerGrid) {
    let boardForMT = playerGrid.cells.flat(); // flatten the grid cells

    // generate leaf nodes using hashed vales
    let leafNodes = boardForMT.map(cellState => {
      let salt = BigInt(Math.floor(Number(generateSecure128BitInteger()) * playerGrid.size)); // gene

      // concatenate the cell state and salt, then hash the result
      const val = window.web3Utils.keccak256(String(cellState) + String(salt));
      return val;
    });

    merkleTreeLevels = [leafNodes]; // array to store the LEVELS of the MT

    while (leafNodes.length > 1) { // build MT
      let lastLevel = [];

      // iterate through leaf nodes pairwise to compute parent nodes
      for (let i = 0; i < leafNodes.length; i += 2) {

        let leftChild = leafNodes[i];
        let rightChild;
          if (i + 1 < leafNodes.length) {
            rightChild = leafNodes[i + 1];
          } else {
            rightChild = leftChild;
          }

         // combine the hashes of left and right children
        let combinedHash = window.web3Utils.keccak256(xor(String(leftChild), String(rightChild)));
        lastLevel.push(combinedHash); // push the combined hash to the last level
      }
      // udate leaf nodes with the parent nodes
      leafNodes = lastLevel;
      // store the current level in the Merkle tree levels array
      merkleTreeLevels.push(leafNodes);
    }
    return leafNodes[0]; // ROOT of the MT
```

8.2 Merkle Proof

Merkle proof is verified after each turn

```
merkleProof: function(row, col) {
    var merkleProof = []; // array to store the MP
    let flatIndex = row * playerGrid.size + col; // calculate the flat index of the cell

    // iterate thru levels of the MT
    for (var arr of merkleTreeLevels) {
```

```
  if (arr.length > 1) { // if current level has more than one node
    // index of the sibling node
    let siblingIndex = flatIndex % 2 === 0 ? flatIndex + 1 : flatIndex - 1;

    // sibling node added to the proof
    merkleProof.push(arr[siblingIndex].toString());

    // update flat index to the parent node
    flatIndex = Math.floor(flatIndex / 2);
  }
}
return merkleProof;
```