Name: Luca G. Pinta
Matricola: 579458

University of Pisa
Peer to Peer & Blockchains
A.Y. 2022-23

Battleship on Ethereum - Project Report
Hit and Sunk!

Contents

1 Implementation

1.1 Front-end

The front-end has been developed by refactoring the initial UI design from the open-source project Battleboat, modified and adapted to be able to use it in the context of an *Battleship contract* interaction. The UI allows to manage a *Battleship* match over a 10x10 board with 5 ships of size:

1. *Patrol boat*: size 2×1
2. *Submarine*: size 3×1
3. *Destroyer*: size 3×1
4. *Battleship*: size 4×1
5. *Aircraft Carrier*: size 5×1

More details on how the contract manage the number and the sizes of the ships are described later in 1.2.
The code follows an *event-based* approach: for local functions, UI update and contract interaction events are registered and relative callbacks are invoked. There isn't a strong separation between function that manage the contract and UI update due to the particular application structure, executed as a **single anonymous function** (*composed by several different utils functions*) registered at application startup .

The front-end functions manage, aside from the UI interaction and update, the creation of the board's **Merkle tree**: one relevant details is represented by the r_i random salt, that according to the paper *"Crypto-Battleships" from University of Israel*, must be *at least* 128 bit to make hard to guess the s_i value. To this aim, the Web Crypto API native function `Crypto.getRandomValues()` has been applied over an `Uint8Array` of 16 bytes to generate the random salt with *high-min entropy distribution*, despite an array of 32 bytes can be used to extend the difficulty of guessing the generated values, obtaining the cells state value.
The **Merkle tree** creation has been implemented from scratch due to the *anonymous single function* structure of the application already mentioned: the use of third-party library was possible only by using the relative CDNs and injecting the dependencies throught the corresponding `index.html`. The OpenZeppelin/merkle-tree library has been evaluated but it's not available through CDNs so it's not injectable; the merkletreejs/merkletreejs despite being available throught CDNs it's not recommended because it's vulnerable to *second pre-image attack and forgery attack for unbalanced tree*. The adopted implementation, both contract-side and frontend-side, is briefly described in Merkle Proofs explained: frontend-side has been used the `web3Utils.keccak256` utils functions, implementing the checking conditions to generate a *balanced Merkle tree*, despite the *unbalanced* scenario is still possible contract-side, as discussed later in 3. Details on contract-side implementation are provided in 1.2.1.

1.2 Smart Contract

The contract has been developed as a single, unique contract `Battleship` that manages multiple `Match`, stored as an `array` named `matchList`. Each `Match` is identified by a `matchID` which is the index position of a given `Match` inside the `matchList` array. This solution has been adopted to avoid costly solutions in term of Gas to manage the `Match` addressing.

Despite the UI allows to play on a 10x10 board with 5 ships, the contract has been developed to manage games on multiple board size with different number of ships. The sizes of the ships can be arbitrary (*if inside the board size limit of* $10x10$) because the contract manage each ship as a set of *cells* of 1x1 dimension each: *to clarify*, by referring the sizes in 1.1, the contract would see 17 (*5+4+3+3+2*) **ships of** 1x1 **each** and not only 5 ships, allowing on client-side multiple complex ships configuration.
The totality of contract's methods use Solidity `modifiers` to verify the correctness of information received by the UI, assumed as *untrusted* source of information: this allows both to reduce the Gas usage by checking working conditions in each `modifier` and simplify the code in each method.

1.2.1 Execution logic

An *honest-players* gameplay execution is pictured in 1, showing each method invokation and events emitted by the contract.

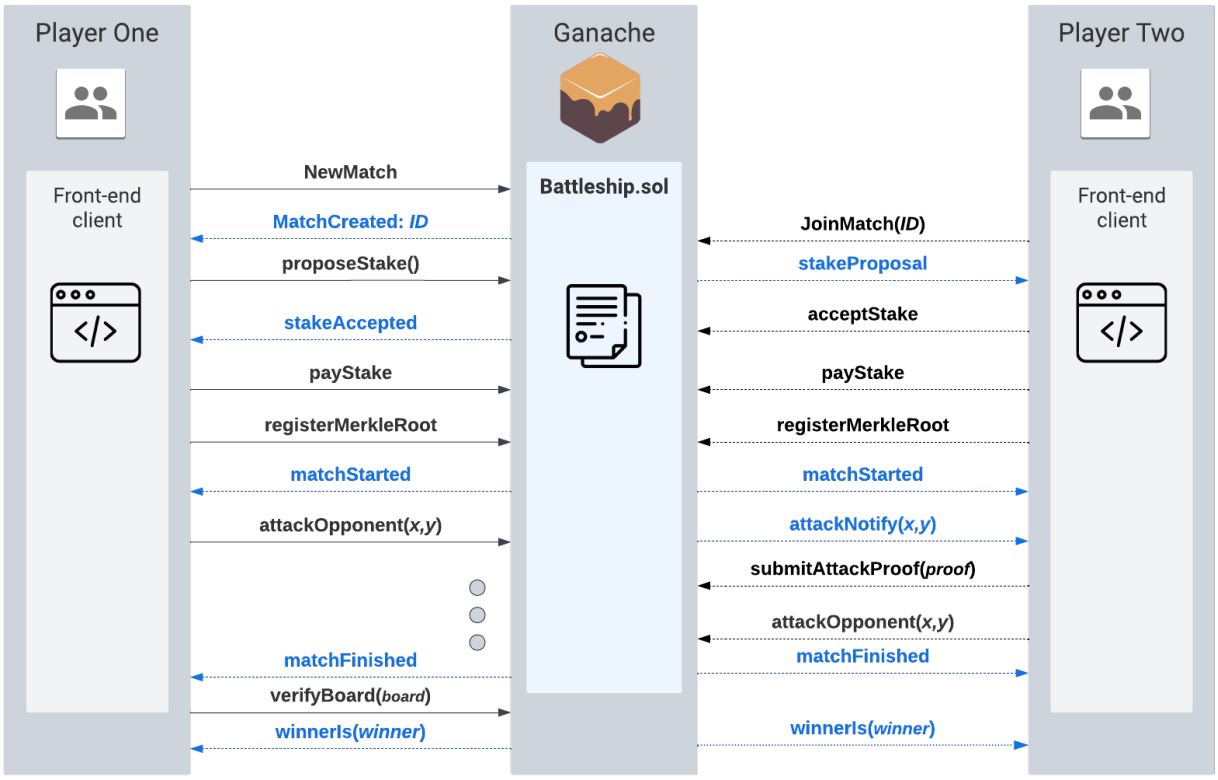

Figure 1: *General workflow schema:* events are in blue, method invokation in black

A player can create a new match by invoking `NewMatch` or join a preexisting match by invoking `JoinMatch` or `JoinRandom`: in the former case, the negotiation stage is performed while in the latter case the negotiation can be carried out only if the match's owner haven't still proposed a *stake amount*, otherwise the joining player is forced to accept the proposed amount from the match's owner without negotiate.

After the negotiation stage, executed by both players invoking `proposeStake, acceptStake` and finally `payStake` to commit the amount to the contract instance, the `registerMerkleRoot` allows players to commit the board to the contract: when both players executed the method, the event `matchStarted` is fired.
Each player launch a torpedo by invoking the method `attackOpponent` which notify to the opponent (*throught the contract that emit the* `attackNotify` *event*) the coordinates of the attack. The opponent, when notified, first send the *Merkle proof* for the shot taken at turn $i-1$ by invoking `submitAttackProof` and then it's allowed to launch an attack for the turn i, with $i \in [1, boardSize^2]$.
The method `submitAttackProof` execute the verification phase without using third-party libraries: from the given *Merkle proof*, recompute the *Merkle root* and compare it against the first registered via `registerMerkleRoot`. The implementation reflect the specific application of the more generalized method to verify a given proof against an already stored *Merkle root*, as cited in 1.1. For each attack, `submitAttackProof` update the residual number of ships of each player, verifying that the number of ships not hitted allows the match to continue.

When this condition it's not satisfied, so a *candidate winner* is identified by the contract instance, the `matchFinished` event is fired to notify both players: the verification phase is performed by the candidate winner player by invoking the `verifyBoard` method, passing the flattened board (*an* `array`), to verify board condition like *board size* and *no ships intersection*. No furter verification are required because the verification on the Merkle proof against the registered *Merkle Root* have been executed after each turn, avoiding the final check on initial *board configuration*. The ship representation adopted contract-side (*described before in 1.2*) allows to simplify those two last verification on the board. The verification phase is concluded by emitting the event `winnerIs` to notify both the loser and the winner (*even in case of cheat detection, reporting the* `cause` *message to both players*).

The *penalty mechanism* of accusing opponent to have left can be triggered only during attack phases, once when both players have committed their board: in case of cheat or timeout passed, the stake is accredited directly to opponent's address without further verifications on the match board. More details on the penalty mechanism are provided in 3.

2 Gas Evaluation

In table 1 are reported the total cost of the gas per method. As required, the evaluation is based on a match with a board of 8x8 with 10 ships per player, assuming that each player *misses* all the guesses.

2.0.1 Test customization

The evaluated costs have been generate by executing `gas_test.js` under `test` folder. It's possible to customize the context of the gas test by changing the following configuration:

```
1  const board = {
2    size: 8,
3    cells: [
4      [0, 0, 0, 0, 0, 0, 0, 0],
5      [0, 0, 0, 0, 0, 0, 0, 0],
6      [0, 0, 0, 0, 0, 0, 0, 0],
```

```
      [0, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 1, 1],
      [1, 1, 1, 1, 1, 1, 1, 1]
    ],
    shipNumber: 10,
  };
```

To execute the test and replicate the data execute in the root folder the command `truffle test` or `npm test`.
Among the gas cost listed in table 1, it's important to highlight that three function have the highest cost for different reason: `NewMatch` due to the creation of a new `Match` object and insertion in the `matchList`; the `proposeStake` function because despite its simplicity, it's invoked multiple times by both the players; finally, the `verifyBoard` because verify the *flattened board* received by the winner, bringing complexity to $O(n^2)$, where n is the board size.
Although the implementation of some methods could be further streamlined, the cost of gas per operation remains in line with that reported in EVM - OpCodes Gas cost.

Method name: *brief info*	**Gas used**
`NewMatch`: create match	243153
`JoinMatch`: join existing match	40814
`JoinRandom`: join random match	48692
`proposeStake`: propose amount	74764
`acceptStake`: accept opponent amount	54447
`payStake`: commit amount to contract	48528
`registerMerkleRoot`: send Merkle root	51064
`attackOpponent`: attack at coordinate (x, y)	40782
`submitAttackProof`: verify $(i-1)^{th}$ turn `Merkle Proof`	49054
`verifyBoard`: send final board for verification	76734
`accuseOpponent`: report opponent to have left	44759
Total `Battleship` cost	**772791**

Table 1: Total Gas cost per method

3 Vulnerabilities

Here are listed the main possible vulnerabilities and the methods implemented to *minimize the risk* of fully carry out an attack:

3.0.1 Re-entrancy attack

The risk for this type of attack has been reduced by applying the *"Checks-Effects-Interactions"* pattern for the methods that execute the `transfer` function.

It consists of three phases:

- *Checks*: Verify inputs, conditions, and permissions before proceeding. Ensure the function's prerequisites are met to avoid unintended behavior.
- *Effects*: Make state changes and perform computations within the contract. Update the contract's internal state variables and data structures.
- *Interactions*: Carry out fund transfers. This ensure that all state changes are finalized before interactions occur.

More information about the specific pattern can be found at at Solidity Docs - Checks Effects Interactions Pattern.

3.0.2 Arithmetic over/under flow attack

The risk for this type of attack has been reduced by manipulating both the `proposedStake` and, after the negotiation stage, the match `stake` as separated entities until the stake is accepted by both parts. The amounts are declared as `uint` and for each function that operate on them has been extensively used the previously mentioned 1.2 `modifiers` to verify the *non-zero*, *positive*, *players-owned* condition. Furthermore, the contract `Battleship.sol` has been compiled used the `solc 0.8.19` compiler and, as the SafeMath.sol developers stated: *"SafeMath is no longer needed starting with Solidity 0.8.+. The compiler now has built in overflow checking"* so the operations are implicitly safe and the use of `SafeMath` library has been avoided to not perform redundant checks on `uint` manipulation even library-side.

3.0.3 Forgery attack for Unbalanced Tree

The implementation is vulnerable to a *forgery attack for an unbalanced tree*, where the last leaf node can be duplicated to create an artificial balanced tree, resulting in the same *Merkle root hash* so the attacker is able to create a *forged proof* that seemingly validates an incorrect data element in the tree, leading to a successful validation despite the incorrect data. A possible solution would be to store contract-side all the levels of generated *Merkle Tree* and store the root only if the resulting tree is balanced, despite this solution imply higher costs of contract execution and added complexity. More information at BitcoinTalk - Block Merkle calculation exploit.

3.0.4 Locking funds

The current implementation of reporting opponent for having left the game allow to manage one accusation at a time, thus if both players accuse each other every round the previous report action is cancelled. The implemented *penalty mechanism* is considered secure if the opponent has left the game and does not perform any action: if the opponent reply by reporting the player, the funds are trapped in the contract and this does not allows to terminate the match to release the funds to any of the player.
A first possible solution to this problem could be to decouple the player reporting functionality, implementing additional checks on the time intervals between one report and the next by the same user, thus identifying the player's incorrect behavior.

3.0.5 Placement and configuration cheat

The winner is forced to validate the original matrix used during the match by using the `verifyBoard` function of the contract. This does not allow an advantageous placement of the ships (*e.g. by superimposing the ships on the same cells*) or to change dynamically ship size and position during the match because at each turn i, the opponent is forced to send the *Merkle proof* of turn $i-1$, checking step by step the *attacks* evolution. The adopted methods reduce the set of possible cheating scenarios.

3.0.6 Unsecure salt generator

The implemented frontend generate a salt r_i of 128 bit, as described in 1.1. In case of a malicious/infected client, it's possible to generate a *weak* or *predictable salt* (*e.g., using a constant value, a simple algorithm, or a short string*), so an attacker could potentially predict the salt and reverse-engineer totally or partially the board configuration of the infected opponent by *brute-forcing* all the possible *Merkle tree* bit configurations.

4 User manual

First start `Ganache` or `truffle develop`. Modify the `truffle-config.js` specifying `host` and `port` of the running `Ganache` instance, as shown:

```
development: {
  host: "127.0.0.1",     // Localhost (default: none)
  port: 7545,            // Standard Ethereum port (default: none)
  network_id: "*",       // Any network (default: none)
},
```

In the root folder execute `truffle migrate` to compile and migrate the contract on the configured instance and `npm install ci` to install the UI dependencies. Finally, by executing `npm run start` the address `http://localhost:3000` will be automatically opened on the default browser.

Regarding the match parameters configuration, the **default ships number** is 17 (*accordingly to the contract-side management of the ships described in 1.2*) with a board size of 10 (*the upper limit size of the board allowed*) but those parameters can be edited by modifying the file `/src/bs-config.json` accordingly, as here reported:

```
{
  "server": {
    "baseDir": ["./src", "./build/contracts"]
  },
  "gameParam":{
    "boardSize": 10,
    "numberOfShips": 17
  },
  "ghostMode": false

}
```

Listing 1: JSON board parameters configuration in `/src/bs-config.json`

The parameter modification **does not change the UI configuration** that remain fixed to the default parameters (*listed in Listing 1*) but, changing the number of ships (*e.g. to* 3) allows a

faster gameplay because, despite the UI displayed configuratio, after 3 ship hitted, the match is considered finished by the contract.

5 Demo

Open http://localhost:3000 from two different browsers with *Metamask* configured on the correct `Ganache` network or an *incognito window* with a **different wallet selected from the default window**. Using two different browsers have shown an incremental latency on long matches while interacting with the contract instance. Differently, within the same browser the execution is highly responsive.
The steps for playing assume that at each step both players confirm the transaction relative to the action they're performing:

1. Initiate a new match by clicking on *"New Match"* button or join a match by ID/randomly. The specific *match ID* is displayed on the top page in the first window.
2. Propose a stake as player A.
3. Accept the proposal or propose an alternative stake as player B.
4. Position the required ships by clicking on the ship name and moving the cursor over your board that will show the ship shadow. Alternatively, click on *"Place randomly and start"* to generate a random ships placement.
5. Alternate turns to shoot at your opponent's board by clicking on the cells of board under *"Enemy fleet"*: when enabled to shoot the cursor will turn into a crosshair.
6. When one of the players achieves the predetermined count of sunk ships, the contract will declare a *candidate winner*, notifying both players.
7. The winner must then submit their match board for verification: this is done by accepting the relative transaction, without perform any additional step.
8. If the checks on the configuration of the board and on its size return a positive result, the winner will receive the credit of the agreed stake. Differently, in the event of a cheat, the agreed sum will go to the opponent without further checks on the validity of his board. In both cases, winner and loser will be notified by an UI alert that terminate the match.

The main page shown a *simplified tutorial* to play the game, each step is described by the **UI alert** that indicates the next step to guide the user through the match.