Chainship - Architecture Plan

Filip Krawczyk, Index: 448308

May 8, 2025

[Programming with Blockchain]

1 Introduction

This document outlines the architecture for Chainship, a decentralized Battleship game implemented on EVM-compatible blockchains. The primary goal is to create a secure, transparent, and fair gaming environment where players can compete for cryptocurrency stakes. By leveraging smart contracts, Chainship eliminates the need for a trusted central server, ensuring that game logic execution, fund handling, and result verification are managed entirely on-chain.

The core component is the `Chainship` smart contract, which manages game rooms, player commitments, turn progression, and prize distribution. Cryptographic techniques, such as commitment schemes (using `keccak256`), are employed to hide sensitive information (e.g., ship placements) until necessary for verification, thereby preventing cheating. The system relies on a frontend interface that interacts with the user's Metamask wallet to communicate with the deployed smart contract.

2 System Architecture

2.1 Components

The system comprises two main components:

1. **Frontend Application:** A web-based interface built with React, running in the user's browser. It provides the game UI (board, ship placement, shooting controls) and interacts with Metamask for blockchain communication.
2. **Blockchain (Ethereum or EVM-compatible):** The decentralized network where the Chainship smart contracts are deployed and executed.
    - **Chainship Smart Contract:** The core logic resides here, managing game states, rules, and funds. It includes the abstract `Chainship` contract and a concrete `ChainshipImplementation` (potentially utilizing a `Multicall` contract for batching function calls to improve gas efficiency and user experience, though not essential for core functionality).

- **Scoreboard Contract:** A contract that keeps track of the number of points for each player. It can be connected to multiple Chainship contracts, allowing users to see their statistics across different game settings or even multiple games in the future.

2.2 Architecture Diagram

The following diagram illustrates the high-level interaction between components:

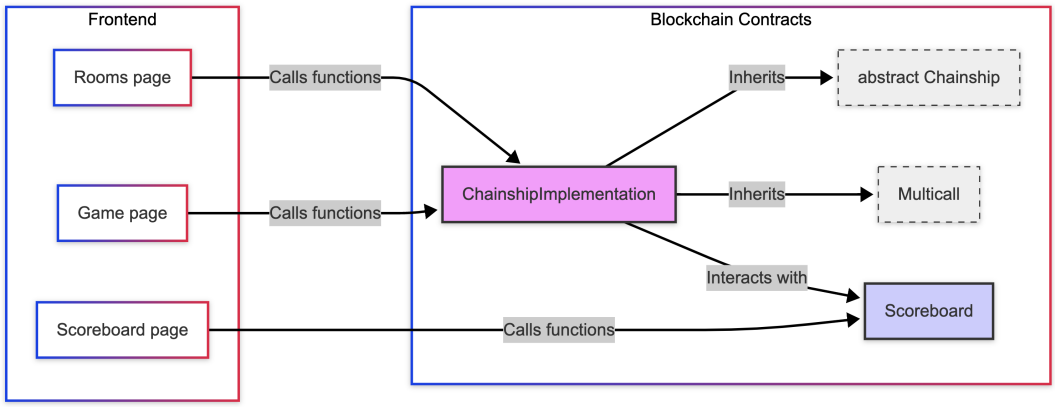

Figure 1: High-Level System Architecture

2.3 User Interaction Flow

A typical game flow involves a sequence of multiple interactions among users, the frontend, Metamask, and the Chainship contract. For simplicity, the diagram below shows only the actions of shooting and answering a shot.

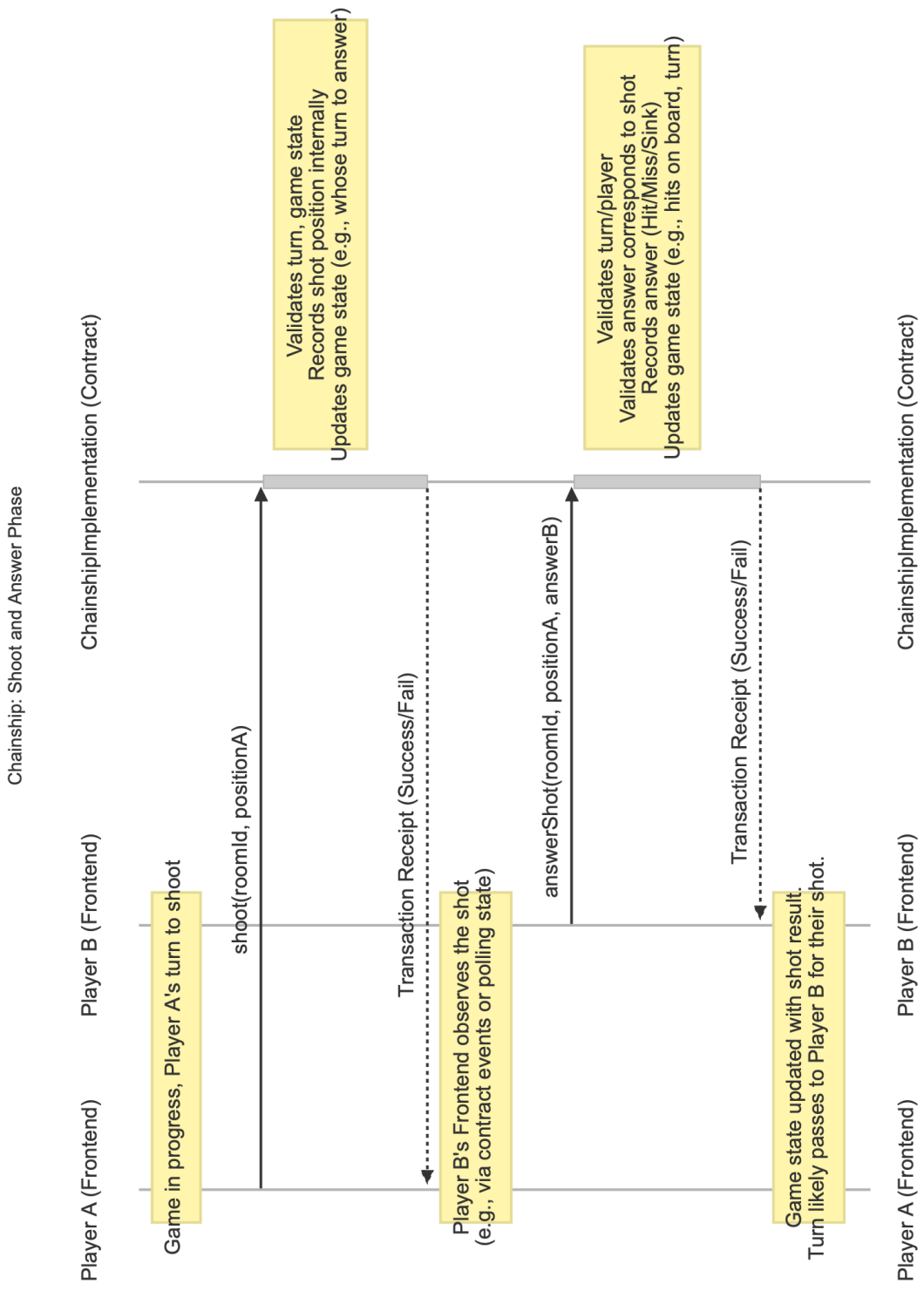

Figure 2: User Interaction Sequence Diagram (Simplified Flow)

3 Smart Contract Design

3.1 Overview

The primary contract, `Chainship`, is abstract and contains the core game logic, state management, and cryptographic verification functions. A concrete implementation, `ChainshipImplementation`, inherits from `Chainship` and provides specific details, such as the commission calculation method. It may optionally inherit from a `Multicall` contract to enable batching calls for gas efficiency and to reduce delays between player actions, although this is not strictly necessary for the core functionality.

3.2 Data Structures

Key data structures within the `Chainship` contract include:

- `RoomId`: A unique identifier for each game room, derived from a user-provided secret and the contract seed.
- `RoomStatus`: An enum tracking the current stage of the game within a room (e.g., `Created`, `Full`, `Shooting`, `Finished`).
- `PlayerData`: Stores information for each player, including their address, board commitment hash, number of shots taken, and hashes accumulating their shots and answers.
- `RoomData`: Contains all data for a specific game room, including its status, entry fee, players' data, current turn indicator, and the deadline for the next action.
- `Position`: A struct representing a position on the game board, with x and y coordinates.
- `Answer`: An enum representing the possible answers to a shot (`Miss`, `Hit`, `Sunk`).
- `rooms`: A mapping from `RoomId` to `RoomData`, storing the state of all active and past games.

3.3 Key Functions

The contract provides several functions for player interaction with the game:

- `createRoom(RoomId, uint256 randomnessCommitment)`: Initializes a new game room, setting the entry fee (via `msg.value`) and registering the creator as Player 1. This function also records Player 1's commitment to their chosen randomness.
- `joinRoom(uint256 roomSecret, uint256 randomnessCommitment)`: Enables a second player to join an existing room. The joining player must provide the correct room secret (which is linked to the `RoomId`) and match the entry fee. Their commitment to chosen randomness is also recorded.
- `submitBoard(RoomId, uint256 boardCommitment, uint256 randomnesDecommitment)`: Players use this function to submit the commitment of their board layout. They also reveal their previously committed randomness, which is then used to determine the first shooter.

- `shoot(RoomId, Position)`: The player whose turn it is fires a shot at a specified coordinate. This action updates their `shotsHash`.
- `answerShot(RoomId, Position, Answer)`: The opponent responds to a shot by indicating a `Miss`, `Hit`, or `Sunk`. This updates their `answersHash`.
- `claimDishonest(RoomId)`: A player can invoke this to accuse their opponent of providing incorrect answers to shots.
- `proveHonesty(RoomId, ...)`: An accused player calls this function to reveal their board layout and complete answer history, aiming to demonstrate they have followed the game rules.
- `proveVictory(RoomId, ...)`: A player initiates this to formally prove they have won the game, typically by demonstrating all opponent's ships have been sunk.
- `claimIdle(RoomId)`: If an opponent fails to act within the specified deadline, a player can call this function to claim victory.
- `receivePrize(RoomId)`: The confirmed winner of a game calls this function to claim the prize pool.

3.4 State Machine

The game progresses through states defined by the `RoomStatus` enum. Figure 4 illustrates these transitions, which are driven by player actions and enforced by the contract logic (e.g., moving from `Full` to `Shooting` only after both players have submitted their boards).

3.5 Scoreboard Contract

The Scoreboard contract keeps track of the number of points for each player. The Chainship contract calls the Scoreboard contract to update player points. The Scoreboard contract authorizes only specific addresses (i.e., game contracts) to call its update functions. Users can query the number of points for any player.

Functions provided by the Scoreboard contract:

- `getPoints(address)`: Returns the number of points for a given player.
- `updateRanking(address player, int256 points)`: Updates the number of points for a given player by adding (or subtracting, if negative) the provided number of points.
- `addGameContract(address)`: Adds a new game contract to the list of authorized game contracts.
- `removeGameContract(address)`: Removes a game contract from the list of authorized game contracts.

3.6 External Components

The Scoreboard contract inherits from OpenZeppelin's `Ownable` contract, which is used to restrict access to the `addGameContract` and `removeGameContract` functions to the contract owner.

3.7 Contracts Diagram

The following diagram shows the structure and relationships of the core contracts:

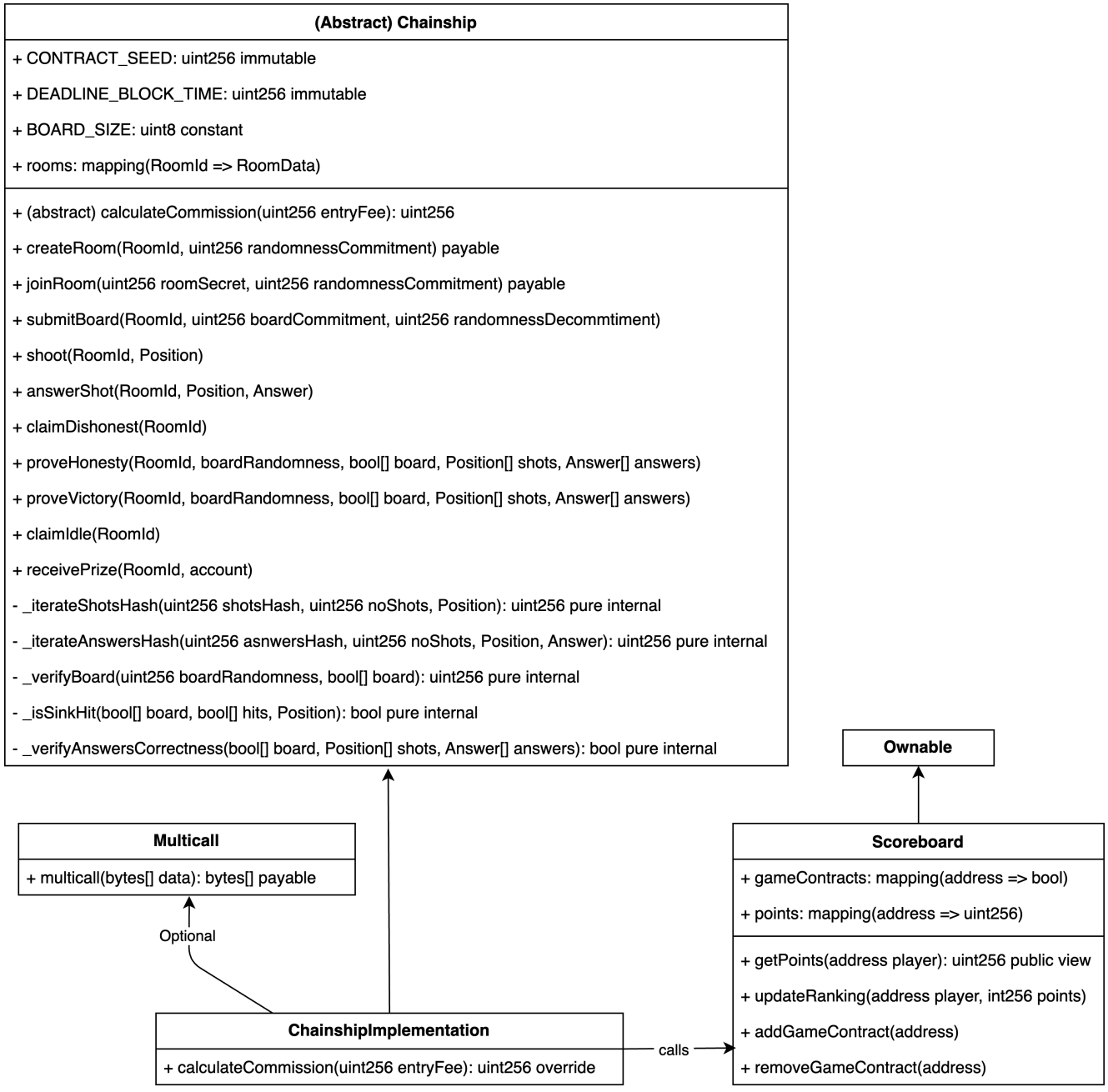

Figure 3: Smart Contract UML Class Diagram

4 Game Logic

4.1 Game State

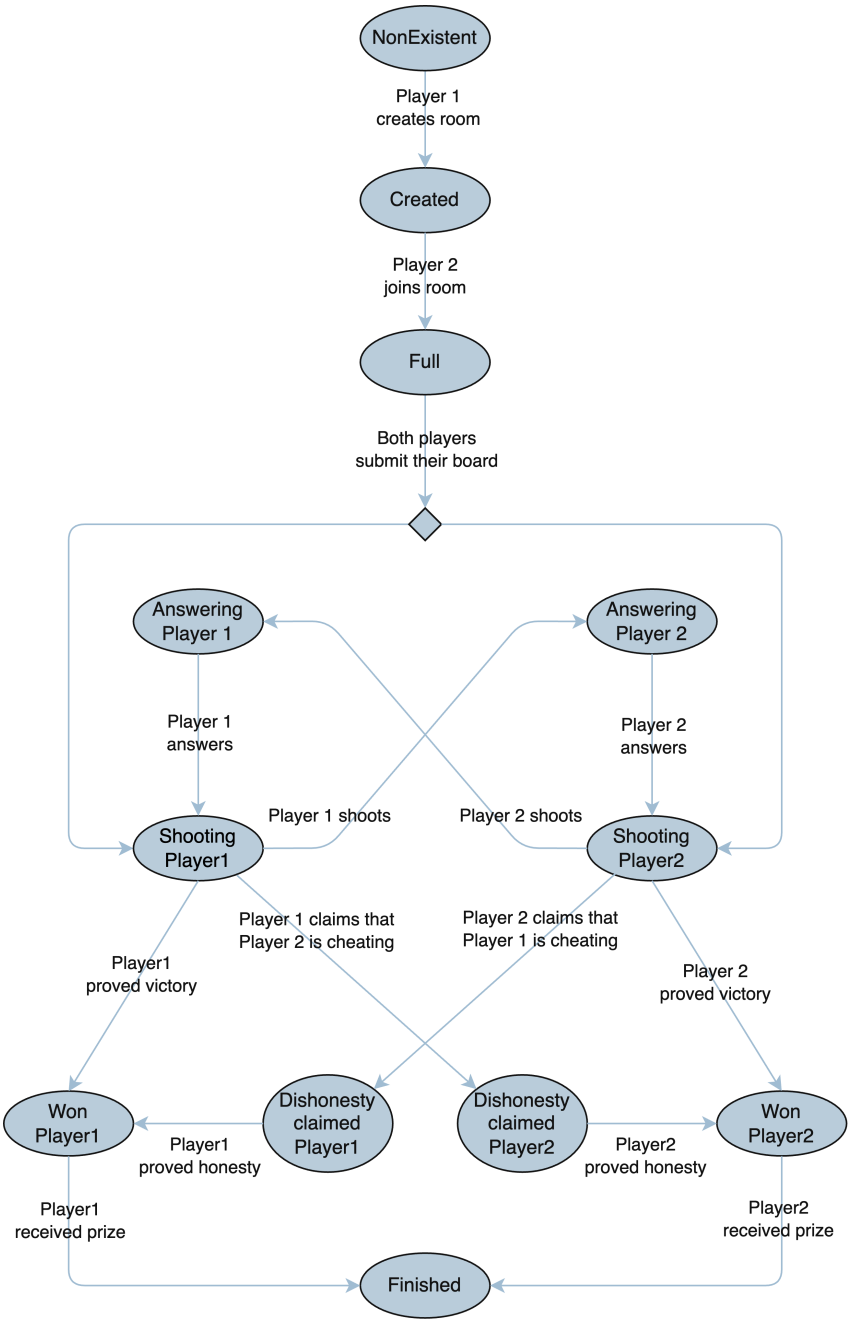

Figure 4: Game State Diagram

4.2 Randomizing First Shooter

Since draws are not allowed, randomizing the first shooter is crucial for fairness. The contract uses a commit-reveal scheme based on player inputs to determine the first shooter. Players commit to their choice of randomness when creating or joining a room. Then, while submitting their boards, they reveal their choices. Both revealed values are combined (e.g., via XOR and modulo operations) to generate a random bit that determines which player starts.

4.3 Proof of Victory

Players can win in several ways:

1. **Sinking all enemy ships:** This requires the winning player to prove that their shots correctly led to sinking all opponent's ships, consistent with the opponent's answers (and potentially revealed board if dishonesty was claimed).

2. **Opponent Dishonesty:** A player can claim their opponent answered incorrectly using `claimDishonest`. If the accused opponent fails to prove their honesty via `proveHonesty` within the stipulated deadline, the accuser wins.

3. **Opponent Idleness:** If an opponent fails to perform a required action (such as `submitBoard`, `shoot`, `answerShot`, or `proveHonesty`) before the `answerDeadline`, the other player can claim victory.

In all cases, after the winner is determined, they must call the `receivePrize` function to claim their winnings.

To prove honesty after a `claimDishonest` call, the accused player invokes `proveHonesty`, providing their board's salt (`boardRandomness`), the board layout (`board`), the sequence of shots they received (`shots`), and the answers they gave (`answers`). The contract then verifies:

1. **Board Commitment:** That `_verifyBoard(boardRandomness, board)` matches the stored `boardCommitment`.

2. **Shot History:** That the provided `shots` sequence hashes to the opponent's stored `shotsHash`.

3. **Answer History:** That the provided `answers` sequence (along with `shots`) hashes to the accused player's stored `answersHash`.

4. **Answer Correctness:** That `_verifyAnswerCorrectness` confirms each answer in `answers` was correct given the `board` and the sequence of `shots`.

To prove victory by sinking all ships, the winner calls `proveVictory`. This involves providing arguments that are the same as those in `proveHonesty`, along with answers received from the opponent. This allows the contract to verify that all the opponent's ships were sunk according to the game rules.

5 Cryptography

5.1 Board Commitment

To prevent players from changing their ship layout mid-game and to hide it from their opponent, a commitment scheme is utilized. A player commits to their board b (represented as a flat `bool[BOARD_SIZE*BOARD_SIZE]` array) combined with a secret random salt r (`boardRandomness`).

$$\text{BoardCommitment}(b, r) = \text{keccak256}(\text{abi.encodePacked}(r, b))$$

The salt r prevents pre-image attacks where an opponent might pre-calculate commitments for all possible valid boards, especially if the board size were small. The player reveals b and r only when required (e.g., during a `proveHonesty` call). The internal `_verifyBoard` function also checks the validity of the board layout itself (e.g., ensuring ships do not touch diagonally and adhere to correct sizes).

5.2 Shots Hash

To efficiently verify the sequence of shots taken by a player without storing each shot individually on-chain, an iterative hashing approach is employed. Let $h_0 = 0$. For the i-th shot at position (x_i, y_i), the hash h_i is calculated based on the previous hash h_{i-1}, the shot number i, and the coordinates:

$$h_i = \text{keccak256}(\text{abi.encodePacked}(h_{i-1}, i, x_i, y_i)) \quad \text{for } i = 1, 2, \ldots, n$$

The final hash h_n (stored as `shotsHash`) represents the entire sequence of n shots. This design allows the contract to store only the latest hash and the shot count (`noShots`). Verification involves the challenged player providing the full sequence of shots, which is then re-hashed iteratively by the contract to check if it matches the stored final hash.

5.3 Answers Hash

Similarly, an iterative hash (stored as `answersHash`) accumulates the answers given by a player. For the i-th shot received at (x_i, y_i) with answer a_i (an `Answer` enum value), the hash is updated:

$$h'_i = \text{keccak256}(\text{abi.encodePacked}(h'_{i-1}, i, x_i, y_i, a_i)) \quad \text{for } i = 1, 2, \ldots, n$$

where $h'_0 = 0$. This enables verification of the entire answer sequence during the `proveHonesty` phase by comparing the recomputed hash with the stored `answersHash`.

5.4 Commit-Reveal for Randomness

To fairly establish the starting player, a Commit-Reveal scheme is employed. Both players, P_1 and P_2, first commit to a secret nonce N_i by submitting its cryptographic hash $C_i = \text{keccak256}(N_i)$ to the smart contract:

$$C_1 = \text{keccak256}(N_1)$$

$$C_2 = \text{keccak256}(N_2)$$

Once both commitments are recorded on-chain, players reveal their nonces N_1 and N_2. The contract verifies these revealed nonces against the previously stored commitments. A shared random value R is then derived from the revealed nonces, for instance, by XORing them and taking the result modulo 2 to select the starting player:

$$R = (N_1 \oplus N_2) \pmod 2$$

This mechanism prevents either player from unfairly influencing the outcome after observing the other's commitment.

6 Estimations

6.1 Unit Tests

Unit tests will verify the correct behavior of individual functions in isolation.

- `Chainship` **Contract:**
  - `constructor`: $\approx$ 1 test case (initial state, immutable variables).
  - `roomSecretToId`: $\approx$ 2 test cases (correct ID generation, different secrets yield different IDs).
  - `createRoom`: $\approx$ 4 test cases (success, entry fee less than/equal to commission, room already exists, event emission).
  - `joinRoom`: $\approx$ 5 test cases (success, room not in correct state, entry fee mismatch, event emission, deadline setting).
  - `_getPlayerNumber` (internal): $\approx$ 3 test cases (player 0, player 1, not a player).
  - `submitBoard`: $\approx$ 7 test cases (success P1 then P2, board commitment zero, wrong room state, past deadline, non-player sender, already submitted, event emissions).
  - `_iterateShotsHash` (internal): $\approx$ 3 test cases (valid position, invalid position, correct hash).
  - `shoot`: $\approx$ 6 test cases (success, wrong room state, not player's turn, invalid position, state changes, event emission).
  - `_iterateAnswersHash` (internal): $\approx$ 3 test cases (valid position/answer, invalid position, correct hash).
  - `answerShot`: $\approx$ 6 test cases (success, wrong room state, non-player sender, invalid position, state changes, event emission).
  - `claimDishonest`: $\approx$ 5 test cases (success, wrong room state, non-player sender, state changes, event emission).
  - `_verifyBoard` (internal): $\approx$ 4+ test cases (valid board, invalid size, various invalid ship placements, correct commitment).
  - `_isSinkHit` (internal): $\approx$ 4+ test cases (not hit, hit not sink, hit and sink, edge cases).
  - `_verifyAnswerCorrectness` (internal): $\approx$ 4+ test cases (correct sequence, length mismatch, various incorrect answer scenarios, hash/hit count correctness).

- `proveHonesty`: ≈ 9 test cases (success, wrong room state, not accused player's turn, board commitment mismatch, shots hash mismatch, answers hash mismatch, incorrect answer position, state changes, event emission).
- `claimIdle`: ≈ 4 test cases (successful claim for opponent not submitting board, successful claim for opponent not acting in turn, deadline not passed, wrong state).
- `receivePrize`: ≈ 5 test cases (success, not winner, game not finished, fund transfer, reverting receive function).
- First-player randomization (commit-reveal): ≈ 3 test cases (valid and invalid commitments, correct determination).

Estimated unit tests for `Chainship`: $\approx$ **78**+.

- `Scoreboard` **Contract:**
  - `getPoints`: ≈ 2 test cases (player exists/does not exist).
  - `updateRanking`: ≈ 3 test cases (success add/subtract points, unauthorized call by non-game contract).
  - `addGameContract`: ≈ 2 test cases (success by owner, failure by non-owner).
  - `removeGameContract`: ≈ 2 test cases (success by owner, failure by non-owner).

  Estimated unit tests for `Scoreboard`: $\approx$ **9**.

Total estimated unit tests: $\approx$ **86**+.

6.2 Integration Tests

Integration tests will verify the interaction between functions and contracts across key user scenarios.

- **Scenario 1: Full Game Flow:** Player 1 creates a room. Player 2 joins. Both players submit boards. Players exchange several shots and answers correctly. Simulate a scenario leading to standard win by proving victory.
- **Scenario 2: Dishonesty Claim:** Game setup as above. Player A makes an intentionally incorrect answer. Player B calls `claimDishonest`.
  - Path 2a: Player A successfully calls `proveHonesty` with correct data, opponent (Player B) loses.
  - Path 2b: Player A provides inconsistent data to `proveHonesty`, call reverts or honesty is not proven, Player B wins.
- **Scenario 3: Game End due to Idleness:**
  - Path 3a (Board Submission): Player 2 joins but fails to submit their board within the deadline. Player 1 calls `claimIdle` and wins.
  - Path 3b (Shooting/Answering): During the game, the active player fails to `shoot` or `answerShot` within the deadline. The other player calls `claimIdle` and wins.

  - Path 3c (Prove Honesty): After a `claimDishonest`, the accused player fails to call `proveHonesty` within the deadline. The accuser calls `claimIdle` and wins.

- **Scenario 4: Scoreboard Interaction:** After a game concludes (e.g., via scenarios above), verify that if the `Scoreboard` contract were integrated, the `Chainship` contract would correctly call `updateRanking` on the `Scoreboard`.
- **Scenario 5: First-Player Randomization:** Test the commit-reveal scheme for determining the first player, covering various inputs and ensuring fairness.
- **Scenario 6: Prize Distribution:** Test the correct distribution of the prize pool to the winner and commission handling.

6.3 Implementation Time Estimation

To estimate the project implementation time for smart contract development, a method based on the number of key operations and the number of involved contracts is used.

- `Chainship` **Contract:** Key operations:
  - `createRoom`
  - `joinRoom`
  - `submitBoard`
  - `shoot`
  - `answerShot`
  - `claimDishonest`
  - `proveHonesty`
  - `proveVictory`
  - `claimIdle`
  - `receivePrize`

  Total key operations for `Chainship`: $L_{op1} = 10$.
- `Scoreboard` **Contract:** Key operations:
  - `updateRanking`
  - `addGameContract`
  - `removeGameContract`
  - `getPoints`

  Total key operations for `Scoreboard`: $L_{op2} = 4$.

Total number of key operations $L_{op} = L_{op1} + L_{op2} = 10 + 4 = 14$. Number of main contracts to implement$L_k = 2$ (`Chainship` and `Scoreboard`).

Estimated project complexity factor $W = L_{op} \times L_k = 14 \times 2 = 28$ units.

Assuming that in one day one can work on 2 units, the project will take 14 days to complete.

7 Summary

The Chainship project aims to deliver a decentralized Battleship game on EVM-compatible blockchains, emphasizing security, transparency, and fairness. The core of the system is the `Chainship` smart contract, which manages all aspects of game creation, progression, and fund distribution, eliminating the need for a central authority. A secondary `Scoreboard` contract provides a persistent ranking system.

Key architectural features include:

- **Decentralized Game Logic:** All game rules and state transitions are enforced by the smart contract.
- **Cryptographic Commitments:** `keccak256`-based commitment schemes are used for board placements and first-player randomization, preventing cheating and ensuring hidden information until reveal.
- **Iterative Hashing:** Sequences of shots and answers are hashed iteratively to save storage and gas costs, while still allowing for on-chain verification.
- **Player-Driven Dispute Resolution:** Mechanisms like `claimDishonest` and `proveHonesty` allow players to resolve disputes on-chain.
- **Modularity:** The system is designed with a main game contract (`Chainship`) and a separate `Scoreboard` contract, potentially allowing for future expansion or integration with other games.
- **Frontend Interaction:** A web-based frontend application will enable user interaction with the game via wallets like Metamask.

The project leverages Solidity for smart contract development and standard cryptographic techniques to build a robust and trustworthy gaming platform. The architecture is designed to minimize trust assumptions and provide a verifiable gaming experience.