MASTERMIND

P2P Project Report

Francesco Lorenzoni (mat. 599427)
Emiliano Sescu (mat. 596883)

28th August 2024

Contents

Chapter 1

Introduction

Preface

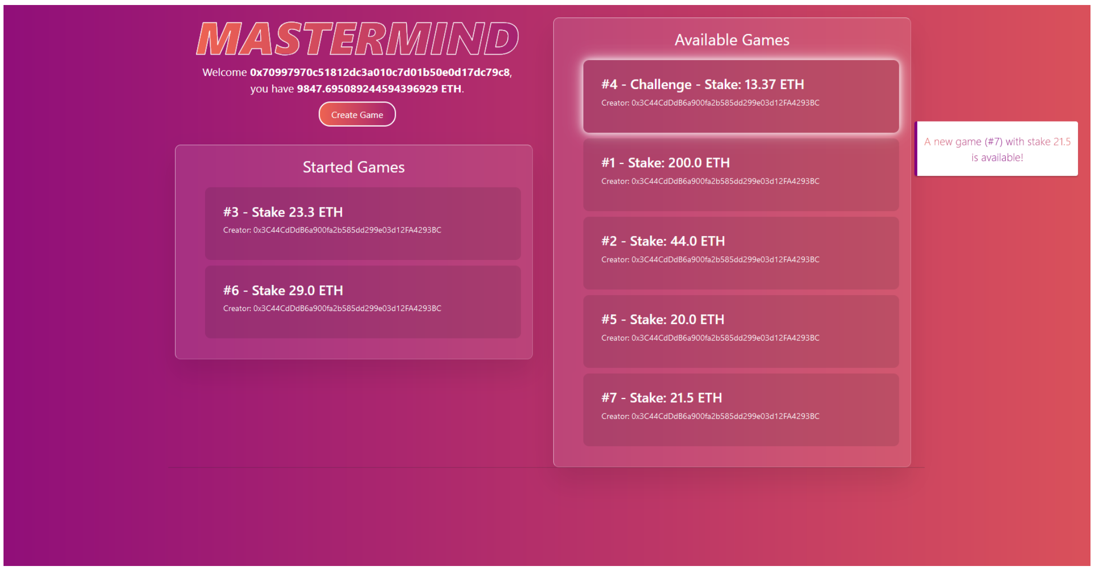
This is the report of the exam project of the *Peer-to-Peer and Blockchain* UniPi course. The assignment consisted in developing a solidity-based version of the famous Mastermind game. The project may be found at github.com/frenzis01/mastermind along with a brief *video demo* in the `media` folder and at the bottom of the GitHub `README`. The developed application consists of a Solidity backend exploiting the *Hardhat* framework, and a *Vite* react-based frontend. The project was jointly developed by **Emiliano Sescu** and **Francesco Lorenzoni**; aside from the contribution and commit history itself, we discussed together almost all design choices and shared with each other the implementations aspects learned throughout the development. For instructions on how to deploy the contract and interact with it using the frontend, please refer to the `README` file located in the project's GitHub repository.

Mastermind is a two-player code-breaking game where one player (the code maker) creates a secret code using colored pegs, and the other player (the code breaker) tries to guess the code within a fixed number of guesses. After each guess, the code maker provides feedback with key pegs: black for correct color and position, white for correct color but wrong position. The code breaker wins a turn by guessing the code within the allowed number of guesses; otherwise, the code maker wins.

1.1 Overview

Players can join the platform by connecting their Metamask Wallet. Once connected, they will be able to use wallet's profiles to log in using a specific address. Note that due to the way wallets are integrated in Metamask, each operation on the blockchain requiring a fee results in a transaction to be confirmed in a wallet pop-up.

From the home page, players can either **create** a game, **join** new games or **resume** previously started ones, as shown in Figure 1.1. Players have the flexibility to leave and rejoin any ongoing game at any time. Highlighted games represent challenges directed to a specific address, which are only visible to the challenged address. There is no handshake protocol between creator and joiner to establish the stake for a game, every player may create a game proposing a stake, and every joiner is free to choose which stake fits best its needs.



Welcome **0x70997970c51812dc3a010c7d01b50e0d17dc79c8**, you have **9847.695089244594396929 ETH**.

Started Games

#3 - Stake 23.3 ETH
Creator: 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC

#6 - Stake 29.0 ETH
Creator: 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC

Available Games

#4 - Challenge - Stake: 13.37 ETH
Creator: 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC

#1 - Stake: 200.0 ETH
Creator: 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC

#2 - Stake: 44.0 ETH
Creator: 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC

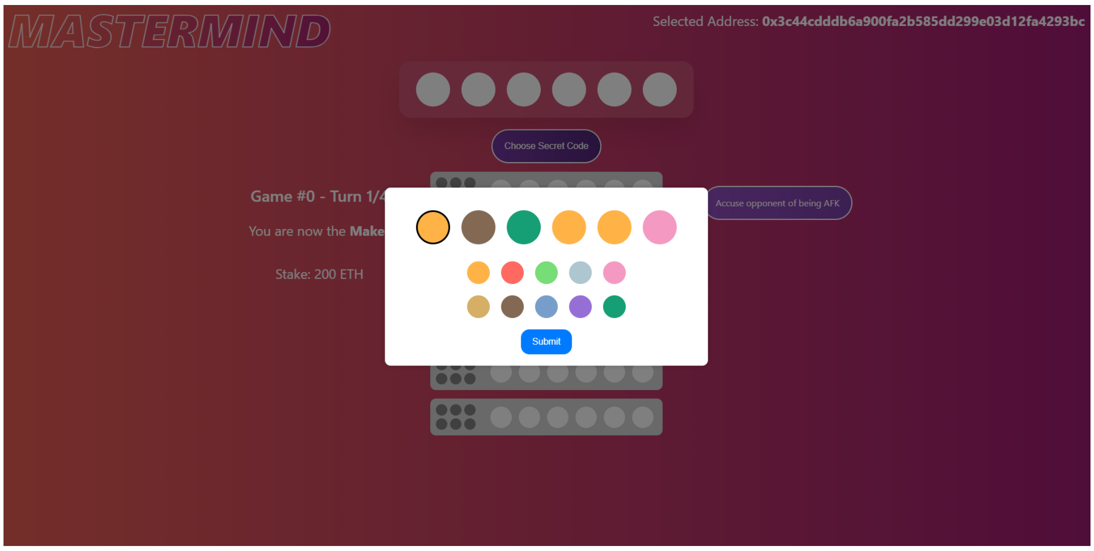
#5 - Stake: 20.0 ETH
Creator: 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC

#7 - Stake: 21.5 ETH
Creator: 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC

A new game (#7) with stake 21.5 is available!

Figure 1.1: Home page

When a player creates a game, a role of either "maker" or "breaker" is randomly assigned; the two boards displayed corresponding to each role are similar, but expose to the user different interactions according to the role assigned. The game creator will be notified of the join when an opponent joins the game; similar notifications are sent for each in-game event. The breaker, regardless of whether they are creator or joiner, always starts the turn, while the maker waits for the `TurnStarted` event. Upon receiving the event, the maker selects a secret code, prepends a *salt*, and publishes the hash onto the blockchain using the modal shown in Figure 1.2. The breaker will receive the hash publication `HashPublished`, and the turn will start.



Selected Address: **0x3c44cdddb6a900fa2b585dd299e03d12fa4293bc**

Game #0 - Turn 1/4

You are now the **Make**

Stake: 200 ETH

Figure 1.2: maker selects secret code

The breaker uses a similar modal to make guesses, and, upon the receival of a `Guess` the maker provides the corresponding feedback (a tuple ⟨`CC/NC`⟩). The rows in the board get colored according to received guesses and feedbacks which are all stored in the blockchain, allowing for game interruption and later resume.

At any point, a player can accuse the opponent of being **AFK** (*Away From Keyboard*) using the *"Accuse opponent of being AFK"* button. The accused player then has a limited number of blocks to perform a game action; the accuser can at any time check whether the "time" has elapsed, i.e. the opponent is not playing and went AFK. In this case the accuser wins the entire game stake, otherwise the game normally proceeds.

Once the breaker either guesses the code or exhausts their attempts, the **turn** concludes. The maker must then publish the secret code in clear text along with the salt. The blockchain verifies that the $hash(salt|code)$ matches the initially published hash. If there is a discrepancy, the current breaker claims the entire game stake. The breaker then validates the feedbacks received (Figure 1.3); if some feedbacks are deemed invalid, they may dispute them and claim the stake if the dispute is correct; Note that if the breaker disputes $fb_1, fb_3, ...fb_k$ and at least one of them was correct, the maker wins the game and obtains the stake.

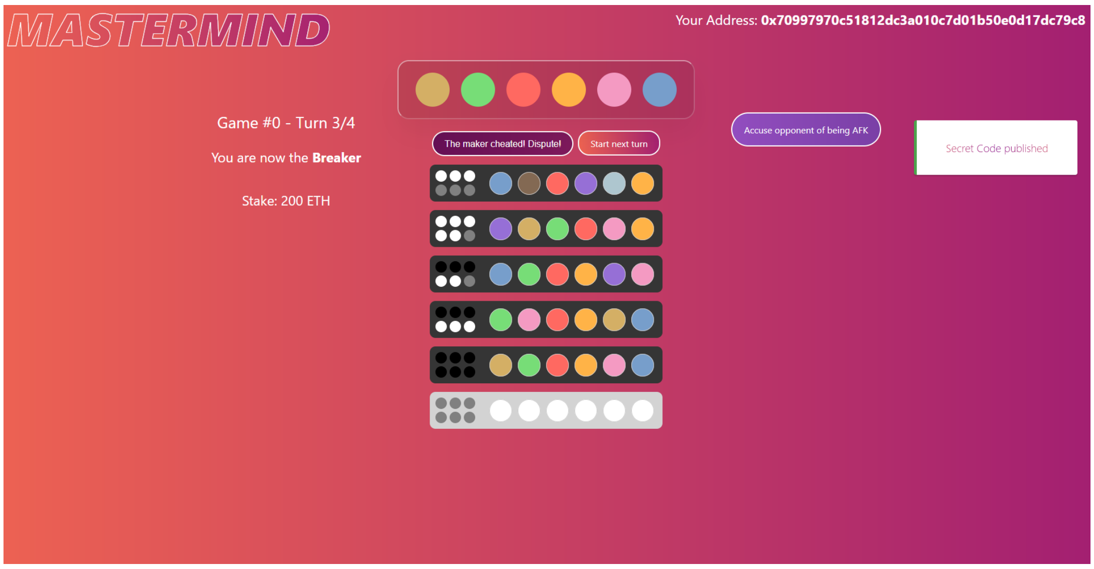
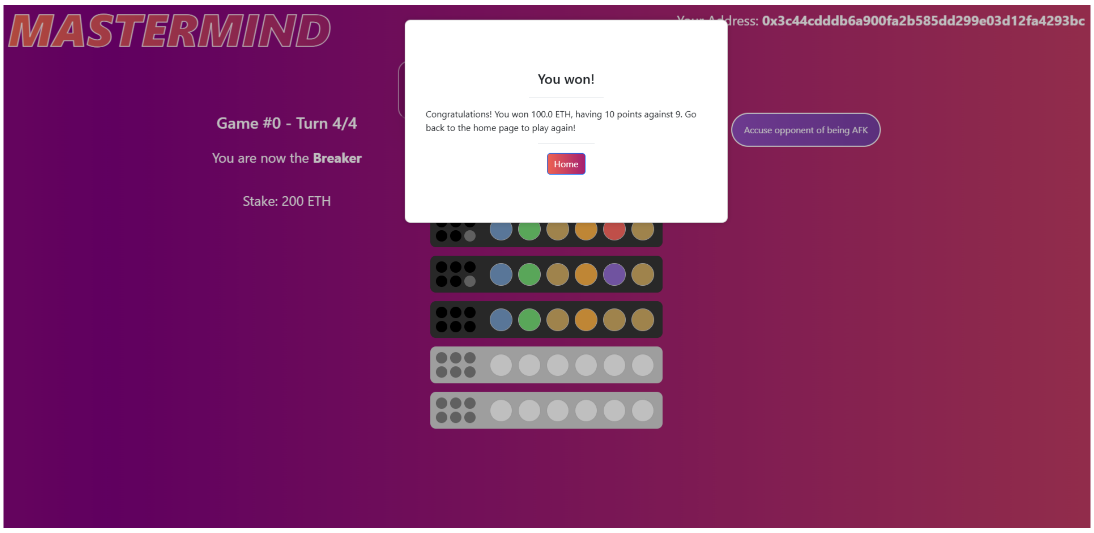

MASTERMIND
Your Address: 0x70997970c51812dc3a010c7d01b50e0d17dc79c8
Game #0 - Turn 3/4
You are now the Breaker
Stake: 200 ETH
The maker cheated! Dispute!
Start next turn
Accuse opponent of being AFK
Secret Code published

Figure 1.3: breaker receives secret code

At the end of each turn, points are awarded to the maker based on the number of guesses the breaker needed to crack the code. Additional points are given if the breaker uses all their attempts without guessing the code. After all turns have been played, the player with more points is declared the winner and receives the entire stake, as shown in Figure 1.4.

In case of a tie, the creator wins.


MASTERMIND
Your Address: 0x3c44cdddb6a900fa2b585dd299e03d12fa4293bc
Game #0 - Turn 4/4
You are now the Breaker
Stake: 200 ETH
Accuse opponent of being AFK
You won!
Home

Congratulations! You won 100.0 ETH, having 10 points against 9. Go back to the home page to play again!

Figure 1.4: Win modal

Chapter 2

Architecture

2.1 Backend

This section discusses the aspects of the contract related to the overall application architecture. Detailed informations about the Solidity **smart contract** itself can be found in the Section 3. Hardhat was used as the development framework for the smart contract.

2.1.1 Contract ABI

The Contract ABI acts as a function selector, defining the specific methods that can be called to a smart contract for execution. These methods and their associated data types are listed in a generated JSON RPC file.

The Contract ABI is automatically generated by Hardhat and saved in the `./frontend/src/contracts` folder as `Mastermind.json`, ensuring that the frontend can use it to interact with the smart contract. This is done by the `deploy.js` script during the smart contract deployment process.

2.1.2 Events

The smart contract emits *events* whenever a game action is performed. This allows the frontend to obtain information about the contract state without explicitly calling getters, enabling asynchronous communication mediated by the smart contract.
We strongly advise to check the **Appendix** 5 for a flowchart of a game highlighting how the events influence and model the proceeding of the game[1].
These emitted events are associated with *event handlers* (see Section 2.2.1) on the frontend.

2.2 Frontend

The frontend was implemented using Vite, a React framework. Although the frontend consists of many elements and components, this section focuses only on those relevant to the project which concern interacting with the Solidity smart contract. Detailed discussions about the React page structure, routing, components in the 'misc' folder, CSS, hooks, and other React-related aspects used to compose the frontend are **not** included.

2.2.1 Pages

The main logic of the frontend is implemented across two pages: **Home** and **Game**. As both pages require interactions with the smart contract, attempting to use them without a connected wallet or without using one of the addresses stored in the wallet will render two respective components: `NoWalletDetected.jsx` and `ConnectWallet.jsx`.

After connecting the wallet to the page, a **provider** is created. This provider is used to interact with the wallet to fetch the selected address, it's balance, and eventually other informations. Additionally, the provider, along with the contract's address and its ABI, is used to define a `Contract` object in the frontend. This `Contract` (named **mastermind**) is used from React to perform explicit calls to the smart contract.

Events emitted by the smart contract enable asynchronous communication from the smart contract to the front end.

[1]In the flowchart, blue squares indicate invocations to contract functions, yellow emitted events, and green user choices

This is also possible thanks to the `Contract` object, which listens to those events and associates handlers to execute upon event capture. Each handler comprises a notification (referenced as a "snack", see Section 2.2.2.3) to inform the user of the captured event, along with actions necessary to update the page state based on information received from the smart contract. The specific events used on both pages will be listed in their own subsection. When the page is unmounted, a cleanup function is called to remove event listeners.

2.2.1.1 Home

This page serves several purposes: **creating** a new game (using the `CreateGameModal.jsx`), displaying **available** games (including *challenges* to a specific address), and showing games that have already started and can be **resumed**. *Challenges* to a specific address are indicated by glowing borders around the corresponding React item, and are always prioritized above other available games, as shown in Figure 1.1.

Creating a new game involves calling either the `createGame()` or `createGameWithJoiner()` smart contract functions, depending on whether the game is directed towards a specific address. Joining a game from the list of available games triggers the `joinGame()` smart contract function. For more informations, refer to Section 3.1.1.

Events listened The *events* that the `Contract` object is set to listen are:

- **GameCreated** → `handleGameCreated()`: adds to available games the new game created.
- **GameJoined** → `handleGameJoined()`: removes from available games the game joined.

2.2.1.2 Game

This is the page used to play Mastermind. Upon loading, it retrieves information about the selected game (from the Home page) calling the `getGameDetails()` smart contract function. Depending on the fetched information, the player can either act as a Maker or a Breaker. Accordingly, either the Maker board or the Breaker board component will be displayed, along by other relevant information on the page. An example of how this page appears can be seen in Figure 1.3, which illustrates a case where the Breaker board is being loaded.

This page contains the invocations of *core* smart contract functions —discussed in Section 3.1— required to play the game. These functions are passed as references to the board components, enabling the buttons within the boards to execute fundamental game actions.

The function for computing the hash of the secret code is called `computeHash()` (more on this in Sec. 3.1.3), and takes as input the secret code and a random salt, which will both need to be exhibited on the end of a turn. Since it is desirable to store them in the client —to automatically insert them in end-turn phase—, they *cannot* be simply assigned to a variable, because they must *persist* even in case of page reload and cannot be retrieved from the blockchain, since they are not there. For this reason they are stored in the browser's `localStorage` by the functions `setCodeSecretMemo()` and `setCodeSeedMemo()`.

Events listened The *events* that the `Contract` object is set to listen may vary depending on the role of the player; they are listed below for each scenario:

Either Maker or Breaker:

- **TurnEnded** → `handleTurnEnded()`: sets the flag `_turnEnded` = true.
- **GameEnded** → `handleGameEnded()`: resets persistent information (`Memo`) stored about the code, displays the `NonClosableModal.jsx` with game result informations, and then sets the flag `_gameEnded` = true
- **AFKAccusation** → `handleAFKAccusation()`: sets an object (`_accusedAFK`) that contains the address of the accused opponent.
- **ResolveDispute** → `handleResolveDispute()`: sets the flag `_disputed` = true and generates the end game message.

Player acting as Maker:

- **Guess** → `handleGuess()`: sets an object (`_lastGuess`) containing the new guess, and removes the object associated with the AFK accusation.
- **TurnStarted** → `handleTurnStarted()`: sets the flag (`_turnStarted`), removes the object associated with the AFK accusation, and calls the function to setup a new turn.

Player acting as a Breaker:

- **Feedback** → `handleFeedback()`: sets an object (`_lastFeedback`) containing the new feedback, and removes the object associated with the AFK accusation.

- ◇ **HashPublished** → `handleHashPublished()`: sets a string (`_codeHash`) that contains the hash of the secret code, and removes the object related to the AFK accusation.
- ◇ **CodeSecretPublished** → `handleCodeSecretPublished()`: sets an object (`_codeSecret`) that contains the secret code, and removes the object related to the AFK accusation.

The flags set in the handlers are passed to and used from the boards to detect specific events in order to act accordingly.

2.2.2 Components

2.2.2.1 Boards

Core components of the Game page, allow to display the board of the current game (as shown in Figure 1.3). There are two boards: `BoardMaker.jsx` and `BoardBreaker.jsx`. Each board displays different buttons based on the possible actions of the maker and the breaker. The Maker Board includes an additional element that shows the code selected by the maker at the beginning of the game. This code is only revealed on the Breaker Board when the maker publishes it.

Both boards also provide useful text messages to the player, depending on the current state of the turn.

2.2.2.2 Modals

A modal in React is a component that overlays other content on a webpage. The following modals are used in the frontend:

- ◇ `CreateGameModal.jsx` contains a form that collects informations for creating a new game, such as the game stake and optionally an address to challenge. When the form is submitted, the submit button triggers a function on the Home page that calls the smart contract function to create the game.
- ◇ `ColorChoseModal.jsx` contains a form for submitting a color combination. This modal is used by the Maker Board to create the secret code (see Figure 1.2) and by the Breaker Board to submit guesses.
- ◇ `ProvideFeedbackModal.jsx` contains a form for submitting a feedback related to the last guess.
- ◇ `NonClosableModal.jsx` is used to display the end game message and includes a button to redirect the player to the Home page, as shown in Figure 1.4.

Although the modals containing a form require valid values to be submitted, any transaction with incorrect values gets **reverted** by the smart contract, which performs all necessary checks.

2.2.2.3 Snacks

Components named “snacks” are used to notify the user of any events received from the *smart contract* (as shown in Figure 1.1 or in Figure 1.3). A global state for storing snacks is maintained, allowing all pages to add a new snack. Only the handlers create new snacks by calling the `addSnack()` function.

Chapter 3

Solidity Smart Contract

The solidity contract is in `contracts/Mastermind.sol`. The variables which could be statically and arbitrarily defined are set as constants at the beginning of the contract, to allow ease of tuning of such parameters.

Everything revolves around a `Game` struct which holds the state of a game. Alterations to such state are logged in the form of *events*, each one associated to a function.

`modifier`s are used to ensure that at the time of function's invocation its preconditions are satisfied, ensuring that the state of the contract is always valid. Functions marked with `view` do *not* modify the state of the contract, and there is **no fee** to execute them, unless they are executed inside a function which do alters the state: in such case, they contribute to the function invocation's fee. `external view` functions are used only by the frontend or by Mocha tests (`.js` files under `tests` folder) to get information on the games, `internal view` ones only by other functions in the contract, while `public view` match both use cases.
The only `payable` functions are `createGame` and `joinGame`.

3.1 Core functions and Choices

3.1.1 Creating and Joining games

The "agreement" on a stake to play on is handled in the frontend, and basically consists in choosing the game with the stake closest to desired one. For what concerns the contract, `createGame()` specifies a stake, and a joiner simply invokes `joinGame()` sending such stake. The two functions (mostly the former) initialize most of the `Game` state, but not all: some informations are set by `startTurn()` —discussed in 3.1.6— , since they need to be initialized not only at the beginning of the game, but at each turn.

3.1.2 Feedbacks and the end of a turn

The logic behind `makeGuess()` and `provideFeedback()` is rather simple, but the latter must handle two special cases:

1. The feedback provided indicates that CC[1] is equal to the length of the code (6 in the screenshots provided earlier), indicating that the breaker has **guessed** the code.
   At this point the turn should end, having the breaker guessed the secret code.
2. The breaker has **exhausted** their available guesses, so the turn should end, with the maker being awarded of extra points, since the breaker could not guess its code.

```
function provideFeedback(
    uint256 _gameId,
    uint256 numCorrectPositions,
    uint256 numCorrectColors    // correct colors but in wrong positions!
) external checkGameValidity(_gameId) turnNotEnded(_gameId) {
    Game storage game = games[_gameId];
    // Ensure feedback is valid
    require(
            numCorrectColors + numCorrectPositions <= game.codeLength,
        "Invalid feedback"
    );
    // Ensure player has not already provided feedback in the current turn
    require(
        isMakerTurn(_gameId),
```

[1]Number of colors in the correct positions

```
        "Code maker has already provided feedback");
    game.feedbacks[game.currentTurn].push([numCorrectPositions, numCorrectColors]);
    game.accusedAFK[msg.sender] = 0; // Reset the AFK accusation if present
    emit Feedback(_gameId, numCorrectPositions, numCorrectColors);
    if (numCorrectPositions == game.codeLength) {
        endTurn(_gameId,true);
        return;
    }
    if (game.guesses[game.currentTurn].length == game.maxGuesses) {
        endTurn(_gameId, false);
    }
}
```

Note that the smart contract is **not** able to check that the feedback is consistent with the secret code chosen by the Maker. The secret code is unknown to the smart contract until the end of a turn when it gets published using `publishCodeSecret()`, but even then, the contract does not check that all received feedbacks are consistent with the given Secret Code: doing so would invalidate the requested *Dispute* feature.

3.1.3 Secret - Commit and reveal

About `publishCodeSecret()` and `submitCodeHash()`, to avoid involving the contract in any way in the hash computation, the contract simply receives a `bytes32 hash` value right after the turn has started, without ever knowing the values which produced it, but actually expecting it had been computing using a precise algorithm and data structure; we'll cover this aspect in a moment.

To inhibit the risk of a brute force attack (or lists of publicly known mappings *value* $\rightarrow$ *hash*), we chose to prepend a *salt*[2] to the secret code before hashing. In the solidity contract, `publishCodeSecret(gameId, secret, seed)` computes the hash by invoking the following function, and checks that the result is equal to the hash submitted at the start of the turn.

```
function hashArrayOfIntegers(uint256[] memory intArray, bytes memory seed) public pure returns (
    bytes32) {
    // Encode the array using abi.encodePacked
    bytes memory encodedArray = abi.encodePacked(intArray);
    // Combine the seed and the encoded array
    bytes memory combined = abi.encodePacked(seed, encodedArray);
    // Hash the encoded array using keccak256
    bytes32 h = keccak256(combined);
    return h;
}
```

The key point is that on the frontend side the hashing function must perfectly mimick the behavior of this one, in order to produce the same hash (given the same inputs) and for the comparison in `publishSecretCode()` to succeed.

Below is listed the frontend code to compute such hash, which required some tuning and even though it looks more complicated than solidity one, it does the exact same thing. The randomness of the salt relies on `window.crypto.getRandomValues()`, which is reported as *"suitable for cryptographic purposes"*

```
// Generate a random 64-character string salt
generateRandomSeed() {
  const array = new Uint8Array(32);
  window.crypto.getRandomValues(array);
  return array; // 32 bytes * 2 hex chars per byte = 64 hex chars
}
// Hash the salt prepended to the serialized array
computeHash(intArray, seed) {
  // Serialize the array as it would be in Solidity
  const abiCoder = ethers.AbiCoder.defaultAbiCoder();
  const types = new Array(intArray.length).fill('uint256');
  const serializedArray = abiCoder.encode(types, intArray);
  // Concatenate the seed and the serialized array
  const combined = ethers.concat([seed, serializedArray]);
  // Compute the hash
  return ethers.keccak256(combined);
}
```

3.1.4 AFK - Away from Keyboard accusation

The contract allows to accuse the opponent of being AFK, and they do not perform actions on the game for B blocks, the accuser may win the entire stake.

[2] This is called `seed` in the code

In solidity there is no setting of timers or triggers, so the act of accusing and claiming the stake is split in two functions:

1. `accuseAFK()`: Which notes in `game.accused[accused]` the current block number.
   Such accusation note (if present) is automatically reset every time the accused player invokes a function associated to an in-game action; `createGame()`, `joinGame()` and `accuseAFK()` itself are not considered in this set).
   It was chosen for `accuseAFK()` not to reset the *accuser* accusation note, since this may lead to the two players to indefinitely accuse theirselves without actually playing the game.
   In order for the function to be invoked it must be the turn of the accused; this is crucial for the correct functioning of the AFK accusation.
2. `verifyAFKAccusation()`: Which checks whether the accused had been AFK for enough time, and if so, ends the game, transferring the game stake to the accuser.

3.1.5 Disputing

When the breaker invokes the `disputeFeedback()` they must attach the *set* of feedback IDs they are disputing.
Such function can be invoked only after the turn has ended, and after the code secret has been published.

The contract, having now the secret code in clear text, may compute the correct feedbacks for the given guesses[3] and compare them with the ones stored in the game state (`game.feedbacks[currentTurn]`).

It is important to note that *all* the feedbacks indicated by the breaker must result invalid for the Breaker to win, if any of them is correct the Maker wins instead; in any case, the game ends.

3.1.6 Starting Turns

There are the two main design choices concerning `startTurn()`, which however affect how the whole game proceeds.

Even if it seems counterintuitive —since it's the maker who has to first publish the secret hash— , it's thebreaker who has to start a turn, the maker is not allowed to do so. We chose this way mostly due to the logic of the end turn phase and of the dispute.
Consider this scenario: the turn has ended, and the maker has revealed the secret code, but the next turn has *not* started yet. At this point the breaker has a fixed period of time to review the feedbacks provided by the Maker and decide whether it's the case to **dispute** some of them. It makes sense that is the Breaker to say at some point "Okay, nothing to dispute, we can move onto the next turn", rather than the Maker to ask the contract whether the time for disputing has elapsed or the Breaker is ready to start the new turn; the latter option would imply also an additional transaction performed by the Breaker to state "Okay, nothing to dispute".

`startTurn()` also handles the "standard" way of ending a game: when the current turn reaches the maximum, an invocation to `startTurn()` results in the normal end of the game, comparison of points, and stake transfer to the winner. A detailed **interaction schema** is provided in Appendix 5.
In case of a tie, the stake goes to the game *creator*

3.2 Testing

Since the contract's first core functions were implemented, further developments were accompanied by Mocha test suites. Such test suites are in test folder. The absence of a GUI and user inputs allowed to quickly write and run (unit) tests. Tests may be ran by executing:

```
npx hardhat test
```

There are three `.js` test files:

1. `Mastermind.js` tests all core functions of the contract, and handle some scenarios. Even if Mocha is fully exploited with independent unit tests, in this case units depend on the previous state, resulting in a more linear testing.
2. `Looped.js` simply executes the same `Mastermind.js` code but multiple times, it was used in late development to obtain more precise gas reports (described in the following section 4.1).
3. `Parallel.js` runs multiple games in a row.

[3]The ones corresponding to the feedbacks indicated by the Breaker

Chapter 4

Considerations

4.1 Gas Evaluation

In Solidity, `mapping` and arrays such as `uint256` serve similar purpose. In general, `mapping` is recommended for highly variable and *sparse* data; on the other hand arrays are recommended for fixed size structures.

In Mastermind while some of the data structures —requiring for multiple values— have fixed-size, e.g. a single guess/feedback, others are not, but are not *sparse*. So we had to decide which one to use.

The implementation of the approaches described in this section concerning the gas consumption may be found in gasreport branch of the GitHub repo

4.1.1 Mapping vs Triple Array for guesses and feedbacks

Table 5.1 and Table 5.2 refer to the gas consumption by using a triple array `uint256[][][]` as a structure for guesses and feedbacks of the whole game. Given a `game`, the i^{th} guess of the j^{th} turn is accessed as follows.

```
    uint256[] guess = game.guesses[j][i]
```

The guess holds values ranging from `0` to `MAX_COLORS`, which represent a guess.

Table 5.4 and Table 5.3 refer instead to the gas report when using a `mapping(uint256 => uint256[][]` instead of the triple array. The mapping seems to lead to considerably higher cost in `startTurn(...)`, probably due to the assignment performed in `resetGuessState()`.

Overall, the solution that implements the map costs slightly more, since the total weighted consumption (based on average cost multiplied by number of calls) is lower for Table 5.1 and Table 5.2 (50,434,149 / 105,992,755) compared to Table 5.3 and Table 5.4 (51,909,235 / 106,627,870). In cases where the number of calls was uneven, the larger number of calls was taken into consideration. The weighted consumption was still in favour of the triple array solution even considering the smaller number of calls.

4.1.2 Explicit maker/breaker vs Inferred

A point unrelated to the topic of complex data structures, but which still may be instructive concerns the implementation of `isCurrentMaker()`.

The approach kept until the last days before submitting was to set a flag (`isCreatorMakerSeed`) at the beginning of the game to establishing whether at a given current turn, a player is the maker or not. This clearly involves some computations and comparisons, which —as may be read here— appear rather verbose in `isCurrentMaker()` implementation.

A different approach would be to have in the `Game` struct, two `address` fields indicating *maker* and *breaker* for the current turn, set at the beginning of each turn inside `startTurn()`. The total weighted consumption (based on average cost multiplied by number of calls) is lower for Table 5.5 (107,927,815) compared to Table 5.2 (109,592,755). So in late-development we decided to switch to the latter approach having two `address` fields, providing a small improvement for the smart contract.

4.2 Vulnerabilities

GitHub reported some potential vulnerabilities due to the *Node.js* packages used by Hardhat and Vite, but they are beyond the purpose of this project. Here we discuss only vulnerabilities to topics mentioned during lectures.

4.2.1 Secret Leak

One possible vulnerability that was detected during the development of the smart contract is the hash of the secret code, since it's of fixed length (6) and only contains number from 0 to 9 (representing the 10 colors available to play the game), it is susceptible to brute force attacks or the use of publicly known mappings ($code \rightarrow hash$).

To address this issue, we implemented a solution, as discussed in Section 3.1.3, by appending a salt to the secret code. The salt adds randomness and enhances security, mitigating the risk of brute force attacks. The randomness of the salt is generated using `window.crypto.getRandomValues()`, which is considered *"suitable for cryptographic purposes"* according to its documentation.

Appending a salt to the secret code, combined with a reliable source of randomness, significantly improves the security of the smart contract by reducing the likelihood of successful brute force attacks.

We ensured that the original code secret never reaches the smart contract, not even as a function parameter.

4.2.2 Re-entrancy attacks

Re-entrancy attacks involve calling multiple times a function before the previous call completes in order to drain funds from the contract. The only function which yields funds to the user is `endGame()` which is marked with `internal`, meaning that it cannot be invoked directly by a user, but only by other functions in the contract. Hence, as other functions may be invoked only in safe states, also `endGame()` should not pose any problems.
However, we added the following statement in the `endGame()` body —because "never say never"—, preventing multiple subsequent invocations to succeed.

```
endGame(...
require(!game.gameEnded, "Game has ended");
// Mark the game as ended
game.gameEnded = true;
...
```

There should be no chance that two executions proceed step-by-step *exactly* in parallel, only sequential executions should be allowed. According to the sparse information we found, some parallelism may happen in the Ethereum Virtual Machine when a function invokes a function from another contract; but we could not link a reliable source on the topic.

4.2.3 SafeMath

`SafeMath` was **not** used in the project. The only mathematical operations involved are increment and decrements on counters on player points and on the number of existing games. Player points are bounded by the constants set in the contract, while reaching overflow in the number of games[1] (2^{256}) would indicate a serious world-wide game addiction issue ☺. Summing up, considering the extent of the project, we chose to avoid using `SafeMath`, obtaining way more readable code.

The only exception is for a multiplication when transferring the Game Stake, which we guarded with an assert to prevent overflow from happening, just as it is done in the SafeMath library.

```
payable(winner).transfer(safeMul(game.gameStake,2));
...
function safeMul(uint256 a, uint256 b) internal pure returns (uint256 c) {
    if (a == 0) {
        return 0;
    }
    c = a * b;
    assert(c / a == b);
    return c;
}
```

[1]no way of decrementing it "below zero"

Chapter 5

Appendix

Solidity and Network Configuration				
Solidity: 0.8.17	Optim: true	Runs: 2000	viaIR: true	Block: 30,000,000 gas
Methods				
Contracts / Methods	**Min**	**Max**	**Avg**	**# calls**
`createGame`	433,538	618,776	497,713	5
`disputeFeedback`	192,225	207,231	198,152	3
`joinGame`	174,588	175,400	175,197	4
`makeGuess`	213,071	232,326	216,787	120
`provideFeedback`	103,410	158,601	114,657	120
`publishCodeSecret`	212,875	213,277	213,115	20
`startTurn`	91,646	108,299	95,092	20
`submitCodeHash`	59,764	62,143	61,193	20
Deployments				
Mastermind	-	-	4,378,855	14.6%

Table 5.1: Parallel.js

Solidity and Network Configuration				
Solidity: 0.8.17	Optim: true	Runs: 2000	viaIR: true	Block: 30,000,000 gas
Methods				
Contracts / Methods	**Min**	**Max**	**Avg**	**# calls**
`accuseAFK`	71,576	71,763	71,645	25
`createGame`	433,538	459,038	445,895	30
`createGameWithJoiner`	473,620	516,232	494,041	40
`disputeFeedback`	200,310	200,472	200,407	40
`joinGame`	170,505	174,588	172,547	20
`makeGuess`	213,071	232,326	218,384	120
`provideFeedback`	103,410	175,930	128,764	130
`publishCodeSecret`	212,797	213,265	213,047	40
`startTurn`	93,886	108,299	100,905	40
`submitCodeHash`	59,725	62,143	60,936	30
`verifyAFKAccusation`	80,400	80,635	80,518	20
Deployments				
Mastermind	-	-	4,378,855	14.6%

Table 5.2: Looped.js test with Triple array `[][][]`

Solidity and Network Configuration				
Solidity: 0.8.17	Optim: true	Runs: 2000	viaIR: true	Block: 30,000,000 gas
Methods				
Contracts / Methods	**Min**	**Max**	**Avg**	**# calls**
`createGame`	388,125	470,617	428,454	5
`disputeFeedback`	193,426	202,354	197,011	3
`joinGame`	174,590	192,502	183,749	4
`makeGuess`	208,627	227,882	212,559	72
`provideFeedback`	98,806	153,846	110,330	72
`publishCodeSecret`	212,800	213,280	213,063	12
`startTurn`	97,984	318,779	245,084	12
`submitCodeHash`	59,734	62,140	60,943	12
Deployments				
Mastermind	-	-	4,484,756	14.9%

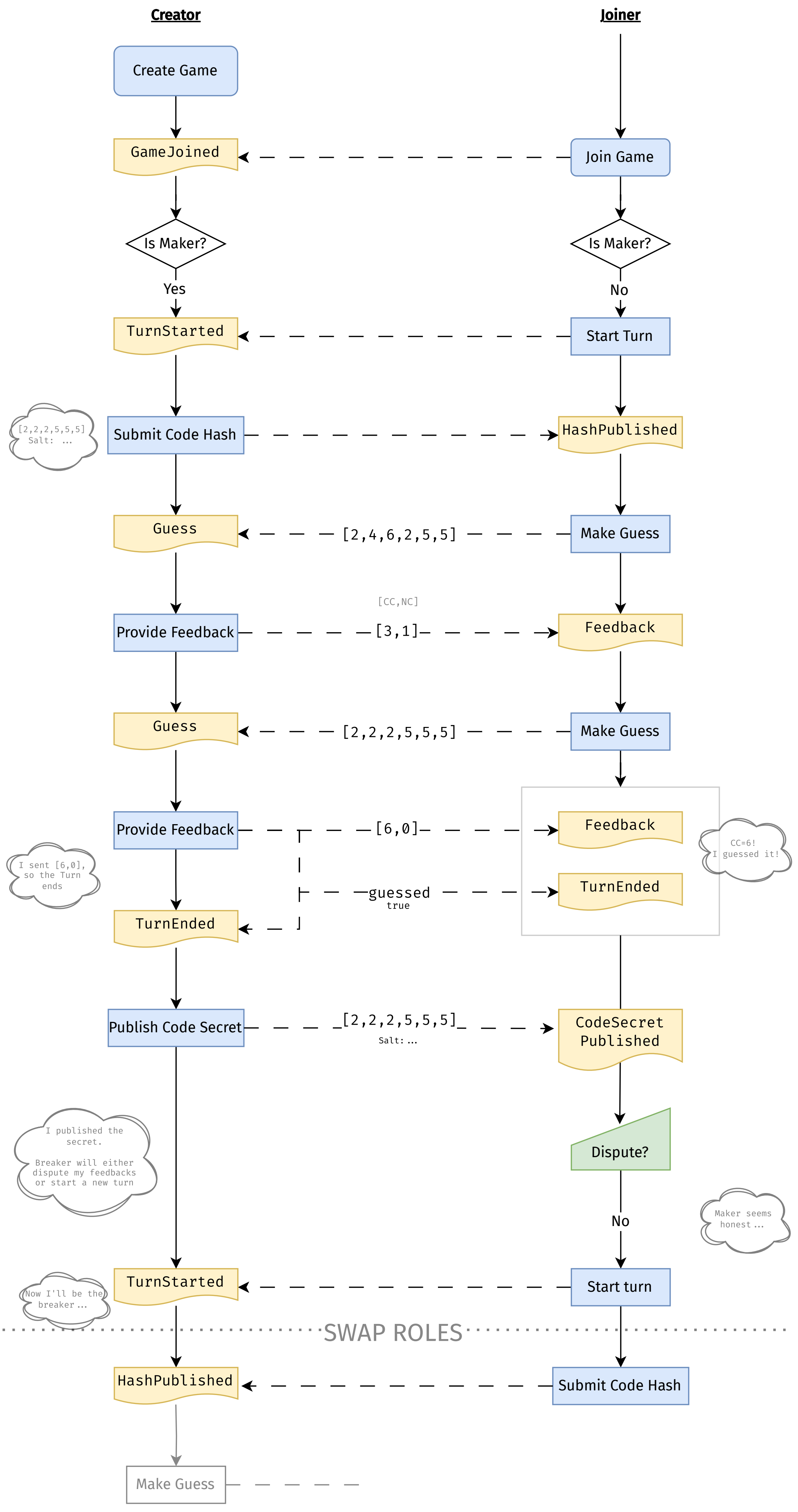
Table 5.3: Parallel.js test with Mapping `mapping(uint256 => uint256[][])`

Solidity and Network Configuration				
Solidity: 0.8.17	Optim: true	Runs: 2000	viaIR: true	Block: 30,000,000 gas
Methods				
Contracts / Methods	**Min**	**Max**	**Avg**	**# calls**
`accuseAFK`	66,968	67,173	67,026	23
`createGame`	388,125	413,625	399,818	30
`createGameWithJoiner`	428,208	470,820	446,142	40
`disputeFeedback`	195,252	195,414	195,333	40
`joinGame`	170,507	174,590	172,549	20
`makeGuess`	208,627	227,882	214,011	120
`provideFeedback`	98,806	171,175	124,065	130
`publishCodeSecret`	212,800	213,268	213,054	40
`startTurn`	95,984	246,408	171,988	40
`submitCodeHash`	59,710	62,140	60,847	30
`verifyAFKAccusation`	80,397	80,632	80,538	20
Deployments				
Mastermind	-	-	4,484,756	14.9%

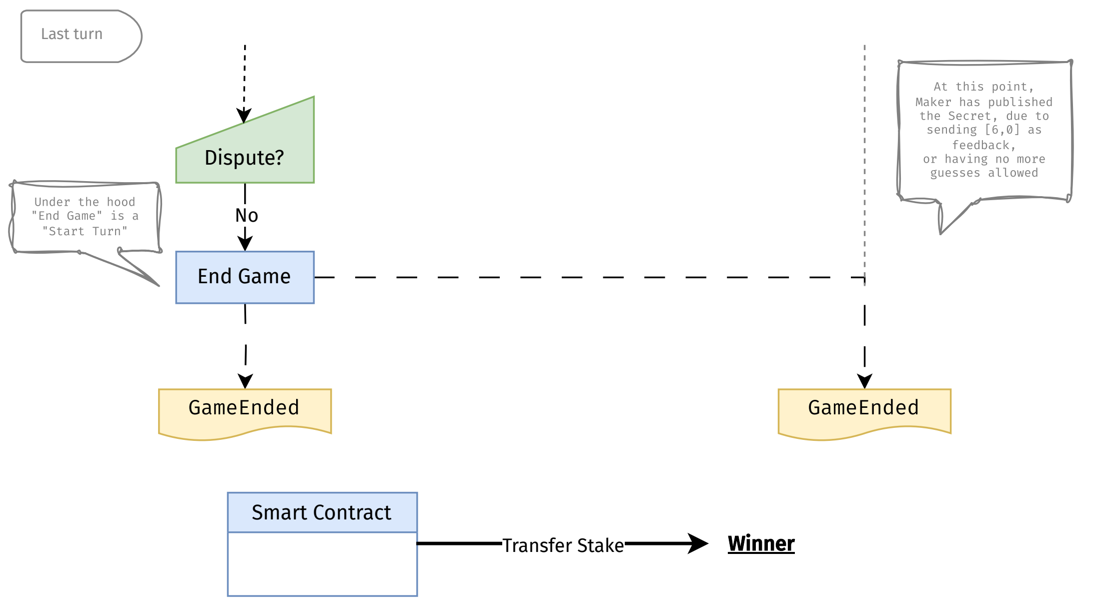
Table 5.4: Looped.js test with Mapping `mapping(uint256 => uint256[][])`

Solidity and Network Configuration				
Solidity: 0.8.17	Optim: true	Runs: 2000	viaIR: true	Block: 30,000,000 gas
Methods				
Contracts / Methods	**Min**	**Max**	**Avg**	**# calls**
`accuseAFK`	71,166	71,452	71,205	21
`createGame`	456,187	481,675	470,527	30
`createGameWithJoiner`	516,180	538,918	524,672	40
`disputeFeedback`	198,431	198,629	198,530	40
`joinGame`	175,268	199,287	183,004	20
`makeGuess`	213,099	232,353	218,411	120
`provideFeedback`	103,423	175,418	128,623	130
`publishCodeSecret`	210,095	210,230	210,163	40
`startTurn`	99,764	107,816	103,768	40
`submitCodeHash`	57,380	59,631	58,578	30
`verifyAFKAccusation`	80,412	80,646	80,506	20
Deployments				
Mastermind	-	-	4,507,180	15%

Table 5.5: Looped.js executed with maker/breaker defined in a variable

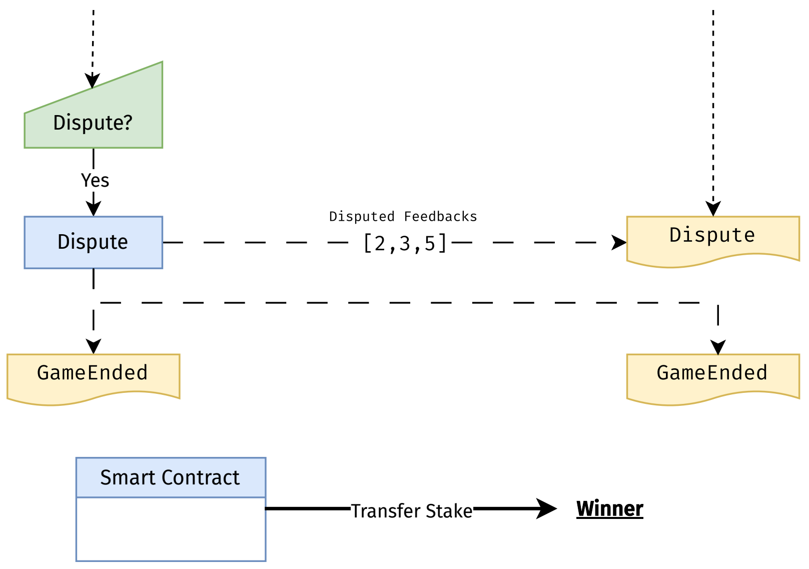

Creator
Joiner
Create Game
GameJoined
Join Game
Is Maker?
Is Maker?
Yes
No
TurnStarted
Start Turn
[2,2,2,5,5,5]
Salt: ...
Submit Code Hash
HashPublished
Guess
[2,4,6,2,5,5]
Make Guess
[CC,NC]
Provide Feedback
[3,1]
Feedback
Guess
[2,2,2,5,5,5]
Make Guess
Provide Feedback
[6,0]
Feedback
CC=6!
I guessed it!
I sent [6,0],
so the Turn
ends
guessed
true
TurnEnded
TurnEnded
Publish Code Secret
[2,2,2,5,5,5]
Salt: ...
CodeSecret
Published
I published the
secret.
Breaker will either
dispute my feedbacks
or start a new turn
Dispute?
No
Maker seems
honest ...
TurnStarted
Start turn
Now I'll be the
breaker ...
SWAP ROLES
HashPublished
Submit Code Hash
Make Guess

TURNS GO ON REGULARLY — 1st SCENARIO



Last turn
Dispute?
No
End Game
Under the hood "End Game" is a "Start Turn"
At this point, Maker has published the Secret, due to sending [6,0] as feedback, or having no more guesses allowed
GameEnded
GameEnded
Smart Contract
Transfer Stake
Winner

TURNS GOES ON REGULARLY until *Dispute* — 2nd SCENARIO



Dispute?
Yes
Dispute
Disputed Feedbacks
[2,3,5]
Dispute
GameEnded
GameEnded
Smart Contract
Transfer Stake
Winner