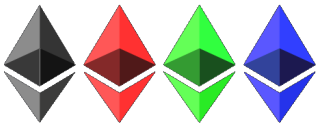

Peer to Peer Systems and Blockchains

---

# Mastermind on Blockchain



Marco Antonio Corallo

531466

# Contents

# Introduction

## Overview

The aim of this project is to develop a blockchain-based version of the *Mastermind* board game. The project includes:

- The development and testing of the game, along with a set of scripts to deploy and interact with it;
- An evaluation of the *gas cost* for the functions provided by the smart contract;
- An analysis of the potential *vulnerabilities* of the contract.

This report presents the main design and implementation choices, along with the gas cost evaluation and the vulnerability analysis.
Additionally, it provides a simple user manual with instructions for deploying the contract and interacting with it, as well as details about a basic front-end developed to enhance the gameplay experience.

## Gameplay and Rules

This section briefly summarizes the rules of the game.
The game requires two players who take on the roles of *CodeMaker* and *CodeBreaker*.
The CodeMaker chooses a secret code, composed of `N` colors. At each turn, the CodeBreaker submits a guess: a sequence of `N` colors. The CodeMaker responds with feedback consisting of two numbers:

- The number of correct colors in the *correct positions*;
- The number of correct colors in the *wrong positions*.

The CodeBreaker continues guessing until they successfully guess the code or reach the maximum number of allowed guesses.

At the end of each turn, the number of guesses the CodeBreaker needed to crack the secret code determines the points awarded to the other player, the CodeMaker. If the CodeBreaker fails to guess the code, `K` extra points are awarded to the CodeMaker.

## A Blockchain-Based Version of the Game

Compared to a standard implementation of the game, a blockchain-based version:

- Ensures that the *secret code* chosen by the *CodeMaker* at the beginning of each turn cannot be modified later;
- Ensures that the feedback provided by the *CodeMaker* to the *CodeBreaker* at each turn is consistent with the secret code chosen at the start of the game;
- Implements a *reward mechanism* for the winning players;
- Defines a *penalty mechanism* for cheating players or players who fail to take action during a turn, thus preventing the game from advancing.

## Technologies

Since the game has been developed on the Ethereum blockchain, the smart contract is written in *Solidity*.
The entire development and testing of the project utilized the *HardHat* framework, with *JavaScript* as the scripting language for the tests.
Additionally, a simple client was developed using *React.js* and *Node.js*.

# Game Phases

## Phases

In this blockchain-based implementation of Mastermind, a game is divided into *phases*. From the creation to the conclusion of a game, there are specific phases during which players can perform certain actions while being restricted from others. The list of phases, along with the capabilities and responsibilities of each player, is briefly explained below:

- `Creation`: This is the first phase of a game, where a player creates the game and waits for another user to join.
- `Declaration`: This is the second phase of the game. Both players must agree on a common stake and declare it, each doing so once. The stake is chosen *off-chain*; thus, if the stakes do not match, it is impossible to determine who cheated, and the game terminates.
- `Preparation`: Once both players have declared the (same) stake, they must pay it. This phase allows them to perform only this action.
- `SecretCode`: In this phase, the CodeMaker chooses the secret code and submits its hash.
- `Guess` and `Feedback`: During each turn, the CodeBreaker submits a guess, and the CodeMaker provides feedback on the previous guess. After the CodeBreaker makes a guess, the phase transitions to `Feedback`, and returns to `Guess` when the CodeMaker submits their feedback.
- `Solution`: This phase occurs when the CodeBreaker guesses the code correctly or the maximum number of turns is reached with-

out breaking the code. The CodeMaker must then submit the solution.

- `Dispute`: This phase is time-bound, during which the CodeBreaker can raise a dispute regarding incorrect feedback they received.
  If a dispute is raised, the smart contract verifies the correctness of the feedback, determines who cheated, and declares the winner. The game then terminates.
  If no dispute is raised within this period, points are updated, roles are switched, and the game returns to the `SecretCode` phase.

- `Closing`: After all rounds have been played, the smart contract declares the winner and transfers the game stake to them. The game then terminates.

## Constants

In this implementation, the number of rounds, turns, colors, and the length of the secret code align with the original version of the game:

- The secret code consists of 4 colors, with duplicates allowed.
- There are 8 turns, meaning 8 guesses and 8 feedbacks.
- A game comprises 4 rounds, allowing each player to take on the roles of CodeMaker and CodeBreaker an equal number of times.

Additionally, both the `AFK` period and the `Dispute` period are set to 36 seconds. This duration is a multiple of the Ethereum average block time, which is currently 12 seconds.
Finally, 3 extra points are awarded to the CodeMaker if, at the end of the round, the CodeBreaker fails to guess the code.

## Finite-State Machine

The life cycle of a game can be represented by the following *finite-state automaton*, where the nodes represent the phases, and the edges are labeled with the smart contract's functions that transition the game from one phase to the next. In this representation, the edges are labeled using a GCL-like syntax for simplicity and clarity.
The automaton also highlights the `AFK` states: when the player whose

turn it is delays their move, the other player can accuse them of being *Away-From-Keyboard*. From this state, the game can:

- Terminate if the accused player fails to make a move within the allocated time;
- Transition to the next phase if the accused player resumes and makes a move.

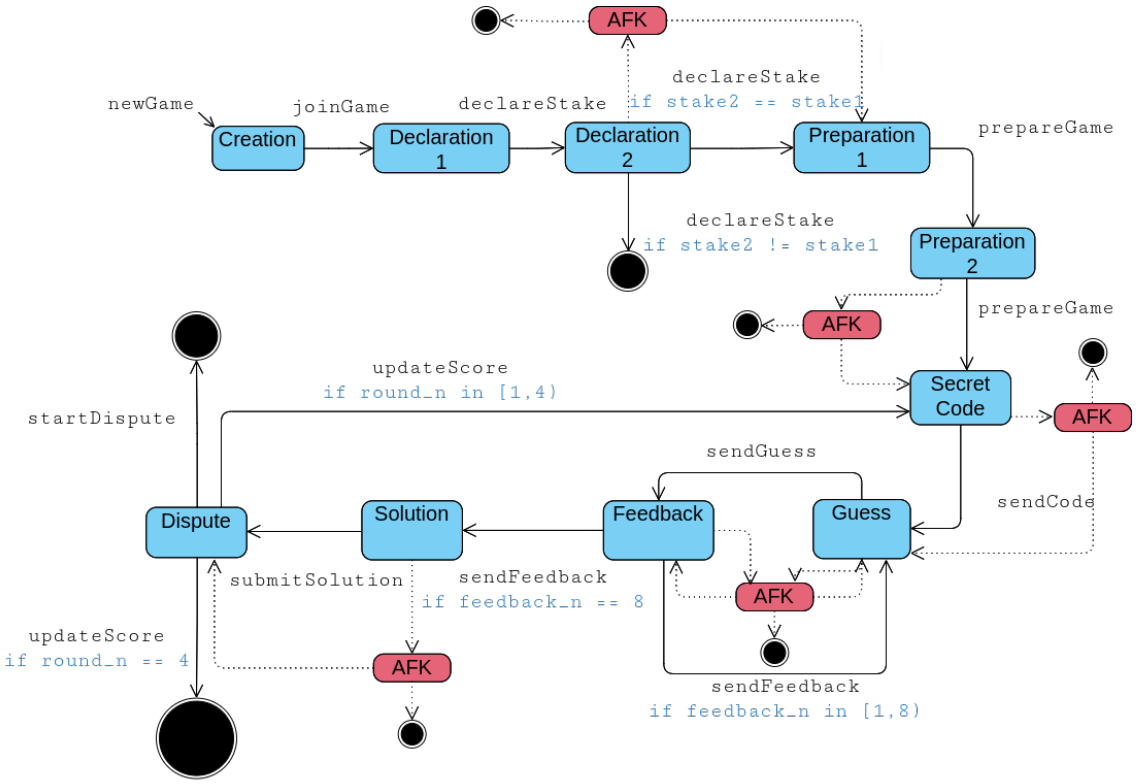



Figure 2.1: Finite-State Machine of a game.
The blue nodes represent the game phases, forming the main game path.
The red nodes represent the AFK states, while the black circular nodes denote termination states.

# Design Choices

In this chapter, the main design choices made during the project development are briefly discussed.

## Data Structures

A key design decision concerns the data structures used, particularly how to implement and handle a single game. Although implementing each game as a separate `Contract` could better leverage object-oriented programming (OOP) principles, it would result in higher costs, especially since there would be a contract for each game.
To balance efficiency and functionality, this implementation defines a custom library, `GameLib`, which provides a `Game` struct and several methods to handle the game.

This data structure stores information about the entire game, such as:

- the CodeMaker and the CodeBreaker;
- the agreed stake;
- the hash value of the secret code;
- the current phase of the game;
- the current round, along with the corresponding guesses and feedbacks;
- the AFK status and the block number where the AFK (or the dispute) started;
- the players' scores.

The smart contract acts as a *game orchestrator*, providing an API to securely and efficiently manage the game. All game-related information is stored within the `Game` struct itself. To achieve this, the smart contract maintains:

- a mapping < `GameID`, `Game` >, where `GameID` is the unique identifier for each game;
- a map of active players;
- a list of *free games*, which are games that have been created but not yet joined.

These data structures allow games to be handled in a simple yet efficient way during every operation.

## Stake Decision

In this blockchain-based implementation of Mastermind, the agreed game stake is determined *off-chain* by the two players. This approach reduces transaction costs and allows faster, simpler communication.
In the provided client, the stake decision mechanism is implemented as a simple **chat**, where the two players can communicate to agree on a common stake value.
Since the stake is decided off-chain, the smart contract cannot verify the agreed value until the players transfer the amount. If the two players transfer different stakes, it is impossible to identify who attempted to cheat. In such cases, the game will be closed.
If the stakes match, only that amount will be transferred, and any transaction with a different value will be *reverted*.

## An Additional Feature

Although the full game specification is satisfied, an additional feature has been implemented to allow players to leave a game. This feature was particularly useful during testing and debugging to shorten game times and increase the test-to-time ratio.
Moreover, there are practical scenarios where this feature could be beneficial. For instance, consider a player A who creates a dedicated game to play with player B, but B is unable to play for some reason.

Without this feature, A would be unable to play other games until B can join again.

When a player leaves a game in the `Creation` phase, the game must be removed from the *free games* list. This operation requires scanning the list to find the `GameID` to remove, which introduces a higher cost compared to other operations. However, this cost can act as a deterrent, encouraging players to proceed with the game rather than abandon it.

A player can leave a game at any phase or turn. When this occurs, an event is emitted, and the other player is declared the winner.

## To *Revert* or to *Punish*, That Is the Question

A key consideration during development was how to handle *minor* cheating attempts. Should they result in punishment or should the transactions simply be reverted?
Minor cheating attempts in this context include:

- The CodeMaker resubmitting the hash to modify the secret code;
- The CodeMaker submitting a solution that does not match the original hash;
- A player declaring a stake amount but transferring a different value;
  ⋮

After careful analysis, the decision was made to simply revert these transactions. These are mild and easily detectable attempts that would not benefit a cheater meaningfully. Reverting transactions also accommodates the possibility of honest mistakes, requiring users to resubmit valid transactions. This approach simplifies the development and maintenance of the smart contract.

Ultimately, the only way for a player to cheat (and be punished) is by submitting an incorrect feedback, as specified in the rules.

## Design by (Smart-)Contract

Beyond the wordplay, the development of the smart contract adhered to the *Design by Contract* approach, leveraging Solidity's `require` statements and `modifiers` to enhance code robustness.

Additionally, since smart contracts contain public code and data accessible by everyone (including other contracts), the development strongly incorporated the *Security by Design* approach. For every operation, the smart contract ensures that `msg.sender` is authorized for the specific game and allowed to perform the corresponding move during the given turn.

As a result, addressing security vulnerabilities played a central role in the smart contract's development. For more details on security considerations, refer to the next chapter.

# Vulnerability Analysis

As previously mentioned, security played a central role in the development of the smart contract. The following sections outline the main vulnerabilities that were identified during the development process, along with the corresponding solutions that were implemented to address them.

## Re-entrancy Attack

A *re-entrancy attack* is a well-known and dangerous vulnerability that has previously resulted in the largest crowdfunding exploit in history, raising over 12 million Ethers.
Re-entrancy occurs when an attacker performs recursive withdrawals to drain all the Ethers locked in a contract by *re-entering* the contract's execution flow before it is completed.
To prevent this issue, two well-established strategies have been implemented:

- Setting a gas limit of 3500 for transferring transactions: this amount is sufficient to send ETH but insufficient to execute additional calls. It also accommodates potential changes in gas costs.
- Updating user balances **before** the transfer: the only method for the contract to access the stake value is via the `popStake` function, which resets the balance after returning the value.

## Improper Authorization

Another common attack vector arises from phishing campaigns that exploit the use of `tx.origin` to bypass authentication mechanisms. To mitigate this risk, `msg.sender` is used throughout the contract

instead of `tx.origin`.
This approach also allows *contracts* to participate as players, enriching the game functionality.

## Timestamp Adjustment

This vulnerability leverages miners' ability to *adjust block timestamps* for personal gain. For instance, miners could manipulate the *dispute* time window, preventing the CodeBreaker from initiating a dispute, or reduce the AFK time to unjustly win the game by accusing the opponent.
To address these issues, time-sensitive logic (e.g., disputes and AFK checks) relies on `block.number`, which considers Ethereum's average block time of 12 seconds, instead of using block timestamps.

## Source of Entropy

As Solidity is a deterministic language, it cannot natively generate random numbers because all miners must produce consistent results. This implementation uses `blockhash(block.number - 1)` as a (limited) source of entropy.
Other methods, such as RanDAO or Chainlink VRF, have been investigated and are identified as potential future works.

## Brute-Force Attack

The secret code in this implementation consists of 4 colors selected from a pool of 6 colors, resulting in $6^4 = 1296$ possible combinations. An attacker could potentially extract the hash from the blockchain and execute a brute-force attack to determine the corresponding combination.
To counter this, the secret code is *salted* with 5 random `uint8` values, increasing the number of possible combinations to $6^4 \cdot 256^5 \approx 1.425 \times 10^{15}$. This makes a brute-force attack computationally impractical.
During the revelation phase, the player must submit both the secret code and the corresponding salt used to compute the hash.
To simplify this process for players, the provided client automatically

generates 5 random numbers (ranging from 0 to 255) and stores them for use during the *revelation* phase.

## DDoS Games Vector

A potential attack could involve a malicious player attempting to fill the list of available games, thereby denying other users the ability to play. This threat is partially mitigated by the intrinsic cost of gas fees. Additionally, in this implementation, a player can create only one game at a time, further limiting the feasibility of such an attack.

## DDoS Against a Single Player

Another possible attack involves targeting a specific player by creating a game for them, preventing them from participating in other games since players can only play one game at a time.
This issue has been resolved by allowing players to decide whether to accept a game invitation or leave it in a *waiting list*. This ensures they are not locked out of other gameplay opportunities.

## Hash Manipulation

A malicious CodeMaker might attempt to submit a new hash after the game has started. However, this is prevented by the contract, which *reverts* any such attempt.

## Frozen Stake

During the stake payment phase, a player could halt gameplay, effectively freezing their opponent's stake. To address this issue, players can accuse the opponent of being AFK or, alternatively, leave the game. Either action will return the stake to the player who already paid.

# Gas Cost Evaluation

The `hardhat-gas-reporter` package was used to evaluate the gas costs, with the following configuration.

| **Solidity and Network Configuration** | |
|---|---|
| Solidity | 0.8.27 |
| Optim | true |
| Runs | 200 |
| viaIR | false |
| Block | 30,000,000 gas |
| Network | ETHEREUM |
| L1 | 27 gwei |

The *Min*, *Max*, and *Avg* columns represent, respectively, the minimum, maximum, and average gas consumption observed during the tests for each corresponding method. These values are accompanied by the number of calls to the method and its average cost in euros.

Naturally, the gas consumption depends on the execution context: a reverted transaction consumes less gas than a fully executed transaction.

Finally, the report includes the total cost of deploying the contract and the average gas consumption. The gas price is fetched dynamically from the live network.

| **Method** | **Min** | **Max** | **Avg** | **# calls** | **EUR (avg)** |
|---|---|---|---|---|---|
| **MasterMind** | - | - | - | - | - |
| AFK | 63,914 | 85,516 | 67,100 | 104 | 6.26 |
| claimStakeByAFK | 187,533 | 225,931 | 210,448 | 12 | 19.62 |
| declareStake | 73,191 | 194,200 | 89,989 | 175 | 8.39 |
| joinGame | - | - | 102,480 | 101 | 9.56 |
| joinGame | 74,294 | 74,306 | 74,300 | 4 | 6.93 |
| leaveGame | 197,764 | 212,280 | 206,471 | 21 | 19.25 |
| newGame | 124,928 | 147,628 | 125,751 | 124 | 11.73 |
| newGame | 120,681 | 123,481 | 121,618 | 6 | 11.34 |
| prepareGame | 82,036 | 90,493 | 84,367 | 140 | 7.87 |
| sendCode | 40,205 | 61,887 | 52,297 | 76 | 4.88 |
| sendFeedback | 62,305 | 64,243 | 62,827 | 492 | 5.86 |
| sendGuess | 79,015 | 80,797 | 79,044 | 466 | 7.37 |
| startDispute | 257,404 | 261,496 | 259,145 | 14 | 24.16 |
| submitSolution | 71,473 | 108,791 | 86,922 | 54 | 8.11 |
| updateScore | 61,533 | 380,524 | 182,252 | 48 | 16.99 |
| **Deployments** | | | | | |
| MasterMind | - | - | 3,428,212 | 11.4% | 319.67 |

Table 5.1: Solidity and Network Configuration Data Table

# The client

To enhance user experience and improve the usability of the game, as well as to facilitate testing, a simple client application has been developed.

The client features a user-friendly front-end for interacting with the smart contract using event listeners, along with a lightweight *Node.js* server to manage the chat functionality for agreeing on the game stake.

The client was developed following the *Component-Based Development* approach, ensuring that each component has dedicated event listeners and well-defined responsibilities.

One of the key design decisions was to store game information in the browser using an *account-based* format. This design choice enables the client to handle multi-account games seamlessly and efficiently.

Additionally, the chat remains accessible to users from the stake decision phase until the conclusion of the game. This simple yet valuable feature allows players to communicate, for example, to agree on playing another game. It also provides a means of confirming synchronization between clients, helping to detect and resolve potential event-miss issues that could otherwise disrupt the game.

Furthermore, during the game selection phase players can view the games created specifically for them by other users.

Ultimately, despite the provided client serves as a basic *stub* for testing and playing Mastermind with ease, it has been designed to be extendable, offering ample opportunities for future enhancements, such as implementing polling mechanisms to retrieve past events.

# Test cases

All the requirements specifications for the development of *Mastermind* have been successfully met, including the optional ones.

The testing process followed a rigorous approach, utilizing the three most common test types to ensure reliability and robustness. Specifically:

- Each component of *Mastermind* was subjected to a comprehensive suite of *unit* tests.
- *Integration* tests were incrementally applied to the entire system, verifying the proper interaction between components and improving reliability.
- *Functional* tests were designed to cover the most common scenarios, including edge cases and potential malicious actions.

The test cases were crafted to cover every game phase comprehensively, including detailed checks for payment and refund operations to validate that the Ether transfers matched the expected amounts.
Most of the tests were executed automatically using the test suite provided by HardHat's `Chai` extension. Following this, each feature was manually tested through the GUI, which also helped identify potential limitations in the front-end.

In addition, the smart contract was successfully deployed and tested on both the Hardhat local testnet and the Sepolia testnet. This deployment ensured compatibility and functionality in different environments, providing additional assurance of the system's robustness.

Overall, more than 60 test cases were designed and executed, encompassing both successful and failing scenarios. This extensive testing

effort resulted in optimal test coverage, ensuring the overall reliability of the system. Code coverage metrics were generated using HardHat's built-in coverage tools and are summarized in the table below:

| **File** | **% Stmts** | **% Branch** | **% Funcs** | **% Lines** |
|---|---|---|---|---|
| **contracts/** | | | | |
| `Constants.sol` | 100 | 100 | 100 | 100 |
| `Events.sol` | 100 | 100 | 100 | 100 |
| `Game.sol` | 100 | 80.71 | 100 | 100 |
| `MasterMind.sol` | 100 | 80.68 | 100 | 100 |
| `Utils.sol` | 100 | 100 | 100 | 100 |
| **All files** | | | | |
| `contracts/` | 100 | 80.87 | 100 | 100 |

Table 7.1: Code Coverage Report

In conclusion, the development of both the smart contract and the client followed a *test-driven development* model, which facilitated the early detection and minimization of bugs. At the time of writing this report, all features appear to work as expected, even if it is well known that *"program testing can be used to show the presence of bugs, but never to show their absence."*

# User Manual

The smart contract is currently deployed on the Sepolia testnet and can be accessed by anyone using its address and the corresponding ABI.

To facilitate testing, playing, and interacting with the client, a set of scripts is available. In particular:

- `npm run start` is the main command to execute and test the entire system. This command concurrently runs two separate clients on ports 3000 and 3001, along with the server chat for the stake agreement.
- `npm run start:client` runs a single client on port 3000, while
- `npm run start:chat` runs only the server chat.
- `npm run start:test`, `npm run start:coverage`, and textttnpm run start:gas execute tests, with the latter two providing coverage analysis and gas cost evaluation, respectively.

Note that, to correctly run the system and interact with the contract, the following are required:

- Node.js;
- the necessary dependencies, which can be installed using `npm install` in the main directory;
- (at least) a Metamask account.

In the end, this report is accompanied by a simple *demo* of a game, making it easier to understand how to interact with the contract and properly use the client. Additionally, the automatically generated *NatSpec* documentation for the contract and its events is provided for reference. These additional resources can be found in the `README` file.