UNIVERSITÀ DI PISA

Peer to Peer & Blockchain
In Dipartimento di Informatica

by
Francisco Moura, Guilherme Campos

June 26, 2024

Contents

1 Introduction

The following report will focus on the final project of the **Peer to Peer & Blockchain** course. This project presents the implementation of the classic Mastermind game as a decentralized application on the Ethereum blockchain. Developed using Solidity, the project leverages smart contracts to facilitate secure, transparent, and fair gameplay between two players. The smart contract handles game state management, enforces rules, and ensures integrity by verifying actions and preventing cheating. The implementation also incorporates a staking mechanism to reward winners and penalize cheaters, enhancing the game's fairness and engagement.

2 Project Organization

The project, as recommended, was developed using the Hardhat framework so, following the framework functioning, multiple files and directories will be created. What we will be concerned over is three of the folders created and that were worked on:

- **Contracts:** folder which contains the solidity file which is the backbone of the project containing the full implementation of the code that handles the game;
- **Test:** folder containing the JS script that tests the validity of the code. All testing was done using this script to validate the well functioning of the code implemented in the solidity file.
- **Scripts:** this folder will contain the script that allows us to deploy the solidity smart contract to the Ethereum blockchain, allowing the game to function in a decentralized way.

3 Project Functioning

3.1 Contract Side

In the solidity code, our game is essentially represented by the struct *Game*, this struct includes information like the creator of the game, the player who joined, each player's points, the current turn going on and the stake of the game. The current turn saved in the game struct is also a struct called *Turn* which will hold onto the guesses and feedbacks made, who is the breaker and the maker in the current turn, and the hash of the secret code.

1. **Game Setup:**
   - The game creation start with the call for *createGame*, where user can define if he wants it to be a closed game with a specific player. User also defines initial stake.
   - Another user can join a game, either by having the ID of the game or joining a random available game. For joining "friend" games, you have to be the friend specified by the creator of the game. The stake to join the game must be the same as the creator.
2. **CodeMaker's Turn:**
   - The CodeMaker calls the function *submitCode* and selects a secret code, which is a sequence of colours, committing its hash to the blockchain to ensure it can't be changed later.
3. **CodeBreaker's Turn:**
   - The CodeBreaker, calling *submitGuess*, makes guesses to determine the secret code. Each guess is a sequence of colours.
4. **Feedback:**
   - After each guess, the CodeMaker has to submit feedback on how accurate the guess was, calling the function *submitFeedback*, it will come in a numeric form:
     - Correct colour and position: 2;
     - Correct colour but wrong position: 1;

    - Wrong colour: 0;
  - This feedback will allow the CodeBreaker to progressively get closer to the right code.

5. **End of Turn:**
   - The turn ends when the CodeBreaker either correctly guesses the code or exhausts the allowed number of guesses.
   - The CodeMaker then reveals the secret code, using *revealCode*, and the smart contract verifies the revealed code against the committed hash.

6. **Points and Penalties:**
   - Points are awarded based on the number of guesses taken by the CodeBreaker.
   - If the CodeBreaker fails to guess the code, the CodeMaker receives extra points.
   - Disputes can be raised if either player suspects foul play, and penalties are enforced accordingly.

7. **Game End:**
   - The game continues for a predefined number of turns, with players alternating roles.
   - The player with the most points at the end of the game wins, the breaker in that turn will call $finishGame$ and the smart contract disburses the stake accordingly.
   - There might be situations where a game ends prematurely, either by because a player was AFK for too much time or a player was dishonest during the game.

8. **AFK and Cheating:**
   - If a player takes too long to make a move, they can be accused of being away from the keyboard (AFK), by calling $accuseAfk$, and penalized in case they take too much time to defend against the accusation.
   - Cheating accusations, called via *accuseCheating*, are handled by the smart contract, if it's confirmed the player was dishonest while giving feedback, he'll be appropriately penalized for it.

3.2 Test Side

The tests implemented via JavaScript and use libraries provided by the Hardhat framework to interact with the smart contract. The purpose of this script is testing that the implementation of the backend of the game works properly, as it was explained before.

Some of the tests used in the script are:

1. Creating & Joining a game with an ID;
2. Creating & Joining a random game;
3. Trying to submit a code;
4. Trying to submit a guess and feedback;
5. Testing AFK mechanism by accusing and verifying AFK;
6. Testing AFK mechanism if player takes too much time to answer after being accused;
7. Try breaking the code and accusing of cheating;
8. Testing if turn changes after previous one ended;
9. Trying to finish the game by normal behaviour.

4 Main Decisions

4.1 Values & Constraints

For the well functioning of the contract, some constraints need to be set so that multiple games run on the same rules. Of all the values used in the contract, some of them are:

- **MAX_GUESSES:** not only the number of guesses the CodeBreaker has, but also the number of feedbacks for the CodeMaker. The only logical restraint is that it needs to be a positive integer.
- **MAX_TURNS:** the number of turns to be played in a single game. This must be a positive integer power of 2 so that in a game, both players play both roles the same number of times.
- **AFK_TIME_LIMIT:** this should be the time it takes before the player accused of AFK is deemed guilty of such. Although values are low in the present code due to testing reasons, a reasonable time should be 3 minutes.
- **DISPUTE_TIME:** the time the CodeBreaker has to review the game and in case it believes the CodeMaker cheated and accuse him of cheating over feedback given to one of the guesses. Again, the value might be low due to testing but, recommended values should be 1 minute in case of experienced players, but 2 should be better to also include new players.
- **EXTRA_POINTS:** this is the number of extra points CodeMaker receives if CodeBreaker did not find the secret code. It is a positive integer.
- **CODE_LENGTH:** it is the number of colours a code should have. Right now, it only accepts codes of length 4 but, in could accept bigger lengths.
- **NUM_COLORS:** this is the number of colours accepted in by the contract. Right now, colours are substituted by numbers that should correspond to a certain colour, should be up to a possible frontend to decide which colours should be, as long as the number equals NUM_COLORS.

4.2 AFK Accusation

The mechanism for accusing a player for AFK consists of three Solidity functions:

- $accuseAfk$: if an opponent is taking too much time to play, can send a warning and setup a window of time when the opponent has to answer. Only saves the address of the accuses and the timestamp of when it was launched.
- $verifyAfk$: if the accusation was not cleared and the time window has passed, the accuser can verify that the opponent has been AFK for the whole time window, and finishes the game.
- $defendAfk$: in case one of the players gets accused of being AFK, if it makes an action during the time window, it clears himself of the accusation and resets the accuser address and the timestamp of when it was made.

On the JS testing side, a mechanism to simulate the passage of time is used to simulate the AFK situation:

$$await\ new\ Promise(r => setTimeout(r, time\_in\_ms));$$

4.3 Cheating Accusation

During a game, multiple turns happen, and during a turn there are multiple guesses and feedbacks made. During feedback, the CodeMaker can be dishonest and lie about the feedback. This type of behaviour is considering cheating. The CodeBreaker, while reviewing the turn after the code is revealed, might find an inconsistency and accuse the CodeMaker of cheating.

The following cheating detection works as shown:

Algorithm 1: Work Distribution

```
function accuseCheating(gameId, guessNmr):
    revealedCode[] ← revealed code array
    guess[] ← guess obtained from guessNmr
    feedback[] ← feedback obtained from guessNmr
    colourCount[] ← array with occurrences of colours
    cheating ← bool to check if it's cheating
    for i ← 0 to revealedCode.length do
        colourCount[revealedCode[i]]++;
    for i ← 0 to revealedCode.length do
        if guess[i] == revealedCode[i] then
            if feedback[i] ≠ 2 then
                cheating ← true;
            colourCount[guess[i]]–;
        else if guess[i] ≠ revealedCode[i] then
            if colourCount[guess[i]] > 0 && feedback[i] ≠ 1 then
                cheating ← true;
            else if colourCount[guess[i]] == 0 && feedback[i] ≠ 0 then
                cheating ← true;
            colourCount[guess[i]]–;
```

5 User Manual

The project is developed using Hardhat and so, to work on it, one must have the framework installed in the computer. Following what was shown to us in the practical classes with professor Francesco Donini, the solidity file is present in the contracts' folder *./contracts*, and in the *./test* folder you can find the JS script to test and interact with the contract on the localhost. The test itself compiles and deploys the contract locally. To test the scripts, one should run the following:

```
npx hardhat node && npx hardhat test
```

After this, we also deployed the project into the Sepolia testnet, as shown in the class, we obtained an Infura API key and after that we got a Sepolia private key to setup the deployment. The deployment script can be found in folder *./ignition/modules* To deploy the project into the testnet, remembering that the Sepolia account associated should have enough Ethereum, one should run the following command:

```
npx hardhat ignition deploy ignition/modules/mastermind.js −−network sepolia
```

6 Potential Vulnerabilities

6.1 Gas Limit & Efficiency

There are a lot of modifications to structures in the blockchain, as well as iterations over arrays and emission of events. All these occurrences can potentially consume excessive amounts of gas, leading to out-of-gas errors. Some solutions could be use other types of data structures and limit the number of events to avoid gas limitations.

6.2 Integer Underflow/Overflow

The contract performs various arithmetic operations over different types of integers without explicit checks for underflow or overflow. This type of vulnerability can lead to "underflow/overflow attack" but, since solidity 8.0, the checks done by default by the compiler and therefore this problem is solved.

6.3 Front-Running Attacks

Some functions on the contract involve actions based on the state of the blockchain (e.g. blockchain.timestamp, block.prevrandao). This dependency can lead to attackers attempting to manipulate these values to their advantage in certain situations.

6.4 AFK Spam

The AFK accusation can be abused throughout the game since, after each opponent play, I can call on accuseAfk without any punishment. The way we deal with accuseAfk on the system, it's not a big problem to deal with the accusation, but it can be gas expensive. We could deal with this problem by introducing a minimum waiting period before an AFK accusation can be made, and ensure that the time limits are reasonable.

6.5 Unrestricted Game Creation

As the contract is, there is no restriction on creating a new game, which can lead to abuse or spam. One way to solve is implementing some type of fee, the more games it creates simultaneously.

7 Gas Evaluation

For analysing the gas expenditure, we need to evaluate each function and its respective gas values.

- *createGame*: 165526
- *joinGame*:
  - specific join: 129556
  - random join: 130728
- *submitCode*: 126459
- *submitGuess*: 105835
- *submitFeedback*: 109376
- *accuseAfk*: 76094
- *verifyAfk*: 35033
- *accuseCheating*: 78355
- *revealCode*: 122113
- *finishGame*: 56299

7.1 Gas Consumed in Normal Game

If we consider a normal game where the combination of colours in the code should be 4, with 6 turns of 5 guesses each, we have to consider the calls to functions of each player.

7.1.1 Gas of Player1 (Creator)

1. 1 x createGame
2. 3 x submitCode
3. {3, ..., 15} x submitGuess
4. {3, ..., 15} x submitFeedback
5. 3 x revealCode
6. {0, 1} x finishGame

7.1.2 Gas of Player2 (Joiner)

1. 1 x joinGame (specific or random)
2. 3 x submitCode
3. {3, ..., 15} x submitGuess
4. {3, ..., 15} x submitFeedback
5. 3 x revealCode
6. {0, 1} x finishGame

7.1.3 Total Gas Consumed

After analysing the possible functions calls of each player, we make the assumption that we play all turns and all possible guesses out, so 6 turns with 5 guesses and feedbacks each. Using previous information, we can make an estimation on how much gas we expend for the occurrence of one normal game:

$$165526 + 129556 + 6 * 126459 + 15 * 105835 + 15 * 109376 + 6 * 122113 + 56299 = \mathbf{5070978}$$

8 Conclusion

In conclusion, this project successfully implements a decentralized Mastermind game on the Ethereum blockchain using Solidity, ensuring secure, transparent, and fair gameplay. The Hardhat framework facilitated efficient development, testing, and deployment of the smart contract.

The game's design includes comprehensive game setup, turn-based play, feedback mechanisms, and end-of-game procedures. Security features address AFK players and cheating accusations, enhancing fairness.

While potential vulnerabilities like gas limits, front-running attacks, and AFK spamming were identified, they can be mitigated in future iterations. Gas evaluation indicates a need for optimization to reduce costs.

In summary, this project demonstrates the feasibility of creating engaging and secure decentralized applications, setting a foundation for future blockchain-based games and applications.