Assignment 4 BDL - s1814033

Pablo Miró

January 2022

1 Introduction

This report describes the design of a Chess game to be implemented in Solidity. I will go over the contracts used in the game, what they do, their data structures and their public APIs. Furthermore, after going over the contracts I will explain how a game is started and finished, evaluate the efficiency of the contracts, possible attacks and other design choices.

2 Contracts

The game will have two different contracts: `Game`, `Chess`, and a library: `GameLogic`.

2.1 Game.sol

This contract will be extended by our `Chess` contract and will contain the main logic a game between two players has. It will define a custom data structure, `struct GameStruct`, to store the important information about the game. It will contain:

- `address`es of player1 and player2 plus `bytes32` of their aliases (`player1, player2, player1Alias, player2Alias`)
- `address whitePlayer:` address of the player with the white pieces.
- `address nextPlayer:` address of player to make the next move
- `address winner:` address of the winner.
- `bytes32 gameId:` a variable containing the id of the game.
- `bool ended:` boolean to check whether the game has ended.
- `bool endsInDraw:` a variable to indicate if the game finishes in draw. In Solidity this is default to `false` so only needs to be changed to `true` when the game finishes as a draw.
- `uint timerDuration:` variable containing the duration of the timers.
- `uint timerStartTime:` variable used for the time at which a timer is started.
- `address playerToStartCheckmateTimer:` a variable to contain the address of the player who has their opponent in check and starts a checkmate timer.
- `int typeOfTimer:` takes a value from enum `TimerType`. See 2.1.1.

2.1.1 Timers

The `TimerType enum` will be made up of values `NO_TIMER, PLAYER1_DRAW, PLAYER2_DRAW, CHECKMATE, ABSENT_OPPONENT`.

1. `NO_TIMER`: indicates timer has not been set.
2. `PLAYER1_DRAW`: indicates timer was set by player1 during player2's turn.
3. `PLAYER2_DRAW`: indicates timer was set by player2 during player1's turn.

4. `CHECKMATE`: indicates timer has been set by player who thinks he/she will win by checkmate.
5. `ABSENT_OPPONENT`: indicates timer has been set by a player when he/she believes the opponent has stopped playing.

2.1.2 public API

- `GameCreated(bytes32 gameId, address player1, bytes32 alias1):` An `event` with 2 `bytes32` and an `address` to indicate that the given `address` has created a game with id `gameId` which is waiting for an opponent.
- `StartGame(bytes32 gameId, address player1, address player2):` An `event` which contains a `bytes32` and two `address`es indicating the second player has joined so the game has started.
- `GameFinished(bytes32 gameId):` An `event` which contains a `bytes32` to indicate the game has ended.
- `TimerStarted(bytes32 gameId, bytes32 playerAlias, uint typeOfTimeout):` An `event` containing two `bytes32` and a `uint` to indicate a timer has been started.
- `GameFinishedAsDraw(bytes32 gameId):` An `event` containing a `bytes32` to indicate a given game has finished as a draw.
- `DrawRejected(bytes32 gameId):` An `event` containing a `bytes32` which indicates that the draw has been rejected by the opponent.
- `Surrender(bytes32 gameId, address playerWhoSurrenders):` An `event` with a `bytes32` and an `address` to indicate a player has surrendered in a given game.
- `CheckMate(bytes32 gameId):` An `event` to indicate a game has ended via checkmate.
- `numberOfWins:` This is a `mapping (address => uint)` containing the `address`es of the players and the number of times each player has won a game.
- `games:` This is a `mapping (bytes32 => GameStruct)` which given a `gameId` will return the corresponding `GameStruct struct`. This `mapping` will contain finished games and games that are still being played.
- `openGames:` This is a `bytes32[]` containing the ids for those games which only have one player and are waiting for an opponent to join.
- `gamesIdsAndPlayers:` This is a `mapping (address => bytes32[])` which, given an `address` of a player, it returns an `bytes32 array` containing the ids of those games in which the player has/is involved with.
- `closeOpenGame(bytes32 gameId) returns (bool):` A function that closes an open game. It `require`s that the caller of the function is `player1` and that there is no `player2`. It will remove the game from `openGames`, change the game's `struct` to mark it as `ended` and emit a `GameClosed event`. The function will return a boolean (`true`).
- `surrender(bytes32 gameId):` A function through which a player can surrender. It should check the game has two players and has not been `ended`. Depending on whether the sender is player1 or player2 it should update `numberOfWins` and `winner` in the `Game struct`. Emits a `Surrender event`.
- `isGameEnded(bytes32 gameId):` A function which returns a `bool` indicating whether a game with id `gameId` has ended. This will use the `ended bool` from the `GameStruct` that corresponds to `gameId`.
- `initGame(bytes32 player1Alias, uint timerDuration) returns (bytes32):` The function to start a game. It should create a new `GameStruct struct`, set the `address` of `player1` with `msg.sender` and use the arguments to update `player1Alias` and `timerDuration` in the `struct`.

It should mark the game as in progress (i.e. `ended = false`), set `typeOfTimer = Timer.NO_TIMER` which indicates there is no timer in progress and set `checkmateTimer = timerDuration`. It must generate a gameId for the game and add the game to `games`, `openGames` and update the `mapping gamesIdsAndPlayers` such that `msg.sender` has this new game added to their games. We do not need to create a `gameId` with a secure randomized process: this id does not affect which player starts playing or gives any advantage to any player. Hence, we can just use a hash function such as `keccak256` or `sha3` with the `block.timestamp` and the address of `player1` to create such id. The function should then emit a `GameCreated` event and return the `bytes32 gameId`.

- `joinGame(bytes32 gameId, string player2Alias) returns (bool):` A function which `requires` the `GameStruct` that corresponds to `gameId` has a player1 (that is, the game has been created), has not ended (`ended = false`) and does not have a `player2`. It will update both `player2` (using `msg.sender`) and `player2Alias` in the `GameStruct struct`. Furthermore, it will add the `gameId` to `gamesIdsAndPlayers` for the `address` of player2 (the address that called `joinGame`, i.e. `msg.sender`) and remove the `gameId` from `openGames`. The function returns a boolean (`true`).
- `startWinTimer(bytes32 gameId) returns (bool):` A function that `require`s the sender is playing the game, the type of timeout from the game is `Timer.NO_TIMER` and the sender is calling the function during the turn of the opponent (hence he/she is not `nextPlayer` from `GameStruct`). Then it will udpdate `timeStartTime` with `now`, `typeOfTimer` with `Timer.CHECKMATE` and will set `playerToStartCheckmateTimer = msg.sender`. Lastly, it will emit a `TimerStarted` event and return a boolean (true).
- `startTimer(bytes32 gameId):` A function that starts a timer for the case in which the sender thinks the opponent is not playing any more. It `require`s that no timer has started, the sender is a player in the game and is not the next person to play (i.e. he/she is waiting for the other player to play but wants to start this timeout because the opponent takes longer than usual). Otherwise it will set `typeOfTimer` from the `GameStruct struct` as `Timer.ABSENT_OPPONENT` and will start the timer, i.e. will set `timerStartTime` to `now`. The function returns a boolean value (`true`).
- `offerDrawToOponent(bytes32 gameId) returns (bool):` A function that checks whether sender is playing the game with id `gameId`, that the `typeOfTimer` of the `GameStruct` indicates the game has no timer and then updates the `typeOfTimer` depending on who offers the draw: `Timer.PLAYER1_DRAW` or `Timer.PLAYER2_DRAW`. It will also update the `timerStartTime` variable from the `GameStruct struct` to contain the time at which this function was called (`now` in Solidity) so that it can be checked when the timer has passed. It will emit a `TimerStarted` event and return a boolean value (`true`).
- `rejectDrawOffer(bytes32 gameId) returns (bool):` A function that will `require` that the sender is playing the game, the timer has been set by the opponent to `Timer.PLAYER1_DRAW or Timeout.PLAYER2_DRAW` (depending on who the opponent is) and that the sender is not the player that has to make the next move. It sets `typeOfTimer` to `Timer.NO_TIMER` and emits a `DrawRejected` event. The function returns a boolean value (`true`).
- `function finishGameWithTimer(bytes32 gameId) returns (bool):` A function to finish the game if the opponent is not playing any more. It will `require` the sender is playing the game, `typeOfTimer` is `Timer.ABSENT_OPPONENT` and that `timerDuration` minutes have passed after the `timerStartTime` variable was updated when `startTimer` was called.It will update the `ended` variable of the game to `true`, it will update the `winner` variable to `msg.sender`, emit a `GameFinished` event and return a boolean value (`true`).
- `function finishGame(bytes32 gameId) returns (bool):` A function that will `require` the game has not ended, the sender is playing the game and that a timer has been started (except for `Timer.ABSENT_OPPONENT`, for this timer `finishGameWithTimer` should

be used) and that the timer has already passed (i.e. that `now` > than `timerStartTime` + `timerDuration`). It will then check if there is a draw (if so the `winner` will be set to `address(0)` and `endsInDraw = true`) or if `typeOfTimer` is `Timer.CHECKMATE`. For the latter case, it will set `playerToStartCheckmateTimer` as `winner`. Then the game will be marked as `ended`, the number of wins of the sender will be updated to have one more victory and the contract will emit a `GameFinished` event. The function will return a boolean value (`true`).

N.B. (this will also be detailed later): If during this timer the opponent has managed to make a move (using `move` from `Chess.sol`) the timer will be removed and the variable will be set to `typeOfTimer = Timer.NO_TIMER` so the player who starts the timer will not be able to claim victory if the opponent manages to escape the check or if the player claimed a checkmate when it was not true. This will also happen for the case when the type of timer is `Timer.ABSENT_OPPONENT`.

2.2 Piece.sol

This will be an abstract contract that will be used for the pieces in the chess board. Its `public` API is:

- `int initialised`: This variable will be set to 0 when an instance of the class is created since that is the default value in Solidity (and because Solididy does not use `undefined`).
- `bool isWhite`: A variable which determines whether the piece is white or black.
- `bool killed`: A variable that represents whether the piece has been killed.
- `canMakeMove(mapping (int => Piece) chessBoard, uint fromIndex, uint toIndex)`: An abstract function that will be implemented by those classes extending this contract.
- `isPathFree(uint fromIndex, uint toIndex)`: An abstract function implemented by those classes that extend `Piece.sol` and checks whether the path is free for them.

  This class will have a constructor which will set `initialised` to 1 and will take an argument for setting `isWhite` to its corresponding value.

2.3 Pawn.sol

This class extends the `Piece.sol` abstract class. Its `public` API contains:

- `bool isInStartingPosition`: A variable to check whether the pawn has not been moved yet.
- `isPathFree(uint fromIndex, uint toIndex) returns (bool)`: A function that checks whether the pawn has any piece of the same colour in `toIndex`. Returns a boolean value indicating whether the path is free or not.
- `canMakeMove(mapping (int => Piece) chessBoard, uint fromIndex, uint toIndex) returns (bool)`: A function that needs to do several things:
  - check if the pawn is black or white. Then it will see if the move is vertical and if the pawn is going to a bigger number (adding 16) for a white piece; or going to a lower number (subtracting 16) for a black piece. See how the 0x88 representation works
  - check if the player intends to make a double move and if the pawn is in its starting position.
  - If any of the above does not hold the method returns `false`. Otherwise returns the result of calling `isPathFree` with the same `fromIndex` and `toIndex`.

2.4 Knight.sol

This class implements the `Piece.sol` interface. Its `public` API contains:

- `int[] knightMoves:` This will be an array of `ints` that should contain the moves that a knight can make using the 0x88 representation. For example, if a knight goes up, up and right it should add 31 to its current position. Consider the same thing, but for the possible 8 moves of a knight.
- `isPathFree(uint fromIndex, uint toIndex):` A knight can jump over pieces so it only checks if there is a piece of the same colour in `toIndex`
- `canMakeMove(mapping (int => Piece) chessBoard, uint fromIndex, uint toIndex):` A function that will return a `bool` indicating whether the knight can make a move. This will return the result from `isPathFree` since the knights can jump over pieces and I will have other checks (such as checking the move is not out of bounds) in `ChessLogic.sol`.

2.5 Bishop.sol — Queen.sol — Rook.sol — King.sol

These contracts extend the `Piece.sol` abstract contract as well. Their public APIs will be almost the same by having different code changing the logic of `canMakeMove`. They do not need to consider colour since their moves are not constrained as with pawns that can only go forward.

- `canMakeMove(mapping (int => Piece) chessBoard, uint fromIndex, uint toIndex):` A function that checks whether the bishop/queen/rook/king can go to the required end position (assuming no obstacles). N.B: When callin it from `King.sol`, and if the king and rooks are in original positions, `canMakeMove` will have to check if the king is moving 2 positions towards a rook and see if the castling is valid. It will update the `castling` flags in `Other struct` accordingly if a rook or the king have been moved (so *castling* cannot be done). It will return `false` if the piece cannot go to `toIndex` assuming there are no obstacles. Otherwise, it returns the result from calling `isPathFree`
- `isPathFree(uint fromIndex, uint toIndex):` A function that checks whether there is any piece in the path going from `fromIndex` to `toIndex`. This will vary for the different pieces (bishop, rook, queen, king) since they have different possible moves.

2.6 ChessUtils.sol

2.6.1 Data Structures

We will have the following data structures in this contract:

- `chessBoard:` A double `mapping` `mapping (bytes32 => mapping (int => Piece)) boardFromId` which, given a `gameId`, returns a mapping that indicates the pieces at the different positions in the board. We will be using the 0x88 representation of the chess board to make moves from one slot to another. Positions are those from 0x88 representation, i.e. {0, ..., 7, 16, ..., 23, 32, ..., 39, ...}
- `Player:` An `enum` containing two values, `WHITE` or `BLACK`.
- `Other`: An struct to store other variables together. This will contain the position of the kings and a flag for *castling*. (`blackKingPosition, whiteKingPosition, isWhiteCastlingPossible, isBlackCastlingPossible`).

2.6.2 public API

- `Other flags: Other struct` initialised in `initBoard`.
- `initBoard(mapping (int => Piece) chessBoard)`: A function to initialise the chess board. This will update those positions that corresponds to the first 16 and the last 16

in a chess board to contain the starting pieces. It should set the `isWhite` variable of the pieces to the corresponding value according to their starting position. It will also initialise an `Other flags` with the beginning positions of the kings and the castlings flag to `true`

- `firstChecks(mapping (int => Piece) chessBoard, uint fromIndex, uint toIndex, int playerColour)`: A function that will get the piece at position `fromIndex` and check the player is moving a piece of his/her colour using the `isWhite` field of the piece. This function will also check that the piece is being moved to a position that is empty (remember Solidity has no `undefined` value so to check a slot is empty we use `board[toIndex].initialised` whose value should be 0 if there is no piece in there) or has a piece from the opponent (thus the corresponding piece will have `isWhite` to the opposite value of `playerColour`. For checking the colour remember that the `Player enum` will set `Player.WHITE = 0` and `Player.BLACK = 1`. This will also check the piece does not stay in the same place and does not go out of bounds.
- `isKingInCheck(mapping(int => Piece) board, int playerColour, int kingPosition)`: A function that will return a `bool` indicating whether the king of the `playerColour` player is in check. It will consider the closest 8 positions to see whether there are any pieces of the opponent that can "kill" it. It will have to consider also the possible moves of a knight and, if some neighbours are empty, it will have to check if there is any piece, such as the rook/queen/bishop that can "kill" it.
- `isMoveValid(mapping (int => Piece) chessBoard, uint fromIndex, uint toIndex, int playerColour) returns (bool)`: A function that will obtain the `Piece` at `fromIndex` and will return the result from calling `canMakeMove` with the same `chessBoard`, `fromIndex` and `toIndex` arguments as this function. It will also `AND` the result with the value obtained from `isKingInCheck(mapping (int => Piece) chessBoard, playerColour, kingPosition)`. N.B: kingPosition can be obtained by using the `Other flags` given `playerColour` (use `flags.black/whiteKingPosition`).
- `makeMove(mapping (int => Piece) chessBoard, uint fromIndex, uint toIndex)`: A function that will change the piece in `fromIndex` and put it in `toIndex`. If there is a piece of the opposite colour in `toIndex` it should replace it in the `chessBoard mapping` with the piece in `fromIndex` and set the field `killed` of that piece to `true`. Otherwise it just changes the `mapping` to have the piece in the new position.
- `move(mapping (int => Piece) chessBoard, uint fromIndex, uint toIndex)`: This function will use `firstChecks` and `isMoveValid`. If both are `true` then it will use `makeMove`.

2.7 Chess

- `Move(bytes32 gameId, address player, uint fromIndex, uint toIndex, mapping(int => Piece) currentBoard`: An event with a `bytes32`, an `address` and two `uint`.
- `TimerEnded(bytes32 gameId)`: An event with a `bytes32`.
- `gamesBoardState`: A `mapping (bytes32 => ChessUtils.ChessGame)` that stores the ChessGame struct of a game given its id, `gameId`.
- `initGame(string player1Alias, uint turnTime) returns (bytes32)`: A function that calls `initGame` from its parent class (and stores the `gameId` which comes from calling `super.initGame(...)`. It will then call `initBoard` from `ChessLogic` with `gamesBoardState[gameId]` to initialise the chess board corresponding to the `gameId` that was just created in the `super` class.
- `joinGame(bytes32 gameId, string player2Alias) returns (bool)`: Function to call the parent class' function `joinGame`. After doing so it will select which player is playing with the white pieces by using a secure randomized process (see 2.12). It will then

update the `address whitePlayer` from the `Game struct` with id `gameId` and also `address whitePlayer` from the corresponding `ChessGame` struct.

- `move(bytes32 gameId, uint256 fromIndex, uint256 toIndex):` A function that checks the sender is a player, gets the `GameStruct` corresponding to `gameId`, checks that it has not ended and that the sender is `game.nextPlayer`, i.e. he/she is the next one to play. Otherwise it throws. It uses the `move` function from `ChessLogic` to make the move. If this does not `throw` (i.e. the move was completed successfully), this function will remove any timer that was running by setting `typeOfTimer` to `Timer.NO_TIMER` and emits a `TimerEnded` if this is the case. It will change `nextPlayer` from the corresponding `GameStruct struct` and emits a `Move` event.
- `getWhitePlayer(bytes32 gameId):` A function to return who the white player is in game with id `gameId`.
- `startWinTimer(bytes32 gameId) returns bool:` This function will `require` the sender is a player of the game and that the opponent is in check. The function will check this by calling `isKingInCheck` with the opponent's colour. Then, `startWinTimeout` will be called in the parent class using the `super` keyword. Note: To get the colour it will use `game.whitePlayer` and see if the address is the same as `msg.sender` which means the opponent has the black pieces (otherwise the opponent has the white ones because we have checked `msg.sender` is indeed playing the game). The function returns a boolean value (`true`).

Recall that by extending the `Game.sol` contract, `Chess.sol` will also have its other methods I have not mentioned, i.e. `surrender`, `offerDrawToOpponent`, `rejectDrawOffer`, `finishGame`, `finishGameWithTimeout`, `startTimeout` ...

2.8 How a game starts and ends

A game starts when a player interacts with the smart contract by calling the `initGame` function. This will generate a `gameId` that will be emitted by the `GameCreated` event so that the player can write it down to tell his/her friend for joining the game.

The second player will use this `gameId` to join the game. The contract will follow a procedure for obtaining who has the white pieces (i.e. starts the game) explained in 2.12. After this, both players will be able to start making moves. Then, the game will be able to finish in 3 different ways:

- A player thinks the opponent has stopped playing so he/she starts a timer. After waiting for `timerDuration` minutes the player can call `finishGameWithTimer` to finish, and win, the game. N.B: if the opponent makes a move before the timer finishes the game will continue (the `move` function removes the timer and `GameStruct.nextPlayer` is updated).
- A player thinks the game is too difficult or that they are getting nowhere so offers a draw during his/her opponent's turn. The opponent can just reject the offer or accept it by waiting `timerDuration` minutes and then using `finishGame`.
- A player has made a checkmate so it starts a timer with `startWinTimer` and after the timer calls `finishGame`. The opponent will not be able to remove this timer since he/she can only remove it by using `move`; and the opponent will not be able to call `move` because this throws if the opponent ends in check after the move (which the opponent will if there is a checkmate).

2.9 Timing rules to ensure a game does not run forever

I already discussed how the timing rules work in section 2.8. A game will not run forever because when one player stops playing the other can just start a timer that will let them finish the game.

2.10 Making moves and notifying the players

Players will interact with the contract to move their pieces by using its `move` function detailed in the above `APIs`. This function will emit `Move` events to notify the players about the move. They record the `address` of the player that makes the move, the start and end position of the piece, and the state of the chess board.

2.11 Design/coding patterns to increase gas fairness and efficiency + tradeoffs

2.11.1 Gas fairness and efficiency

We use a `guard check` in the functions with `require` for validating the inputs of the user (so the user does not pass on arguments which are invalid e.g. in our contract we can have playerColour = 0 or 1 so passing another number is not valid). We use the `require` method because it uses opcode `REVERT` which returns the gas that has not been consumed.

Another pattern to take into account would be the `checks-effects-interactions` pattern. This basically implies that we should always check who is the `msg.sender`, if they sent enough funds, if the arguments are in range... Having done so, then we can let the function proceed to make changes. This will improve the security and fairness of the contract since the function will not make unnecessary changes when it should not.

Another proposal that I have in mind has to do with the computational cost of making a single move. For every single move there are several checks that have to be made and this means the cost of such transactions will be more expensive. I will discuss this again in 2.13.

The game could be modified such that players can place a bet before starting the game. I did not add this in the API, but it will simply mean to update the `GameStruct` with the amount of the bet and give it to the winner (or to both players in a draw). In such case we would use the `pull-over-push` pattern which will make the contract more secure and fair by letting the users withdraw their own money.

A different way to increase gass efficiency would be to restrict access to some functions such as `isPathFree` that will not be used by the players. Other functions could be moved to `internal` e.g. `canMakeMove`, `isKingInCheck`... since this will reduce the cost of the contract.

Minimising the use of `storage` data will also be beneficial for the cost of the contract. Using less `state variables` and keeping non-permanent data in memory plus using mappings instead of arrays when possible will reduce the cost. The code could also use smaller data types e.g. `bytes8` and pack several of them into a single `32 bytes` slot.

2.11.2 Trade offs

Due to the gas fees for making transactions, each call to `move` will cost more money if we keep the computation of valid moves on-chain. Moreover, due to the difficulty of checking whether there is a checkmate, it seems much more efficient to make the player claim he/she has done a checkmate and start a timer that would let the opponent show if that was true (otherwise the player can finish the game with the timer). Hence, the players will have to get used to setting timers and then verifying they have ended if they want to play chess in such contract.

Another important thing will be that the player starting the game will have to pay for more fees to initialise the game. Moreover, the second player will have to pay some extra fees to compute who is the white player. With the current gas fees in Ethereum, the developer might consider moving to the Solana blockchain... Furthermore, if we consider the players are betting money and we are using the `pull-over-push` pattern, the contract will be fair in the sense that the winner will the one paying for finishing the game/getting the bet. The other interactions will also be fair: each player pays for their own moves and nothing else apart from initiating/joining (each player pays for one), offering a draw (one player pays for the offer and the other accepts/rejects it) and finishing the game (winner pays for it).

2.12 A secure randomized process to choose which player gets to play white in every game

The first idea that could be to use values such as `block.timestamp/number/hash/coinbase`, `now`... However, these can be manipulated by a malicious miner and they are shared within the same block to all users.

In order to obtain a random number much more safely we could use, similarly to coursework 2 in the matching pennies contract, a `commitment-reveal scheme`: The first player to start the game will commit to a value by sending its hash to the contract. Then, when the second player joins the game they commit to another value. After this, the two players reveal their original choices, which will be verified by the contract, and the contract will use the `AND` of their choices as solution. If the result is 0, player1 will be white; if the result is 1, player 1 will be black. If we let players choose bigger numbers, the contract might `AND` the digits after `AND`ing the values in order to obtain a 0 or a 1.

On a side note, I would like to note that there exists services such as chainlink (a decentralised blockchain oracle network) with which we could obtain randomness on-chain, and avoiding to have the random value tampered/manipulates by anyone. See its documentation for using randomness.

2.13 Design choice according to attacks discussed in lectures

2.13.1 Griefing

The smart contract will not suffer a griefing attack because it is not dealing with ETH, so it cannot run out of money. If we implemented it to let the contract hold a bet we could avoid this by making use of the `pull-over-push` pattern (with a `withdraw` function and avoid `send()` and `throw` in our code.

2.13.2 Reentrancy

Similarly as with a griefing attack, the contract will not suffer a reentrancy attack since it does not hold ether. In case we did add the bet option, the contract will be implemented by resetting every variable and updatting the users' balances before sending any amount. Again, since we would use the `pull` and `checks-effects-interactions` pattern we will mitigate the risk for this attack.

2.13.3 Front-running

A front-running attack would not make sense in the contract since this will check that the sender of a transaction is indeed playing a given game. If they are not player1 or player2 the transaction will be reverted. Further, there is no benefit in making a transaction before another (not even for the players because for making moves the variable `nextPlayer` checks who has the next turn).

2.13.4 Other possible hazards

Another possible attacks would have to do with integer overflows/underflows, but with a Solidity higher than 0.8.0 this is checked automatically (otherwise we would just use the `SafeMath.sol library`). On the other hand, there might be a problem if the code needs some sort of randomness, but we already discussed that the code will be designed to use a commitment scheme.

Last but not least, it might be the case that the owner of the contract or any player is malicious. The contract is defined so that the address belonging to the owner has no privilege over any other address (and the code is deployed in Ethereum so it is publicly available for anyone to see it). With respect to players being malicious, they could not really place several moves and the only thing they could really do would be to stop playing (but if they are placing a bet they would lose it), which means they will lose the game.

2.14 Parts off-chain to reduce cost.

The principle change we would have to make for the players to pay for less gas would be to let the validation of the moves be done off-chain. This includes checking whether a given move will result in the player been in check or checking whether a move is in bounds and does not interfere with any piece during its way to the end position.

Furthermore, checking whether there is checkmate could also be done off-chain. This is because the contract will have to go and see every piece of the opposite colour and whether they can get to the king's position without being blocked. This will cost quite some gas since we will have to do this check for every piece of the opponent.

2.14.1 How would they retain trust/security guarantees of an on-chain game

In order for maintaining trust and security as if this were done off-chain, we could set up a KYC process with a trusted third party as we discussed in coursework 3 section 3. We would have each player own a token from a certificate authority with which they could sign their moves and send them to this trusted party that will validate them and could even be given permission to perform them. Then, this party will sign its execution and send it back to the players so that they can obtain the updated chess board, verified by the trusted party.