Assignment 2

Pablo Miró
BDL - Blockchain and Distributed Ledgers

November 24, 2021

Contents

1 High-Level decision making

The smart contract implements the matching pennies game defined in the coursework document. In this section I explain how the game can be played following my implementation. Later on I will argue why I made these implementation choices.

- Both players register to the game and bet 1 ETH.
- First, player A chooses a move (1 for *Heads*, 2 for *Tails*) and a *salt* value. Then they compute the hash of the concatenation of the move and salt (h(move—salt), with h being the keccak256 hash function) , i.e. their *commitment*; and this is what they will send to the contract.
- Having player A committed to a move, player B computes their commitment and sends it to the smart contract.
- Once both players have committed to their values, they will reveal their choices to the smart contract. The contract will check they are being honest and assign to each player their corresponding choice of move.
- After the revelation of their commitments, one of the players finishes the game and the contract updates each player's balance.
- The game is finished, addresses and other state values are restored, and the winner can withdraw its money form the contract.

I divided the implementation in several phases: Registration, commitment, reveal and result:

1.1 Registration

In this section both players register to play the game and send 1 ETH to the contract: the game is defined such that the winner gets 1 ETH as reward. I did not allow sending more money to keep the smart contract for playing the game and nothing more. Giving more functionality than needed to the smart contract (e.g. acting as a bank) is not a good practice. This can be prone to error and vulnerabilities. Moreover, it is to mention that inside the contract I tried to keep good practices by encapsulating the logic and making every function have a unique purpose. There is also no point on allowing players to bet more money because the game is set so that the reward will be 1 ETH anyway.

On a separate note, I added a function so that the first player to register can cancel their bet in case they do not have any friends to play with.

1.2 Commitment

There are several ways I thought for implementing this phase after the registration. I will explain the different options considered and what I chose to do in the end.

My first thought was that both of the players have to make a commitment so that nobody can cheat. After trying this in the code, I believed to have found a way for player A to cheat. Remember player A would win the game if both players choose the same move (either both choose *Heads* or *Tails*), whilst player B is the winner if the choices are different. Therefore, I thought that if player A waits for player B to play, player A can send the same hash. By doing so, player A will have committed to the same move as player B and player A will win. After thinking about this problem, I found a counter-argument for why player A would not immediately win and how I could fix this. Let us consider player A sends the same hash as player B. There are two different outcomes:

- Player B reveals its choice. Then, player A can check the blockchain and player B's transaction to see what is the (value, salt) pair player B committed to. Therefore, player A will be able to reveal their commitment and they will win because both players have chosen the same value (both *Heads* or *Tails*). Player A wins.
- Player B realises player A has sent the same commitment and decides not to reveal their commitment because they know they would lose. In this case, if player B does not want to reveal his commitment, player A will not be able to reveal it neither (player A does not know what they committed to, they only have the hash). In this case the smart contract would get lock.

This is a big problem, but there is a simple solution for it: We can just fix this by making player A commit to a value first. I changed the *play* function I had to send the commitment for two different ones, *play1* and *play2*, so that player A sends the commitment before player B (player A will use *play1* and player B *play2*).

In light of the above, there are two possible ways I considered for implementing the commitment phase.

1. On the one hand, it is not compulsory to make both players commit to their choices: we can have the first player playing the game committing to a value while the second one does not make a commitment. This way, even if player A is the second person to play, they will have to send the decrypted choice so player A cannot just send the same hash as player B (either 1 or 2).
2. On the other hand, we could make both players commit to their values and put a restriction (with a *require* statement or a *modifier*) so that they cannot send the same hash or make player A the first one to play.

I chose to use the latter option and force player A to make a commitment first. By doing this, both players will have to call the smart contract twice: to send the commitments and, as we will see later, to reveal them. I chose this because it means that both players will have to pay for the same amount of fees, as I will discuss later.

N.B. The smart contract provides a function for computing the hash value given a choice and a salt. It *should NOT* be used by the players in a real game and is only there for testing purposes. If a player computes their hashes using the function in the smart contract, the other player could see what value they are committing to by looking at the blockchain.

1.3 Reveal

After both players commit to a value they will have to reveal it so that the smart contract can check they are honest and compute the winner. The players will call `revealChoiceFromCommitment( string memory revealedChoice, string memory salt)` with the value and salt they used to compute the commitment they sent; and the contract will check this pair of (value, salt) is the one which corresponds to their commitment. If this is the case, the contract will assign the corresponding choice to the player. Otherwise, it will leave their choice as *NA*.

Both players will have to call a very similar function for committing to their results and the same one for revealing their commitments. Consequently, they will have to pay for almost the same amount of fees ($\pm$ some small amount that can deviate due to congestion of the network).

1.4 Result

Once both players have revealed their commitments, they will be able to finish the game using the public function `finishGame()`. This function will compute winner of the game and update a mapping that contains the balances of both players. After this, the winner will be able to obtain the reward from the contract using the `withdraw()` function.

1.5 Detailed answers to specific implementation choices

Who pays for the reward of the winner

The reward of the winner is paid by the player who loses the game. Both players send 1 ETH to the smart contract so that they can register to play the game. After playing the game, the smart contract will update the mapping *balances* which indicates how much money from the contract is owned by which player.

Without loss of generality, if player A wins the game the contract will update their balance by adding 2 ETH to it, while player B's balance will not be updated.

How is the reward sent to the winner

In the beginning I wrote the code so that the contract will send the money to the winner when the `finishGame()` function is called after both players have revealed their results.

Nevertheless, I chose to add a mapping which will contain the corresponding ETH to each player after the game. Players can withdraw their money by calling the `withdraw()` function. They can also play several games and withdraw their money in the end. I will explain why I decided to do so in the following section.

How is it guaranteed that a player cannot cheat

There are several ways a player could try to cheat or :

- As I mentioned in the commitment section before, player A could send the same hash as player B and hope that player B reveals their commitment. This could happen in player B did not check player A's hash and thought player A was a honest person. This was fixed by making player A send a commitment first.
- By making the players commit to their values concatenated with a salt value, they will not be able to determine what the other player has chosen. If we did not have the commitments, after the first player had sent a move, the second player could see this move and choose the move that would make them win.

Furthermore, I want to mention that there is a timer (only 5 minutes long) starting after the first player reveals their commitment. The reason being that the second player could abandon the game if they realise they are going to lose. If this happened the game could not be finished and the winner would not be able to retrieve their reward. However, by adding the timer, the winner can finish the game and withdraw its money after 5 minutes if the other player did not reveal their move.

Data type/structure used for the pick options

I decided to create a *Choices* `enum` for storing the possible moves as integers. I found out that `storage` in solidity is very expensive. In the beginning I had different variables for the addresses, commitments and choices of each player. However, I modified it and created a `Player struct` which contains all those attributes. I can keep track of the players by using a `players mapping` from `uint8` to `Player structs`, accessing these by using `players[1]` for player A and `players[2]` for player B.

2 Gas evaluation

2.1 Cost of deploying and interacting with the contract

(I give more details about the cost of interacting with the contract at the end of the coursework, alongside the transaction history)

Deployment of the contract

- Transaction hash: 0x3f355e91c2e804f99ea1a7c6e270f07c1523d5abfd907398d9b472327d4a49b1
- Transaction cost: 1813791 gwei
- Deployment Address: 0x49022a93251D1eaeCd3B20438c38EE42FF9c5f5f

There are several ways the player can interact with the contract. The players can check the values for several public constants (the amount of the bet and the time of each timer set in the contract). There are also public view/pure functions that the players can be used to check how much time is needed for the timers to end, for checking their balances or even knowing what player they are. There is no cost associated with them because they are only reading data from the blockchain to give an intended result.

On the other hand we have functions that change the state of the contract. These can be `payable` or `non-payable`. These functions will create a transaction if we call them and consequently will cost gas. Using the register function in the contract usually costs between 67000 and 69000 gas.

Calling the `play1` and `play2` functions costs almost the same amount, between 44000 and 45000 `gwei`. Nevertheless, the cost for revealing the commitment will be a bit more expensive for the first player to reveal. This is because the first player to reveal their choice starts the 5-minute timer in case the other player does not want to reveal their commitment. The cost for the first player was around 40000 gas units, while the second one had to pay around 25000 units.

Last but not least, interacting with the contract to finish the game costs around 63000 gas units, and withdrawing money from it around 20000 units. An example of a transaction history alongside the gas fees that both players paid can be found here. (I decided to put the cost of interacting the contract with the transaction history to keep it more organised).

2.2 Fairness of the contract

The contract is pretty much fair in the sense that both players have to follow almost the same steps in order to play the game. Both players will pay similar fees for registering, sending their commit to the contract and revealing the values committed (first player to reveal will have to pay a bit more to set the timer). As a consequence, the game is fair in that respect. Nevertheless, there are two actions which might not be completely fair: finishing the game and withdrawing the money:

The game could be finish automatically after both players have revealed their results: I could add a conditional statement in the end of `revealChoiceFromCommitment()` function that will check whether the two players have revealed their choices. If they have, it would call `finishGame()` and the game would be over. The problem here is that the last player to reveal the results will have to pay for more fees when they reveal their choice. Because of this reason, I decided to avoid adding this condition and make public the `finishGame()` function. Once both players have revealed their results (or the timer is over) they can call `finishGame()` to update their balances and reset the game. This is still not completely fair because one of the two players would have to call `finishGame()` and pay fees for the transaction. Even so, I think this is a better solution than making pay more to the last player that revealed their commitment since it does not force a single player to pay more fees and this can be chosen by the players. The way I thought about this is the following: after both players have revealed the results, they will know who is the winner so they could either make the winner call `finishGame()` to pay for the fees or choose any player at random.

The other action that might not be complete fair is the withdrawal of the reward. Once the game is finished and the balances have been updated, the winner will have to pay for the fees of the transaction to withdraw the money. A possible alternative would be to send the

corresponding money to the winner after computing the result of the game. However, it is possible that the player who loses the game is who calls `finishGame()` so this player would be paying even more fees for sending the money to the other player. I think that in this case it is more fair that the winner pays only the corresponding fees for withdrawing the reward. Moreover, having a `withdraw()` function is more secure as I will discuss, so in this case it is a good trade-off to make.

Having considered several options for changing the code, I made the code as fair as possible by trying to make both players do the same transactions so that the fees are equal. Nevertheless, due to the existence of a single winner and having to compute the result, there are a few actions that have to be carried out only once by a single player. I for one think that letting the players choose the winner to pay for the fees of `finishGame()` (or use the `Random` library in `Python` for choosing it randomly); and that making the winner pay for receiving the reward is more reasonable than making the loser pay for it.

2.3 Techniques to make the contract more cost efficient/fair

One of the things that I mentioned above was storing `structs` for each of the two players instead of storing the corresponding data of each one. This change of the `storage` turned out to make a big difference. Moreover, I modified the type of the `TIMEOUT` constants to `uint32` from uint since $2^{32} - 1 = 4294967295$, (i.e. Sun Feb 07, 2106) and `Solidity` will probably not be used for making applications by then. I modified other `uint` types to either `uint32` or `uint64`. I did not use `uint8` because this is a more expensive type than uint due to Solidity having to downscale the number.

Another technique I used to make the contract more cost efficient was to use `bytes` and `integers` instead of `strings` in the contract, for example when players are revealing their choices (using a `Choices enum`).

I have discussed already several techniques I used to make the contract more fair. I would like to stress out again my thought of making *only* the first player commit a value. Nevertheless, I found out that this was pretty unfair for the players and decided that making them both commit to a value would be a better idea. This way, both of them will have to reveal these commitments and the fees they pay will be much more similar.

3 Possible hazards and vulnerabilities

There are several ways the contract can be vulnerable. In this section I will explain how the contract could be found vulnerable and how I avoided them.

3.1 Griefing

To understand what a griefing attack is, it is important to know that a smart contract will call its default function when it receives money without any extradata or function call. This type of attack is based on the error handling when a contract sends money somewhere else using the `send()` function.

A smart contract should check for the failure or success of `send()`, but most of them do not. A send can fail when there is not enough gas to execute the default function at the contract that is receiving the money. Therefore, a `griefing` wallet could have a default function that takes an enormous amount of gas to run. This wallet could lock smart contracts by engaging with them so that when these contracts use `send()`, they fail and get disrupted.

There is a recommendation for the smart Ethereum Developer to `throw` if a call to `send()` fails, but this is still a problem because the `griefing` wallet could just lock the contract by making it throw all the time. Dimitris gave an example in lecture 4 where we have a contract

that keeps an array of investors and adds a new investor if they invest more than the current smallest investment.

This griefing attack will not happen in the contract because I avoided using `send()` and `throw`. I wrote the contract so that this does not have to send the money to the players, but they are the ones who have to withdraw it from the contract. This is known as "pull over push"; my contract using the "pull" design pattern.

3.2 Reentrancy

A *reentrancy* attack is a very powerful one that can be devastating for the contract. This attack can completely drain the smart contract of its ether, and it can also sneak its way into the code. A *reentrancy* attack can occur when one of the functions from the smart contract calls an external (untrusted) source before it resets/finishes/resolves the state of the contract. To give an example, a contract should update the balance of a user before sending a given amount to it; this way the contract prevents the *fallback* function of the receiving address making it give them more money. To generalise this example, if the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the effects were resolved.

This is an attack that will not happen with my code due to the smart contract resetting the values of the players when the game is finished, and updating their balances accordingly before transferring any amount.

3.3 Front-running

Front-running basically consists on placing a transaction with some knowledge of a future one. The attacker can see a transaction and then place another one using this knowledge; but the attacker will do so with more gas so that the miner puts their transaction earlier than the first one.

This is something that does not affect the way matching-pennies is defined. The order of the bets do not matter, and the only problem we can find is having player A copying player B's choice. Since we are making player A make the commitment first, it does not matter if player B places their transaction earlier in the blockchain: player B is not incentivised to copy player A's hash (player B would lose). Moreover, because of the commitment scheme, player B will not have any 'insider' knowledge after seeing player A's commitment.

3.4 Other possible hazards

Some applications might required data to be private during some amount of time and they would have to be careful what source of randomness they use to prevent attackers reveal the data. For my contract we need to keep private the choices and we use a *commitment scheme* for doing so. Another possible problem could be unbounded operations, *underflows* and *overflows*; but these are not vulnerabilities which can be exploited in the contract to win the matching-pennies game.

4 Trade-offs

There are several trade-offs I had to make for writing the smart contract. The fastest, cheapest way for the game would be to send their moves alongside their bets without any commitments, and having the contract computing the winner and sending the rewards after the second player chose a move. This, however, is everything but secure or fair. I decided to make both players register first without making a move so that they will have placed their bets in the contract. Moreover, I chose to use a commitment scheme so that the players will not be able to cheat by looking at each other's bets. This will increase the cost of the contract since they will have to

reveal their commitments, but it is a vital trade-off to make in order to make sure players are not cheating.

A different trade-off which makes the code a bit less efficient has to do with the way the contract sends money to the winner. I chose to make the winner withdraw the money from the contract. By encapsulating the logic this way we increase the security of the contract. We make sure the balance is updated correctly and remove any possible reentrancy attack.

Last but not least, it might be pointed out that the game could be finished at the same time the second player reveals their commitment. I chose not to do it this way so that both players pay for the same fees until the reveal phase, and then they can choose at random who will finish the game (they can also look at each other's choices after the reveal phase and make the winner finish the game). This gives more flexibility on how the gas is distributed so that it can be as fair as possible and not force the loser pay for more fees.

5 Fellow student's contract

5.1 Description

The code of my fellow student had a different approach to mine. She starts by making player A send the commitment alongside the bet to the smart contract, while player B will not have to send any commitment. Player B will send the bet alongside a single `boolean` value (*true* or *false*) depending on whether player B chooses *Heads* or *Tails*. After this, she makes player A reveal their commitment and the contract makes the payment to the winner automatically.

After looking at the code there are several things which could be improved. To begin with, she had an expiration time of 24 hours for player B to get their money back in case player A did not reveal their choice. I thought 24 hours was too much for playing such a simple game and we decided using 10 minutes as the expiration time. Furthermore, I suggested that it would be good to have some logic for player 1 to get their money back if they sent a commitment, but did not find anyone to play with. She added a `cancelBet()` function.

I saw several problems with the contract regarding how fair it was. To begin with, the players have to use different functions for interacting with the game so this means the gas they will pay will not be similar. She based her code on making the game faster, but she agreed with me that it would be better for both players to pay more similar fees. In her code, player A will have not only to pay for fees for revealing their choice, but also for sending money to the winner (even if player A did not win the game). In the end, she decided to encapsulated the code as I suggested so that when player A reveals the choice, the contract does not directly pay the reward to the winner. Nevertheless, she did not make player B commit to any value and therefore she is giving more importance here to efficiency than fairness (player A will have to pay for more fees).

5.2 Vulnerabilities and possible exploits

The code from my fellow student was well structured, but still there were a few things that I spotted for making it more secure. To begin with, the logic was not well encapsulated and there were several functions which did more than what the function's name suggested. For example, `revealChoice(bool choice, uint256 nonce)` was not only revealing the choice from player A, but also sending the money to the winner.

The other problem I found was that she was resetting the addresses - commitments - choices of the players after paying the winner, while it should

```
function revealChoice(bool choice, uint256 nonce) public {
    require(player2 != address(0));
    require(block.timestamp < expiration);

    require(keccak256(abi.encodePacked(choice, nonce)) == bytes32(player1Commitment));

    if (player2Choice != choice) {
        payable(player2).transfer(address(this).balance);
        emit Payout(player2, betAmount*2);
    } else {
        player1.transfer(address(this).balance);
        emit Payout(player1, betAmount*2);
    }
    emit Reveal(player1, choice);

    // Reinitialising values
    player1 = payable(address(0));
    player2 = payable(address(0));
    player1Commitment = bytes32(0);
    player2Choice = bool(false);
    expiration = 2**256-1;
}
```

Figure 1: `revealChoice` does more than revealing

be done before to avoid any *reentrancy* attack. The
way she fixed it was by encapsulating the code as I
suggested: `revealChoice` will not pay to the winner, but it will update the winner's balance. The winner will be able to obtain their corresponding amount by calling a separate `withdraw()` function. Moreover, she made `revealChoice` reset the state variables so that the game can be played safely again.

6 My Smart Contract

Before showing the transaction history and the code of my contract, I want to single out some changes that I made after the other student and me played my game. I realised that the fees being payed by the players were not as equal as I was expecting. Going through the code I realised the following problem:

- I used to have another timer for player A to cancel their bet if they did not find a player B (or player B did not register). I was starting this timer when player A called the `register` function so player A would pay a little bit more fees by changing the state of the variable `startTimer2`. I fixed this by removing that timer and creating a `cancelBet()` function player A could use instead. Now when both players call `register`, they pay almost the same amount of fees.

- The fees players pay when revealing their choices are not the same. First player who reveals their choice will pay more fees because they will be starting a timer. This has been discussed already: if a player does not reveal their commitment for whatever reason (probably because the other player revealed and they realised they lost), the contract will get stuck. I decided not to change this and leave that timer even if the first player to reveal has to pay for more fees. This approach is more fair than having player B not sending a commitment because player B would be paying much less fees in this later case.

I am going to provide the transaction history where I realised the fees are not equal for registering, and then the transaction history for my final code.

6.1 Transaction history

Note: In my code I used 1 and 2 for referring to players A and B. Transaction cost in units of `gwei`.

Deployment

- Transaction hash:
- Transaction cost:
- Address of smart contract:

Player A calls `register()`

- Transaction hash: 0x9ae34fbbf75120353c486963b6bd7296dc59012a5060c9c218c56556be100a45
- Transaction cost: 87704

Player B calls `register()`

- Transaction hash: 0x03134ab6645b53ce485f2052f26b886a053a1c554ecd72563cbfd7820d51f8a2
- Transaction cost: 68655

Player A sends commitment (calls `play1(bytes32 commitment)`)

- Transaction hash: 0xe51e3d95a0e0ffddc5d38bc52500620443e88a2a8511076a391f82ace817804b
- Transaction cost: 44035

Player B sends commitment (calls `play2(bytes32 commitment)`)

- Transaction hash: 0xdd9ea6a0cf9e54981d3f8895ff3d16dbd27362715b7c0d28db918f5013de262f
- Transaction cost: 44958

Player A reveals commitment (calls `revealChoiceFromCommitment`)

- Transaction hash: 0x879915e1002a11c5dea4e6195e763fcb9691f294457923ca3ff4bc0626cd7c12
- Transaction cost: 41582

Player B reveals commitment (calls `revealChoiceFromCommitment`)

- Transaction hash: 0xd5ae0a0c1724aac972583baca7bbda8deafc88be8def557a59879744806745c5
- Transaction cost: 23265

Player A finishes the game (calls `finishGame()`)

- Transaction hash: 0x9bb1f538a4dfe551809fff3a8b1dc153feef5ed5e545e372860967c88723963f
- Transaction cost: 63940

Player A wins and withdraws their money plus reward (calls `withdraw()`)

- Transaction hash: 0x7ae542fe336b2066c0c597638648487d776154899f1816ed9e101548cd6332ac
- Transaction cost: 19761

6.2 Transaction history of updated smart contract

Deployment

- Transaction hash: 0xb13d157de2d40339833369cf14aa870b5860bd78a89623ad3d650e2a039a559b
- Transaction cost: 1819054
- Address of smart contract: 0xA3CE40c7Ca390eE4a0BA822FBD9a693A2138E169

Player A calls `register()`

- Transaction hash: 0x0a9e742bcbbcceacd6163994b90c6c4c1a471c97474577cd81e16415160cf702
- Transaction cost: 67669

Player B calls `register()`

- Transaction hash: 0x295f9bd149e2021e456142c9e4cfd2f6754f03dfc715588a8af81ebc572daec7
- Transaction cost: 68633

Player A sends commitment (calls `play1(bytes32 commitment)`)

- Transaction hash: 0xf0a139c6f39a9b83bbd28e4bd7d34d87b9e8783f8cb8f052a75e79626ce000eb
- Transaction cost: 44013

Player B sends commitment (calls `play2(bytes32 commitment)`)

- Transaction hash: 0x58abbb03335ce7f3834c51315091d245b884bd9588f06ce6687b4cbc223e154c
- Transaction cost: 44936

Player A reveals commitment (calls `revealChoiceFromCommitment`)

- Transaction hash: 0x448dedae47cb1de7aa49dc8f1f641a803b2bce072d55764997d77949581ba0bc
- Transaction cost: 41560

Player B reveals commitment (calls `revealChoiceFromCommitment`)

- Transaction hash: 0xa9b288141f8cf54eb4f64726bb77b41ca10b3bd2608a9ae83d6bb4c68e16c048
- Transaction cost: 23479

Player A finishes the game (calls `finishGame()`)

- Transaction hash: 0x385fac649c7ba51d00f4faec3fbdafbbef4262b7313d9f9788a0ffeb866db09f
- Transaction cost: 60426

Player A wins and withdraws their money plus reward (calls `withdraw()`)

- Transaction hash: 0xdd7f3468f0c8200261974385686308 5bcb76bb058360c59fce73fb015ce4929f
- Transaction cost: 19739

Code

```
// "SPDX-License-Identifier: UNLICENSED"
pragma solidity >=0.7.0 <0.9.0;

contract MatchingPennies {

    uint constant public AMOUNT_TO_BET = 1 ether; // Value of the bet of each player
    uint32 constant public TIMEOUT = 5 minutes; // after TIMEOUT results the game can be finished

    uint private startTime;

    enum Choices { Heads, Tails, NA }
    enum Results { Player1, Player2, Draw }

    struct Player {
        address _address;
        bytes32 commitment;
        Choices choice;
    }

    mapping(uint32 => Player) players;
    mapping(address => uint) private balances;

    modifier isCorrectAmountSent() {
        require(msg.value == AMOUNT_TO_BET);
        _;
    }

    modifier isAddressNotRegistered() {
        require(msg.sender != players[1]._address && msg.sender != players[2]._address);
        _;
    }

    function register() public payable isCorrectAmountSent isAddressNotRegistered returns (string memory) {
        if (players[1]._address == address(0)) {
            Player memory player1 = Player(msg.sender, 0, Choices.NA);
            players[1] = player1;
            return "You have been assigned to Player 1";
        } else if (players[2]._address == address(0)) {
            Player memory player2 = Player(msg.sender, 0, Choices.NA);
            players[2] = player2;
            return "You have been assigned to Player 2";
        } else {
            return "Sorry, there are two players already. You will be able to play when they finish! ;)";
        }
    }

    // Choice is "1" for Heads and "2" for Tails
    // Salt is a random string to compute the hash value. Similar as how passwords are stored in your laptop.
    // e.g. with trueValue = "1" and salt = "HelloWorld!" we are commiting to Heads!
    function computeCommitment(string memory choice, string memory salt) public pure returns (bytes32 commitment) {
        return keccak256(abi.encodePacked(choice, salt));
    }

```

```
54      modifier isPlayer1() {
55          require(msg.sender == players[1]._address);
56          _;
57      }
58
59      // Store player's commitment if he/she is registered and has not played
60      function play1(bytes32 commitment) public isPlayer1 {
61          if (players[1].commitment == 0) {
62              players[1].commitment = commitment;
63          }
64      }
65
66      modifier hasPlayer1Committed() {
67          require(players[1].commitment != 0);
68          _;
69      }
70
71      modifier isPlayer2() {
72          require(msg.sender == players[2]._address);
73          _;
74      }
75
76      function play2(bytes32 commitment) public isPlayer2 hasPlayer1Committed {
77          if (players[2].commitment == 0) {
78              players[2].commitment = commitment;
79          }
80      }
81
82
83      function getChoiceFromDecryptedCommitment(bytes memory decryptedCommitment)
            private pure returns (Choices) {
84          // Get first byte
85          bytes1 firstByte = decryptedCommitment[0];
86          // 0x31 corresponds to char '1' in HEX
87          // 1 corresponds to heads; 2 to Tails
88          if (firstByte == 0x31) {
89              return Choices.Heads;
90          } else if (firstByte == 0x32) {
91              return Choices.Tails;
92          } else {
93              return Choices.NA;
94          }
95      }
96
97      modifier haveAllPlayersCommitted() {
98          require(players[1].commitment != 0 && players[2].commitment != 0);
99          _;
100     }
101
102     modifier isAddressRegistered() {
103         require(msg.sender == players[1]._address || msg.sender == players[2].
                _address);
104         _;
105     }
106
107     function revealChoiceFromCommitment(
108         string memory revealedChoice,
109         string memory salt
110         ) public isAddressRegistered haveAllPlayersCommitted returns (Choices)
                {
111
112         bytes memory decryptedCommitment = abi.encodePacked(revealedChoice,
                salt);
```

```
113         bytes32 commitmentFromInputValue = keccak256(decryptedCommitment);
114         Choices choice = getChoiceFromDecryptedCommitment(decryptedCommitment);
115 
116         if (msg.sender == players[1]._address && commitmentFromInputValue ==
                players[1].commitment) {
117             players[1].choice = choice;
118         } else if (msg.sender == players[2]._address &&
                commitmentFromInputValue == players[2].commitment) {
119             players[2].choice = choice;
120         } else {
121             return Choices.NA;
122         }
123 
124         if (startTime == 0) {
125             startTime = block.timestamp;
126         }
127 
128         return choice;
129     }
130 
131     modifier canComputeResult() {
132         require(
133             (block.timestamp > startTime + TIMEOUT) ||
134             (players[1].choice != Choices.NA && players[2].choice != Choices.NA
                    )
135         );
136         _;
137     }
138 
139     function finishGame() public canComputeResult returns (Results) {
140         Results result;
141 
142         bool player1Wins = players[2].choice == Choices.NA ||
143             players[1].choice == players[2].choice;
144 
145         if (players[1].choice == Choices.NA && players[2].choice == Choices.NA)
                {
146             result = Results.Draw;
147         } else if (player1Wins) {
148             result = Results.Player1;
149         } else {
150             result = Results.Player2;
151         }
152 
153         address payable address1 = payable(players[1]._address);
154         address payable address2 = payable(players[2]._address);
155 
156         reset();
157         updateBalances(address1, address2, result);
158 
159         return result;
160     }
161 
162     function reset() private {
163         startTime = 0;
164         Player memory emptyPlayer = Player(address(0), 0, Choices.NA);
165         players[1] = emptyPlayer;
166         players[2] = emptyPlayer;
167     }
168 
169     function updateBalances(address payable address1, address payable address2,
            Results result) private {
170         if (result == Results.Player1) {
```

```
171             balances[address1] += AMOUNT_TO_BET * 2;
172         } else if (result == Results.Player2) {
173             balances[address2] += AMOUNT_TO_BET * 2;
174         } else {
175             balances[address1] += AMOUNT_TO_BET;
176             balances[address2] += AMOUNT_TO_BET;
177         }
178     }
179
180     function withdraw() public {
181         uint balance = balances[msg.sender];
182         balances[msg.sender] = 0;
183         payable(msg.sender).transfer(balance);
184     }
185
186     modifier canPlayer1CancelBet() {
187         require(msg.sender == players[1]._address && players[2]._address ==
                address(0));
188         _;
189     }
190     function cancelBet() public canPlayer1CancelBet {
191         Player memory emptyPlayer = Player(address(0), 0, Choices.NA);
192         players[1] = emptyPlayer;
193         balances[msg.sender] += AMOUNT_TO_BET;
194     }
195
196     function checkBalanceInContract() public view returns(uint) {
197         return balances[msg.sender];
198     }
199
200     function getContractBalance() public view returns (uint) {
201         return address(this).balance;
202     }
203
204     function whoAmI() public view returns (string memory) {
205         if (msg.sender == players[1]._address) {
206             return "You are player 1";
207         } else if (msg.sender == players[2]._address) {
208             return "You are player 2";
209         } else {
210             return "You are not playing the game";
211         }
212     }
213
214     function timeForTimerToEnd() public view returns (uint) {
215         if (startTime != 0) {
216             uint remainingTime = startTime + TIMEOUT - block.timestamp;
217             if (remainingTime < 0) {
218                 return 0;
219             } else {
220                 return remainingTime;
221             }
222         }
223         return TIMEOUT;
224     }
225
226
227 }
```