School of Informatics
Blockchains and Distributed Ledgers

Assignment 2

Erodotos Demetriou (s2187344)

Game High-Level Description

Matching pennies is a zero-sum game used in game theory. In particular, we have two players who want to bet against each other. In this game, each player will randomly pick a number 0 or 1 (we can think of it as true/false or head/tails). If both players place the same bet, *playerA* will get the reward of 1 ETH; else, if both players choose different numbers *playerB* wis. This is a zero-sum game since the reward of the winner is the loss of the other player. The diagram below illustrates the aforementioned winner/loser scheme we have described.

		Player A Bet	
		0	1
Player B Bet	0	*Player A*	*Player B*
	1	*Player B*	*Player A*

At any moment, only two players can participate in this game. They have a specific time limit to complete their round so other players can play the game later. Each player who wants to join the game has to bet 1 ETH and pay extra transaction gas fees.

It is essential to mention that when a player wants to play, he has to do this in a two-step process. Firstly, he has to obscure his bet. This is because any transaction data are visible on public-permissionless blockchains. For this reason, we use a hash function to locally hash a player's bet concatenated with a random salt that the player chooses.

$$Obscured\ Bet = H(bet||salt), \quad where\ H : Keccak256Hash$$

Using this technique, when *playerA* submits his bet on the Smart Contract, nobody can view his real bet value and thus not cheat on him. Afterward, *playerB* has to place his bet in the same manner.

When this phase completes, players' bets are locked, and they can not make any changes. Subsequently, we move to the second phase of the game, where each player has to provide his real bet and salt to validate their initial bet inputs.

We have to mention that whenever a player interacts with the Smart Contract, an event is emitted to notify the other player to progress. Each player has 10 minutes to respond, else the one who played last can request a refund and cancel the game. The player who did not responded in time, loses his ETH, which are kept by the contract.

Finally, any of the two players can call the Smart Contract evaluate function to calculate the winner. The winner can immediately withdraw his reward from the Smart Contract and reset the game settings allowing another pair of players to compete.

The above process describes the core concept behind the implementation of the game. Furthermore, during the realization of the program, several programming techniques have been used to facilitate the flawless and safe execution of the Smart Contract. We discuss such good programming practices and considerations in the "Potential Hazards and Vulnerabilities" section.

There follows a detailed description of each Smart Contract Variables, Functions, and Events:

Variables

_playDeadline: This is a public uint256 that stores the last time a player interacted with the game.

_playedLast: This is a public uint256 that stores the address of the last player.

_adr_playerA: This is a public uint256 that stores the address of playerA

_adr_playerB: This is a public uint256 that stores the address of playerB

_bets: This is a public mapping which stores a players address and its corresponding Bet. The bet is a struct as shown below

Bet: This is a struct consist of a string which is the real bed, a bytes32 which represents the hidden bet , and a boolean variable that indicates if the bet is valid.

_locked: This is a public uint8 that takes values 0,1. When its value is 1 the Smart Contract is considered locked, and no one else can place new bets.

_playersJoined: This is a public uint8 that stores how many players joined the game on a round. It takes values 0,1,2.

_winner: This variable stores the address of the winner.

Events

event Play: Whenever a player interacts with the Smart Contract to place his bet, a new event is emitted to notify the other player to place his bet. This event displays informatin such as the address of the player who interacted last and his index.

event Winner: Whenever a player wins the game, this event is emitted, so the winning entity knows. This event displays the winner address.

event NewGame: Whenever the Smart Contract is available and can host a new round this event is emitted.

Functions

giveHiddenBet(): This is a payable function. The player provides a bytes32 value representing its bet. This is an obscured representation of it since we want it to be safe so playerB won't cheat. The function requires that it is not locked, that the player has sent 1 ETH, and that the player has not already placed another bet. After these checks are complete, the Smart Contract variables are updated, and the player's bet is final. Eventually, this function emits an event to notify the other player to progress.

giveRealBet(): This external function can be called after both players have placed their hidden bet. It receives as input the real bet and salt of each player and validates and updates their hidden bet. Eventually, this function emits an event to notify the other player to progress.

eveluateWinner(): This external function is called to determine which player won the game. The constraint for this function is that both users' bets have to be validated. Eventually, this function emits an event to notify players about the game outcome.

requestRefund(): This is an external function and can be called by the player who played last, and only if 10 minutes have passed from his previous interaction. Then the gameReset function is called, and one ether is returned to him.

withdraw(): This is an external function that can be called only by the game-winner. If this requirement is satisfied the gameReset() function is called, and two ether are returned to the winner.

gameReset(): This is an internal function which can be called by requestRefund() and withdraw() functions. Its purpose is to reset the Smart Contract variables and prepare the game to host a new pair of players. Finally, this function delivers the requested amount of ether to the proper recipient.

Gas Costs Evaluation

This section measures and evaluates the gas cost we expect to have when deploying and interacting with the Smart Contract.

Gas costs appear in the following table, and they are unquestionably high. The Smart Contract owner has to pay 1,801,345 gas units to deploy the code, which is approximately $995 (price on 26/10/21). This is definitely a high amount of gas which makes the Smart Contract not so gas efficient.

In regards to the players interacting with the Smart Contract, the fees are much lower. Namely, during the initial phase of submitting a hidden bet, *playerA* has to pay 135,643 and *playerB* 84,620 gas units, translating to approximately $61 and $44, respectively. We can notice a considerable difference between the fees paying each player, making this a bit unfair.

For the next step, both players need to call the giveRealBet() function, which will cost them 81,007($42) and 83807 ($43) accordingly. Finally, the evaluation, withdrawal, and refund functions will cost 40,423($20), 43,717($20), 54,648($28) gas units.

Commenting on the above findings, it is apparent that *playerA* has a disadvantage against *playerB* in terms of gas fairness. In total, *playerA* has to pay 216,650 gas units while *playerB*

Gas Costs

Contract Owner Fees	
Contract Deployment	1,798,945

Players Fees	***Player A***	***Player B***
giveHiddenBet()	135,643	84,620
giveRealBet()	81,007	83,807
evaluate()	40,423	
withdraw()	43,717	
requestRefund()	54,648	

168,427 gas units, occurring a difference of 48,223. To mitigate this imbalance, we propose that *playerB* call the evaluateWinner() function narrowing the gap to 7800 gas units. Regarding refund requests and withdrawals, it is more than fair that the transaction invoker pays the proportional fee for himself.

Potentially, we could argue that the Smart Contract can be improved in terms of gas efficiency. Such improvements would be to use a different structure to store the players' bets. Using a string in the Bet struct is not ideal, but the validation process could not be achieved using bytes32 data type. There was an issue when it came to concatenating the real bet and salt.

Potential Hazards and Vulnerablities

Developing a Smart Contract for the Ethereum Network can always be challenging. This is because, on a public-permissionless blockchain, everything is observable by everyone. This fact makes it difficult when it comes to securing users' data. Besides that, a developer has to be alert to write code that is attack-resistant. Ethereum Blockchain and specifically Smart Contracts expose a broad spectrum of vulnerabilities, enabling an adversarial entity to exploit them for its interest.

When developing the Matching Pennies game, we took into consideration possible attacks. The following list presents vulnerabilities and mechanisms to moderate them.

DoS(Denial of Service) - Griefing: An attacker attempts to make a Smart Contract get stuck when executed. In the case of the Matching Pennies game, a player might grieve and stop playing to halt the Smart Contract or make his opponent lose money. This is not wanted since the other player's ETH will get stuck, and no one else will be able to play the game.

Mitigation: To counter this type of attack, we implemented a time limit mechanism. When a player interacts with the Smart Contract, a timer is initiated. The other player has only 10 minutes to play his move. If the time limit expires, the last player can request a refund and cancel the game. The funds of the misbehaved player will be kept as punishment. If a single player tries to halt the program (i.e., only one player joins the game), the Smart Contract owner can join, to force the first player to play. If the first player refuses to play, he will lose his money since the contract owner can request a refund, which will reset the game. Using this technique, the Smart Contract can never halt.

DoS(Denial of Service) - Pull Vs. Push: When sending the reward to the winner, there are several ways to do it. If we decide to send the reward to the player, he might have a malicious fallback function that will halt the transaction on a loop and make the Smart Contract run out of gas.

Mitigation: It is a good practice to use a pull mechanism where each player has to initiate a withdrawal. As a result, each external transaction is isolated and reduces any problem with gas limits since the Smart Contract balance will not be affected. The user who initiates the transaction will have to pay for it.

Re-Entrancy: An attacker might try to take advantage of the Smart Contract withdraw function by executing a reentrancy attack. In more detail, when he runs a withdrawal, he can lead his transaction to a malicious fallback function on another Smart Contract that can recursively call again the withdraw function, trying to get more ETH.

Mitigation: In order to avoid such unpleasant attacks, we execute the code of the Smart Contract in a particular way. When there is a withdrawal invocation, we check some constraints to ensure that the transaction sender can withdraw ETH. After that, we will make any changes to the state of the Smart Contract and eventually make the call that sends the requested ETH to the recipient. It is important to make the ETH transfer after changing the Smart Contract state. If a reentrancy attack occurs on the next transaction, it will be stopped because checks will evaluate the transaction according to the previously updated state.

Front-Running: This attack happens on the Miner level. An attacker might clone your transaction and put a much higher gas limit on it. This results in the inclusion of his transaction to the next block instead of yours. This is inconvenient since another player might steal your spot in the game.

Mitigation: There is no straightforward solution since the problem lies at the transaction mining level. For the Matching Pennies game, front-running will not have a significant effect, except that a player might steal the position of another one. The disfavored player will have the opportunity to play in another moment.

Overflow/Underflow: Matching Pennies game does not face this problem since it does not receive any integer value from the transaction sender.

Randomness Source Exposure: Matching Pennies game does not face this problem since it does not use any random value during the code execution.

Delegation: Matching Pennies game does not face this problem since it does not use code from other Smart Contracts or Libraries.

Other good practices:

- Use call() instead of transfer() or send(). Using call() might be insecure, but transfer() and send() can forward only 2300 gas. In the future gas costs might change, and 2300 gas might not be enough. Consequently, we must apply some checks when using the call() function to secure our transactions. View the example below.

```
1         (bool success, ) = msg.sender.call.value(amount)("");
2         require(success, "Transfer failed.");
```

- If you need to have a callback function, keep it simple.
- Any public-permissionless blockchain reveals transaction details to the ledger. Due to this fact, in data-sensitive applications such as the Matching Pennies game, we need to hide users' data. We can do this by following a two-phase process: committing a value obscured by hashing the bet and some salt and then exposing the real value.

Security vs Performance

- security vs performance trade-offs

Fellow Student Contract Analysis

In this section, you find the review of a Smart Contract a fellow student implemented.

Vulnerabilities :

1. Used *trasnfer()* function to transfer rewards or refunds.
2. Update the Smart Contract state after transfering funds.
3. A player is allowed to play the game by himself.
4. The *claimReward()* function can be called by both players.

Exploitation :

1. The Smart Contract developer took the decision to use *trasnfer()* function instead of *call()*. From one point of view this implementation logical if we consider that *trasnfer()* was introduced after the DAO attack to mitigate Re-Entrancy attack. However, *trasnfer()* does not allow to forward more than 2300 gas. This is good only if the gas costs wont change. In the future, if gas costs change the Smart Contract might be problematic.
2. When someone trie to claim a reward or a refund there are some checks, then funds are transfered and eventually the Smart Contract variables are updated. In the example below an adversarial player1 might try to call *gameReset()* in line 1 before any bet is revealed. So, he will validly pass the *require* statements and procced to the line 11. From this point he will be able to perform a Re-Entrancy attack by calling again *gameReset()* and since the Smart Contract variables have not been updated yet, the attacker will pass any check and be able to withdraw 2 ETH instead of 1 that is allowed.

```
1  function resetGame () public { // allows people to play again, but doesn
       't deal with deposited ETH
2      require (msg.sender == player1 || msg.sender == player2, "You are
           not a player in this game" );
3
4      // if both players joined the game and one of them have revealed
           their choice, then the reset is not allowed
5      if (player1 != address(0) && player2 != address(0)) {
```

```
6           require ( !bets[player1].isValid && !bets[player2].isValid, "A
                player has already revealed choice, so you can no longer
                reset, game needs to be finished" );
7       }
8       // refund each player
9       if (player1 != address(0) ) {
10          require ( !bets[player1].isValid, "Player 1 has already revealed
                choice, so you can no longer reset, game needs to be
                finished" );
11          refundBet(payable(player1));
12      }
13      if ( player2 != address(0)) {
14          require ( !bets[player2].isValid, "Player 2 has already revealed
                choice, so you can no longer reset, game needs to be
                finished" );
15          refundBet(payable(player2));
16      }
17
18      choices[player1] = false; // these two lines allow each player to
            play again possibly
19      choices[player2] = false;
20
21      player1 = address(0); // allows another two players to play
22      player2 = address(0);
23  }
```

3. The particular Smart Contract enables a player to play the game by himself. As a result, an adversarial entity that wants to stuck the code forever, can place a bet and never reveal it. (Grief) Consequently, the game will be locked and unavailable to other players.

4. The *claimReward()* function can be called by both players. As you can observe in the following figure, claim reward can be called by both players at any time. According to lines 6 - 9 the actual winner is the one who calls the function first. As a result, the Smart Contract is vulnerable to Front-Running attacks. For instance, the loser of the game, might call the *claimReward()* function using more gas and steam the money from the winner.

```
1   function claimReward() external {
2       require (msg.sender == player1 || msg.sender == player2, "You are
            not a player in this game" );
3       require ( bets[player1].isValid, "Player 1 did not reveal a valid
            bet" );
4       require ( bets[player2].isValid, "Player 2 did not reveal a valid
            bet" );
5
6       if (choices[player1] == choices[player2]) {
7           winner = payable(msg.sender);
8           transferReward(); // calls on a separate function to transfer
                the 2 Eth
9       }
10      else {
11          winner = payable(bets[msg.sender].opponent);
12          transferReward();
13      }
14
15      // reset the game for the next two players
16
17      choices[player1] = false;
```

```
18      choices[player2] = false;
19
20      player1 = address(0);
21      player2 = address(0);
22  }
```

Smart Contract Execution History

Owner Address: 0x79BE6e946368520419BF4A20aC45b28fd3a5b2bA
Player A Address: 0xE083f2644739ef27EC126e42288253F0b9AdFB85
Player B Address: 0xd29Fd58d75aE640415D23166802EA3bd66Ddfd04

Player A Balance: 2 ETH
Player B Balance: 2 ETH

- **Phase 0 - Smart Contract deployment**

  Deployment tx: 0x66014298bc638b7a72b29ef643c15960910470bed2918bcceaeec0bfdeda76ae
  Contract Address: 0xF87a42464eEf144fF0C81cE8d5E927548CF4695D

- **Phase 1 - Submit hidden value**

  Player A input: 0x3eb0fa86b29ff88ffdd4458cd1f554dd6ad43237a86e38c862ab6c440a387964
  Player A tx: 0x742e2b4bd030a2e1130fae0f5f9ff413407fd0502af2a1106dec3d9461453e15

  Player B input: 0xf7f905159c4867bf40ccc7667b940bf77402f6daddc3055b3b2256cb0a291365
  Player B tx: 0xfe3bdff40b02b116befa96647f6b31bf8a80c212aeeeda5b7b20774c6994b981

- **Phase 2 - Submit real values**

  Player A input: 0, 123
  Player A tx: 0xb497dd616183eeccc8eba62c2989e5d4661e4af39b28bf9b43b63b2114fc1742

  Player B input: 0, 234
  Player B tx: 0xba1edb21440455e86c75a59f74205cd6874c7c993f2a866e3ce198b8232ceb9e

- **Phase 3 - Calculating Winner**

  Evaluation tx: 0x623271aba3304cb1a4983979ae5521a3edb0989dba845982e9ea38d36be9bdfa

- **Phase 4 - Winner withdrawal**

  Withdrawal tx: 0x91686c19055f98eb6b2450e0804b537dc6f62ce3b2ad843fcf47b71a0f4cdf32

Player A Balance: 2.9997 ETH
Player B Balance: 0.9998 ETH

Implementation Code

```
pragma solidity 0.8.0;

/// @title Matching pennies game
/// @author Erodotos Demetriou
contract Game {
    uint256 public _playDeadline;
    address public _playedLast;
    address public _adr_playerA;
    address public _adr_playerB;

    mapping(address => Bet) public _bets;

    struct Bet {
        string _realBet;
        bytes32 _hiddenBet;
        bool _isValid;
    }

    uint8 public _locked = 0;
    uint8 public _playersJoined = 0;
    address public _winner;

    event Play(address indexed _playerAddress, uint8 _playerNumber);
    event WinnerAnnounced(address indexed _winner);
    event NewGame(string _newGame);

    /// @notice Takes 1 ETH as bet stake and set contract state accordingly
    /// @param _bet This is an obscured 32-byte string produced after
    /// hashing (real_bet || salt)
    function giveHiddenBet(bytes32 _bet) public payable {
        // Perform checks
        require(
            _locked == 0,
            "There are already 2 players. Wait for the next game to start!"
        );
        require(msg.value == 1 ether, "You must bet 1 ETH");
        require(
            _bets[msg.sender]._hiddenBet == bytes32(0),
            "You have already put your bet"
        );

        // Change the smart contract state
        _playersJoined += 1;
        _bets[msg.sender]._hiddenBet = _bet;
        _playDeadline = block.timestamp + 10 minutes;
        _playedLast = msg.sender;

        // Lock the contract if both players beted
        // and emmit events to announce their participation
        if (_playersJoined == 2) {
            _locked = 1;
            _adr_playerB = msg.sender;
```

```
53              emit Play(msg.sender, 2);
54          } else {
55              _adr_playerA = msg.sender;
56              emit Play(msg.sender, 1);
57          }
58      }
59
60      /// @notice Receives the players real bets
61      /// and their salt and check the initial bet validity
62      /// @param _realBet A string representing the real bet
63      /// @param _salt The salt that the message sender used
64      /// to create his initial obscured bet
65      function giveRealBet(string memory _realBet, string memory _salt)
            external {
66          require(_playersJoined == 2, "Wait for player #2 to join the game");
67          require(
68              keccak256(abi.encodePacked(_realBet, _salt)) ==
69                  _bets[msg.sender]._hiddenBet,
70              "Error: Provided invalid input: Abort"
71          );
72
73          _bets[msg.sender]._realBet = _realBet;
74          _bets[msg.sender]._isValid = true;
75
76          _playedLast = msg.sender;
77          _playDeadline = block.timestamp + 10 minutes;
78      }
79
80      /// @notice Calculates the game winner
81      function evaluateWinner() external {
82          require(
83              _bets[_adr_playerA]._isValid && _bets[_adr_playerB]._isValid,
84              "Error: Players did not provide their real bet"
85          );
86
87          if (
88              keccak256(abi.encode(_bets[_adr_playerA]._realBet)) ==
89              keccak256(abi.encode(_bets[_adr_playerB]._realBet))
90          ) {
91              _winner = _adr_playerA;
92          } else if (
93              keccak256(abi.encode(_bets[_adr_playerA]._realBet)) !=
94              keccak256(abi.encode(_bets[_adr_playerB]._realBet))
95          ) {
96              _winner = _adr_playerB;
97          }
98
99          // Emit event
100         emit WinnerAnnounced(_winner);
101     }
102
103     /// @notice Let a player to stop the game and get
104     /// refund in case his opponent griefs
105     function requestRefund() external {
106         // Checks
107         require(
108             block.timestamp > _playDeadline &&
109                 msg.sender == _playedLast &&
110                 _winner == address(0),
111             "You are not allowed  to request a refund yet!"
```

```
        );

        gameReset(1 ether);
    }

    /// @notice Allows the winner to withdraw his reward
    function withdraw() external {
        // Checks
        require(msg.sender == _winner, "You are not the winner!");

        gameReset(2 ether);
    }

    /// @notice Send money to the winner or the
    /// refund requestor and reset game variables for a new round
    function gameReset(uint256 _value) internal {
        _locked = 0;
        _winner = address(0);
        _playersJoined = 0;
        _bets[_adr_playerA] = Bet("", bytes32(0), false);
        _bets[_adr_playerB] = Bet("", bytes32(0), false);
        _adr_playerA = address(0);
        _adr_playerB = address(0);
        _playDeadline = 0;

        // Reward/Refund transfer
        (bool success, ) = msg.sender.call{value: _value}("");
        require(success, "Error: Withdraw unsuccessful");

        // Emmit event
        emit NewGame("New game spots available");
    }
}
```