BDL Assignment 2

Kareem Hussein - s2258311

October 2021

1 Introduction

The matching pennies game is a classical game in game theory played by 2 players, each hypothetically starting out with a penny. Each player flips his penny, and the outcomes are compared. If the outcomes are the same - both heads or both tails - the first player wins; otherwise the second player wins. Assuming no prior knowledge, the game's outcome is entirely probabilistic, where players have equal probabilities of winning due to the equality of the probabilities of the two outcomes [1]. This report describes a smart contract-based implementation of this game

Smart contracts are programs which automatically execute and document the terms of a protocol over the blockchain, a distributed peer-to-peer public database [2]. The described protocol can take a variety of forms, including that of a game that can be played between different parties over the network. This is particularly useful for a game such as matching pennies since transactions are secure and irreversible, therefore providing an advantage in integrity as long as the code is securely implemented.

2 Contract Design

The implemented contract describes a matching pennies game played by 2 different ethereum accounts over the blockchain, where each account wagers 1 ether to join the game, and the winner wins the loser's ether. The game's execution 'phases' follow the steps of a cryptographic commitment scheme, where players must first commit encrypted representations of their choices before revealing them to each other afterwards - only after further commits are blocked [3].

Following the structure described above, players must first join the game by calling the "joinGame" function, passing their commitment in a transaction with a 1 ether value. After both players join, players can then call the "revealValue" function to reveal their respective values, where the contract makes sure that the hash of the revealed value matches up with the commitment made by the same player earlier. It also ensures that the commitment value passed in the reveal phase matches up with that originally committed by that player.

After both players successfully reveal their votes, another function - "calculateWinner" - can then be called to calculate the winner, whose winnings are added to a mapping representing balances held by different addresses. The winner can then call a function to withdraw their winnings afterwards, a choice which follows the "pull over push" design pattern, and prevents possible gas issues that could arise from making a transfer during a wider transaction.

The primary motivating factor behind the design choices outlined above is the prevention of cheating strategies by adversarial players, which is achieved in a variety of methods. Firstly, the usage of a commitment scheme ensures the confidentiality of the committed values during the commitment phase. Afterwards, the outcome of the game cannot be altered, since further commitments are blocked. Moreover, a value can only be revealed by the account that made the respective commitment, thus preventing manipulation by malicious parties.

Furthermore, rather than restricting the number of valid options to a binary "head" or "tails" - thus invalidating the use of a commitment scheme - players can use any string for the value. This prevents adversaries from comparing the commitments made by other players with a list of what would otherwise be the only two possible hashes. The value's length is then used for the comparison, where the even lengths are "heads" and odd lengths are "tails".

The motivation behind that choice described above over a more obvious option - such as using an integer value - was partially motivated by usability. It was noted that if a user were to externally hash their value using an online tool to come up with their commitment value, it would produce the hash of a string representation of the passed in data, even if it was numerical. Furthermore, the use of even and odd lengths allows the program to accept any input string from the user, prevent a whole class of errors that could arise from accepting a commitment hash of what would later turn out to be an illegal string.

Finally, the contract also contains a function, "resetFailedGameState", for handling resetting the state in order to start a new game in case both players attempt to grief, a scenario described further in section 4. This resets the game-specific storage variables kept in the contract, which are a mapping of addresses to their commitments, an array of player addresses, and an array of vote structs. Vote structs keep track of the revealed values, where each "Vote" contains the length of the value string as well as the player who committed that value. This state is reset by first iterating over the mappings kept between player addresses and their commitments and setting the respective values to zero, and then resetting the player addresses array, votes array and revealPhaseEndTime using the "delete" keyword.

3 Gas Evaluation

This section describes an evaluation of the contract's gas consumption, efficiency and fairness. Table 1 shows the gas consumption of the transactions representing the different steps in the program in conjunction with the execution steps shown in table 2. The steps are both ordered by the chronological order of a set of

Step	Gas Consumed
Contract Creation	1787922
Player1 Commit	88290
Player2 Commit	96196
Player2 Reveal	92463
Player1 Reveal	81306
Player1 calculateWinner	115410
Player1 withdrawBalance	36991

Table 1: Execution steps, gas consumption and the charged parties

transactions made on a deployed instance of the contract where the game was played.

As seen in the table above, the gas efficiency of the contract was unfortunately not prioritised during the development process, with the focus instead being on contract security. This is particularly reflected in the high deployment cost, which is raised due to the need to initialise multiple functions during the contract's construction. Furthermore, it is exacerbated by the amount of security checks taking place at the start of each function, making it more expensive for all involved parties.

In the case of gas fairness, while table 1 shows a sizeable discrepancy between the gas paid by players 1 and 2 over the commit and reveal phases, this is inflated by the fact that player2 was both the second to commit yet first to reveal in this run. If the same order were to be maintained in the commit and reveal phase, the only major source of discrepancy would be the calculateWinner function.

Before addressing that function, however, it is interesting to explore the reason behind the apparent balance in gas usages. As seen in section 7, the gas consumption is higher during the commit phase for the second player due to the need to set a value for the reveal phase end time at that stage then emit a log marking the start of the reveal phase. As for the reveal phase, it is likely more expensive for the first player due to the addition of a first element to a new array - particularly an array of structs.

With regards to the "calculateWinner" function, however, the current approach definitely highlights a shortcoming in the contract's gas fairness, since while it is not forced on any of the players - and indeed it is left up to the players to decide whether they want to incur this gas just to find out who won -, it is often unfair to the person that calls that function, especially when that person is not the winner.

Recognising the issues outlined above, some suggestions for improving them are therefore suggested. It was noted that with the large amount of checks and forks in logic currently in place in the core functions to handle situations, it might be useful to analyse the ones currently in place and check for redundancies to simplify.

Furthermore, it might be better to split some of the functions up into smaller functions for handling different scenarios. A notable example is the calculateWinner() function, which contains code for handling scenarios for an honest game, a game where 1 player griefed and a game where both griefed. Having this function only handle the scenario where none of the players griefed would likely reduce its gas consumption by a considerable margin, which would also benefit the contract's position with respect to gas fairness.

Finally, while the gas consumption of the resetFailedGameState function was not tested on the testnet, perhaps this gas could be refunded to the account that calls the function, especially considering that this call is made after the contract's account balance is increased by 2 ether due to both players getting punished for griefing.

4 Security Considerations

Listed below are some of the security considerations and related decisions made during the development process as well as any related and other trade-offs.

It was first noted that a malicious party could grief by joining a game then never revealing their committed value. To prevent this action from keeping the game permanently suspended, a timer was implemented such that when the second player commits their value, a reveal phase end time is set for one day after that block's timestamp. Consequently, calculateWinner() can be called even when the number of votes is less than 2, as long as the end timestamp has passed. If only 1 player revealed their value by that point, that player automatically wins the game and 2 ether get added to his balance in the contact, which ensures that the griefing player is penalised for their attempt. If both players attempted to grief, the "resetFailedGameState" function can be called by a player willing to play another game after the end timestamp.

However, a vulnerability in this contract's security ultimately lies in the fact that if a malicious miner were to participate in the game and withhold the other player's transaction from being mined into the blockchain, this automatically provides them with a method to defraud all of their opponents. This concern motivated the choice of prolonging the reveal phase time period, since if this contract were to be deployed to the ethereum network, it would be difficult for a singular miner to hold onto a transaction for a whole day due to the network's short block time [4].

It was also noted that in a commitment scheme such as this one, there is a reasonably large attack surface for front-running attacks unless some precautions were taken. To solve this, the "even" player is set to be the first player that commits a value, rather than the first to reveal.

These decisions are made to first ensure an adversarial player cannot front-run the other player's transaction, disguising it as theirs. Furthermore, they prevent a scenario where the attacker attempts to exploit being placed as the first player by revealing first. In this scenario, the attacker would commit the same hash as the other player, then when the other player attempts to reveal their value, the attacker would front-run a transaction with the same parameters, automatically giving themselves the win.

Attempts at re-entrancy attacks were also considered, and a couple of decisions were made. The checks-effects-interactions pattern , and the "transfer" function is used instead of the "call" function in the withdrawBalance() function to ensure that there is a limited amount of gas for the fallback function of the person calling the function.

Finally, a local implementation of the SafeMath add function was made and used wherever addition were made in the contract to prevent integer overflows.

5 Tradeoffs

The design decisions outlined above in sections 2 and 4 mainly emphasised security over other considerations, and trade-offs were therefore made with respect to user experience, gas fairness and efficiency.

This was particularly apparent with the user experience, where users have to hash their strings externally before committing them, then having to go through multiple iterations of waiting and calling different functions before being able to collect their winnings. This is caused by the use of the commitment scheme and the pull design pattern, among other choices.

6 Security Analysis of a Different Contract

In order to provide wider context around the statements made above, an analysis of a peer's deployed contract was also conducted in order to compare and contrast the different design decisions. In this case, the analysed contract is deployed on the testnet at address 0x86557b1e51ED27803D04EBE1be948Ff159AcD1c6.

A quick look at the contract's code shows that while there was a larger emphasis on the number of implemented features, such as the addition of support for multiple simultaneous games, many of the underlying design patterns are also shared.

At a high-level, the analysed contract also uses a commitment scheme to dictate the flow of the game. However, there is a slight modification in high-level flow in comparison to the contract analysed elsewhere in this report. Rather than having each player commit his value while joining the game, the first player to initiate a game instead only sends the required ether, and then receives a game id which he needs to share with the other party in order for them to join. The second player has to then call the acceptGame function, passing in the same id, in order to join that game. Players can then begin to commit their values afterwards. The code for the two described functions is shown below:

Past this point, individual games' execution flows are largely similar, with a similar amount of checks on the address calling each of the value commitment and value revealing functions along with checks on the reveal value matching up with the correct commitment. A minor difference here is that unlike my contract, this contract does not require the sending of the commitment alongside

```
function startNewGame() public payable msgValue() returns (uint256 gameId) {
    require(!maxGamesReached, errors[0]);
    uint256 id = numGames;
    if (numGames + 1 < numGames) {
        maxGamesReached = true;
    } else {
        numGames++;
    }

    Game memory newGame;
    newGame.isGame = true;
    newGame.status = GameStatus.Initiated;
    newGame.player1 = msg.sender;
    newGame.gameBalance += msg.value;
    games[id] = newGame;
    emit StatusUpdate(id, newGame.status);
    return id;
}

function acceptGame(uint256 _gameId) public payable msgValue() isGame(_gameId) gStatus(_gameId, GameStatus.Initiated, errors[5]) {
    Game storage game = games[_gameId];
    assert(game.gameBalance + msg.value > game.gameBalance);
    require(msg.sender != game.player1, errors[12]);

    game.player2 = msg.sender;
    game.gameBalance += msg.value;
    game.status = GameStatus.CommitPhase;
    emit StatusUpdate(_gameId, game.status);
}
```

Figure 1: startNewGame() and acceptGame()

the revealed value in the reveal function, instead sufficing with checking the reveal against the stored commitment. This is shown below.

```
function revealChoice(uint256 _gameId, string memory _choiceStr) public isGame(_gameId) isPlayer(_gameId) gStatus(_gameId, GameStatus.RevealPhase, errors[4]) {
    Game storage game = games[_gameId];
    uint8 playerId = msg.sender == game.player1 ? 0 : 1;
    require(!game.revealed[playerId], errors[7]);
    require(game.choiceCommits[playerId] == keccak256(abi.encodePacked(_choiceStr)), errors[8]);

    bool pChoice = bytes(_choiceStr).length % 2 == 0;
    game.choices[playerId] = pChoice;
    game.revealed[playerId] = true;
    if (game.revealed[0] && game.revealed[1]) {
        game.status = GameStatus.AwaitingWinner;
    }

    emit ChoiceRevealed(_gameId, playerId + 1, pChoice);
    emit StatusUpdate(_gameId, game.status);
}
```

Figure 2: revealChoice()

Overall, the utilisation of a well-checked commitment scheme in the code means that a winning strategy for an adversarial party to be able to cheat in the game was not found. However, two ways were identified through which the contract could be griefed.

Due to a lack of time checking throughout the process of the game, the only way a game could be terminated after 2 players have joined (i.e. after the second player commits his 1 ether by calling joinGame()) is for the game to be completed by both players. Therefore, a malicious party could simply create a game, send the id to a victim then never play. In this case, both the attacker and the victim's 1 ether are locked up in the game forever. It must be noted, however, that due to the support for multiple simultaneous games, this would not hold up the contract, and therefore the presence of this feature mitigates the impact this move otherwise could have had.

Apart from the method described above, a malicious miner could also force a victim to pay gas for a cancelled game by creating a game and sending its id, then when a victim sends a transaction to join the game, the attacker front-runs that transaction with a transaction of their own where the game gets cancelled, which would cause the victim's transaction to fail. However, the gas impact of such a move would not be too drastic, since the function would simply revert early on in the execution and only charge the victim for the gas consumed so far.

7 Transaction History & Code

An instance of the contract described in sections 2 to 5 has been deployed to the BDL private chain at address 0x7Ec10570D33F519e558413119E55F1650E2CdCD6. This section contains the code in that contract, as well as a log of transactions produced by 2 accounts playing the game.

```
// SPDX-License-Identifier: AFL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract MatchingPennies {

    event LogWinnerFound(string, address);
    event PlayerJoined(string);
    event CommitPhaseOver(string);
    event RevealPhaseOver(string);
    event InvalidFunctionCall(string);

    struct Vote {
        address player;
        uint vote;
    }

    mapping(address => uint) private balances;

    mapping(address => bytes32) private commitments;
    address[] private players;
    Vote[] private votes;
    uint private revealPhaseEndTime;

    fallback() external {
        emit InvalidFunctionCall("An invalid function was called in
             this contract! Fallback triggered instead.");
    }

    function resetFailedGameState() public {
        require(players.length == 2, "This function can only be
            called in case of a failed game, but the current game
            still hasn't started!");
        require(votes.length == 0, "This function can only be
            called in case of a failed game, but this game has a
            winner! Please call calculateWinner() instead.");
```

```
        require(revealPhaseEndTime != 0, "This function can only be called in case of a failed game, but there is no game currently in progress!");
        require(block.timestamp > revealPhaseEndTime, "This function can only be called in case of a failed game, but the current game is still in progress!");

        resetState();
    }

    function joinGame(bytes32 commitment) public payable {
        //  either we are in a clean slate new game or we are resetting a previously full game only AFTER the game time ran out.
        require(players.length < 2, "Game already full! Please join when this round is over!");
        require(commitments[msg.sender] == 0, "Can't play the game against yourself!");
        require(msg.value == 1 ether, "Amount sent is NOT 1 ether!");

        commitments[msg.sender] = commitment;
        players.push(msg.sender);
        emit PlayerJoined("A new player has joined the game");

        if (players.length == 2) {
            revealPhaseEndTime = add(block.timestamp, 1 days);
            emit CommitPhaseOver("The commit phase is now order. Registered players can begin revealing their values. The reveal phase will end in 1 hour.");
        }
    }

    function revealValue(bytes32 _commitment, string memory _vote) public {
        require(players.length == 2, "Can't reveal values before both players have joined!");
        require(commitments[msg.sender] != 0, "Only players who have previously joined the game can call this function!");
        require(commitments[msg.sender] == _commitment, "Invalid parameters!");
        require(commitments[msg.sender] == keccak256(abi.encodePacked(_vote)), "Invalid parameters!");
        require(votes.length < 2, "Both votes in this round already revealed!");
        require(block.timestamp < revealPhaseEndTime, "Can't reveal a value after the reveal phase has ended!");

        if (votes.length == 1) {
            require(votes[0].player != msg.sender, "You have already revealed a value, can't do that again!");
        }

        votes.push(Vote(msg.sender, bytes(_vote).length));

```

```
        if (votes.length == 2) emit RevealPhaseOver("Reveal phase over. The winner can now be calculated");
    }

    function calculateWinner() public {
        require(revealPhaseEndTime != 0, "Can't call calculateWinner() unless both players have joined the game!");
        require(votes.length == 2 || block.timestamp > revealPhaseEndTime, "Not all votes have been revealed!");

        address winner;
        bool winnerFound = true;

        //  if nobody griefed and the game ended as expected.
        if (votes.length == 2) {
            if (votes[0].vote % 2 == votes[1].vote % 2) {
                winner = players[0];
            } else {
                winner = players[1];
            }
        //  if one player griefed, reward the other player with an auto win
        } else if (votes.length == 1) {
            winner = votes[0].player;
        //  if both players griefed, both get punished. Nobody gets their ether back.
        //  need to keep this condition in case a different person calls this function
        } else {
            winnerFound = false;
        }

        resetState();

        if (winnerFound) {
            emit LogWinnerFound("A winner has been found!", winner);

            //  using safemaths-esque add to support solidity version 0.7 overlflow safety
            balances[winner] = add(balances[winner],2 ether);
        }

    }

    function withdrawBalance() public {
        uint balance = balances[msg.sender];

        balances[msg.sender] = 0;

        payable(msg.sender).transfer(balance);
    }

    function resetState() private {
        commitments[players[0]] = 0;
```

```
        commitments[players[1]] = 0;

        delete players;
        delete votes;
        delete revealPhaseEndTime;
    }

    function add(uint a, uint b) private pure returns (uint) {
        uint c = a + b;
        require(c >= a);
        return c;
    }
}
```

Listing 1: Smart Contract Code

Step	Transaction Hash
Contract Creation	0x6e78622de008716ba92bc32dce52d9da065be31b4dd6f9770db78c4b0e564a3c
Player1 Commit	0x346845d6222ddcab3b5fe6a86050c1a87c087144cf60d8d0d074120b26721105
Player2 Commit	0xfff0677c62988ffc643009924ea26d5566ea7c96a7979ab5dfa404b7be0b0082
Player2 Reveal	0x8890962c60022c8a361c8b86302181e2f0c268f103a890119c73320f02f710af
Player1 Reveal	0x04e8c63e6421de47cd8c3a84dc0253618387bce8c6e1ea1f586b4e29e12a40e5
Player1 calculateWinner	0x1db5db4268195a2f2402098fd70bc3c2077b359fc59abfaed0d873a3e28b83cb
Player1 WithdrawBalance	0xe667283777d12a77f74abdb0d0668bef68b72985e38a0a8aa7dec4f7d444543d

Table 2: Transaction History for a Matching Pennies Game

References

[1] R. S. Gibbons, *Game Theory for Applied Economists*. Princeton University Press, 1992. [Online]. Available: https://doi.org/10.1515/9781400835881

[2] V. Buterin, "A next-generation smart contract and decentralized application platform," 2015.

[3] O. Goldreich, *Foundations of Cryptography: Basic Tools*. USA: Cambridge University Press, 2000.

[4] "Ethereum average block time chart," Oct 2021. [Online]. Available: https://etherscan.io/chart/blocktime