Blockchains & Distributed Ledgers CW1

B178937

November 2023

1 Introduction

This smart contract facilitates a straightforward game between two participants, enabling them to each select a number within a specified range. The chosen numbers are then added, and if the sum is even, Player A emerges victorious; otherwise, Player B claims the win.

2 High level design

This contract was designed to be used by a secure front end application, which would make it more user friendly than simply interacting with the contract. We break down the high-level design decisions made into subcategories: functional decisions, transactional decisions, security design decisions, and data structure choice.

2.1 Functional Decisions

In determining the specific functional requirements for the smart contract implementation, the challenge of balancing gas efficiency, security, and simplicity presented itself. The overarching functional requirements outlined in the assignment were clear, but the devil was in the details of implementation.

Two distinct approaches emerged: one emphasizing streamlined functionality for optimal gas efficiency and security, and the other incorporating more intricate features at the cost of increased gas expenditure and potential vulnerabilities. The first approach adheres to a straightforward model: two users join the game by paying the minimum entry fee, select their numbers, and witness their balances being updated and can withdraw their winnings. This swift process then readies the contract for the next pair of players.

The alternative approach introduces complexities. After players submit their numbers and witness balance adjustments based on the game outcome, a "play again" button becomes available. Both players must press this button or withdraw their winnings which resets the contract for subsequent players. Moreover, a restriction on the number of consecutive games prevents a pair from monopolizing the contract.

Given the primary focus on security and gas efficiency, the chosen design opted for simplicity. This design allows players a single round, promptly freeing up the contract for new participants. The decision prioritizes simplicity to mitigate potential vulnerabilities and ensure a seamless experience for all users interacting with the contract.

In the game, participants are required to pay an entry fee of 10^{-9} ETH to join. The game progresses in distinct stages, starting with both players entering the game. Following their entry, each player submits a hash computed using the Keccak256 algorithm. This hash is formulated as **Keccak256**(address ‖ number ‖ nonce), where *address* denotes the player's Ethereum address, *number* is their chosen number ranging from 1 to 100, and *nonce* is a randomly generated nonce. After both players have successfully submitted their hashes, the next phase involves revealing their chosen number and nonce. This revelation allows the smart contract to validate the submitted hash and ensure the integrity of the game. The contract then updates the game balances accordingly, and players can proceed to withdraw their winnings. In instances where a player's revealed number and nonce fail to authenticate the hash, or if the chosen number falls outside the permissible range, the game requires the players to resubmit their hashes. This mechanism ensures fairness and adherence to the game's rules.

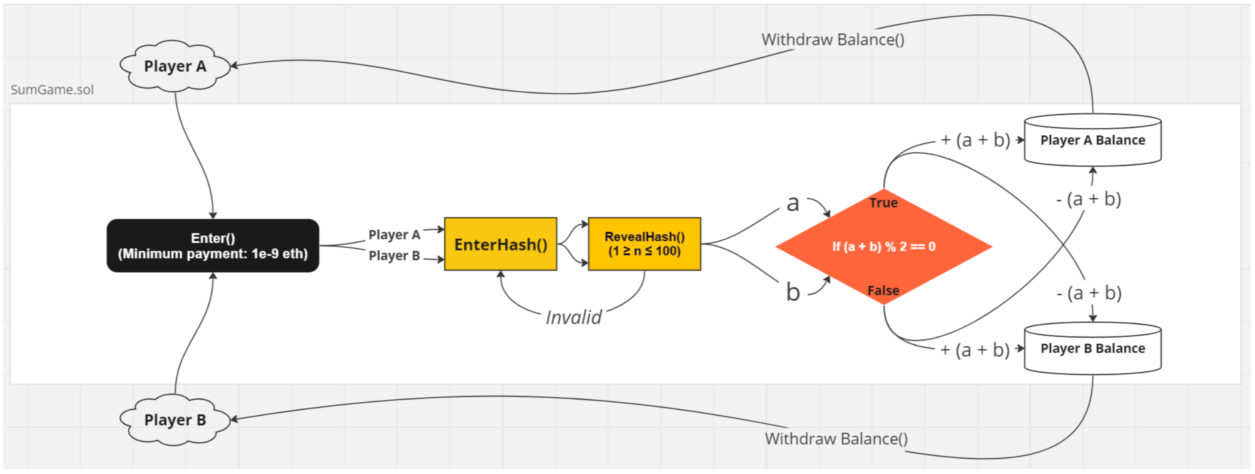

Figure 1: Figure 1: *B178937.sol* state diagram

2.2 Transaction & Gas Fee design decisions

In the domain of transaction and gas fee considerations, the design decisions are intricately woven to ensure fairness, efficiency, and robustness in the game's economic dynamics.

At the core of the transactional architecture is the principle that users bear the responsibility of covering gas fees for any function calls or game interactions. This deliberate choice serves a dual purpose. Firstly, by placing the responsibility on users to pay gas fees, the contract avoids potential vulnerabilities that could arise if the contract itself were to cover these costs, especially in scenarios where the contract lacks sufficient funds. Secondly, this approach shifts the risk associated with frivolous or excessive function calls squarely onto the user, discouraging behaviors that could potentially disrupt or deplete the contract's resources.

To augment fairness and user autonomy, the contract is designed to be as gas-efficient as possible. Users are empowered to set their preferred gas limits, providing flexibility in their interactions with the contract. Notably, the gas efficiency is a paramount consideration, ensuring that users' costs align with the value and experience they derive from participating in the game.

Upon the determination of a winner, the withdrawal process further exemplifies the user-centric approach. Users, including the winner, pay for the gas required to access their funds. This mechanism not only adds an element of equity to the process but also aligns with the principle that each user bears the cost associated with their interactions with the contract.

To minimize gas consumption, the contract incorporates several strategic design decisions. Event emissions are limited to essential occurrences, optimizing user-friendliness without unnecessary gas overhead. The codebase is streamlined with a conscientious reduction in state variables and functions, promoting simplicity and efficiency. Variable packing is employed wherever feasible, and the EVM optimizer is utilized to streamline bytecode execution.

In the realm of fund transfers, the contract adheres to the pull over push principle. By using the transfer method, users initiate the fund transfer, ensuring security against potential attacks and leveraging inherent advantages of this approach. This method not only aligns with the user-centric philosophy but also contributes to a more secure and reliable fund transfer mechanism.

There is no inherent bias or unfairness in gas payments between the players. Each player covers the cost of their own interactions with the contract, and there is no scenario where one player subsidizes the gas costs for the other, although we will discuss the specifics later. This approach aligns with fairness principles in decentralized applications, where users are responsible for their individual transaction costs.

As a caveat, it's acknowledged that the current contract deals with relatively small quantities of wei, and in an improved version, considerations for larger winnings would be prudent. The suggestion is made to explore higher denominations, on the order of tens of thousands of wei, to better align with the economic realities of gas costs and user engagement.

2.3 Security design decisions

This contract places paramount emphasis on security, with a meticulous design that incorporates multiple best practices to ensure a cheat-proof environment for players. The foundational principle governing this contract is robust access control, recognizing its pivotal role in establishing trust. By minimizing access to critical functionalities, the contract builds a secure foundation where players can confidently engage.

Visibility is another cornerstone in the security architecture of this contract. The careful restriction of visibility, particularly tailored to the context of the game, is deemed essential. This ensures that only the necessary information is exposed, reducing the attack surface and enhancing overall security.

Input validation stands as a crucial safeguard against potential exploits. By rigorously validating user inputs, the contract mitigates the risk of manipulative actions from both legitimate users and potential attackers. This proactive approach to input verification fortifies the contract against malicious activities.

Several strategic design choices further enhance the security of this contract. The commitment to utilizing the latest version of Solidity allows us to leverage the most up-to-date security features and fixes. The decision to employ the pull over push principle in transactions minimizes susceptibility to various attacks, including reentrancy and denial-of-service attacks. Additionally, the contract meticulously avoids reliance on block timestamps in its logic, a deliberate measure to prevent potential manipulation by miners. This foresighted choice contributes to the overall resilience against external interference.

The contract proactively addresses a broad spectrum of security threats, including but not limited to reentrancy, front-running, off-chain storage inspection, delegate call attacks, denial-of-service, and integer overflows & underflows. Our approach integrates established defensive strategies to ensure a solid safeguard against these vulnerabilities. Notably, the contract has been fortified to mitigate the risks associated with off-chain storage inspection, effectively protecting sensitive data and game integrity. Furthermore, rigorous analysis using advanced vulnerability detection tools such as Oyente and Mythril has confirmed the robustness of our security measures, as indicated by their reports stating, '*The analysis was completed successfully. No issues were detected*'. This comprehensive security assessment provides an additional layer of assurance regarding the contract's resilience to potential exploits.

2.4 Data Structure choices

In the design of the SumGame smart contract, careful consideration was given to the selection and optimization of data structures and state variables. These choices were driven by the twin objectives of minimizing computational cost and optimizing storage utilization.

Simplification from Structs to Variables: Initially, I considered employing structs to encapsulate related data points. However, this approach, while organizationally appealing, introduced unnecessary computational overhead. To streamline the contract and promote gas efficiency, I pivoted to using simple state variables. This change not only reduced the complexity of the contract but also facilitated tighter variable packing, thereby minimizing storage costs.

State Variables: The contract employs a set of carefully chosen state variables, each serving a distinct purpose:

- `address payable playerA` and `address payable playerB`: These variables store the addresses of the two players participating in the game.
- `uint256 playerABalance` and `uint256 playerBBalance`: These variables keep track of the balances of each player, representing their stakes in the game.
- `bytes32 playerAHash` and `bytes32 playerBHash`: These variables store the hashed commitments of each player's chosen number and nonce, crucial for the commit-reveal scheme that ensures fairness and security.
- `uint8 playerANumber` and `uint8 playerBNumber`: Represent the numbers chosen by each player, confined to the range [1, 100].

- `bool gameState`, `bool winnerPicked`, and `bool hashesSet`: Boolean flags that indicate the current state of the game, whether a winner has been picked, and if both players have submitted their hashes.

Each data type is carefully chosen to be the minimum size necessary to hold the required values, ensuring efficient storage without sacrificing functionality.

Immutable Variables: To enhance gas efficiency, the contract employs an immutable variable `uint32 public immutable minimumWei = 1000000000`. As immutable variables are assigned a value only once and stored directly in the contract code, they offer a gas-efficient alternative to regular state variables.

Event Declarations: Events like `WinnerPicked`, `HashEntered`, `GameReset`, and `InvalidReveal` are defined to emit minimal logs upon key actions within the contract. These serve as efficient means for external entities to monitor contract activities.

Considerations: While designing the data structures, I had to balance the requirements of gas efficiency, security, and clarity. For instance, while individual state variables require more declarations than a single struct, they simplify the contract's interactions and reduce gas costs due to more efficient storage access patterns.

3 Gas Efficiency and Fairness

3.1 Deployment and Interaction costs

Compiled using the latest version of Solidity **^0.8.23** and optimized using the EVM optimizer set to 100000, our contract prioritizes execution efficiency over deployment cost. This approach is particularly beneficial for contracts with frequent interactions, as it reduces the gas fees incurred during regular operations.

The deployment of the contract on the Sepolia testnet can be viewed here, with the gas used totaling 1,048,003.

In terms of interaction costs, the table below outlines the gas usage for different functions of the contract. It is important to note that in each sequence of interactions, Player B always follows Player A. This consistent pattern, where Player A initiates the function calls and Player B responds, is maintained throughout the gameplay, from the initial entry to the final withdrawal. This order of interactions is a significant factor in understanding the dynamics of the contract's operation.

Function	**Player A**	**Player B**
Enter	67,928 gas	90,051 gas
EnterHash	51,814 gas	56,951 gas
RevealHash	32,296 gas	46,664 gas
Withdraw	35,354 gas	41,864 gas

Table 1: Gas Usage for Contract Interactions (Player A Acts First)

An important aspect to consider in the practical implementation of this game is the relationship between potential winnings and the gas costs associated with playing. In the current design, the reward for winning the game is determined by the sum of the two chosen numbers, which ranges between 2 and 200 wei. However, the gas costs incurred during the gameplay are significantly higher than these potential winnings. This discrepancy indicates that, in its current state, players are likely to incur a net loss, making participation in the game economically unfeasible.

For the game to be viable and attractive to players in a real-world scenario, the stakes need to be substantially higher. A revised model where the potential winnings are scaled up – for instance, to hundreds of thousands of wei or more – could offset the gas costs, thereby creating an incentive for players to engage with the contract. It's crucial to ensure that the rewards not only cover the transaction fees but also offer a meaningful profit to the participants, aligning the game's design with practical economic considerations.

3.2 Fairness Analysis

In the observed interactions with the smart contract (Table 3), we notice a consistent pattern where the gas costs for Player B are slightly higher than those for Player A. This variation is attributed to the order of transactions. When Player B initiates each stage of the game first (Table 2), the gas costs are reversed, with Player B incurring less gas than Player A. This implies that the player who acts second tends to pay more due to additional state changes that occur after the first player's action.

Function	**Player B**	**Player A**
Enter	67,928 gas	90,051 gas
EnterHash	51,814 gas	56,951 gas
RevealHash	32,260 gas	46,643 gas
Withdraw	35,354 gas	49,878 gas

Table 2: Gas Usage for Contract Interactions (Player B Acts First)

This variation in gas costs is not an inherent unfairness in the contract design but rather a consequence of the Ethereum blockchain's state update mechanism. When a player acts first, they benefit from a slightly lower gas cost due to fewer state changes compared to the player who acts second. In practical deployment, this incentivizes players to act quickly, and is potentially balanced out over multiple function calls. Therefore, while there is a minor difference in gas costs depending on the order of actions, the overall design does not favor one player over the other.

3.3 Efficiency Improvements

Throughout the development of this contract, a variety of strategies were employed to enhance gas efficiency. One key approach was the optimization of storage requirements for state variables. After evaluating different data structures such as structs, arrays, and simple variables, the decision to use simple state variables was made to facilitate variable packing. This resulted in a more compact and efficient use of storage.

In addition to this, the contract's game state variables were refined, transitioning from the more storage-intensive enums to leaner boolean values which only occupy a single bit. The choice of data types was also meticulously considered, ensuring each state variable utilized the minimum size necessary to hold its value, thereby reducing storage and execution costs.

Logical simplifications were applied extensively across all functions. Modifiers were utilized to eliminate repetitive code, and visibility modifiers were judiciously applied to balance gas savings and security. The messages in 'require' statements were kept concise, not exceeding 32 characters to ensure they occupied only a single memory block.

A conscious decision was made not to initialize any values in the constructor, further saving on deployment gas costs. Writing to storage, a relatively expensive operation, was minimized and deferred until absolutely necessary. The choice of using 'bytes32' over 'string' or 'bytes' was also a gas-saving measure, as 'bytes32' is more efficient for fixed-size data.

Lastly, the number of events emitted by the contract was carefully considered. Only essential events for potential front-end applications were included to avoid unnecessary gas expenditure. These measures collectively contributed to a contract that is not only efficient in terms of gas consumption but also robust and secure. For more insights on gas-efficient Solidity practices, refer to this source.

4 Security Analysis

4.1 Hazard mitigation

Addressing security in this smart contract involved a multi-faceted approach, combining thorough testing with proactive defense against known vulnerabilities. The contract underwent extensive unit testing, ensuring its resilience in various scenarios, including edge cases and potential attack vectors. These tests, serve as a testament to the robustness of the implementation.

In terms of specific security measures, the contract is designed to be safe from reentrancy attacks by adhering to the check-effects-interactions pattern, updating balances before any external calls. Leveraging Solidity version **ˆ0.8** provides automatic protection against arithmetic overflows and underflows, further bolstered by our rigorous input validation.

The contract's design intentionally avoids vulnerabilities associated with self-destruct attacks, delegate calls, and external contract interactions. By not relying on its ether balance for core logic, it remains unaffected by attacks involving forcible ether sending. Additionally, the absence of delegate call functionality and external contract interactions negates associated risks.

We also ensured our contract mechanics is free from randomness-related vulnerabilities, as our contract does not rely on random number generation. This choice is crucial in maintaining the integrity and predictability of game outcomes.

A significant consideration in our design was the handling of potential denial-of-service attacks. By employing a pull-over-push strategy for withdrawals and ensuring that any transfer failures result in transaction reversion, we mitigate the risk of funds being locked due to failed transfers. Additionally, our contract does not contain unbounded loops or operations, further reducing the likelihood of DoS attacks.

The contract uses *msg.sender* for authentication, steering clear of the vulnerabilities associated with *tx.origin* phishing. This choice is critical for ensuring that only legitimate participants can execute contract functions.

Block timestamp manipulation attacks are rendered irrelevant as our contract's logic does not depend on block timestamps. This design decision is crucial for maintaining the integrity and fairness of the game outcomes.

While the contract has been thoroughly secured against a range of attacks and vulnerabilities, a notable challenge that remains is the risk of user inactivity. In scenarios where a participant becomes inactive, there's a potential for locking the contract's state and the funds involved. This situation underscores the necessity for implementing external mechanisms or utilizing front-end applications that can effectively manage or substitute inactive players, thus maintaining the game's flow and preventing deadlock situations.

Additionally, although our contract itself does not directly utilize randomness in its core logic, the security of the game partly hinges on the trustworthiness of the front-end application interacting with the contract, particularly concerning the generation of random nonces for hashing. If there are vulnerabilities or weaknesses in the randomness of these nonces generated by the front end, it could inadvertently provide attackers with an opportunity to deduce a user's chosen number within computationally feasible timeframes. This reliance on external applications for nonce generation introduces an indirect vulnerability that users must be aware of, emphasizing the importance of secure and robust front-end interactions with the contract.

Furthermore, the contract incorporates a commit-and-reveal scheme to safeguard against front-running attacks, ensuring the privacy and fairness of players' number submissions. This mechanism prevents any party from gaining unfair advantage by observing and reacting to other players' actions on the blockchain. The effectiveness of this scheme, however, is partly dependent on the security of the nonce generation process, as previously mentioned.

The implementation of the Keccak256 hashing algorithm in our smart contract offers a high degree of security against potential attacks. Its collision resistance ensures that it is exceedingly difficult for attackers to find two distinct inputs that produce the same hash output, thereby maintaining the integrity of the hashed values. Furthermore, Keccak256's pre-image resistance means that it is nearly impossible to reverse-engineer the original input from its hash output, protecting against brute-force attacks. However, it's crucial to acknowledge that while these features make Keccak256 robust, no cryptographic method is entirely infallible. Therefore, combining unique and unpredictable input parameters enhances the overall security, making it more challenging for attackers to compromise the system.

In summary, the contract's security measures are comprehensive, addressing known vulnerabilities and ensuring robust protection against potential attacks. The combination of these measures with the commit and reveal scheme forms a solid foundation for a secure and fair gaming experience.

4.2 Tradeoffs and Implementation details

In the development of this smart contract, a series of tradeoffs and decisions were made to balance various aspects such as gas efficiency, privacy, security, code organization, user friendliness, and fairness. These tradeoffs reflect the inherent complexities and challenges in designing a robust and practical blockchain-based application.

Gas Efficiency vs. Privacy/Security: A significant tradeoff was between enhancing gas efficiency and ensuring the privacy and security of the game. The implementation of a commitment scheme added considerable complexity to the contract, impacting gas costs. However, this complexity was a necessary compromise to ensure the privacy of player choices and safeguard against cheating or front-running, thus enhancing the overall security and integrity of the game.

Code Structure/Organization vs. Gas Efficiency: Another tradeoff involved balancing the desire for a well-structured, easy-to-understand codebase with the need for gas efficiency. While a more structured approach using data structures like structs and an extensive set of events could enhance code readability and organization, it was necessary to limit these aspects in favor of reducing gas consumption. This decision meant opting for simple variables over structs and minimizing the number of emitted events, ensuring that each transaction remained cost-effective.

User Friendliness vs. Security: The introduction of a commitment scheme, while crucial for preventing cheating and front-running, added a layer of complexity that could be perceived as less user-friendly. This decision emphasized security over simplicity, ensuring that players could not gain unfair advantages by observing others' actions on the blockchain. The complexity of this mechanism, however, might necessitate additional steps or interactions from the users, potentially impacting the overall user experience.

Security vs. Performance: Throughout the development, ensuring a high level of security was paramount, which often meant making choices that could impact performance. For example, the use of rigorous input validation and adherence to the check-effects-interactions pattern for mitigating reentrancy attacks were vital for security but added additional computational steps, thereby affecting performance.

Fairness vs. Efficiency: Ensuring fairness in the game, particularly in terms of gas costs for both players, was a key consideration. The contract was designed to minimize instances where one player might consistently bear a higher gas cost than the other. However, achieving this balance was challenging due to the dynamic nature of gas prices and the order of transactions, which could inherently lead to some discrepancies in gas costs between players. For example, the second player to withdraw must pay for the resetGame gas fees, but this is unavoidable and the design can only try to make the discrepancy a small as possible.

In summary, the development of this contract involved carefully navigating these tradeoffs to create a solution that balanced efficiency, security, and fairness. The choices made were geared towards optimizing the contract for its intended use-case, ensuring that it could function effectively in the real-world blockchain environment while adhering to the principles of secure and fair gameplay.

5 Fellow student analysis

The contract being analyzed can be found here. Each player wagers 200 wei and chooses a number between 1 and 100. The game concludes when both players have selected their numbers. The sum of these numbers determines the winner: if it's even, Player A wins; if odd, Player B wins. The winner receives their original bet plus the combined sum as a reward, while the other player loses an equivalent amount. The game features events to signal the start and end of each game, along with the declaration of the winner. Players can also reset the game once a round is completed, allowing for continuous play.

5.1 Vulnerabilities

The contract, while implementing a simple number-guessing game between two players, exhibits several security vulnerabilities and design flaws that could be exploited by attackers or lead to undesirable game outcomes. Here's an analysis of each point:

Public Variables and Cheating Risk: The contract's critical variables, such as the numbers chosen by players, are private but can still be read by any external observer once set. This lack of privacy means that a third party could easily read these values directly from the blockchain, enabling them to predict or influence the game's outcome. In a more competitive or high-stakes scenario, this transparency could lead to cheating or strategic manipulation by external observers.

Front-Running Vulnerability: Due to the transparent nature of blockchain transactions, front-running is a significant risk. In this context, a malicious user can monitor pending transactions (like Player A choosing a number) and quickly submit their own transaction (as Player B) with a number that ensures their victory. This problem is exacerbated because the players' numbers are immediately visible and not concealed or committed in any encrypted form.

Reentrancy Attack Risk: The contract's endGame function is vulnerable to reentrancy attacks. The state variable gameEnded is set to true after Ether transfers have occurred. This sequence could potentially allow a malicious actor to re-enter the contract and execute the endGame logic again before the state is updated, leading to unexpected behavior or loss of funds.

```
winner = payable(playerA);
payable(winner).transfer(betAmount + winnerReward);
...
emit WinnerDeclared(winner, winnerReward);
gameEnded = true;
```

Listing 1: Reentrancy Vulnerability

Denial of Service (DoS) via Push Payments: The contract uses a 'push' instead of a 'pull' strategy for transferring Ether winnings. If one of the transfers fails (for example, if a player's account is a contract that rejects Ether), the entire transaction could revert, potentially locking the contract and making it unusable. This approach opens up the possibility of a deliberate or accidental DoS attack.

Inactivity of Players: Similar to the DoS risk, the contract does not handle the scenario where a player becomes inactive after the game starts. If one player chooses their number and the other does not, the game can become indefinitely stalled, with no mechanism to reset or address the inactivity.

Public Reset Function: The resetGame function is public and can be called by anyone, not just the participating players or an administrative entity. This design could lead to scenarios where an external party disrupts the game flow by prematurely resetting the game state, potentially erasing the progress of an ongoing game.

```
function resetGame() public {...}
```

Listing 2: Public reset function

Each of these vulnerabilities represents a significant risk in the context of smart contract games, where transparency, fairness, and security are paramount. Addressing these issues would require a combination of improved privacy measures (like using a commit-reveal scheme), rethinking the game logic (especially regarding Ether transfers and state updates), and implementing robust access controls.

5.2 Gas Efficiency

In analyzing the NumberGame smart contract, several opportunities for enhancing gas efficiency are evident. First, the contract exhibits code redundancy, which could be streamlined by employing Solidity modifiers or simplifying functions. Such optimizations would eliminate repetitive code, leading to reduced gas consumption for each transaction.

Action	Gas Used
Deployment	898,249 gas
enterNumber PlayerA	70,260 gas
enterNumber PlayerB	95,202 gas
resetGame	25,725 gas

Table 3: Gas Usage for Contract Functions

```
function PlayerA(uint8 _numberA) external payable gameNotEnded {
        require(msg.sender != playerA && playerA == address(0));
        require(msg.value == betAmount);
        playerA = msg.sender;
        require(playerA != playerB);
        require(_numberB >= 1 && _numberB <= 100);
...
function PlayerB(uint8 _numberB) external payable gameNotEnded {
        require(msg.sender != playerB && playerB == address(0));
        require(msg.value == betAmount);
        playerB = msg.sender;
        require(playerA != playerB);
        require(_numberB >= 1 && _numberB <= 100);
```

Listing 3: Repetition example (Require statements removed)

Another area of potential improvement lies in the use of the contract's constructor. The current implementation may include unnecessary initializations. Simplifying or removing these initializations could lead to significant gas savings, especially during the contract deployment phase.

```
constructor() {
    gameEnded = false;
}
```

Listing 4: Unnecessary Constructor

The use of immutable for certain variables, like betAmount, is another opportunity for optimization. Marking variables as immutable can reduce storage costs, as these variables are written to the contract's bytecode instead of storage, making them cheaper to access. Additionally, variable packing - a technique to store multiple variables in a single storage slot - is not utilized in the current contract. Implementing variable packing can significantly optimize storage usage and, in turn, reduce gas costs.

5.3 Fairness

Regarding fairness, the NumberGame contract reveals an inherent imbalance in gas costs between the two participating players. Notably, the player who enters their number second (typically PlayerB) consistently faces higher gas fees. This discrepancy stems from the fact that the second player's action triggers the endGame function. This function involves additional computational steps, including transferring balances, which naturally incur higher gas costs.

Moreover, the contract's resetGame function introduces another element of unfairness. The gas cost of this function is entirely borne by the player who calls it. As a result, one player may incur additional costs for resetting the game while the other player does not, creating an uneven playing field.

6 Source code and address

My contract can be found at **0x2b87b99D316D37A7DeB257759d6958ce330BB6F3** on the Sepolia testnet. The source code can be found here or below:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.23;

```

```solidity
contract SumGame {
    address payable playerA;
    address payable playerB;
    uint256 private playerABalance;
    uint256 private playerBBalance;
    bytes32 private playerAHash;
    bytes32 private playerBHash;
    uint32 public immutable minimumWei = 1000000000;
    uint8 private playerANumber;
    uint8 private playerBNumber;
    bool public gameState;
    bool public winnerPicked;
    bool public hashesSet;

    event WinnerPicked(address winner, uint8 winningAmount);
    event HashEntered(address player);
    event GameReset();
    event InvalidReveal(address player);

    modifier gameInProgress() {
        require(gameState, "Game not in progress");
        _;
    }

    modifier mustBePlayer() {
        require((msg.sender == playerA || msg.sender == playerB), "Must be a player");
        _;
    }


    function enter() external payable {
        require(!gameState, "Game in progress");
        address sender = msg.sender;
        uint256 value = msg.value;
        require(value >= minimumWei, "Payment less than minimum to play");
        require(sender != playerA, "Cannot enter twice");
        if (playerA == address(0)) {
            playerA = payable(sender);
            playerABalance = value;
        } else {
            playerB = payable(sender);
            playerBBalance = value;
            gameState = true;
        }
    }


    function enterHash(bytes32 hash) external mustBePlayer gameInProgress {
        bytes32 currentHash = (msg.sender == playerA) ? playerAHash : playerBHash;
        require(currentHash == bytes32(0), "Hash already set");
        require(hash != bytes32(0), "Invalid (default) hash");
        if (msg.sender == playerA) {
            playerAHash = hash;
        } else {
            playerBHash = hash;
        }
        emit HashEntered(msg.sender);

        // Set hashesSet to true if both players have entered their hashes
        if (playerAHash != bytes32(0) && playerBHash != bytes32(0)) {
            hashesSet = true;
        }
    }

    function validHash(uint8 number, uint256 nonce) private view returns (bool) {
        bytes32 hash = (msg.sender == playerA) ? playerAHash : playerBHash;
        return ((number > 0 && number < 101) && keccak256(abi.encodePacked(msg.sender,
    number, nonce)) == hash);
    }

    function revealHash(uint8 number, uint256 nonce) external mustBePlayer gameInProgress {
        require(hashesSet, "Hashes not submitted yet");
        bool isPlayerA = (msg.sender == playerA);
        uint8 currentNumber = (isPlayerA) ? playerANumber : playerBNumber;
        require(currentNumber < 1, "Number already revealed");
        uint8 otherNumber = (isPlayerA) ? playerBNumber : playerANumber;
        if (validHash(number, nonce)) {
            if (isPlayerA) {
                playerANumber = number;
            } else {
                playerBNumber = number;
            }
            if (otherNumber > 0) {
                pickWinner();
            }
        } else {
            emit InvalidReveal(msg.sender);
            restartGame();
        }
    }

    function restartGame() private {
        playerAHash = 0;
        playerBHash = 0;
        playerANumber = 0;
        playerBNumber = 0;
        hashesSet = false;
        emit GameReset();
    }

    function pickWinner() private {
        uint8 sum = playerANumber + playerBNumber;
        address winnerAddress;
        if (sum % 2 == 0){
            playerABalance += sum;
            playerBBalance -= sum;
            winnerAddress = playerA;
        } else {
            playerBBalance += sum;
            playerABalance -= sum;
            winnerAddress = playerB;
        }
        winnerPicked = true;
        emit WinnerPicked(winnerAddress, sum);
    }

    function getBalance() external view mustBePlayer returns (uint256) {
        if (msg.sender == playerA) {
            return playerABalance;
        } else {
            return playerBBalance;
        }
    }

    function withdraw() external mustBePlayer gameInProgress {
        require(winnerPicked, "Winner must be picked before you can withdraw.");
        uint256 withdrawingBalance = (msg.sender == playerA) ? playerABalance :
    playerBBalance;
        require(withdrawingBalance > 0, "Your balance is 0.");
        address payable withdrawingAddress = payable(msg.sender);

        if (msg.sender == playerA) {
            playerABalance = 0;
        } else {
            playerBBalance = 0;
        }
        withdrawingAddress.transfer(withdrawingBalance);

        if (playerABalance < 1 && playerBBalance < 1) {
            resetGame();
        }
    }

    function resetGame() private {
        playerABalance = 0;
        playerBBalance = 0;
        playerANumber = 0;
        playerBNumber = 0;
        playerAHash = bytes32(0);
        playerBHash = bytes32(0);
        playerA = payable(address(0));
        playerB = payable(address(0));
        gameState = false;
        winnerPicked = false;
        hashesSet = false;
        emit GameReset();
    }
}
```

Listing 5: B178937.sol

7 Bibliography

- Lecture slides
- https://yos.io/2021/05/17/gas-efficient-solidity/
- https://docs.soliditylang.org/en/v0.8.7/internals/layout_in_storage.html
- https://solidity-by-example.org/calling-contract/
- https://github.com/Consensys/mythril