On-Chain Commit-Reveal Lottery Application

Irfan Bozkurt, 2017400261

Introduction

This report presents an overview of a fully decentralized on-chain lottery application. The application leverages non-malleable user commitments to eliminate the reliance on on-chain random number generation (RNG). By offloading the RNG process to end users, the application ensures a transparent and tamper-proof lottery system. The key design principles of the application include the separation of purchase and reveal phases, cryptographic commitments, and secure reward claiming.

The lottery operates in rounds, each consisting of a purchase phase and a reveal phase. During the purchase phase, which spans a one-week period, users can purchase lottery tickets to participate in the ongoing lottery. To participate, users must submit a SHA-3 hash of two values: a random number and their address. This cryptographic commitment ensures the integrity of the process and prevents manipulation by malicious entities. The reveal phase follows the purchase phase and lasts for another week. In this phase, participants are required to reveal the random numbers they submitted in the previous phase. Reveal actions must originate from the same address that made the cryptographic commitment, further enhancing security.

Technique

To ensure a fair and unbiased lottery system, the on-chain lottery application implements a perfectly decentralized lottery-style non-malleable commitment technique. The following considerations are taken into account to eliminate unfair advantages and maintain transparency:

- User Decisions: Any decision made by a user that could influence the outcome of the lottery, such as using a blockhash, timestamp, or user-submitted random number, is considered unfair. To prevent this, the contract ensures that everything it sees is visible to the public, eliminating the possibility of manipulating the system.
- Getting Rid of On-chain Number Generation: This eliminates the risk of fraudulent miners manipulating the lottery results. Users are responsible for generating their own random numbers and maintaining the security of their cryptographic commitments. The application does not require block-related information, enhancing its resistance to external manipulation.
- Timing of Number Generation: The contract delays the generation of the random number until after the entry into the lottery has been closed. This prevents users from knowing the number beforehand and gaining an unfair advantage.

Rules & Steps

1. Purchasing round starts
2. Each user generates their own secret random number (N) locally and hashes it with their address (msg.sender) using the SHA-3 algorithm. This creates a commitment (hash) unique to each user.

   ```
   ethers.utils.keccak256(
     new ethers.utils.AbiCoder().encode(
       ["uint256", "address"],
       [rnd, signer.address]
     )
   )
   ```

   Users submit these hashes while buying the ticket.
3. Once the purchase round ends, the reveal round begins. Users are required to submit their actual random number (N) to the contract. The contract verifies that the submitted random number matches the original commitment. If a user fails to reveal a valid random number within the specified time, their deposit is forfeited.
   All revealed random numbers (N) are XOR'd together to generate the final random number that determines the winner.
4. After the reveal round ends, any of the corresponding read methods of the contract will trigger the winner-selection algorithm.
   The formula **XOR % numUsers** is used to select the participant who will win the lottery.

Determination of the Winner

Once the purchase and reveal phases conclude, users can interact with the contract to check if they are winners or claim their rewards immediately. The contract populates the winners' positions based on a sample size of revealed tickets.

The first-place winner is determined using an XOR operation on the sample size. The second and third-place winners are selected through subsequent XOR operations with cycle breakers. In other words, selection of the winner is random while 2nd and 3rd can be determined by anyone if they know the winner

The prizes are calculated based on the total money collected, total money carried out by the previous contract, the position of the winning ticket, and the ticket type. The contract calculates the prizes proportionally, with different ticket types having varying prize distributions.

Implementation Nuances

The biggest point to see as a difference from traditional back-end applications in this system is that each chain of execution must be triggered by an external entity. One just cannot tell the contract to execute the winner-selection algorithm at a certain time automatically. They must order the execution at the time of execution manually.

As an example, when reveal round for lottery N finishes, the contract is able to calculate the winners. But for this to happen, somebody must make a call and trigger the contract to do so.

This is not a problem, generally, when the state of the contract is only dependent on user funds, or other user-related information. Everyone can deposit some funds, interact with a contract as much as they have money, and get out.

But this lottery application does not come with such ease. Users' success here is dependent on other users' behavior (random numbers sent by them), as well as the states of the past rounds (because lotteries inherit ungiven prizes from before).

To handle such complexity, contract must clearly define what the inputs of each round will be, what the arithmetics is, as well as keeping pointers signifying which lottery round was calculated the last time. This enables the contract to start at the lastly-left point of execution and fill the gap up to now.

Such a base enables the engineer to implement basic "checking functions" that

1. Controls if the lastly calculated information is up-to-date,
2. Triggers calculation if not so

If we can design the contract in this way so far and trigger these "checking functions" with every (or almost every) call to the contract, then this lottery logic can be tamed on-chain.

A Note on Efficiency and Fairness

With contracts needing to manage a list of entities comes risk of iteration, which is costly to do on-chain. A naive approach, for instance, to determine the winner after the reveal round can be iterating through all the revealed tickets and XORing them.

This way of programming should always be considered as last resort, for when it's impossible to do the aggregation (in our case, XOR aggregation) on-the-fly.

Another side effect of triggering a bulky iteration is delegating all the gas cost of that iteration to the one who calls it. This is unfair to that person as the cost of arithmetics is off-loaded to them only.

What should be done instead, as I do in this work, is aggregating the data whenever there's new input to the sample group. We can XOR the random numbers as they are revealed. This will shrink the winner-detection logic down to O(1) after the reveal round, as well as partitioning the overall gas cost of the contract to everyone participating.

Alternatives

Please note that this contract is no fit for production usage because although hard, there exists some scenarios enabling a miner to manipulate the result. The contract implements what's known as the best pattern for source of randomness, the commit-reveal scheme.

Use cases like lotteries or gambling are traditionally reliant on random number generation, and perhaps the best gambling application would be possible with a strong ChainLink integration. This wouldn't give away the decentralized nature of a web3 application as ChainLink offers decentralized oracle networks.

Another possibility could be making another chain mine randomness and transmitting the new blocks of random numbers to the Ethereum network. This option only sounds different from the previous one, because bridges require off-chain validation of transactions.

However, this work represents the strongest base that randomness can get to without relying on oracles or bridges.

Tests

This contract has been tested using a massive 750-lines file that comes with unit-ish tests for each and every function, and tests for more complex integrative scenarios. Here’s a sneak peek of differend kinds of tests:

```
base monatery functionality
  ✓ depositEther
  ✓ withdrawEther
  ✓ getBalance
time arithmetics
  ✓ getCurrentLotteryInPurchase
  ✓ getCurrentLotteryInReveal
state variables
  ✓ getTotalLotteryMoneyCollected
  ✓ getTicketPrice
  ✓ getLastOwnedTicketNo
  ✓ getIthWinningTicket
  ✓ getIthOwnedTicketNo
  ✓ getLotteryNos
business logic
  buyTicket
    ✓ success
    ✓ failure
  collectTicketRefund
    ✓ success
    ✓ failure
  revealRndNumber
    ✓ success
    ✓ failure
  checkIfTicketWon
    ✓ success
    ✓ failure
  collectTicketPrize
    ✓ success with one entry
    ✓ success with two entries
    ✓ success with three entries
    ✓ failure
mass usage
  ✓ 5 tickets  ✓ 10 tickets  ✓ 100 tickets  ✓ 200 tickets
```

See full implementation for tests in **Lottery.test.js** by running **npm hardhat test**

Average Gas Usage

Solc version: 0.8.9
Optimizer enabled: true, Runs: 1000
Block limit: 210000000 gas

Here is the hardhat-generated average gas usages.

```
|  Methods
·······················|·············|·············|··············|············
 Method                ·   Min        ·  Max        ·  Avg       # calls
·······················|·············|·············|··············|············
 buyTicket             ·   82661  · 116937  ·  84533  ·     336  ·
·······················|·············|·············|··············|············
 collectTicketPrize    59224  ·256693  ·  133950  ·    21  ·
·······················|·············|·············|··············|············
 collectTicketRefund  -  ·          -  ·      55933  ·      2  ·
·······················|·············|·············|··············|············
 depositEther          43955  · 43967  ·    43955  ·     339  ·
·······················|·············|·············|··············|············
 revealRndNumber  52030  · 118449  ·  85397  ·     329  ·
·······················|·············|·············|··············|············
 withdrawEther            -  ·          -  ·      33863  ·      2  ·
·······················|·············|·············|··············|············
|  Deployments                          ·                               % of limit
·······································|·············|·············|············
|  Lottery                         ·          -  ·     1866400  ·    0.9 %
·--------------------------------------|-------------|-------------|------------
```

Run the Project

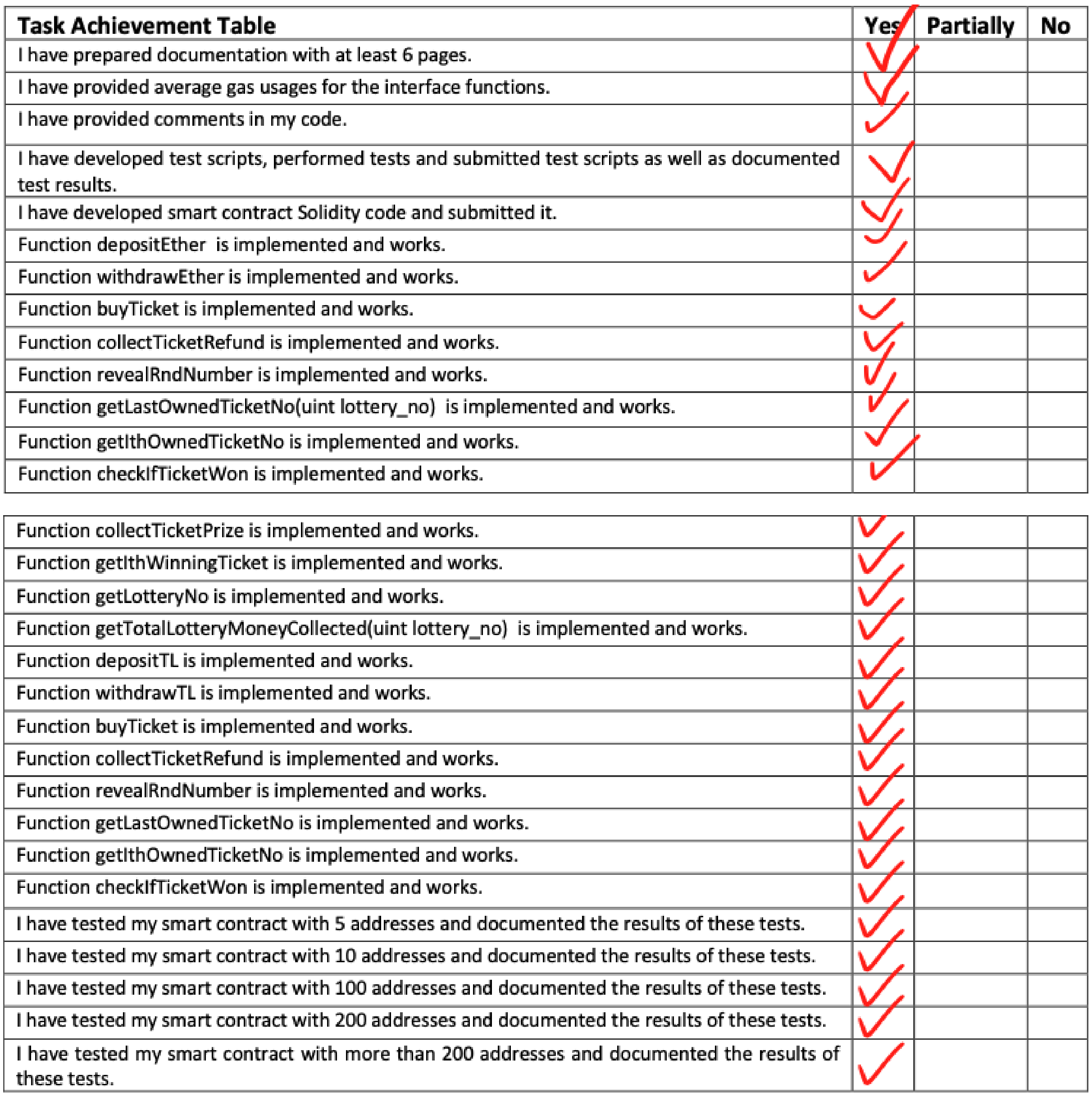
Please see README.md

Conclusion

The decentralized on-chain lottery application presented in this report offers a transparent and secure platform for lottery participation.

By leveraging non-malleable user commitments and offloading the random number generation process to participants, the application ensures fairness and prevents manipulation by external actors. The separation of purchase and reveal phases, along with a enrichened commitment schema, enhances the security and trustworthiness of the system.

Task Achievement Table



Task Achievement Table	Yes	Partially	No
I have prepared documentation with at least 6 pages.	✓		
I have provided average gas usages for the interface functions.	✓		
I have provided comments in my code.	✓		
I have developed test scripts, performed tests and submitted test scripts as well as documented test results.	✓		
I have developed smart contract Solidity code and submitted it.	✓		
Function depositEther is implemented and works.	✓		
Function withdrawEther is implemented and works.	✓		
Function buyTicket is implemented and works.	✓		
Function collectTicketRefund is implemented and works.	✓		
Function revealRndNumber is implemented and works.	✓		
Function getLastOwnedTicketNo(uint lottery_no) is implemented and works.	✓		
Function getIthOwnedTicketNo is implemented and works.	✓		
Function checkIfTicketWon is implemented and works.	✓		
Function collectTicketPrize is implemented and works.	✓		
Function getIthWinningTicket is implemented and works.	✓		
Function getLotteryNo is implemented and works.	✓		
Function getTotalLotteryMoneyCollected(uint lottery_no) is implemented and works.	✓		
Function depositTL is implemented and works.	✓		
Function withdrawTL is implemented and works.	✓		
Function buyTicket is implemented and works.	✓		
Function collectTicketRefund is implemented and works.	✓		
Function revealRndNumber is implemented and works.	✓		
Function getLastOwnedTicketNo is implemented and works.	✓		
Function getIthOwnedTicketNo is implemented and works.	✓		
Function checkIfTicketWon is implemented and works.	✓		
I have tested my smart contract with 5 addresses and documented the results of these tests.	✓		
I have tested my smart contract with 10 addresses and documented the results of these tests.	✓		
I have tested my smart contract with 100 addresses and documented the results of these tests.	✓		
I have tested my smart contract with 200 addresses and documented the results of these tests.	✓		
I have tested my smart contract with more than 200 addresses and documented the results of these tests.	✓		