

CmpE 483 - Blockchain Programming

Homework-1, Spring 2022

Fikri Cem Yılmaz, 2017400141

Zehranaz Canfes, 2017205138

M. Furkan Atasoy, 2017400216

1. Abstract

The objective of this project was to implement an autonomous decentralized lottery using solidity and smart contracts. A lottery round takes two weeks and is divided into two stages. After the previous lottery is completed, a new lottery round should begin. **ERC20** token contract is used to implement TL tokens and **ERC721** token contract is used to implement tickets. The tickets should be transferrable and should be represented as tokens. To achieve these tasks, various functions should be implemented and tested. Achieving an autonomous system is challenging and thus requires inspecting details and calculating possible failures in the system.

2. Introduction

Like each lottery, the implemented autonomous decentralized lottery has winners and losers. Being a decentralized blockchain-based system, buying and revealing tickets require hashing and key controls. Here, for hashing, securely generated random numbers are used. When one wants to buy ticket, he/she should provide a random number that belongs only to them and use it to reveal the tickets. The i th prize P_i will be given with the following formula:

$$P_i = \lfloor M/2 \rfloor + (\lfloor M/2^{i-1} \rfloor \bmod 2) \text{ and } i = 1, \dots, \lfloor \log_2(M) \rfloor + 1$$

For each functionality, an amount of gas is used. The aim is to keep the amount of gas used minimal and therefore the implementation should be optimized. Also, there are a lot of functionalities and therefore there should be a proper design and a systematic, organized implementation. Thus, we have worked on the readability of the code and

thought on every variable and functionality to decrease the possibility for security issues and wrong implemented autonomous system.

3. Methodology

To achieve an autonomous lottery system, we have implemented several functionalities that are connected with each other.

As required, we used the ERC20 contract for the Turkish lira currency. The implementation for TL token can be found in *tl.sol* file.

```
function takeAmount(uint256 amount) public{ }
```

This function is to distribute TL currency to the contributors to the lottery. This way, anyone can take the amount they want using this function.

We used ERC271 contract to implement the lottery. The lottery consists of hash, dictionary to keep the balance mapped to the address, private addresses of deployer and currency deployer. The currency is defined to be TL. Each hash has a value, isHashed, and status fields. The status field is used to keep the status of the ticket. If *status* = 0, then it is hashed, if *status* = 1, then the ticket is refunded, if *status* = 2, then the ticket is revealed, and finally, if *status* = 3, then the prize is collected.

Lottery also contains the a private token id to keep the count of tokens. For each ticket, a hash value is stored in the mapping *ticket_hashes*. *Lottery_to_tickets* keeps the ticket_ids for each lottery and *revealed_tickets* keeps the revealed tickets only to determine the winners of the lottery. To search for users and tickets we have implemented the field *lottery_to_account_to_tickets* and *ticket_to_lottery*. The first one is used to access the tickets belonging to a specific user. Here the user's address is used to access them. The latter is to access the lottery number using the ticket id.

For each user, the last bought tickets are kept in a dictionary. Moreover, after the lottery ends, the random numbers *N* of each users is XOR'd and to determine the winning user, we use

$$(N \% numberOfParticipants)$$

After each reveal, we store the XORd value in a dictionary called `xored_value`. Since this is a lottery, there should be a total amount collected from this lottery. This is calculated and then stored in a dictionary *collected_from_lottery*.

The instance of lottery is constructed using the ERC 271 contract and uses TL's deployment address. Token id is initialized with 1 and the deployment time i.e. instance creation time is kept using the timestamps. Each lottery is 10 TL and therefore the unit cost is initialized to be 10.

Now that each variable is explained and discussed above, the functionalities that we implemented can be explained.

a. `depositTL`

This is a simple method to deposit the given amount of currency. Here it is checked whether the transfer is happening or not i.e. if the transfer is approved or if there is enough money to be transferred. If the transfer occurs, then the balance dictionary is updated and the amount transferred is added to the dictionary.

b. `withdrawTL`

This is a method to withdraw money from the account. The balance is checked to see if there is enough currency to be withdrawn and if the transfer is approved. After the checks are passed, then the balance dictionary is updated by subtracting the amount from the balance.

c. `checkBalance`

This is to return the balance dictionary content.

d. `Transfer`

This is a method to transfer tickets among the users. Users are allowed to transfer tickets only after the lottery ends. With this, we aim to prevent conflicts caused by changing the ownerships. Therefore, firstly, it is checked whether the lottery round has ended or not. This is done by using the current lottery's id and the current's ticket lottery id.

e. `buyTicket`

This is a method to buy tickets for the lottery. To buy lottery tickets, one needs to give out a hash number that was generated using a private random number. If the submission time is ended or there is not sufficient currency in the account. Since each ticket is 10 TL, this amount should be taken from the users buying tickets. The bought token id is added to the lottery and it is inserted to the user's list of last bought tickets. After this trade, the amount of currency collected from users is increased by 10 TL. Moreover, this token id is added to other lists to monitor the user-lottery-ticket relationships (*lottery_to_account_to_tickets*, *ticket_to_lottery* mappings). Given the hash value as input, the hash instance is created and its status is set to 0 i.e. is hashed. The counter token id is increased by 1.

f. `collectTicketRefund`

If the lottery round ends and the user fails to reveal his/her ticket, then he/she can ask for a refund. For this refund to happen, the validity of the ticket and whether the user is authorized to get a refund for his/her ticket is checked i.e. if the *msg.sender* is equal to the *ownerOf(ticket_no)* or not. If all checks are passed, then it is checked whether the ticket was revealed or not. For this purpose we check the status of the lottery i.e. whether it has ended or not. Also, the hash's status is checked. It should be hashed and has the status 0. If all tests are passed, then the status of the hash is set to 1 i.e. *refunded* and the half of the money (5 TL) is refunded to the balance of the user.

g. `revealRndNumber`

This is to reveal the submission period ends (after 4 days), the users can reveal their tickets and they have 3 days to do so. The validity of the ticket and the permissions are checked. Similar to `collectTicketRefund`, the status of the hash is checked to see if the ticket was revealed before or expired before. Then using the random number, the correctness of the hash is checked. If they are not the same, then the reveal does not happen. This is required as for every blockchain distributed systems to increase the security. After each reveal, the XOR'd values are recalculated and the status hash is set to *not hashed*.

h. `calculate_iterator`

This is an internal log function to calculate the iterator *i* as given by the project instructions. It returns the iterator after calculating it.

i. `collectTicketPrize`

The prize of the ticket is collected using this function. The validity and the permissions of the tickets are checked before proceeding. Here, we implemented a for loop to calculate the prizes. This for loop lasts for number of iterations calculated using the *calculate_iterator* function. As the iteration number increases i.e. the amount of money collected, the amount of gas used for this functionality increases since the for loop time will also increase. Using the ticket number, we get the lottery number. And this lottery number is used to determine the winning index i.e. winner. And the prize amount is calculated using the function:

$$P_i = \lfloor M/2 \rfloor + (\lfloor M/2^{i-1} \rfloor \bmod 2)$$

The status of the ticket's hash is set to 3 i.e. prize collected and the balance of the user is updated with the prize amount.

j. `checkIfTicketWon`

This function does the same operations as the *collectTicketPrice* except this function returns the amount instead of paying it.

k. `getIthWinningTicket`

This is to get the *i*th winning ticket. To achieve this, first, some checks are done. The number *i* and the lottery number are checked to see if they are valid i.e. greater than 0. Then the status of the lottery is checked i.e. whether it has ended or not. If it did not end yet then the function stops. Else, the number *i* should be less than the iteration number and if this is also true then the function works on. The winner is calculated as in *collectTicketPrice*. Both the winner ticket and the gained amount are returned.

l. `getLotteryNo`

Gets the unix time and calculates the lottery number corresponds to given time. Returns the lottery number.

m. `getTotalLotteryMoneyCollected`

It takes *lottery_number* as input and returns the amount of collected money for that lottery round. The lottery number cannot be 0 and the lottery should have already started.

n. `getLastOwnedTicketNo`

This function takes the *lottery_number* as input and returns the user's last owned tickets list. The lottery conditions are the same as *getTotalLotteryMoneyCollected* and the user should be checked to see if she/he bought any ticket for this lottery.

o. *getIthOwnedTicketNo*

This is a similar function to *getLastOwnedTicketNo* but it returns the *ith* owned ticket instead of the last one for a given lottery. The ticket number as well as the ticket's hash status are returned.

After completing our implementations and tests, we proved that the implemented autonomous distributed system for lottery works as required.

4. Numerical Analysis

Average Gas Usages for the Interface Functions

- a. *depositTL*
101772 gas
- b. *withdrawTL*
73145 gas
- c. *checkBalance*
23657 gas
- d. *buyTicket*
231532 gas
- e. *collectTicketRefund*
107140 gas
- f. *revealRndNumber*
87475 gas
- g. *collectTicketPrize*
78784 gas
- h. *checkIfTicketWon*
57528 gas












- i. `getIthWinningTicket`
66224 gas
- j. `getLotteryNo`
24673 gas
- k. `getTotalLotteryMoneyCollected`
26819 gas
- l. `getLastOwnedTicketNo`
31617 gas
- m. `getIthOwnedTicketNo`
32227 gas

Gas used in deployments:

- TL deployment: 1290682 of 1290682 gas
- Lottery deployment: 5189688 of 5189688 gas

5. Tasks

Task Achievement Table	Yes	Partially	No
I have prepared documentation with at least 6 pages.	<input checked="" type="checkbox"/>		
I have provided average gas usages for the interface functions.	<input checked="" type="checkbox"/>		
I have provided comments in my code.	<input checked="" type="checkbox"/>		
I have developed test scripts, performed tests and submitted test scripts as well documented test results.	<input checked="" type="checkbox"/>		
I have developed smart contract Solidity code and submitted it.	<input checked="" type="checkbox"/>		
Function <code>depositTL</code> is implemented and works.	<input checked="" type="checkbox"/>		
Function <code>withdrawTL</code> is implemented and works.	<input checked="" type="checkbox"/>		
Function <code>buyTicket</code> is implemented and works.	<input checked="" type="checkbox"/>		
Function <code>collectTicketRefund</code> is implemented and works.	<input checked="" type="checkbox"/>		
Function <code>revealRndNumber</code> is implemented and works.	<input checked="" type="checkbox"/>		
Function <code>getLastOwnedTicketNo(uint lottery_no)</code> is implemented and works.	<input checked="" type="checkbox"/>		
Function <code>getIthOwnedTicketNo</code> is implemented and works.	<input checked="" type="checkbox"/>		
Function <code>checkIfTicketWon</code> is implemented and works.	<input checked="" type="checkbox"/>		
Function <code>collectTicketPrize</code> is implemented and works.	<input checked="" type="checkbox"/>		
Function <code>getIthWinningTicket</code> is implemented and works.	<input checked="" type="checkbox"/>		
Function <code>getLotteryNo</code> is implemented and works.	<input checked="" type="checkbox"/>		
Function <code>getTotalLotteryMoneyCollected(uint lottery_no)</code> is implemented and works.	<input checked="" type="checkbox"/>		
Function <code>depositTL</code> is implemented and works.	<input checked="" type="checkbox"/>		
Function <code>withdrawTL</code> is implemented and works.	<input checked="" type="checkbox"/>		

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 getlthOwnedTicketNo 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.			

We did not test our smart contract with more than 20 addresses since hardhat provided us with default 20 addresses. We tried to increase the number to 200 however due to some operational problems we failed to try.

6. Conclusion

We have completed the tasks successfully. This blockchain application makes distributed autonomous lottery possible.