# BCOLN 2019 – Challenge Task Report

## **Introduction**

In the course of the module Blockchain and Overlay Networks in 2019 the students were asked to implement a challenge task. This should be solved in teams and then presented. The theme of this year's task was to deploy a lottery as a decentralized app (dApp) on the Ethereum Blockchain. The lottery system should determine the random number with the help of an oracle, which like the lottery is also implemented as a Smart Contract.

Lottery systems have been in existence for a very long time and the integrity and correctness of a central authority (lottery organizer) must always be trusted. Since ever larger sums are handled thereby at least the temptation is given to try to take influence on the drawn numbers. Would it not be a good idea at this point to carry out the lottery on the blockchain and thus ensure a decentralized and in the best case fraud-proof system?

### **Requirements**

The system requirements essentially comprise the four points listed below:

1. The lottery must be fully functional, produce the output in graphical or textual form, and be implemented using Smart Contracts only. In addition, the random number must be provided by an Oracle Contract.
2. The oracle must be implemented as a separate Smart Contract. Thereby it does not matter whether the number is generated on the blockchain or externally.
3. The contract must automatically distribute the winnings to the winner, divide the winnings if several players have chosen the right number or cumulate the winnings for the next lottery if no winner is determined.
4. The contract and its functionality must be documented in a self-contained report.

The remainder of this report is structured as follows. In section 1.2 a certain background knowledge about the oracle is given and which possibilities exist to generate a random number.

Section 1.3 deals with the design decisions made, which are then explained in more detail in section 1.4, the implementation. Finally, a summary is presented in 1.5 and the conclusions drawn.

## **Background**

Implementing a lottery system in a "normal" environment may not be considered very demanding. A group of people report their numbers to a central office, the central office determines a random number and the winner receives the money. However, if we operate in the context of blockchains, there are some problems that were not considered important before. Just the basic question of what a random number is and how it is created is of great importance. The following subsections are dedicated to some approaches for generating random numbers, each of which will discuss the advantages and disadvantages.

The two main difficulties encountered are the following

1. How is a random number generated? The code in Solidity should be deterministic because it runs decentralized. A clock or something similar is not available to determine a random.
2. How does a smart contract know when the time has come to draw a winner? Again, there is no reliable time available.

Since the code should run deterministically, a generator is needed to generate the same number on multiple nodes if the code is called multiple times. This actually completely contradicts the term "random". The solution to this problem is to generate a random number once and use it on different nodes. So the code is run several times on several nodes, but it produces a unique number.

### **Why Using Random() is Not a Good Idea**

The Random function consists of the block timestamp and the block difficulty. Both variables cannot be influenced by the players, but by the miners to a certain extent. Accordingly, the use of the random() function for a lottery where unconditional independence should be guaranteed is not optimal.

In general, according to [1] it is **not advisable** to build your random on a **block variable**. This includes for example the coinbase (address of the miner of the block), the previously mentioned difficulty and timestamp as well as the block number or the gas limit. This is for the simple reason that they can be influenced by the miner.

### **Is it Better to Use The Blockhash?**

Some implementations suggest using a past blockhash, to generate a random. This is bad because an attack can create an exploit contract with the same PRNG code and then call the target contract via an internal message. The "random" numbers for the two contracts will be the same.

Some contracts suggest to use a future blockhash according to [1]. This approach is slightly better than the others and works as follows. One player makes a bet and the house stores the block number of the transaction. In a second call, the player asks the house if it can draw the winner. The house fetches the previously saved block number and generates a random one from the hash. The problem is that the blockhashes are only available from the last 256 blocks for scaling reasons. All older blockhashes will be zero. If the second call is not made within 256 blocks, the blockhash and random will be zero.

Another approach is to calculate a private seed with the random. It is true that private variables cannot be called by other contracts, but off-chain the variable can be accessed via web3.eth.getStorageAt() using web3. The blockchain should therefore not be used to store secrets in plaintext.

### **Maybe Using an External Service?**

Another possibility is to use an external service via a URL connector. This can be used, for example, to request a number from Random.Org. However, this is somewhat in keeping with the meaning of the blockchain, since it again relies on a central location. If this risk is considered to be negligible, it is still a good possibility.

### **All good things come in four**

A promising approach, the commit-reveal approach, as the name implies consists of two different phases. In a first phase, the participants commit a cryptographically secured secret in the form of a hash. In the second phase, all parties reveal their secrets in plain text. These will then be compared with the hash and if they are correct, they are merged with the other secrets with an XOR operation and then used as a seed for a random value.

This approach should logically not be based on a single instance and should hash the user's secrets together with their address to ensure that only the actual owner of a secret can reveal it. If a secret is not revealed, no random number will be generated and a new attempt will be made. To avoid that someone does not reveal, it can be added that each commit will cost an amount and this amount will only be refunded to the user if he revealed his secret.

## **Design**

In the course of implementing the Challenge Task, we decided to implement the Commit Reveal approach, as we believe it is the most promising method.

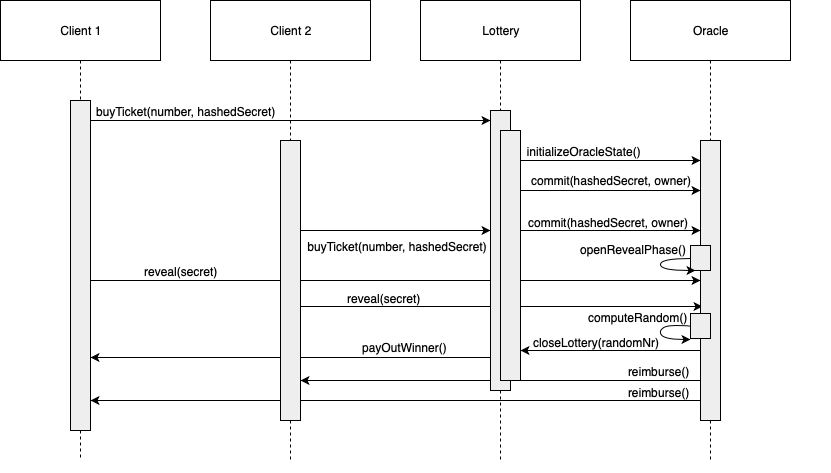
The general procedure of the lottery is as follows. The lottery is opened, which at the same time initialize the state in the oracle. After that all participants of the lottery can buy an arbitrary number of tickets and commit the hash values of their secrets. As soon as there are enough players, the players, which committed something, will have to reveal their secrets. If all players have revealed in a certain time interval, the random number will be calculated and the lottery will be dissolved according to the rules of the game. If more than one person wins, the prize is split. If only one person wins, this person receives the entire pot. If no one wins, the pot is taken to the next round. Fig. [1] shows the sequence diagram of the lottery. It models the interaction between two different clients, the lottery and the oracle. In a first step, a client buys a ticket. This triggers the lottery to initialize the state of a new oracle to find a random number. While buying the ticket also the first hashed secret has been committed by the user. Afterwards for each client buying a ticket, a hash is added to the state of the oracle.  This is possible as long as the oracle state is open for new commits. When enough commits are gathered, the oracle state is set to reveal mode, and all clients that committed a secret now need to reveal it, in order for the oracle to produce a random number. When everything worked out and all secrets are revealed, a random number is generated and the function payOutWinner() is called, which pays the jackpot to the winner(s). Finally, each user, who revealed his/her secret, will get back the cost for committing a hashed secret.

Fig 1. Sequence diagram of the lottery

## **Implementation**

The development of the Smart Contracts was done with the help of Truffle (5.0.18) and Ganache (v1.3.1). The tests were all written with Web3js as this is included with truffle and it offers many more possibilities than development and testing in Remix with plain solidity does. The class diagram shown in fig. [2] is used to explain the components of the Challenge Task in more detail.

### **Lottery**

The lottery contract is the point of contact for players of the lottery. On the one hand it offers the possibility to buy a ticket. It can be decided by the player, whether with or without commitment of a secret, since it should be left to each user whether he would like to contribute to the improvement of the random number or not (he loses his own money if others collude). On the other hand, the contract controls the determination of the winner based on the number provided by the oracle and the associated payout of the jackpot, as well as auxiliary functions such as the refund of the ticket price if the lottery cannot be carried out.

### **Oracle**

The oracle is responsible for determining a random number. It does this using the commit/reveal approach. First a new oracle state is initialized, where the minimum required number of secrets, the minimum amount to be deposited per commit and the modulo variable, which finally defines the number range are defined. The final goal of the oracle is to generate a random output.

Once the oracle state is initialized, participants can send their hashed secrets to the oracle. If there are enough hashes, the contract changes to the reveal phase, in which all participants who have made a commit have to reveal their secrets. These secrets are then merged by XOR operation. Finally, a number is generated from this value and the modulo determined at the beginning. The oracle also offers some private help functions, such as resetting the participants or refunding the amounts under consideration of who has revealed and who has not (or in the best case all have revealed and all get the money back).

### **Deployment of the Contracts**

Fig 2. Class diagram of the lottery smart contract

Lottery or oracle specific parameters, which can differ between different lottery setups, are defined in the deployment of the contracts.

For the deployment of the lottery contract the maximum possible number to pick for a ticket, the price of a ticket and the cost for the commitment of a secret has to be defined. Additionally, the address of the deployed oracle contract needs to be given, that the lottery can connect to the oracle. As a last parameter, the modulo of the oracle to calculate the random winning number has to be set. For a lottery, which makes sense, the modulo should be one higher than the maximum possible number and could be derived from this parameter. Otherwise it makes no sense when a number, which is higher than the maximal possible winning number from the oracle, could be picked or vice versa, if the final winning number could not be picked, because it is too high. However, for testing cases we need to have a predictable, deterministic, “random” number and because of that we need to set the modulo equal to 1 for test cases, that we always get 0 as winning number. The usage of this deterministic number will be explained in the next section. For the deployment of the oracle contract no additional parameters are needed, because the variables of the oracle are initialized by a call from the lottery, when the first ticket is bought. The needed parameters like the cost for a commit can be set there. The parameters of the lottery contract can be set in a separate deployment file called “2\_deploy\_contract”.

## **Evaluation & Conclusion**

The evaluation was carried out using Web3 tests. The main focus was on the functionality of the contract and the different outputs. The test cases each include the following options:

* No one wins pot in the next round
* Several people win, pot is split
* One person wins, pot is paid to that person.

Since it is relatively difficult to calculate the effective amount of ether in an account (gas must be included when an action is executed) we had to make an approximation to determine the amount of money paid out and paid in. The tests were divided into separate test files for the lottery and oracle contract. In addition, a demo test was written for the presentation of the results to simulate the three scenarios mentioned above (no winner, one winner, several winners). For the test cases we had to guarantee constant "random numbers" and thus to reproduce the interactions in the lottery, that the assertions always stay true. To do this the modulo parameter is set to 1 in the deployment process, that the random winning number will always be 0 as mentioned in the previous section. The tests only pass if the modulo is 1, because otherwise the assertions are false. To get other numbers than 0 as winning number, the modulo need to be set one higher than the maximum number parameter in the deployment file as mentioned before.

Finally, it can be concluded that the possibility of conducting a lottery on the blockchain is somewhat cumbersome due to the lack of a standard random function. The commit-reveal approach works, but after setting it takes a relatively long time until a random value is output and only works if all parties cooperate, which cannot always be guaranteed. It also requires more effort from the individual participants. You don't have to buy a ticket just once, as usual, but you also have to take the reveal step, which is an extra step. But besides using an external service like random.org (which is perceived as somewhat suboptimal by the writer as it is a centralized place) no other method was found that offers similar security as the commit reveal approach. It remains to be seen if there will be a development or if this state of "missing randomness" will remain.

## **Interface**

To interact with the contracts on the chain, 3 methods are available. The first one, being the truffle console, where instances of the contracts can be retrieved or deployed manually. By using the contracts’ methods directly, tickets for the lottery can be bought and the secrets can be revealed. The second method makes use of a JavaScript CLI, where the game can be played using a simplified Interface that automatically generates random secrets for the bought tickets. The third method is an NPM and Vue.js based webapp that offers the same functionality as the CLI but as a graphical user interface from a web browser. All accounts on the local blockchain are displayed and can participate in the lottery. Unfortunately, while displaying blockchain and contract data was possible, using the contract’s methods was not possible due to some unidentifiable error on web3.js’ side.

## **References**

[1] Predicting Random Numbers in Ethereum Smart Contracts, <https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620>, Online, Accessed last: 13.05.2019