Report

High level description of the smart contract

This is the description of the version 1.1 of the Lottery contract. Some minor changes may apply with the other versions of the contract, but the main working principles remain the same.

The smart contract emulates a lottery, with a fixed threshold or pot (amount of wei of the prize) and rate, which arbitrary establishes the value of our lottery tokens (or tickets) in wei. For this study it will be assumed that $rate \in \mathbb{N}_{>0}$.With the purpose of storing the amount of wei and tokens (equivalent to the amount of wei) deposited in the contract, two private state variables are created, *weiRaised* and *tokensRaised*. Two mappings are used (which can be understood as the common *hashmap* data structure) to relate the addresses (or users) which buy tokens for the lottery and take account of the amount of tokens each one has :

- A *tokenBuyers* map, which relates the index of a buyer and the correspondent address.
- A *tokenBalances* map, whose keys are going to be the addresses of the token holders and the values the amount of tokens each one has.

I have implemented a third mapping *weiRefunds* which relates addresses with their wei refunds they are entitled of. This data structure stores the wei the users can claim to be transfered, being this wei amount achieved in two ways:

- If the address trespass the threshold while buying tokens (more on this later).
- If the address is the holder of the winner token number.

The public *winnerAddress* state variable stores the address value of the winner of the last performed lottery, so the participants can have feedback of the results. In the case that no lottery has been performed yet, it will point to the empty address 0x0.

Furthermore, two unsigned integer variables called *firstBuyer* and *lastBuyer* will determine the address index range of each lottery, which will permit having an unconstrained number of different lotteries within the same contract without having to delete the store data of past rounds.

Besides of that, the public state variable *lotteryFinished* will store a boolean to account if the current lottery has finished (i.e.,if the wei threshold deposit has been reached, the lottery has been ran and the winner address has been found). It avoids that a malicious user could try to run the lottery more than once or call it when a winner address has been already found.

Now that the contract state variables have been explained, a high level description of the contract functions will be performed.

After the contract has being created, the addresses can get lottery tokens exchanging them with wei via the *buyTokens()* function. The input is going to be validated, to check if the message actually contains some ether, the *threshold* has not yet been reached and the lottery has not finished. After that, if the amount of wei of the message plus the wei already raised surpass the threshold, the difference will be stored as a refund in wei. The next step will be to calculate the number of tokens the participant is going to be issued for the amount of wei (with the refund already substracted). If it is the first time the address participates in **this round**, the buyer index will be increased. Finally, the amount of tokens are stored in the participant's balance.

Assuming that many participants have already engaged in the lottery, the threshold has been reached and the lottery has not yet been performd, one of the user will call the*runLottery()* function. This function makes an internal call to the function *calculateWinner*. This is an internal **constant** function, which constant meaning not writing or updating any of the state variables of the contract. It first computes the number of the last mined block and then computes the hash of the block, which is going to be typecasted to an unsigned integer of 256 bits. After that, this number is going to be modularized to the threshold and returned to the local variable of the *runLottery()* function, *winnerNumber*. After that, a for loop is going to iterate through all the indexes of the participants of the lottery, which each of them have a determined amount of tokens. If the *winnerNumber* is in the range of tokens hold by a participant, its address is set to be the *winnerAddress* and the *lotteryFinished* state variable is set to true. After each iteration, the balances of tokens of each participant address is cleared. When this loop has finised, the prize is updated to the mapping of *weiRefunds* and then all the variables related to the state of the lottery are updated so a new lottery can start again.

Finally, the external fuction *untrusted_withdrawRefund()* can be called by the users of the lottery to claim for their refunds, stored in the *weiRefunds* map. The first step is storing the refund of the caller in a local variable, updating their map value to zero and then transfering the amount of ether they are accredited to. It should be noted that in this case it is used the *transfer* function, which it has a implicit assertion, which will revert the action in case the send call fails[1]. The principle behind this mechanism is called **pull over push payments**, which will be later discussed in later sections.

Analysis of how a winning stakeholder is selected

In this section, it will be described and analyzed how a *winnerAddress* is selected. With this purpose, it will be exposed each step chronologically in the logic of the execution of the smart contract, ressembling a debbuger.

The first step will be to calculate the token winner number, assigning to *winnerNumber* the return value of the function *calculateWinner()*. This function will retrieve the number of the last mined block, get the hash of it and typecast the hexadecimal value of the hash in the uint format of 256 bits, to provide randomness to the number selected. To transform this "random" number in our desired range we just apply to it the modulo of the number of tokens issued.

While we assume that the random number obtained with the hash of the block follows a random distribution $Y \epsilon \{0, ..., n-1\}$ (where $n = 2^{256}$), when we map this element to our desired range with the modulo operation, we are introducing a deviation from the uniform distribution, commonly known as **modulo bias**. We need to compute how far is our distribution $f(y) \epsilon \{0, ..., m-1\}$ (where $m = threshold$ in the Lottery) from the uniform distribution for the same range X, e.g the statistical distance of both distributions), with :

$$f(s) := s \cdot mod m \tag{1}$$

From [2,Chapter8], we can proceed as follows:

We need to compare the distribution f(Y) with the normal distribution in the same range. Let Z be uniformly distributed over $\{0, ..., qm-1\}$ with $n = qm - r$, where $0 \leqslant r < m$ s.t. $q = [n/m]$.

With this setting of Z, we avoid completely the modulo bias for Z, having all the elements in its distribution the same number of images when applying $f(Z)$.

Taking in account that the statistical distance depends only on the distribution of the random variables,we have:

$$\Delta[X; f(Y)] = \Delta[f(Z); f(Y)] \leqslant \Delta[Z; Y] \tag{2}$$

As we have seen:

$$\Delta[Z; Y] = r/qm < 1/q \leqslant m/n \tag{3}$$

So the upper bound of the statistical distance of X and $f(Y)$ will be:

$$\Delta[X; f(Y)] < m/n \tag{4}$$

And in the particular case of the Lottery v1.1, it will be:

$$\Delta[X; f(Y)] < \frac{threshold}{2^{256}} \approx \varepsilon \tag{5}$$

being ε a negligible value.

After this analysis of how the random ticket number is achieved, we can approximate our number distribution to the uniform one, as long as *threshold* $<< n$. Now, we shall proceed with the second part of this analysis.

Once we have obtained the winner number, we wil go over all the particant's buyer indexes of this round to check who "holds" the winner token number. This **lottery protocol** implemented will be explained easily with an example, where the number of tokens or threshold is equal to 10, and the lottery is run automatically:

- P1 first buys 3 tokens. He holds t_i, with $i = (0, 1, 2)$.
- P2 then buys 5 tokens. He holds t_j, with $i = (3, 4, 5, 6, 7)$.
- P1 buys another 2 tokens. He holds now t_i, with $i = (0, 1, 2, 3, 4)$ and P2 holds t_j with $j = (5, 6, 7, 8, 9)$.
- If the winner number $w < 5$ P1 wins. Else P2 wins.

From this example we can recall two properties of the protocol:

- The token indexes a participant holds can potentially be modified during the buying tokens proccess.
- Participants can arbitrary modify these indexes. However, taking in account from our previous analysis that the distribution of the winner numbers performs as the uniform one. This can be regarded then more as a feature than as a flaw.

When the winner address is found, the *lotteryFinished* state will be changed to True. This state variable is needed to avoid any reentrancy or TOD malicious attack[3,6], so we can assure that each lottery is only run once. Furthermore, with each permutation, the token balances of each address is going to be cleared.

When the *for-loop* is ended, the pot is added to the wei account of the winner address and both the *weiRaised* and *tokensRaised* are cleared. Finally, the index range of the round of the lottery (namely *lastBuyer*)is changed to make possible the beginning of a new lottery and the lottery state is restarted.

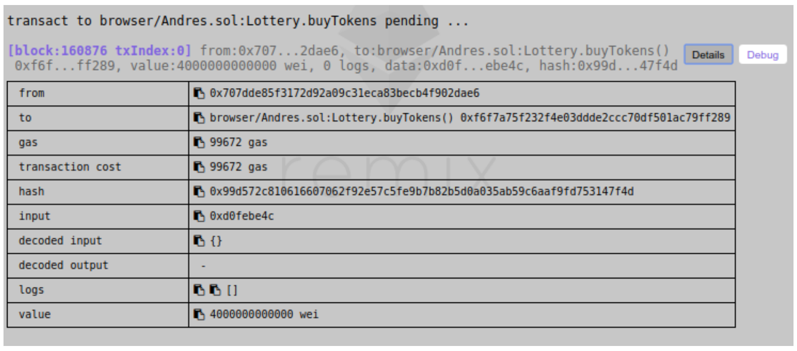
Contract deployment and engagement with other users

The Remix in-browser Solidity compiler provides an user-friendly interface which permits a fluid testing and interaction with contracts. Before the deployment of the contract in the private blockchain, an extensive testing and trial phase was performed in the JavaScript VM environment .

For the deployment of the contract in the private blockchain these steps were required:

- Creation of a MetaMask account, which is an easy-to-use Ethereum wallet. After this, I sent the account address to receive the 5 Ether needed to take part in our private blockchain.
- To interact with the blockchain, we need to connect to our Custom RPC (i.e. Remote Procedure Calls). Once we are connected, our ether balance can be seen in the MetaMasmk wallet.

- After the testing phase in the JavaScript VM environment, I deployed the contracts to the private blockchain. For that, we need to set-up first the Injected Web3 environment in Remix, while we are logged in our MetaMask account.
- Now, a contract can be created. When clicking the button *Create* in Remix, a pop-up from MetaMask will appear, in which we have to set up the gas price for the transaction and submit it. When the contract is succesfully deployed to the network, we have to copy the address where the contract was deployed, to further interact with it.
- To interact with the deployed contract, we need both the source code and the address where it was deployed, so I joined a group of 4 students to interchange the source code of our contracts and their addresses.
- From this point I interacted with my contract and other students contracts, passing values to the public or external functions of them, i.e. the public interface of the contracts. To submit the transaction, the default gas price of 20 Gwei was used.
- The first interactions that we performed were not monitored. In a later stage, we realised that no Block Explorer (as the one provided by Etherscan in the Ethereum real network) was available or currently working for our private chain. Therefore, to record a sample of the transactions that were sent to our deployed contract we ask ed the issuer of each transaction, to send each other the transaction hash from their Remix browser. Our solution was to share screenshots of the transaction details of Remix, as the one shown below.



```
transact to browser/Andres.sol:Lottery.buyTokens pending ...

[block:160876 txIndex:0] from:0x707...2dae6, to:browser/Andres.sol:Lottery.buyTokens()
 0xf6f...ff289, value:4000000000000 wei, 0 logs, data:0xd0f...ebe4c, hash:0x99d...47f4d
```

from	0x707dde85f3172d92a09c31eca83becb4f902dae6
to	browser/Andres.sol:Lottery.buyTokens() 0xf6f7a75f232f4e03ddde2ccc70df501ac79ff289
gas	99672 gas
transaction cost	99672 gas
hash	0x99d572c810616607062f92e57c5fe9b7b82b5d0a035ab59c6aaf9fd753147f4d
input	0xd0febe4c
decoded input	{}
decoded output	-
logs	[]
value	4000000000000 wei

Figure 1: One of the transaction with version 1.1

It should be noted that the contract I deployed for the demo interactions with other peers of my group is the Lottery.sol version 1.0. The main difference between the v1.0 and the v1.1 (*see Appendix* is that in the first one many state variables and functions are set up to public, so my class peers could easily check and monitor the evolution of the state of the lottery. It also has a much smaller pot, so the users could spend some ether in my contract and play with it without worrying of running out of balances. On the other hand, the version 1.1 provides a much more opaque interface, which would be suited for a more real application and not for learning purposes as the 1.0 (the last days I deployed the v1.1 contract too, and a lottery was run with this version as well). The v1.1 has also a bigger pot, which is more reasonable taking in account the proportion of the execution costs to interact with the lottery, as the pot of 1.0 (10000 wei) is already less than the transaction cost base of 21000. With this conditions, it is obvious that no participant would be incentivized in the Main net conditions, where ether has real value.

While the other versions (v1.2,v1.3 and v1.4) were tested in the JavaScript VM, they were not deployed in the private blockchain. The reason for this is that to interact with a contract, both the address where it was deployed and its source code were needed.Also, the pot of the new versions was increased, so that could make other classmates potentially reluctant to interact with it (or the threshold to be reached). Furthermore, I think that the working principles of the private blockchain have been tried and succesfully validated with the deployment of two of the versions.

Usage of Gas

First of all, it should be made clear the distinction between **transaction costs** and **execution costs**. The first ones are related with the cost of sending the smart contract source code to the Ethereum blockchain. In contrast, the latter ones are defined by the actual execution costs of the Ethereum Virtual Machine (EVM)[4] while interpreting the code of the contract, so these are the ones which we are going to be most interested on. However, the transaction costs are also going to be taken in account for the sake of completeness.

For that reason, a table which shows both the transaction and execution costs is going to be showed. In this table, the gas costs of a contract creation and the purchase of tokens are going to be compared. For the latter case, we have to take in account these possible scenarios:

- *buyTokens(msg.value) A* $\rightarrow$ This is the first purchase of tokens made in this round of the lottery by any player and $msg.value < threshold$.
- *buyTokens(msg.value) B* $\rightarrow$ This is not the first purchase of tokens made in this round of the lottery but it is the first purchase made by this player and $msg.value + weiRaised < threshold$.
- *buyTokens(msg.value) C* $\rightarrow$ This is not the first purchase of tokens made in this round of the lottery but it is the first purchase made by this player and $msg.value + weiRaised > threshold$.
- *buyTokens(msg.value) D* $\rightarrow$ This is not the first purchase of tokens made in this round by this player and $msg.value + weiRaised < threshold$.
- *buyTokens(msg.value) E* $\rightarrow$ This is not the first purchase of tokens made in this round by this player and $msg.value + weiRaised > threshold$.
- *buyTokens(msg.value) F* $\rightarrow$ The threshold is already reached, so the payment is reverted.

It should be also noted that the *msg.value* **did not alter the gas costs** while performing the token purchase, so it can be assumed that *msg.value* is an arbitrary value of wei which fulfills the conditions stated for each of the possible scenarios.

Contract interaction	TX_{cost}	$Exec_{cost})$
create Contract	560241	387885
buyTokens() A	124116	102844
buyTokens() B	79116	57844
buyTokens() C	99672	78400
buyTokens() D	38335	17063
buyTokens() E	58891	37619
buyTokens() F	21863	591

Table 1: Gas costs of interactions with Lottery(v1.1)

To give a sense of how expensive are the rest of the possible interactions with the contract in comparison with the contract deployment and the different *buyTokens()* scenarios, the next table is presented:

Contract interaction	TX_{cost}	$Exec_{cost})$
runLottery()	72234	123196
withdrawRefund()	19413	13141

Table 2: Rest of gas costs of interactions with Lottery(v1.1)

It should alse be mentioned, that to get the values of informative state variable for the participants such as *winnerAddress* or *lotteryFinished*, the gas costs only apply when called by another contract. Therefore, it will be free for the users to know who the winner was or if the lottery is running and accepting transactions.

Analysis of the contract's fairness

The analysis of the fairness of the contract will be based on the results displayed in the Table 1 of the last section, focusing exclusively in the execution costs. To compare the gas costs while calling *buyTokens()* we have to focus on the 3 possible scenarios of the first tokens' purchase by an user in a determined round of a lottery. This scenarios are cases A, B and C of the latter section (*see last section for a further description.*

As we can clearly see, **the lottery is not gas fair for all the users**, being $A > C > B$ in terms of costs of gas to buy tokens.
The user who has the most expensive gas costs is the first one who buys tokens in a lottery round (A). In this case he sets to non-zero values both the *tokensRaised* and *weiRaised* state variables from a zero value (in addition to the setting of non-zero to the *tokenBalances* and *tokenBuyers* mappings, which are common for all the scenarios). The reason to this gas cost increase is that setting an storage value to non-zero from zero costs 20000 gas, in contrast to the 5000 of changing an storage value from non-zero to other non-zero value. This addresses the optmization for storage of zeroes of the EVM, incentivizing its usage in the code with the gas costs[4].

Following the same reasoning, we can easily explain why the scenario C is more expensive than B. In C, several computations inside the if-statement will produce minor increases, but the setting to non-zero of the value of *weiRefunds* for the user's address key will provide the 20000 gas increment that makes the gap with case B.

Optimization of the lottery

The development of a lottery in a decentralized system as Ethereum is a challenging problem. The approach of this study will be to create different versions of the lottery, each of them pursuing to highly optmize different characteristics of the game. It is reasonable to argue that a very secure or totally fair for the users lottery will be less gass efficient and vice-versa. Therefore, in this section, the possible variants and subsequent trade-offs are going to be compared.

Security

The security of the smart contract should be the priority and before it is not fulfilled, we cannot start to think in developing more efficient or fair systems. As mentioned earlier, the version 1.1 of the Lottery provides a more opaque interface, but both versions 1.0 and 1.1 follow the recommendations and best practices for smart contract development in Solidity [1].

The untrusted_withdrawRefund() is labelled that way to signal that this function wil call external contracts that are potentially unsafe. Also, in this function, *transfer()* is used, to prevent reentrancy, one of the most common bugs [3] in Ethereum development. The mere existence of this function and the additional mapping *weiRefunds()*, even to the cost of increasing gas usage, try to minimize the risk of fails in external calls, isolating each of the transfers for each participant, the also know **pull over push payments** [1]. It is also used the good practice of updating the state variables before any interaction with external contracts (as in lines 123-127) in v1.1. It has also been try to assure the correctness of inputs given through *require()*, and the usage of up to the date and recommended functions as *transfer* [1]. These assertions prevent our contract to reach states with an overflow vulnerability, for example.

To perform a sanity check of the lottery contract, the source code was tested with Oyente, a tool which can detect most of the common bugs analysing the EVM bytecode through techniques such as static analysis that check all possible executions of the code [5]. These were the results obtained:

EVM Code Coverage	89.3%
Callstack Depth Attack Vulnerability	False
Re-Entrancy Vulnerability	False
Assertion Failure	False
Timestamp Dependency	False
Transaction-Ordering Dependence (TOD)	False

Table 3: Oyente tool results for Lottery(v1.1)

Even though the static analysis results are just presented for the version 1.1 in this report, every smart contract code that is presented in the Appendix was required to pass all tests.

Another security issue that is worth to analyse is the use of the **block number** to calculate the winner ticket. The reason is that a miner could not publish a block, or wait to publish it, if he has interest in the lottery result. This could be an issue if the pot of the lottery was close to the rewards of adding a block to the blockchain. However, the lottery pot was deliberately chosen to be much less than this value, to completely dis-incentivize any kind of attack from a miner.

As a final remark, it should be discussed the *rate* chosen for the contract. Even though a rate was implemented to design a lottery which suits the one required by the project, the *rate* was set to 1. This could have been modified easily, but making the relation 1 to 1 wei to tokens seemed the more reasonable option. The first point is that Solidity does not support float types, so if the rate is bigger (or lesser) than 1, we easily reach scenarios where we lose precision. Having this point in mind, setting it to 1 does not present any disadvantadge, but enables the users to customize their tokens purchase at the maximum (they can even participate with 1 wei, which otherwise would not be the most smart approach, taking in account the transaction costs of the token purchase) and preserving all the precision of the system.

Gas efficiency

As we have seen in previous sections, the most expensive operation used in these contracts is writing and storing data to state variables. Especially, setting to non-zero a variable which was previously set to 0 is highly expensive. For that reason, to optimize the efficiency of gas usage, the state variables are going to be reduced as much as possible, while preserving a secure contract.

Performing this effort was rewarding, because I found out that the state variable *tokensRaised* was not actually needed for the proper functionality of the v1.1.
Furthermore, the feature of storing the refunds of the participants in a mapping was pruned.

To participate in the lottery, the msg.value added to the already *weiRaised* cannot surpass the threshold. The *weiRaised* state variable was set to public, so potential participants were able to know how is the maximum purchase permitted. Clearly, this changes impose heavy constraints in the usability of the lottery, making the proccess of buying tokens more complicate. On the other hand, the winner address will receive the prize automatically when the lottery is performed, with no need to claim his refunds.

In the following table, we can see the comparison of gas execution costs between the version 1.1 and the gas optimized version 1.2:

Contract interaction	*v*1.1	*v*1.2
create Contract	387885	308204
buyTokens() A	102844	82381
buyTokens() B	57844	52381
buyTokens() C	78400	461(reverted)
buyTokens() D	17063	11600
buyTokens() E	37619	461(reverted)
buyTokens() F	591	461

Table 4: Gas costs comparison between v1.1 and v1.2

As we can observe, the gas consumption has been reduced. However, taking in account the features that are missing in this new version, and that the gas savings are at maximum around the 20% for buying tokens, they will not have a great effect in the overall contract's gas costs. Taking everything into account, it is arguable that these gas optimizations are not effective enough to justify the trade-off made in usability and features of the lottery.

Gas fairness

In past sections, the gas fairness of the contract has been analyzed, reaching the conclusion that the contract is not fair for all the users and *buyTokens()* will vary in gas costs depending on the state of the lottery at the moment that the user perform the purchase. The main difference in gas costs was changing the *zeroness* of the state variables of the contract. If we recall the scenarios of the last section, following this reasoning we can easily overcome the difference of gas costs for the scenario A with a naive solution. The creator of the contract can carry this surplus of gas by setting the initial *weiRaised* to $\alpha > 0$ while constructing the contract, and setting the *threshold* limit to $threshold + \alpha$. The index of the buyers *firstBuyer* and *lastBuyer* can be also set to non-zero in the contructor, so the creator of the contract can carry their gas cost too.
The overcoming of the difference of gas costs between scenario B and scenario C is more difficult to solve. One possible solution, in the same line as in the version 1.2, could be to not accept payments in which $msg.value + weiRaised > threshold$. However, as we have argued before, this modification impose a too strict constraint which limits too much the interaction with the contract. A possible solution, which improves the gas fairness (while not cancelling it) is to impose a dynamic threshold. In this case, if $msg.value + weiRaised > threshold$ we set $threshold = msg.value + weiRaised$. The user who makes the last purchase and surpasses the threshold is still in disadvantadge, because his call needs to change the threshold state variable (from a non-zero value to another non-zero value, which means an increment of 5000 gas), but its position is improved from the previous arrangement. This small difference appears to be impossible to overcome with my lottery contract logic and without pruning the features of the standard lottery.

However, if we set the *weiRaised* to public, we incentivize the participants to try to purchase the exact number of tokens that are missing to fill the pot, in the case that they want to reduce their gas costs. This solution is far from perfect, owing to the intrinsic rules of the blockchain, which do not assure that the state of the contract will remain the same since the participant checked the *weiRaised* and his transaction for purchasing tokens is actually deployed in the blockchain. It also imposes the added feature that the users will not know the pot of lottery when they buy their tokens. They will just know that the pot is going to be in $range \in \{threshold, ..., 2 \cdot threshold - 1\}$. The implementation of these changes is stored in the version 1.3 of the Lottery.

Nevertheless, there is a point in the whole analysis of the fairness of the lottery (and the correct alignment of incentives derived from it) that we have not studied so far. With the implementation of the 4 first versions of the contract, somebody (or another contract) has to create the lottery, and somebody has to run it. The issue here is that we have assumed so far that the lottery contract was going to be created (actually by us) and that any of the participants would be incentivized to run it.
The second assumption can be approached as a **game-theoretic challenge**. This was one the main reasons why the pot was increased by a factor of 1000X from version 1.0 to version 1.1. With a bigger pot, and considering the scenario that the amount of participants is limited to our private blockchain, we impose that every participant (or at least most of them) will hold higher stakes in the lottery, so the possibility to win the prize will outweight the gas costs of calling *runLottery().* Even with this added feature, one can argue that a selfish participant will not call this function, expecting that any of the other lottery users will do it. If all the participants followed this behaviour, the lottery would be never run. To overcome this problem, an easy solution will be to add some kind of expiration time to the lottery after the threshold has been reached, so the players are incentivized to run it. However, another approach could be that the lottery could be only run by the creator of the contract (permitting us to explore another feature of Solidity, the **modifiers**). This solution will permit us to substract some portion of the prize and assign it to the address who deployed the contract, so it could be rewarded for creating the contract and running the lottery. It is clear that this solution is not perfect. It adds centralization and the creator is similar to a third trusted party. Nevertheless, if the reward of running the lottery has a positive net gas outcome, the creator of the lottery will be incentivized to run it. The challenging question would be setting a fair amount to reward the creator, taking in account the oscillating gas prices. An accurate answer to this issue will require a further study, and would end up with a reward value which is in perfect equilibrium with the incentives of the contract creator and the participants of the lottery. This equilibrium has a critical importance. If the reward is too low, nobody will create the contract of the lottery (or even worse, creating it and not running it). If the reward is too high, the incentives for users to participate in the lottery will decrease. As previously said, this study is out of the scope of this work. Nevertheless, a *naive* implementation, where the creator of the contract is rewarded with the 10% of the pot will be implemented, just for didactical purpose, in the version 1.4.

APPENDIX

Source code

This is the version 1.0. Wide and transparent interface. Learning, R & D and testing purposes.

```
pragma solidity ^0.4.4;


contract Lottery {

  uint constant public threshold = 100000;
  uint constant public rate = 1;
  uint public weiRaised = 0;
  uint public tokensRaised = 0;

  // For testing purposes all maps are set to public.
  mapping (uint => address) public tokenBuyers;
  mapping  (address => uint) public tokenBalances;
  mapping  (address => uint) public weiRefunds;

  address public winnerAddress;

  uint private firstBuyer = 0;
  uint private lastBuyer = 0;


  bool public lotteryFinished = false;

  function buyTokens() external payable {

    require(msg.value > 0);
    require(weiRaised < threshold);
    require(!lotteryFinished);

    uint weiAmount = msg.value;
    //Check if the threshold is surpassed wih the token sale.
    if(weiAmount + weiRaised > threshold){
      //Calculate the surpassed difference on of the threshold,which will be
          refunded.
      uint refund = weiAmount + weiRaised - threshold;
      weiRefunds[msg.sender] += refund;
      //Update the amount of wei which is going to be exchanged for tokens.
      weiAmount = weiAmount - refund;
    }

    //Calculate number of tokens to be created
    uint256 tokens = weiAmount * rate;

    //Update state of the Wei raised.
    weiRaised = weiRaised + weiAmount;

    //if new participant in this round
    if(tokenBalances[msg.sender] == 0){
      tokenBuyers[lastBuyer] = msg.sender;
      lastBuyer++;
    }

    tokenBalances[msg.sender] += tokens;

    tokensRaised = tokens + tokensRaised;

  }


  function runLottery() external{

    require(weiRaised == threshold);
    require(!lotteryFinished);


    uint winnerNumber =  calculateWinner();
    uint tokenIndex = 0;
    //iterate through all the participants in this round of the lottery
    for(uint buyerIndex = firstBuyer; buyerIndex <= lastBuyer; buyerIndex++){
        address buyer = tokenBuyers[buyerIndex];
        tokenIndex += tokenBalances[buyer];

        if( (tokenIndex > winnerNumber) && !(lotteryFinished)){
            lotteryFinished = true;
            winnerAddress = buyer;
        }
        // clear all the token balances of all players
        tokenBalances[buyer] = 0;
    }
    //write assertion for non-empty return statement

    //Update the prize for the winner!
    weiRefunds[winnerAddress] += weiRaised;
    //set the state for a new lottery
    weiRaised = 0;
    tokensRaised = 0;
    firstBuyer = lastBuyer;
    lotteryFinished = false;
  }

  function calculateWinner() public constant returns(uint) {

    uint256 blockNumber = block.number - 1;
    uint256 blockHash = uint(block.blockhash(blockNumber));

    // above I get a "random" number
    // make the random number to be in my desired range (1,1000)
    uint256 winnerNumber = (blockHash % threshold);
    return winnerNumber;
  }
  // This function is used to claim refunds and prize. PULL OVER PUSH!

  function withdrawRefund() external {
       uint refund = weiRefunds[msg.sender];
       weiRefunds[msg.sender] = 0;
       msg.sender.transfer(refund);
   }
}
```

Version 1.1. Ethereum Main Net purpose application lottery. Opaque interface. Increase of pot from v1.0. (*An improvement would be to omit the *tokensRaised* state variable, but I wanted to attach the actual source code I used for the rest of the project.)

```
pragma solidity ^0.4.4;

/**
 * @title Lottery
 * @author Andres Monteoliva Mosteiro --version 1.1
 * @dev Lottery game, with a prefixed pot where users can participate via the
 * participate the purchase of tokensRaised.
 * @author Andres Monteoliva Mosteiro
 */

contract Lottery {

  uint constant public threshold = 100000000;
  uint constant public rate = 1;

  uint private weiRaised = 0;
  uint private tokensRaised = 0;


  mapping (uint => address) private tokenBuyers;
  mapping  (address => uint) private tokenBalances;
  mapping  (address => uint) private weiRefunds;

  address public winnerAddress;

  uint256 private firstBuyer = 0;
  uint256 private lastBuyer = 0;
  bool public lotteryFinished = false;


/**
 * @dev Function to buy tokens for the Lottery. It cannot be called from
 * inside the contract.
 */

  function buyTokens() external payable {

    require(msg.value > 0);
    require(weiRaised < threshold);
    require(!lotteryFinished);

    uint256 weiAmount = msg.value;
    //Check if the threshold is surpassed wih the token sale.
    if(weiAmount + weiRaised > threshold){
      //Calculate the surpassed difference on of the threshold,which will be refunded.
      uint refund = weiAmount + weiRaised - threshold;
      weiRefunds[msg.sender] += refund;
      //Update the amount of wei which is going to be exchanged for tokens.
      weiAmount = weiAmount - refund;
    }

    //Calculate number of tokens to be created
    uint256 tokens = weiAmount * rate;
    //Update state of the Wei raised.
    weiRaised = weiRaised + weiAmount;

    if(tokenBalances[msg.sender] == 0){
      tokenBuyers[lastBuyer] = msg.sender;
      lastBuyer++;
    }

    tokenBalances[msg.sender] += tokens;
    tokensRaised = tokens + tokensRaised;

  }

/**
 * @dev Function which runs the Lottery . It is required that the threshold
 * is been reached. Only external calls.
 */
  function runLottery() external{

    require(weiRaised == threshold);
    require(!lotteryFinished);


    uint winnerNumber =  calculateWinner();
    uint tokenIndex = 0;

    for(uint256 buyerIndex = firstBuyer; buyerIndex <= lastBuyer; buyerIndex++){
          address buyer = tokenBuyers[buyerIndex];
          tokenIndex += tokenBalances[buyer];

          if( (tokenIndex > winnerNumber) && !(lotteryFinished)){
              lotteryFinished = true;
              winnerAddress = buyer;
          }
          // clear all the token balances of all players
          tokenBalances[buyer] = 0;
      }


      //Update the prize for the winner
      weiRefunds[winnerAddress] += weiRaised;
      //Clear state. New lottery ready.
      weiRaised = 0;
      tokensRaised = 0;
      firstBuyer = lastBuyer;
      lotteryFinished = false;
  }

/**
 * @dev Calculates the winner number, achieving the "randomness" from the hash
 * of the last block mined.
 * @return An uint which holds the winner number.
 */
  function calculateWinner() internal constant returns(uint) {

    uint256 blockNumber = block.number - 1;
    uint256 blockHash = uint(block.blockhash(blockNumber));

    //Retrieving a random number within the desired range.
    uint256 winnerNumber = (blockHash % threshold);
    return winnerNumber;
  }

  /**
 * @dev Function to claim refunds and prices. Untrusted- interaction with
 * untrusted contracts. Pull over push for external calls.
 */
  function untrusted_withdrawRefund() external {
       uint256 refund = weiRefunds[msg.sender];
       weiRefunds[msg.sender] = 0;
       msg.sender.transfer(refund);
   }
}
```

Lottery version 1.2 .Highly gas optimized lottery.

```
pragma solidity ^0.4.4;

/**
 * @title Lottery
 * @author Andres Monteoliva Mosteiro ---version 1.2 (Lottery with gas
    optimization)
 * @dev Lottery game, with a prefixed pot where users can participate via the
 * participate the purchase of tokensRaised.
 * @author Andres Monteoliva Mosteiro
 */

contract Lottery {

  uint constant public threshold = 100000000;
  uint constant public rate = 1;

  uint public weiRaised = 0;

  mapping (uint256 => address) private tokenBuyers;
  mapping  (address => uint256) private tokenBalances;

  address public winnerAddress;

  uint256 private firstBuyer = 0;
  uint256 private lastBuyer = 0;
  bool public lotteryFinished = false;


/**
 * @dev Function to buy tokens for the Lottery. It cannot be called from
 * inside the contract.
 */

  function buyTokens() external payable {

    require(msg.value > 0);
    require(msg.value + weiRaised <= threshold);
    require(!lotteryFinished);

    //Calculate number of tokens to be created
    uint256 tokens = msg.value * rate;
    //Update state of the Wei raised.
    weiRaised = weiRaised + msg.value;

    if(tokenBalances[msg.sender] == 0){
      tokenBuyers[lastBuyer] = msg.sender;
      lastBuyer++;
    }

    tokenBalances[msg.sender] += tokens;

  }

/**
 * @dev Function which runs the Lottery . It is required that the threshold
 * is been reached. Only external calls.
 */
  function runLottery() external{

    require(weiRaised == threshold);
    require(!lotteryFinished);


    uint winnerNumber =  calculateWinner();
    uint tokenIndex = 0;

    for(uint256 buyerIndex = firstBuyer; buyerIndex <= lastBuyer; buyerIndex++){
        address buyer = tokenBuyers[buyerIndex];
        tokenIndex += tokenBalances[buyer];

        if( (tokenIndex > winnerNumber) && !(lotteryFinished)){
            lotteryFinished = true;
            winnerAddress = buyer;
        }
        // clear all the token balances of all players
        tokenBalances[buyer] = 0;
    }

    uint256 prize = weiRaised;
    //Update state before interaction with external contracts.
    weiRaised = 0;
    //Send the prize to the winner
    winnerAddress.transfer(prize);

    firstBuyer = lastBuyer;
    lotteryFinished = false;
  }

/**
 * @dev Calculates the winner number, achieving the "randomness" from the hash
 * of the last block mined.
 * @return An uint which holds the winner number.
 */
  function calculateWinner() internal constant returns(uint) {

    uint256 blockNumber = block.number - 1;
    uint256 blockHash = uint(block.blockhash(blockNumber));

    //Retrieving a random number within the desired range.
    uint256 winnerNumber = (blockHash % threshold);
    return winnerNumber;
  }

}
```

Lottery version 1.3 .Lottery with gas fairness optimization.

```
pragma solidity ^0.4.4;

/**
 * @title Lottery
 * @author Andres Monteoliva Mosteiro ---version 1.3 Gas fairness optimization
 * @dev Lottery game, with a prefixed pot where users can participate via the
 * participate the purchase of tokensRaised.
 * @author Andres Monteoliva Mosteiro
 */

contract Lottery {


  uint constant public rate = 1;
  uint  public threshold = 100000001;

  uint private weiRaised = 1;

  mapping (uint256 => address) private tokenBuyers;
  mapping  (address => uint256) private tokenBalances;

  address public winnerAddress;

  uint256 private firstBuyer = 1;
  uint256 private lastBuyer = 1;
  bool public lotteryFinished = false;


/**
 * @dev Function to buy tokens for the Lottery. It cannot be called from
 * inside the contract.
 */

  function buyTokens() external payable {

    require(msg.value > 0);
    require(msg.value < threshold);
    require(!lotteryFinished);

    uint256 weiAmount = msg.value;
    //Check if the threshold is surpassed wih the token sale and change threshold.
    if(weiAmount + weiRaised > threshold){

      threshold = weiAmount + weiRaised;

    }

    //Calculate number of tokens to be created
    uint256 tokens = weiAmount * rate;
    //Update state of the Wei raised.
    weiRaised = weiRaised + weiAmount;

    if(tokenBalances[msg.sender] == 0){
      tokenBuyers[lastBuyer] = msg.sender;
      lastBuyer++;
    }

    tokenBalances[msg.sender] += tokens;

  }

/**
 * @dev Function which runs the Lottery . It is required that the threshold
 * is been reached. Only external calls.
 */
  function runLottery() external{

    require((weiRaised == threshold) && weiRaised > 0);
    require(!lotteryFinished);


    uint winnerNumber =  calculateWinner();
    uint tokenIndex = 0;

    for(uint256 buyerIndex = firstBuyer; buyerIndex <= lastBuyer; buyerIndex++){
        address buyer = tokenBuyers[buyerIndex];
        tokenIndex += tokenBalances[buyer];

        if( (tokenIndex > winnerNumber) && !(lotteryFinished)){
            lotteryFinished = true;
            winnerAddress = buyer;
        }
        // clear all the token balances of all players
        tokenBalances[buyer] = 0;
    }


    uint256 prize = weiRaised;
    //Update state before interaction with external contracts.
    weiRaised = 0;
    //Send the prize to the winner
    winnerAddress.transfer(prize);

    firstBuyer = lastBuyer;
    lotteryFinished = false;
  }

/**
 * @dev Calculates the winner number, achieving the "randomness" from the hash
 * of the last block mined.
 * @return An uint which holds the winner number.
 */
  function calculateWinner() internal constant returns(uint) {

    uint256 blockNumber = block.number - 1;
    uint256 blockHash = uint(block.blockhash(blockNumber));

    //Retrieving a random number within the desired range.
    uint256 winnerNumber = (blockHash % threshold);
    return winnerNumber;
  }

}
```

Lottery version 1.4. Lottery owned by the contract creator. Increase of fairness regarding the lottery creator.

```
pragma solidity ^0.4.4;

/**
 * @title Lottery
 * @author Andres Monteoliva Mosteiro --version 1.4 Owner optimization
 * @dev Lottery game, with a prefixed pot where users can participate via the
 * participate the purchase of tokensRaised.
 * @author Andres Monteoliva Mosteiro
 */

contract Lottery {

  uint constant public threshold = 100000000;
  uint constant public rate = 1;

  uint private weiRaised = 0;
  uint private tokensRaised = 0;

  address public owner = msg.sender;


  mapping (uint256 => address) private tokenBuyers;
  mapping  (address => uint256) private tokenBalances;
  mapping  (address => uint256) private weiRefunds;

  address public winnerAddress;

  uint256 private firstBuyer = 0;
  uint256 private lastBuyer = 0;
  bool public lotteryFinished = false;


/**
*@dev Modifier which  restricts a feature of the contract for all the users
*excepting the owner (creator) of the contract.
*/
  modifier onlyOwner {
      require(msg.sender == owner);
      _;
  }


/**
 * @dev Function to buy tokens for the Lottery. It cannot be called from
 * inside the contract.
 */

  function buyTokens() external payable {

    require(msg.value > 0);
    require(weiRaised < threshold);
    require(!lotteryFinished);

    uint256 weiAmount = msg.value;
    //Check if the threshold is surpassed wih the token sale.
    if(weiAmount + weiRaised > threshold){
      //Calculate the surpassed difference on of the threshold,which will be
          refunded.
      uint refund = weiAmount + weiRaised - threshold;
      weiRefunds[msg.sender] += refund;
      //Update the amount of wei which is going to be exchanged for tokens.
      weiAmount = weiAmount - refund;
    }

    //Calculate number of tokens to be created
    uint256 tokens = weiAmount * rate;
    //Update state of the Wei raised.
    weiRaised = weiRaised + weiAmount;

    if(tokenBalances[msg.sender] == 0){
      tokenBuyers[lastBuyer] = msg.sender;
      lastBuyer++;
    }

    tokenBalances[msg.sender] += tokens;
    tokensRaised = tokens + tokensRaised;

  }

/**
 * @dev Function which runs the Lottery . It is required that the threshold
 * is been reached. Only external calls.
 */
  function runLottery() external onlyOwner(){

    require(weiRaised == threshold);
    require(!lotteryFinished);


    uint winnerNumber =  calculateWinner();
    uint tokenIndex = 0;

    for(uint256 buyerIndex = firstBuyer; buyerIndex <= lastBuyer; buyerIndex++){
        address buyer = tokenBuyers[buyerIndex];
        tokenIndex += tokenBalances[buyer];

        if( (tokenIndex > winnerNumber) && !(lotteryFinished)){
            lotteryFinished = true;
            winnerAddress = buyer;
        }
        // clear all the token balances of all players
        tokenBalances[buyer] = 0;
    }


    //Update the prize for the winner
    weiRefunds[winnerAddress] += (9*weiRaised)/10;
    weiRefunds[owner]+= weiRaised/10;
    //Clear state. New lottery ready.
    weiRaised = 0;
    tokensRaised = 0;
    firstBuyer = lastBuyer;
    lotteryFinished = false;
  }

/**
 * @dev Calculates the winner number, achieving the            *"randomness" from
     the hash
 * of the last block mined.
 * @return An uint which holds the winner number.
 */
  function calculateWinner() internal constant returns(uint) {

    uint256 blockNumber = block.number - 1;
    uint256 blockHash = uint(block.blockhash(blockNumber));

```

```
        //Retrieving a random number within the desired range.
        uint256 winnerNumber = (blockHash % threshold);
        return winnerNumber;
    }

    /**
   * @dev Function to claim refunds and prices. Untrusted- interaction with
   * untrusted contracts. Pull over push for external calls.
   */
    function untrusted_withdrawRefund() external {
          uint256 refund = weiRefunds[msg.sender];
          weiRefunds[msg.sender] = 0;
          msg.sender.transfer(refund);
     }
}
```

Transactions history

Account address

Address : 0x22a32dE7633c11E0eb5A75fD39d04eA3A4F5244C

I had an issue with Metamask, so I do not have the full tx history of my account. When logging in in different DICE machines, I had to restore the password with the Seed Words. The unexpected bug was that, every time I logged in a different DICE, the transaction history vanished.

Account addresses of interacting peers

- 0x707dDe85f3172d92A09C31ecA83becb4F902Dae6
- 0x4A8cAcEcc537c0B71B8D86A11c55928e0C8663F8
- 0x14c1B930989c59e44c2172A9240cdb5AE2f153aB
- 0x281e9743edfa828e1fad6a5e79572f749d0e1b40

Contract addresses and interactions

Contract: v1.0
Address: 0xa5c7d6cd599ccf0910266340aa8abc1a43e898f0
Interactions:

- 0xea315167f5bfc2c5d36d3442ceda6cfdf5c4a6a3ae653bcf4ae76910b2126cb9
- 0x62f8b6cf96ddcca642aa6b5c72dba39ca7fa75b34d1e9b048a643353c0e56c72
- 0x01856554a69163d58c7b0ceb8a9c9f525b99f8df703e0094858669707977a8e4
- 0x161b224f985802d4e4ba62e0455ca0b5964b936e5bd3a351af753b8abb44aa5c
- 0x286994a5770c9c0654c4a3f5b4958f7b3bc93b336cdbd5d4a39bd5ab8f78cb00
- 0xe54c928680a3c64695256c0fd805302b5c1abdd8b41475e737825c4d3c15eaf1
- 0x7785ca899ac980166caf8a2e5eddd9d483946cdae532aaeb1cc101b182bec5f0

Contract: v1.1
Address: 0xf6f7a75f232f4e03ddde2ccc70df501ac79ff289

In this case, I asked my peers to take screenshots of their transaction details in the Remix Browser to overcome the lack of a blockchain explorer. I think it is a bit redundant to attach all the pictures. I hope that the first example makes clear the whole picture. It should also be noted that in the previous example, the hashes of the transactions were acquired *manually*, so it may be incomplete.

References

[1] Recommendations for Smart Contract Security in Solidity, https://consensys.github.io/smart-contract-best-practices

[2] Shoup,V. A Computational Introduction to Number Theory and Algebra.

[3] Atzei, N., Bartoletti, M., Cimoli, T. (2017, April). A Survey of Attacks on Ethereum Smart Contracts (SoK). In *International Conference on Principles of Security and Trust* (pp. 164-186). Springer, Berlin, Heidelberg.

[4] Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper, 151.* ISO 690

[5] Luu, L., Chu, D. H., Olickel, H., Saxena, P., Hobor, A. (2016, October). Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (pp. 254-269). ACM.

[6] Delmolino, K., Arnett, M., Kosba, A., Miller, A., Shi, E. (2016, February). Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security* (pp. 79-94). Springer Berlin Heidelberg.

[7] Solidity Documentation, https://solidity.readthedocs.io