



**School of Informatics, University of Edinburgh**

---

**Blockchains and Distributed Ledgers  
INFR11144**

# **Ethereum: Smart Contract Lottery**

by

Jinming Cui  
s1732097

**Informatics Coursework Report**

---

**School of Informatics**  
<http://www.informatics.ed.ac.uk/>

**November 2017**

# Ethereum: Smart Contract Lottery

Jinming Cui  
s1732097

SCHOOL of INFORMATICS  
Blockchains and Distributed Ledgers  
INFR11144  
November 2017

## 1 Introduction

"Smart Contract", first proposed by Nick Szabo in 1996, provides new ways to formalize and secure digital relationships which can be more functional compared to paper-based. Functioning the same as a program, smart contract is deployed and self-executed on blockchains, which runs with the terms of agreement between consensus peers. In this project, the smart contract is developed on Ethereum - a decentralized concurrency platform and contract is developed using solidity: a contract-oriented programming language.

This project aims to build a smart contract lottery which performs the same as real lottery system. Every user who runs lottery contract should be able to buy lottery, and get their reward if they win the game. More specifically, the contract has the following features:

1. Buyer can buy tokens from contract with ether, the exchange rate between token and ether is fixed. If someone wins, the contract will transfer entire balance to winner.
2. When the contract receives transaction from buyer, it will check if the money is divisible by exchanging rate, if not, refund the buyer with ether amount (money modular exchanging rate).
3. When receiving transaction, contract will also check if its balance will reaches threshold after receiving transaction. If yes, refund with amount (balance after transaction - threshold).
4. When balance reaching threshold, the contract would choose a winner with probability proportional to the number of tokens buyers hold.

Since hash can be seen as pseudo-random function, the contract uses hash (Sha3) of previous block to get randomness, though using blockhash to get randomness is not recommended by the online official document of solidity.

The contract has been deployed at the experimental blockchains of university of Edinburgh.

## 2 Original Design

This section would introduce the original design of the contract. The original contract aims to implement basic functions, for example users should be able to buy tokens from the contract. We assume that lottery buyer can arbitrarily enter any amount of ethers, if the amount of money can not be divided by exchanging rate, the contract will refund users. Additionally, contract will select a winner during its last transaction, after which the contract balance will reaches its threshold. More complicated factors such as contract security, fairness and gas consumption are not considered in this design. Appendix A shows the code of original contract.

## 2.1 Selling Tokens and Storing Token Records

Original contract has only 1 payable function "buyLottery()" which enabling contract to sell tokens. Storing records is achieved by using "tokens", a local mapping between uint and address. Notice every issued token has a unique index (uint), the mapping is able to store every token's index and its corresponding owner(buyer). Figure 1 gives an example of how users interact with contract and how contract saves users' buying records.

Now assuming we have 3 users who want to interact with contract, user 1 transfers  $v_1$  ethers to contract, user 2 transfers  $v_2$  and user 3 transfers  $v_3$ . We do consider situations where users transferred an arbitrary number of ethers to the contract. Then buyLottery function would judge if the transferred ethers can be divided by a pre-fixed exchange rate every time it is called. If not, the contract will refund user with excess ethers and proceed calculating the amount of tokens user are buying. Once finished, the mapping would assign  $n$  (amount of tokens calculated by buyLottery) indexes with buyers address thus buying information would be stored in the contract.

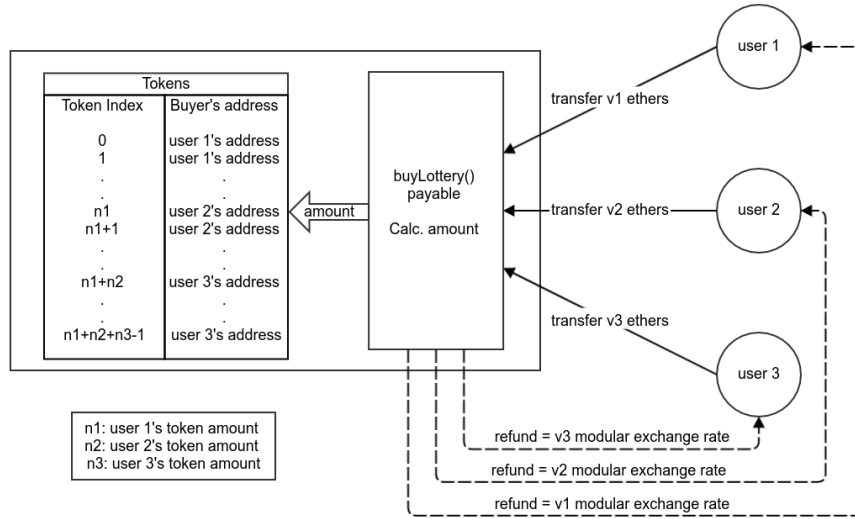


Figure 1: An example of original contract

## 2.2 Choosing winner

As it is introduced above, the contract stores buying records in a mapping. In this subsection, we would explain how winner is selected using mapping and hash. After winner selection, contract would send its balance to winner's address and clear mapping.

Following Figure 6 shows the code for winner selection. A random number ranges from (0,tokenMax-1) is selected first using hash of previous block, whoever holds token of this number would be recognized as the winner. Since we have saved buying information in mapping between token number and its buyer's address, we can simply find out winner's address by calling "tokens[winner]".

```

if(reachedThreshold){
    uint winner = uint(block.blockhash(block.number-1)) % tokenMax;
    tokenNum = 0;
    tokens[winner].transfer(threshold);
}

```

Figure 2: Code of original contract

Additionally, blockhash function in solidity returns a 32 byte hash value. Here are few assumptions of designing this algorithm:

1. Hash is uniformly distributed at range(  $0, 2^{256} - 1$  ).
2. Miners will not calculate current block hash until start calculating next block.

Since we have a mapping between token index and its owner address. We only need to choose a random token index ranging from 0 to the total number of tokens in contract, then "tokens[random token index]" would be winner's address. By using hash value of previous block, we can get a random value ranging from 0 to  $2^{256}$ . If we further divide random value by contract token amount, we will have a pseudo-random value between 0 and contract token amount. However, we still need to prove that the statistical distance of ideal distribution and this distribution is negligible.

Now let D be the ideal distribution of index selection, and the contract has n tokens in total. The probability distribution function of D is:

$$D \leftarrow \{0, 1, 2, 3, \dots, n-1\} \quad (1)$$

Let U be the distribution of index selection in this contract, and g is generator of a cyclic group of order n. the probability distribution function of U is :

$$U = \{x \leftarrow \{0, 1, 2, 3, \dots, 2^{256} - 1\} : g^x\} \quad (2)$$

The statistical distance between D and U is:

$$\Delta(D, U) = \frac{1}{2} \sum_{i=0}^{n-1} |Prob(D = i) - Prob(U = i)| \quad (3)$$

Now consider if  $2^{256}$  modular n equals to 0, which means that U is uniformly distributed as D.

$$Prob(U = i) = \frac{\frac{2^{256}}{n}}{2^{256}} = \frac{1}{n} = Prob(D = i) \quad (4)$$

which means U distribution is exactly the same as D distribution, and statistical distance between these two distributions is 0.

In this contract, we set the contract token number to be 64, which divides  $2^{256}$ . So this distribution is the same as the ideal distribution.

### 2.3 Deployment

After completing the code and testing contract on the virtual machine, we deployed the original contract onto real blockchains network.

Table 1 illustrate the gas consumption for deploying the original contract and buying tokens from the contract. We notice that gas consumption for different transactions is not a constant value, which means that transactions will charge different "Transaction fee".

Activity	Gas (transaction)	Gas (execution)	Value
Create Contract	303602	303602	0
Token Acquisition	Inconstant	Inconstant	

Table 1: Gas Consumption for Different Activities

By changing the environment in Remix to Injected Web3, we are able to interact with our deployed contract using our MetaMask account. However when considering users are buying different number of tokens in a series of transactions, the gas consumption is not a constant. Table 1 shows an example of users buying different number of tokens. it indicates that in this contract people who wants to buy more tokens will have to pay more for their transaction gas, therefore the fairness for every transaction is not satisfied.

Buyer's Address	Gas (transaction)	Gas (execution)	Tokens
0xf0F61a3f2128DeEaf42e0bb67a1025AeF3978323	374003	374003	16
0xB1514A87d9C46b62720EF50701AFd0dEA04b4A3a	374003	374003	16
0x707dDe85f3172d92A09C31ecA83becb4F902Dae6	703762	687392	32

Table 2: Gas Consumption for Transactions with Different Values

### 3 Improvements

Based on limitations of the original contract, a new version of contract has been developed to overcome those limitations. Function "chooseWinner()" has been added so that the contract can choose winner after the contract reaches its threshold, and we have some minor changes on detailed codes in order to reduce average gas consumption and make the lottery fair to every buyer. Figure 3 shows an example of improved contract.

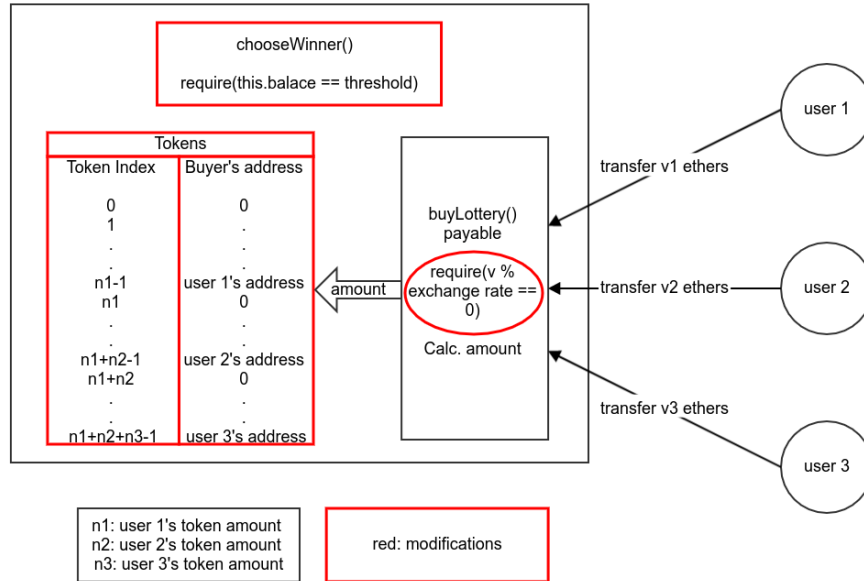


Figure 3: An example of improved contract

In this section, we made modifications mainly in 4 aspects. First `buyLottery` function

#### 3.1 Avoiding Loops in buyLottery Function

As is shown in Figure 4, the original design includes a for loop in order to store users' buying records. Apparently, `token[i]=msg.sender` would be executed "amount" times, which equals to the amount of

```

for(uint i=tokenNum;i<tokenNum+amount;i++)
    tokens[i] = msg.sender;
tokenNum = (amount +tokenNum);

```

Figure 4: Code of original contract

tokens buyer wants to buy in this transaction. "tokenNum" in this case refers to how much tokens have left in the contract.

Nevertheless, in the newer version of lottery contract some modifications have been made to avoid loop. As shown in Figure 5, for every transaction, user information will only be stored once. Thus transaction will be charged same amount of gas.

```

function buyLottery() public payable {
    require(tokenNum<=tokenMax);
    require(msg.value % exchangeRate == 0);
    tokenNum = tokenNum + msg.value / exchangeRate;
    tokens[tokenNum] = msg.sender;
}

```

Figure 5: Code of improved contract

## 3.2 Adjusting ChoosingWinner Function

Figure 6 shows some of the code of buyLottery(). If a user happens to carry out the last transaction of the contract, the gas consumption for this identical transaction is higher than other transactions for the reason that it would additionally calculate the winner and transfer contract's balance to winner.

```

if(reachedThreshold){
    uint winner = uint(block.blockhash(block.number-1)) % tokenMax;
    tokenNum = 0;
    tokens[winner].transfer(threshold);
}

```

Figure 6: Code of original contract

A solution to this is creating a new function containing all the codes. By adding a require function, the ChooseWinner function can only be called after contract balance reaching its threshold. Even though this solution can make every transaction consume same gas, it requires a user volunteers running this function to generate a winner.

## 3.3 Reducing Gas Consumption

Considering general gas consumption problems, "Ethereum: A Secure Decentralized Generalized Transaction Ledger" describes the gas consumption cost for every execution step. Since creating variables, changing variables values, adding variables etc. will all consume gases, the optimization would be to include as less code as possible in buyLottery.

Moreover, we notice that gas consumption for conditional statement consumes more gas as shown in appendix, so additionally conditional statement has been exchanged to substitutes. Figure 8 and Figure 3 shows buyLottery Function of original contract and improved contract.

```

function chooseWinner() public {
    require(tokenNum>tokenMax);
    uint winner = uint(block.blockhash(block.number-1)) % tokenMax;
    while(tokens[winner]==0)
        winner ++;
    resetLottery();
    tokens[winner].transfer(threshold);
}

```

Figure 7: Code of improved contract

```

function buyLottery() public payable returns(uint amount){
    uint refund = 0;
    bool reachedThreshold = (this.balance - msg.value % exchangeRate) >= threshold;
    if(reachedThreshold)
        refund = this.balance - threshold;
    else
        refund = msg.value % exchangeRate;
    amount = (msg.value - refund) / exchangeRate;

    for(uint i=tokenNum;i<tokenNum+amount;i++)
        tokens[i] = msg.sender;
    tokenNum = (amount +tokenNum);

    if(reachedThreshold){
        uint winner = uint(block.blockhash(block.number-1)) % tokenMax;
        tokenNum = 0;
        tokens[winner].transfer(threshold);
    }
    msg.sender.transfer(refund);
}

```

Figure 8: Code of original contract

### 3.4 Avoiding Zero Value in Transaction

Further a document explains that gas consumption for assigning a variable differs from initializing a variable. Originally, our contract have a counter tokenNum, which initially is 0. So when first buyer interacts with contract and increase tokenNum by 1, it will cost additionally 200 gas. Thus during the initialization of our contract, we set tokenNum to be 1, which solves this problem. The gas can be seen at appendix of this report.

### 3.5 Deployment & Conclusion

This section shows deployment processes and transaction history of the improved contract. Table 3 and Table 4 shows the result of improvements. After modification the improved contract does achieves fairness considering gas consumptions. Moreover the average gas consumption is also reduced by avoiding using loops, and removing redundant variables. The full transaction history can be seen at appendix.

Buyer's Address	Gas (transaction)	Gas (execution)	Tokens
0xf0F61a3f2128DeEaf42e0bb67a1025AeF3978323	48066	48066	16
0xB1514A87d9C46b62720EF50701AFd0dEA04b4A3a	48066	48066	16
0x707dDe85f3172d92A09C31ecA83becb4F902Dae6	48066	48066	32

Table 3: Gas Consumption for Transactions with Different Values

Activity	Gas (transaction)	Gas (execution)	Value
Create Contract	303602	303602	0
Token Acquisition	48094	48094	Any value

Table 4: Gas Consumption for Different Activities

## 4 Acknowledgement

This report would not have been done without the help by Jiayu Zhai and MaqinGao, who participated in the deployment test of the contract.



## A Original Contract

```
pragma solidity ^0.4.0;

//0x557a8a60ed8bf997cdcda1366c146609a4adea42

contract Lottery {
    uint tokenNum = 0;
    uint exchangeRate = 0.001 ether;
    uint tokenMax = 64;
    uint threshold = exchangeRate * tokenMax;

    mapping(uint=>address) tokens;

    function buyLottery() public payable returns(uint amount){
        uint refund = 0;
        bool reachedThreshold = (this.balance-msg.value%exchangeRate) >= threshold;
        if(reachedThreshold)
            refund = this.balance - threshold;
        else
            refund = msg.value % exchangeRate;
        amount = (msg.value - refund) / exchangeRate;

        for(uint i=tokenNum;i<tokenNum+amount;i++)
            tokens[i] = msg.sender;
        tokenNum = (amount +tokenNum);

        if(reachedThreshold){
            uint winner = uint(block.blockhash(block.number-1)) % tokenMax;
            tokenNum = 0;
            tokens[winner].transfer(threshold);
        }
        msg.sender.transfer(refund);
    }

    function tokensLeft() constant public returns(uint){
        return (tokenMax - tokenNum);
    }
}
```

## B Improved Contract

```
pragma solidity ^0.4.0;

//0x8d29a3073301f1503e1657a1fdeb364d9a346009

contract Lottery {
    uint tokenNum = 1;
    uint exchangeRate = 0.001 * 1 ether;
    uint tokenMax = 64;
    uint threshold = exchangeRate * tokenMax;

    mapping(uint => address) tokens;
```

```

function buyLottery() public payable {
    require(tokenNum<=tokenMax);
    require(msg.value % exchangeRate == 0);
    tokenNum = tokenNum + msg.value / exchangeRate;
    tokens[tokenNum] = msg.sender;
}

function chooseWinner() public {
    require(tokenNum>tokenMax);
    uint winner = uint(block.blockhash(block.number-1)) % tokenMax;
    while(tokens[winner]==0)
        winner ++;
    resetLottery();
    tokens[winner].transfer(threshold);
}

function tokensLeft() constant public returns(uint){
    return (tokenMax - tokenNum + 1);
}

function resetLottery() private{
    uint i = 0;
    while(tokens[i]!=0){
        tokens[i]=0;
        i++;
    }
    tokenNum = 1;
}
}

```

## C Transaction Screenshots

## Creating original contract

[illegible]

## 1st transaction

```
transact to browser/cui_v1.sol:Lottery.buyLottery pending ...
```

```
[block:161892 txIndex:0] from:0xf0f...78323, to:browser/cui_v1.sol:Lottery.buyLottery() 0x557...dea42, value:160000000000000000 we
i, 0 logs, data:0x8fc...cdd63, hash:0x4f5...e7fb4
```

from	0xf0f61a3f2128deea42e0bb67a1025aef3978323
to	browser/cui_v1.sol:Lottery.buyLottery() 0x557a8a60ed8bf997cdcdal366c146609a4adea4
gas	374003 gas
transaction cost	374003 gas
hash	0x4f5d24c4c03894e2b5dd84a36d90ec7ebf962ald24dafab7bacld5c67251e7fb4
input	0x8fccdd63
decoded input	{}
decoded output	-
logs	[ ]
value	160000000000000000 wei

## 2nd transaction

[block:161892 txIndex:1] from:0xb15...b4a3a, to:browser/Jamie.sol:Lottery.buyLottery() (0x557...dea42, value:1600000000 wei, 0 logs, data:0x8fc...cdd63, hash:0x812...f2c09) [Details](#) [Debug](#)

from	0xb1514a87d9c46b62720ef50701afd0dea04b4a3a
to	browser/Jamie.sol:Lottery.buyLottery() 0x557a8a60ed8bf997cdcd41366c146609a4adea4
gas	374003 gas
transaction cost	359003 gas
hash	0x812530863f947c629d5477b2482f201d7a4ec1d18e9ae61985b286763af2c09
input	0x8fccdd63
decoded input	{}
decoded output	-
logs	[ ]
value	160000000000000000 wei

## 3rd transaction

from	0x707dde85f3172d92a09c31eca83becb4f902dae6
to	browser/Lottery_v1.sol:Lottery.buyLottery() 0x557a8a60ed8bf997cdca1366c146609a4adea42
gas	703790 gas
transaction cost	687392 gas
hash	0x16953elf142f86188cae0f76b36af899a93d20178f3588dc7ad3025fb5ea57d6
input	0x8fccdd63
decoded input	{}
decoded output	-
logs	[ ]
value	320000000000000000 wei

### Creating improved contract

remix

```
buyLottery pending ...
8323, to:browser/cui_v2.sol:Lott
09...a7c25
```

```
eeaf42e0bb67a1025aef3978323  
sol:Lottery.buyLottery() 0x8d29a3073301
```

Details Debug

remix

```

tery pending ...
to:browser/Jamie2.sol:Lottery.buyLott
sh:0x545...f962d

```










```
46b62720ef50701afd0dea04b4a3a
2.sol:Lottery.buyLottery() 0x8d29a307
```

Details

remix

e85f3172d92a09c31eca83becb4f902

as remix

from	 0x707dde85f3172d92a09c31eca83becb4f902dae6
to	 browser/Lottery_v2.sol::Lottery.chooseWinner() 0x8d29a3073301f1503e1657a1fdeb364d9a346009
gas	 50580 gas
transaction cost	 46818 gas
hash	 0x317baa8ac9a7592c428a959b288b96c9f2d4301d43bac54fd278a609fa02d295
input	 0xcd38aa87
decoded input	 {}
decoded output	-
logs	  []
value	 0 wei

1.0 gas costs - Sheet1

Param	Gas price Computed	Actual	
DUP	3	3	3
SWAP	3	3	3
PUSH	3	3	3
ADD	3	3	3
MUL	5	5	5
SUB	3	3	3
DIV	5	5	5
SDIV	5	5	5
MOD	5	5	5
SMOD	5	5	5
ADDMOD	8	8	8
MULMOD	8	8	8
EXPBASE	10	10	10
EXPBYTE	10	10	10
SIGNEXTEND	5	5	5
LT	3	3	3
GT	3	3	3
SLT	3	3	3
SGT	3	3	3
EQ	3	3	3
ISZERO	3	3	3
AND	3	3	3
OR	3	3	3
XOR	3	3	3
NOT	3	3	3
BYTE	3	3	3
SHA3BASE	30	30	30
SHA3WORD	6	6	6
ECRECOVER	3000	3000	3000
SHA256BASE	60	60	60
SHA256WORD	12	12	12
RIPEMD160BASE	600	600	600
RIPEMD160WORD	120	120	120
IDENTITYBASE	15	15	15
IDENTITYWORD	3	3	3
ADDRESS	2	2	2
BALANCE	20	20	20
ORIGIN	2	2	2
CALLER	2	2	2
CALLVALUE	2	2	2
CALLDATALOAD	3	3	3
CALLDATASIZE	2	2	2
CALLDATACOPYBASE	3	3	3
CODESIZE	2	2	2
CODECOPYBASE	3	3	3
GASPRICE	2	2	2
EXTCODESIZE	20	20	20
EXTCODECOPYBASE	20	20	20
GCOPYWORD	3	3	3
BLOCKHASH	20	20	20

1.0 gas costs - Sheet1

COINBASE	2	2
TIMESTAMP	2	2
NUMBER	2	2
DIFFICULTY	2	2
GASLIMIT	2	2
POP	2	2
MLOAD	3	3
MSTORE	3	3
MSTORE8	3	3
SLOAD	50	50
STORAGEADD	20501.05604	20000
STORAGEMOD	5229.126595	5000
STORAGEKILL	-10042.80285 5000, plus 15000 refund	
JUMP	8	8
JUMPI	10	10
PC	2	2
MSIZE	2	2
GAS	2	2
JUMPDEST	1	1
GLOG	349.0623426	375
GLOGTOPIC	377.2953554	375
GLOGDATA	7.705502127	8
CREATE	31180.58924	32000
CREATEDATA	193.8991181	200
GCALL	40	40
GCALLVALUETRANSFER	6705.502127	9000
GCALLSTIPEND		2300
GCALLNEWACCOUNT	24435.08711	25000
RETURN	5	0
STOP	1	0
SUICIDE	10	0
GSUICIDEREFUND	-24435.08711 24000 refund	
MEMWORD	3	3
QUADCOEFFDIV	0.0019697933 512 (divisor)	
GTX	24762.53694	21000
GTXDATANONZERO	67.75945113	68
GTXDATAZERO	4.234965696	4