Università di Pisa

Computer Science Master Degree

# TRY Lottery in Solidity
# P2P Final term

*Author:* Andrea ZUPPOLINI

ACADEMIC YEAR 2021/2022

# 1 The Code

Since the lottery is based on the assignment of NFTs as prizes I have decided to implement 2 smart contract:

1. **TryLottery.sol**, the contract implementing all the logic of the lottery. This contract has a reference to the NFT contract.
2. **Collectbiles.sol**, the contract inheriting from the standar ERC721, where all the operations regarding NFTs are implemented.

From ERC721 the following function are used:

```
constructor(string memory _name, string memory _symbol)
ERC721(_name, _symbol) {}
```

Since the contract inherits from ERC721 it also has to call the standard's constructor passing both the name of the collection and its symbol.

```
_mint(ownerAddress, _tokenIds);
```

The minting function creates a non fungible token with the token ID given in input and sets the ownerAddress as its owner. These tokens are unique and completely different one from another since they are identified by their tokenID. If one tries to mint another token with the same ID the _exists() function of ERC721 is called and throws an error if there is alredy a token with that ID.

```
_setTokenURI(_tokenIds, tokenURI);
```

The token URI (unique resource identifier) is a sort of metadata added to the NFT to make it unique. From the user point of view the tokenURI is what characterize the NFT. For simplicity in the test I have used random words as URI of the NFTs but usually an IPFS link should be provided in order for the resource to exist in a fully distributed way.

```
transferFrom(_from, _to, tokenId);
```

This function is used to transfer the ownership of a token from an address to another. Clearly an external user can not transfer an NFT that he doesn't own. NFTs can only be transferred by their owner or approved addresses that are particular users that the token owner makes operate in its guise.

In this implementation even if the lottery manager can call the mint function whenever he wants the owner of the NFT is the smart contract of the lottery itself, in order to prevent permissions issues due when the prizes will have to be transferred. The lottery contract mines new NFTs using the owner's funds and stores them in a data structure after assigning them a class. When the Collectibles will be rewarded the Smart contract will be able to call the transferFrom function because it is the owner of the tokens.

# 2 Gas Costs

In this section we will analyze the gas cost of different function implemented inside the Try-Lottery smart contract.
With reference to the buy() function of the smart contract, the execution of this function cost **141569** gas units.

```
function buy(uint8[6] memory userNumbers) public payable
    {
    checkRound();
    require(lotteryStatus, 'Lottery is not opened');
    require(msg.value >= ticketPrice, 'Not enough Ether sent');
    require(numberOfCollectibles > soldTickets, 'Not enough Collectibles');
    tickets.push(Ticket(userNumbers, soldTickets));
    ticketOwners[soldTickets] = msg.sender;
    emit TicketBought(msg.sender, tickets[soldTickets]);
    soldTickets++;
    }
```

Following the instructions order, it receives in input an array of 8 int8 number stored in memory, so we consume **3** gas per element stored. Then the function has to perform the checkround() function and 3 require statements. The former ,assuming that its body is not executed because the round is still valid, reduces to a comparison that only requires an **SLOAD** which cost 200 gas unit. The latter, even if there is no official trace of how much does actually the require statement cost, can be estimated around the 20 units of gas.
Then we push a new struct to the dynamic array of tickets. The new Ticket created is saved first in memory and then with the push operation moved to the storage. Since **SSTORE** operations cost **20.000** units of gas per element, the most expansive operation of this function is this, using **120.000** units of gas, plus the cost of loading soldTickets from storage, **200** gas.
Another storage store is performed when we save the address of the owner by accessing the mapping, so we reach **140.000** which is nearly the total cost of the function execution. Also the emit event has a cost, since we have 2 indexed parameters the overall cost of the event should not go over 1200 gas units. We have another storage access to update the state variable soldTickets and that would cost again **20.000** gas units.
The reason why our estimate is bigger than the actual cost can be different, in first place the solidity compiler manages to do different packing operations on both arrays and small integers. Secondly when we update the soldTickets if we are changing a non-zero variable to a non-zero variable the gas cost should be lower around **5000** gas, source: Ethereum Improvements proposals.
The second function we are going to analyze is the givePrizes() function, which is responsible of rewarding the token to the winner.

```
function givePrizes(uint256 ticketNumber, uint8 class) private
    {
    uint256 class_length = collectibles[class-1].length;
    uint256 rewardTokenId = collectibles[class-1][class_lenght - 1];
    collectibles[class-1].pop();
    nftContract.transferNft(address(ticketOwners[ticketNumber]),
                            rewardTokenId);
    emit TicketRewarded(ticketOwners[ticketNumber],
                        tickets[ticketNumber], class);
    numberOfCollectibles--;
    }
```

We have 2 input parameters both stored in memory, at the cost of 3 gas each. We create then the parameter class_length which needs a storage access at the cost of **200** gas. Then make another access to retrieve the rewardTokenId so, again, 200 gas.
Popping from dynamic array has a really low cost, since it is basically setting that position to zero and updating the lenght, which, again has the cost of updating storage and depends on its previous value as stated above. The transferNft function is only a wrapper for the standard transferFrom() function built in ERC721, its cost is estimated to be around 51.000 gas units, plus the cost of accessing a memory parameter from the Collectibles NFT contract we have close to 53.000 gas units.
Emitting the event costs around 375 + 375*number of indexed variables, and then we have to add the cost of accessing 2 storage variables. The owerall cost is around 2000 gas units. The last operation is updating the state variable NumberOfCollectibles, SSTORE operation should usually cost 20.000.

# 3 Logs

The following logs have been provided to analyze the lottery:

1. **TicketBought**, whenever a user buys a ticket this event is emitted giving information on the ticket owner and the bought ticket.

2. **TicketRefunded**, when the lottery owner suddenly closes the lottery all the buyers are refunded the ticket price. This event shows the amount of Ether refunded by the lottery. Of course there could be the chance to emit this event for every single refunded buyer but this would have been more expansive in terms of gas.

3. **TicketRewarded**, whenever a ticket is lucky enough to win a prize, this event is emitted giving information on the winner address, the winning ticket, and the class of the won collectible.

4. **WinningTicketExtracted**, when the lottery draws the winning numbers it emits an even containing the ticket numbers, in order to make clear for the users wheter they are winning something or not.

# 4 Vulnerability

Smart contract can be subject to different attacks. The contract deployed here make use of randomness without accessing an external oracle but using a source of randomness internal to the blockchain itself. In particular, this is the function generating randomness:

```
    function createRandom(uint number) private returns(uint){
        return uint(keccak256(abi.encodePacked(blockhash(block.number),
                                block.timestamp,seed++))) % number;
}
```

This procedure makes use of the cryptoghaphic hash function Kekkak256 called on the parameters generated by packing together the actual block.number, the block.timestamp and a seed. All of those operation are then % number. Since the contract code is public everyone can analyze the code and find vulnerabilities. In particular someone could be able to predict which is the number ingoing to the Kekkak256 hash function and use it to predict the winning number. The seed here is a private attribute so it is not directly avaiable to external users. However, with some calculation it would be possible to predict which will be the seed value at the moment of the computation and with a delayed transaction a malignant user could manipulate the lottery, the choice made here to use it was only for the purpose of evidencing dangerous behaviour inside the contract.
Also the use of timestamp is dangerous since the miners can manipulate the timestamp to predict the contract behaviour. In this case the timestamp could be substituted by the average block mined during a week which is certainly harder to predict. In this project also the **Ownable contract** was used and the main TryLottery inherits from it. This was helpful to easily manage who should have control over the smart contract implementing the lottery. In particular I've used it to ensure that only the contract owner (the lottery manager) could perform certain operation.
When a function is declared to be **onlyOwner** every other external calls made to it are rejected is the msg.sender is not the contract owner. A good part of the facilities given from the Owner contract could be also manually implemented, for instance, checking that msg.sender == owner. However the ethereum developers encourage the use of those premade facilities to avoid common mistakes that could be made while moving though all the blockchain system.

# 5 How to run the code

All of the project was tested and deployed on remix, and the code is also available on GitHub. Please notice that some of the test execution is heavily influenced by the VM. I suggest to run them on Ganache even if it is slower. The test scripts have been written in Javascript using Ehterjs as package to communicate with the blockchain. All of those tests require an instance of the TryLottery.sol contract, in particular the **address of the contract** needs to be set in the parameter **cadd** (contract address) of the Javascript test. There are different test:

1. **LotteryFullTest.js**, this is a full test of the lottery, 60 different NFTs are minted and then 2 different addres (not the owner) buy tickets. The duration of the lottery is set to 26 blocks, since each block contains 1 transaction, we have 24 ticket transaction and 1 transaction to open the lottery. Thanks to this, the last call to checkRound() function will start the draw and award the winners. Please notice that this test is particularly heavy and the RemixIDE might be crashing. I've been testing it using a Ganache environment.

2. **LotteryTest**, this test contains a call to a function made for testing purpose. This function allows the owmer to manually set the winning ticket. This is only needed to test that Ticket are correctly awarded to the users. This test mints 20 NFTs and 3 different users buy 8 tickets. Each of those is built to be the winner for its particular class. So at the end of the test all the 8 tickets will be awarded, one per class.

3. **RefundTest**, this last test is aimed to check if the refund mechanism is working. Again NFTs are minted and tickets are bought. Then the lottery operator suddenly closes the lottery and all the money spent on the tickets are refunded to their buyers.

## 5.1 Run the code directly on Remix

The following list of instruction will be the function you ahve to call in order to manually test the lottery:

(a) opening the folder with all the project (namely with the folders contracts and test), go on remix, select the TryLottery.sol contract and compile it. After having compiled deploy the contract on your VM;

(b) set the number of block you want this lottery to last calling the function **setRoundDuration** with the desidered number of rounds. The default is 10;

(c) Opening the contract the first thing you have to do is minting some NFTs, the contract won't let you buy tickets without available rewards.Put some text in the data field of the function "mint" and call it a good number of times, each collectible is assigned to a class incrementally. For example, if you mint 3 Collectibles they will respectively belong to the classes 1,2,3.;

(d) You can now open the lottery, call the **startNewRound** function thtt will open the lottery;

(e) using the exact ticket prize of 30000000 gwei, you can call buyRandomTicket() if you intend to buy a random Ticket or simply buy() providing it a list of number in the form [1,2,3,4,5,6];

(f) You can buy as many tickets as you want (as long as you have enough collectibles for EACH CLASS of collectibles);

(g) Pressing **CheckRound()** the lottery will check if M block have been minted (you can use it to skip to the end) and will close the lottery awarding the prizes.

(h) Eventually, you can call **CloseLottery()** BEFORE the lottery is closed in order to refund all of the tickets.