CMPE 483 Sp. Top. in CMPE Blockchain Programming Spring 2018 Homework 1

Kağan Sarı
2011400207
Mahmut Uğur Taş
2017700132
Sevda Çopur
2012400054

April 16, 2018

1 Environment Setup

```
# install dependencies truffle and ganache-cli
npm install
# start ethereum testrpc on 127.0.0.1:8545
./node_modules/.bin/ganache-cli
# simulate contract for 2 rounds
# period is 50 blocks, can be changed on migrations/2_deploy_contracts.js
./node_modules/.bin/truffle test
```

2 Assumptions

- Period can be optional. We decided to define period of the lottery as the parameter period in constructor function. This way we can simulate the life cycle of the contract much faster. Period is given by deployer and must be divisible by 2 since each stage is half-period.
- At least 3 participants are required. Otherwise reward is added back to the balances of participants. They can withdraw their money afterwards.
- One ticket can win multiple times in a round. Winners are choosen by XOR'ing the revealed numbers and the last 3 numbers decide the indexes of winner tickets, but same ticket can be choosen twice or thrice. We ignored this situation because it would have disrupted the randomness in selection of winners.
- Payout function can be called implicitly, meaning that if round ends but payout function is not called and winners are not selected then it is called in checkPayout modifier when someone tries to buy ticket in the next round. Cost of the function is unpredictable because of the for loops. It would consume too much gas on a large scale. A workaround would be to block submission functions and force user to call payout function explicitly.
- Someone buys ticket in every round. This is an extreme case. For example if nobody plays the game for whole 3 rounds, then payout function should be called and the contract should iterate 3 periods. If someone tries to buy ticket in this state, her money will be returned 3 times until the contract catch up with up-to-date block.

3 Implementation

The project consists of 2 main parts. One is the solidity contract `contracts/Lottery.sol`, and the other is the javascript test file `test/lottery.js` . We have 2 dependencies defined in `package.json` , truffle and ganache-cli. The test code connects to the network defined in `truffle.js` which is the default network that is run by `ganache-cli` . The other files belong to truffle framework and `migrations/2_deploy_contracts.js` is used to deploy the contract to the connected network with period of 50 blocks.

After the contract is deployed to the network, it immediately starts to run in submission stage. Only available function is `purchaseTicket` during this stage. Stage control is performed by modifiers like `inSubmission` and `inReveal` . Buyers send ticket fee in ether to this function together with their hash. Hashes are calculated outside of the blockchain, giving the concatenation of random number generated by user and the address of the user as input to `keccak` (`sha3`) function . For each puchase, a new instance of `Ticket` struct is created having relevant properties of the ticket. Tickets are stored in state variable as mapping with hashes as keys. Hashes are also stored in another array to be used later to delete mapping values starting a new round. Sent values are also added directly to the `reward` as a state variable.

Contract automatically switches to reveal state after half-period of blocks mined. Submission is not allowed this time. Ticket owners submit the random numbers they generated when bought their tickets calling `submitNumber` function. Contract validates the number hashing it together with the address of the sender of the message, sets necessary properties of the ticket like `number` and `submitted` flag. Then the ticket is pushed to another state variable array `tickets` which contains submitted ones.

After period ends, a new round starts and contract waits for `payout` function to be called. If the function is not called but a user tries to buy a new ticket in the new round, then `payout` function is called by `checkPayout` modifier. `payout` function checks the submitted tickets. If the length is less than 3, money is shared among the ones who submitted their numbers accordingly. Else, all numbers are XOR'ed and the last 3 random numbers correspond to the winner tickets. The XOR'ed number modulus the length of the submitted tickets array become the index of the winning ticket in the array. Winner numbers are also hold in another state variable `winnerNumbers` for further use. Their rewards are calculated according to the type of their tickets and their rank. Prizes are added to the values in `balances` state variable accordingly. After all values related to the reward are set, `restart` function is called. It starts a new round deleting used state variables. Thereafter, contract is ready to accept users to purchase new tickets and so on

3.1 Structure

In this section, we will introduce the structures which we defined in our smart contract. Actually, lottery contract has two important roles by design, Ticket and Gambler who owns the ticket. The design can be compound of both structures, but we estimated that ticket-based lottery contract will cost less amount of gas when we compared with other options. Therefore, The main component of our lottery contract is the *Ticket* which is defined as in (Listing 1)

```
1     struct Ticket {
2         address owner;
3         uint hash;
4         TicketType ticketType;
5         uint number;
6         bool submitted;
7     }
```

Listing 1: Ticket

- **owner:** Address of the wallet who owns the ticket.
- **hash:** Keccak256 sum belong to concatenation of number and owner's address.
- **ticketType:** Type of the ticket which can get 3 possible kinds as defined in (Listing 2).
- **number:** Candidate number in the lottery.
- **submitted:** A boolean field that is used for checking if the ticket is submitted to lottery or not in reveal stage.

We have used *enum* structure storing ticket types because it is known that it contains finite set of values. Instead of storing them as string, we thought that we will eliminate considerable storage cost. The definition of *enum* is as following :

```
1     enum TicketType { Full, Half, Quarter }
```

Listing 2: Types of Ticket

3.2 Global Storage

```
1     Ticket[] public tickets;
2     mapping (uint => Ticket) public ticketMap;
3
4     uint public period;
5     uint public lastStartBlock;
6     uint public reward;
7
8     mapping(address => uint) public balances;
9     uint[] public hashes;
10    uint[3] public winnerNumbers;
```

Listing 3: Global Variables in Lottery Contract

- **tickets:** stores the submitted ticket numbers during reveal stage.
- **ticketMap:** stores purchased tickets in addition to submitted ones.
- **period:** stores how many block the lottery takes
- **lastStartBlock:** the block number which last lottery started
- **reward:** total prize in pool which corresponds to M in description.
- **balances:** only stores the balance of the winners of lottery.
- **hashes:** stores the hashes of senders.
- **winnerNumbers:** statically defined array because only 3 ticket wins each round.

3.3 Modifiers

The use of function modifiers allows to make code highly readable at the same time setting restrictions such that executing the function if only called by contract owner.

```
1     modifier inSubmission();
2     modifier inReveal();
3     modifier afterReveal();
4     modifier hasMoney();
5     modifier checkPayout();
```

Listing 4: Modifiers in Lotter Contract

- **inSubmission:** controls the stage if it is in first half of period
- **inReveal:** controls the stage if it is in second half of period
- **afterReveal:** controls the current round has ended or not
- **hasMoney:** controls whether the sender has ether in balance or not.
- **checkPayout:** if round ended but restart function not called, then restart, payout and continue

3.4 Interface

```
1     function Lottery(uint _period) public ;
2     function keccak(uint number) public constant returns(uint);
3     function getTicketType() private constant returns(TicketType);
4     function getTicketPrice(TicketType ticketType) private pure returns(
          uint);
5     function purchaseTicket(uint hash) public payable checkPayout
          inSubmission;
6     function submitNumber(uint number) public checkPayout inReveal;
7     function restart() private ;
8     function payout() public afterReveal returns(uint[3]);
9     function withdrawal() public hasMoney;
10    function() public payable;
```

Listing 5: Interface of the contract

- **Lottery:** constructor which sets period and lastblocknumber.
- **keccak:** returns sha3 of (number+addressOfSender)
- **getTicketType:** returns ticket type by looking msg.value.
- **getTicketPrice:** returns ticket price by checking ticket type.
- **purchaseTicket:** generates new ticket with given hash, pushes it to mapping and adds ticketPrice to reward.
- **submitNumber:** validates with hash and add relevant ticket to submitted tickets.

- **restart:** reset the state variables for a new round.
- **payout:** applies XOR-distance to evaluate winner tickets and assign balances to winners.
- **withdrawal:** sends reward to winner account at once.
- **No name function:** It's a fallback function that accepts ether to be sent to a contract by accident.

4 Test

Test script `lottery.js` runs the lottery 2 times successively. Randomly generated tickets are submitted with their hashes to the contract. Invalid tickets, invalid ticket prices, reveal and payout functions are tested to throw error here. Total reward is expected to equal to sum of ticket fees. Then EVM is mined until half of the period with `eve_mine` RPC method which is only available in ganache (formerly ethereum-testrpc). Then the numbers of previously generated tickets are submitted. Invalid numbers, trial to reveal again, submission and payout functions are tested to throw errors here. After payout is done, Calculation of rewards and balances of winner accounts are tested with corresponding values computed in javascript.

Thereafter, same operations are performed for the subsequent round in order to validate there is no error on clearing contract state.Example outputs of `ganache-cli` and `truffle test` can be found in `example-outputs` folder