Introduction

The goal of the assignment was to implement the functions `mayor_or_sayonara` and `open_envelope`, providing some gas estimations/measurements of all the functions of the contract and security considerations about `mayor_or_sayonara` and `compute_envelope`. The assignment was done using the Truffle framework and Ganache.

To implement the two functions I introduced two more state variables, that represent the counter of the yay votes and nay votes, a boolean variable and a new modifier (`protocolDone`) to ensure that the function `mayor_or_sayonara` is called only once and thus prevent the flooding of events.

To test and get the evaluations I used the `ganache-cli` to get more than 10 accounts and so to test the contract with different quorums. To run both tests, first remember to run `ganache-cli -a 101` because the last test will use 100 voters (+ 1 is the escrow address).

I provided a base test included in the file `test_base.js` to check the correctness of the functions, emitted events, modifiers and require statements. To assert the correctness of emitted events I used one npm dependency called truffle-assertions. It can be installed with the command:

```
npm install truffle-assertions
```

The contract was then tested (file `test_quorums.js`) with four different quorums (25, 50, 75 and 100 voters) and with random doblon and souls. I also tested the case in which there is a quorum of 25 positive doblon and the case in which there are 25 negative doblons.

Gas evaluation

To get the gas evaluation I used the function `estimateGas` provided by the framework and, when possible, I got the gas written inside the receipt. The function `estimateGas` was only used to estimate the gas cost of the function `computeEnvelope`.

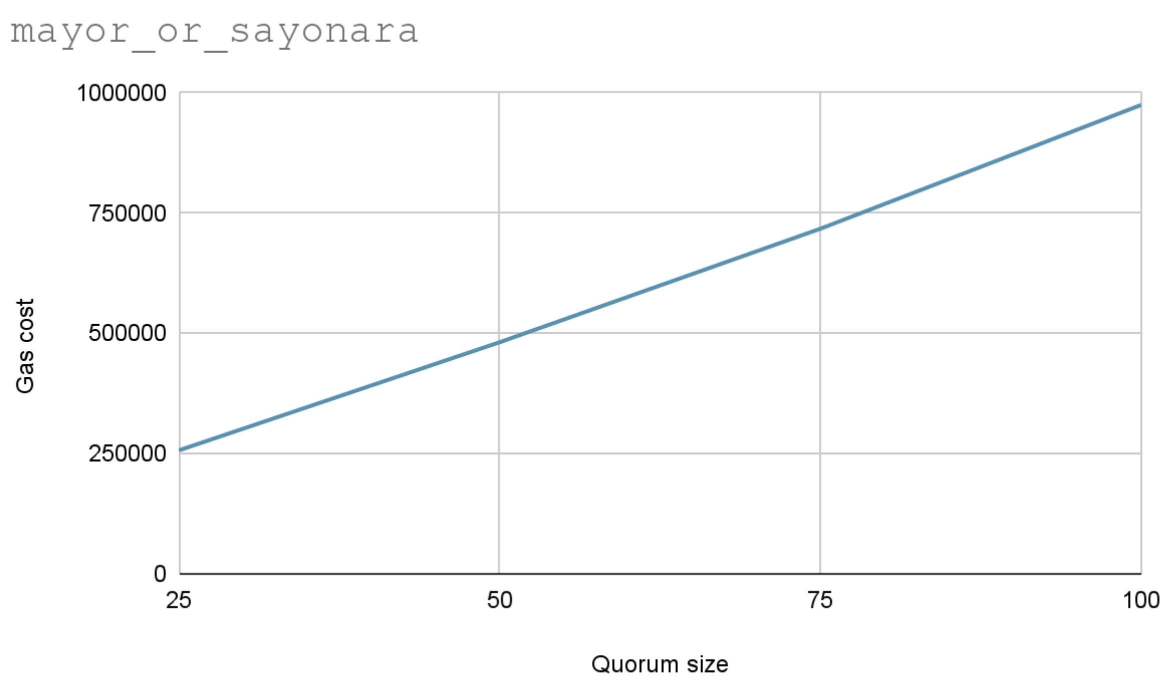
To estimate the constructor I simply created another contract that acts as a factory. It has a simple function that will create the Mayor contract and then I examined the receipt of this function to get the gas used.

All the results are written inside a text file called `output_quorums.txt`.

As we can see the `constructor`, `cast_envelope`, `compute_envelope` and `open_envelope` have a gas consumption that does not change with the quorum size. There are some variations in `cast_envelope` and `compute_envelope` that I think are due to the different parameters of the Keccak algorithm.

The `open_envelope` has two sensibly different costs. This is strange because, in both the branches of the `if` statement that checks the `_doblon`, the commands are the same: read from a variable, sum and an increment. This can be a problem because an attacker can look at the gas cost and derive that the sender has cast a positive or negative vote.

The interesting part is about the `mayor_or_sayonara` function that has a loop whose execution depends on the size of the voters array. The gas cost grows linearly with the quorum size.



The following table contains all the gas costs of the functions defined and implemented in the smart contract.

`constructor`	1.206.529
`cast_envelope`	(min 53.011, max 53.023)
`compute_envelope`	(min 22.992, max 23.004)
`open_envelope` (positive vote)	(min 114.913, max 159.913)
`open_envelope` (negative vote)	(min 95.691, max 140.691)
`mayor_or_sayonara` (quorum of 25)	~256.700
`mayor_or_sayonara` (quorum of 50)	~481.460
`mayor_or_sayonara` (quorum of 75)	~717.460
`mayor_or_sayonara` (quorum of 100)	~974.810

Security issue about `mayor_or_sayonara`

The main issue about `mayor_or_sayonara` is due to reentrancy, if the state is modified after the transfer function an attacker could invoke it again before its termination. To avoid this, I simply updated the state variables before the transfer and, mainly for avoiding the emission of useless events, I used another state variable and modifier to guarantee that the function is called only once.

Another issue, not related to security but scalability, is the gas limit of a block. If the quorum is too big we maybe could not confirm or kick the mayor because the gas cost of the function exceeds the limit established for the block.

Security issue about `compute_envelope`

The function is declared pure and will be computed in a local node. But, when the contract is deployed to the Ethereum network, it will be possibly called internally to the blockchain (i.e. by another contract). When the transaction will be mined, all the nodes of the network will know its parameters such as the doblon, the sigil and how many souls were put in.

Questions

One of the main problem that the inventors of Bitcoin faced is that of“double spending”. Imagine that you have 1 BTC in your wallet, you decide to buy a car, go to the car dealer, and you pay with your BTC. After having received the car, you decide to buy a sailboat, but you do not have any more money. Would you be able to buy the sailboat by somehow 'doubles spending' the 1 BTC you had earlier? Show how an attacker can perform a double spending attack and which are the countermeasures that Bitcoin defines to protect the system from this attack.

The double-spending attack consists of creating two different transactions that have linked, as input, the same output of a previous transaction.

The first attack consists of the generation and propagation into the network of two transactions by a buyer that is not a miner. There are then two cases:
- if both the transactions go to the MemPool of an honest miner then it will realize that there is a conflict between them and will include only one in the block and discard the other.
- if the two transactions are validated simultaneously by two different miners then there will be a fork on the blockchain where each transaction is in a different fork. Now only one fork will be inserted into the long-term blockchain and the other will "die".

The second attack consists of the presence of a miner that is also a buyer. Then, if the miner is not honest, even if it includes both the transactions into the block it will be rejected by the other nodes when it is propagated and the miner will lose the reward.

The third attack exploits selfish-mining. The attacker starts mining in a stealth mode keeping a private fork without broadcasting it to the network. The attacker then spends all his bitcoin in the honest fork but does not include the transactions in his private fork. When the private chain becomes longer than the honest one it will broadcast it into the network that will accept it due to the longest-chain rule. To implement the attack it is needed to mine 6 blocks into the private chain before the rest of the network finds another extra one (this can happen if the attacker controls 51% of total hashing power).

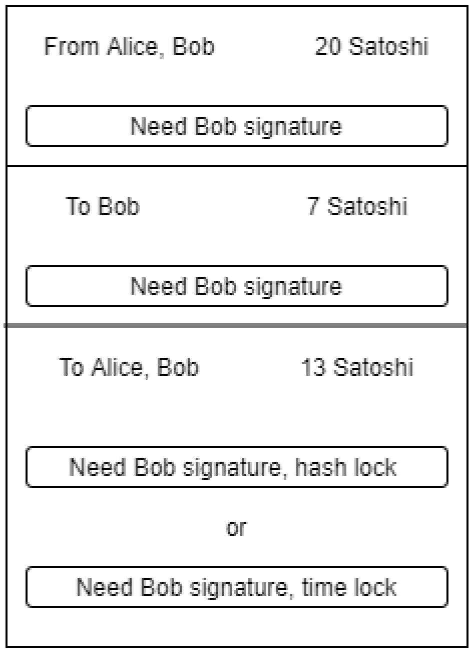
Alice and Bob have several trade relations and decide to open a channel of the Bitcoin's Lightning network. Initially each of them decides to fund the channel with 10 Satoshi. After the last transaction, Alice tries to cheat Bob, by publishing the second transaction, which is more favorable to her, because the state of the channel, after the second transaction, assigns 13 Satoshi to Alice and 11 Satoshi to Bob, while, after the last transaction, 11 are assigned Satoshi to Alice and 9 Satoshi to Bob. Describe how Bob can avoid that the scam is successful, highlighting the functionalities of the blockchain that are exploited to prevent the scam.

The protocol avoids this type of scam by adding disincentives: if one of the party try to cheat not publishing the most updated balances of the closure transactions the counterparty will obtain all the funds of the channel. It uses two mechanisms:

- a time-lock is used to lock the bitcoins in output to make them spendable only in the future. Can be implemented by using a specific time in CLTV or by using a relative time in CSV.
- a pre-image (or secret) is a string randomly generated and impossible to guess. The string is cryptographically hashed through a hashing function. A hash-locked transaction includes the hash of the pre-image in the output script to lock the output.

Alice and Bob start with the opening transaction that include the funding of 20 satoshi and generate a commitment transaction for each payment (only one commitment can be confirmed to prevent double spending). Each of these transactions differ only in the output locking script and are symmetrical. For each payment Alice and Bob send each other the pre-image used in the previous commitment (to reveal the previous secret) and generate the new commitment transaction that will contain a hash of a newly generated secret.

So, now suppose that Alice publishes in the blockchain the commitment transaction received by Bruno related to the first payment. It will be in this form (more or less):



If Bob notices the transaction in the blockchain he can claim the output related to the 7 satoshi with its signature and can also claim the 13 satoshi because he knows the pre-image of this transaction (because before generating the second commitment Alice exchanged the secret chosen for the first payment). Alice instead will have to wait for the expiration of the time-lock. Bob need to check periodically the blockchain because he can apply the anti-cheat measure only before the time-lock expires.