Peer to Peer Systems and Blockchains - Final term

Leonardo Vona
545042

May 27, 2021

1 Ethereum and Smart Contracts

1.1 Implementation

1.1.1 `open_envelope()`

In order to avoid that a voter opens its envelope multiple times, the `Refund` struct has been modified, adding a `bool voted` variable, that is set to `true` into the `open_envelope()` function when a voter opens its envelope for the first time. If a voter tries to open again its envelope, a control on the `voted` attribute prevents this possibility, assuring to allow the execution of the ballot only when all the envelopes have been opened.

```
function open_envelope(uint _sigil, bool _doblon) canOpen public payable{
    require(envelopes[msg.sender] != 0x0, "The sender has not casted any votes");

    bytes32 _casted_envelope = envelopes[msg.sender];

    bytes32 _sent_envelope = compute_envelope(_sigil, _doblon, msg.value);

    require(_casted_envelope == _sent_envelope, "Sent envelope does not correspond to the
        one casted");

    require(souls[msg.sender].voted == false, "The sender has already opened its envelope
        and voted");

    //the sender is a valid voter and didn't already opened its envelope

    voters.push(msg.sender);
    voting_condition.envelopes_opened++;

    _doblon ? yaySoul += msg.value : naySoul += msg.value;

    souls[msg.sender].soul = msg.value;
    souls[msg.sender].doblon = _doblon;
    souls[msg.sender].voted = true;

    emit EnvelopeOpen(msg.sender, msg.value, _doblon);
}
```

Listing 1: `open_envelope()` implementation

1.1.2 `mayor_or_sayonara()`

For the implementation of the `mayor_or_sayonara()` function, the state attribute `bool ballot_completed` and the modifier `ballotNotCompleted()` have been added to the contract. These elements ensure that the `mayor_or_sayonara()` function is executed only once, for the reasons explained in section 1.3.

```
// The mayor_or_sayonara() function can be executed only once
modifier ballotNotCompleted() {
    require(!ballot_completed, "The ballot has been already completed");
    _;
}
```

Listing 2: `ballotNotCompleted()` modifier

```
function mayor_or_sayonara() canCheckOutcome ballotNotCompleted public {
    ballot_completed = true; // The function can be executed only once

    bool winning_vote = yaySoul > naySoul;

    // Each "losing" voter is refunded
    for (uint i = 0; i < voters.length; i++)
```

```
8         if ( souls[voters[i]].doblon != winning_vote )
9           payable(voters[i]).transfer(souls[voters[i]].soul);
10
11      if (winning_vote) {     // The mayor is confirmed
12          candidate.transfer(yaySoul);
13          emit NewMayor(candidate);
14      } else {    // The mayor has lost
15          escrow.transfer(naySoul);
16          emit Sayonara(escrow);
17      }
18  }
```

Listing 3: `mayor_or_sayonara()` implementation

1.2 Cost evaluation

The execution cost of the functions included in the contract has been evaluated considering all the possible cases. The following list shows the results of the evaluation (for each case is reported the *execution cost*).

- `constructor`: 1148453 *gas*
- `compute_envelope`: 1496 *gas* (spent only if called by a contract)
- `cast_envelope`
  - Normal flow: 31447 *gas*
  - The caller has already voted: 5283 *gas*
  - The quorum has been reached: 2525 *gas*
- `open_envelope`
  - Normal execution
    - First call: 120413 *gas*
    - First time the vote expressed by the actual caller is different from the vote expressed by the first caller[1]: 105403 *gas*
    - Other case: 90413 *gas*
  - The quorum has not been reached: 2677 *gas*
  - The sent envelope does not correspond to the one casted: 5062 *gas*
  - The sender has not casted any votes: 3598 *gas*
  - The sender has already opened its envelope: 6078 *gas*
  - All the envelopes have already been opened: 2525 *gas*
- `mayor_or_sayonara`
  - Normal case
    - Quorum 3; 2 positive votes, 1 negative vote: 60265 *gas*
    - Quorum 3; 0 positive votes, 3 negative votes: 48429 *gas*
    - Quorum 6; 3 positive votes, 3 negative votes: 94935 *gas*
  - Not all the envelopes have been opened: 2221 *gas*
  - The ballot has been already completed: 3059 *gas*

[1] e.g.: if the votes from the opened envelopes are, in order, `false`, `false`, `true`, `false`, `true`, then only the third call will fall in this case.

1.3 `mayor_or_sayonara()` considerations

The `mayor_or_sayonara()` function has to handle the transfer of soul to the losing voters and to the mayor or the escrow; any of these actors may be malicious and try to steal ether exploiting a potential re-entrancy vulnerability. In fact, if the function is not re-entrant, an attacker could use the fallback mechanism to re-enter the function before its termination and potentially steal ether.

In order to avoid this vulnerability, the check-effects-interaction pattern can be used. The pattern consists in checking conditions, updating the internal state and then interact with other subjects / contracts. In the `mayor_or_sayonara()` function, the check-effects-interaction pattern can be applied in such a way that a global variable `ballot_completed` is set before refunding the losing voters and giving the remaining soul to the mayor or the escrow. Then, if a malicious user tries to exploit the fallback mechanism or tries to call the `mayor_or_sayonara()` function again, it will simply not be executed. In order to avoid multiple executions of the function, the `ballotNotCompleted()` modifier has been added to the contract, which will pass if and only if the ballot has not been initiated yet.

Another possible vulnerability is that the `mayor_or_sayonara()` function may never been executed. If a malicious voter casts an envelope and never opens it, or casts an envelope that is not a real vote, but just a random bytes32, the condition that allows the execution of the ballot will never be reached. In these cases the useless envelopes will count to reach the quorum, but, since they will never be opened, the condition that all the casted envelopes have been opened will never be reached, and then every attempt to execute the `mayor_or_sayonara()` function will fail. As a consequence, all the funds sent to the contract by the honest voters will never be released and will just be lost.

1.4 `compute_envelope()` considerations

The Ethereum smart contracts are transparent by nature and then the parameters of each function are publicly visible, inside the data field of the relative transaction. This means that anyone could inspect the transactions and retrieve the public key relative to the caller of a function, the parameters and obtain the result. In the case of the `compute_envelope()` function, these considerations involve that the secrecy of the sigil is not guaranteed anymore. Indeed, a malicious entity may inspect the blockchain and obtain the sigil of a voter that wants to compute its envelope. Another result is that even the envelope is not secret anymore, and possibly anyone could retrieve the doblon and the soul of each voter before the voting phase is actually closed. In addition, if the attacker knows to whom are associated the public keys of the voters, the votes are not private anymore.

2 Bitcoin and the Lightning Network

2.1 Question 1

A first attempt to perform a double spending attack is to send two transactions to the network, spending the same bitcoin twice (linking the inputs of the two transactions to the same output). In this case, both the transactions will go in the Mempool of the miners, but only one transaction will be inserted in the next block from a honest miner; the second one will not be confirmed by the miner, and the transaction will be considered invalid. However, it could happen that the two transactions are validated simultaneously by two different miners, and the blockchain forks. While the split is in progress, any of the two chains may be orphaned and then only one of the two transaction will ultimately be inserted on the long-term chain. This mechanism protects the system from a double spending attack, with the consequence that a seller should usually await six confirmations to be sure that a transaction has been definitely validated by the system.

Another attempt to spend the same bitcoins twice may be performed by an attacker that is also a malicious miner. In fact, the attacker may validate a block including two transactions which spend twice

the same bitcoins. Nevertheless, other nodes will check the validity of the block and will reject it, bringing to a failure of the attack.

A possible double spending attack can be carried out by a malicious miner "reversing" the longest chain. In order to perform the double spending, the attacker mines in stealth mode a private branch of the blockchain which is not broadcasted to the rest of the nodes. The attacker spends her / his bitcoins, which are delivered to the seller, and after six confirmations the good will be sent to the buyer. While the transaction is recorded on the truthful public chain, the attacker does not include it to her / his private blockchain. Then, the attacker creates a longer chain with respect to the honest branch of the blockchain and broadcasts her / his version of the chain to the rest of the network. Due to the longest chain rule, the chain version of the attacker will be accepted by all the miners. If the attack succeeds, the truthful chain is abandoned and the transactions in that branch are no more valid, but the attacker has already received the good and is able to spend the bitcoins again.

However, in order to add blocks faster than the rest of miners and build a longer chain, the malicious miner needs more hashing power (at least 51% of the total hashing power) than the rest of the miners. In order to force the switch to her / his branch of the chain, the attacker should mine indicatively six blocks in her / his fork, before the rest of the network has found any extra block. The probability of succeeding is infinitesimal, unless she / he is able to solve Proof of Work puzzles at a rate approaching all other miners combined, which may happen in the Bitcoin network if the malicious miner controls the 51% of the hashing power.

A solo miner will likely never be able to control the 51% of the hashing power to mine new blocks in Bitcoin. However, a mining pool could obtain that level of control. For example, in June 2014 the Ghash.io mining pool reached the 55% of the hashing power, but the event caused a loss of confidence on Bitcoin, and consequently a decline in the value of the bitcoin currency. As a result, miners started to leave Ghash.io mining pool and the system somehow self-regulated, coming back to a "safe" condition.

2.2 Question 2

In order to avoid cheating both parts must have an interest to publish the most updated balances of the closure transaction. Basically, the protocol introduces disincentives: if someone tries to cheat, the other party can punish her / him and take all the amount of bitcoins registered in the channel. In particular, the anti-cheating protocol of the Bitcoin Lightning Network exploits two mechanisms: *time locks* and *hash pre-images (aka secrets)*.

The time lock mechanism "locks" bitcoins in output, making them spendable only at some time in the future. The time when the bitcoins will be unlocked can be specified in absolute (CheckLockTimeVerify, CLTV) or relative (CheckSequenceVerify, CSV) terms. If the time specified is absolute, it will be associated to a specific block of the chain, and then to a specific date and time. Otherwise, in CSV the user specifies the number of blocks that must be added to the chain, therefore the relative time from the time lock creation, before unlocking the bitcoins.

Relatively to the hash pre-image mechanism, a pre-image (or secret) is a string randomly generated and "impossible" to guess. The string is cryptographically hashed through a hashing function, then anyone who knows the secret can easily reproduce the hash, but not the other way round (pre-image resistance). A hash-locked transaction include the hash of the pre-image in the output script, and the bitcoins relative to the transaction (the output) may be unlocked only if the secret of the hash is presented in the unlocking script.

Then, when two parties open a channel, they generate two symmetric commitment transactions, which differ only in the output locking scripts. A new commitment transaction is generated for each payment between the two parties and the outputs of these transactions contains time and hash locks to guarantee anti-cheating. In particular, for each payment, the two parties both generate a new secret and send the hash of the new secret in combination to the pre-image used in the previous commitment transaction (revealing her / his previous secret) to the other party; after having received the other party's previous secret and the hash of the actual pre-image, each party generates the commitment, but just only one of

the two will be confirmed, in order to avoid double spending.

In case one of the two parties decides to cheat, registering on the blockchain a previous transaction more favourable to her / him, the other party can punish the cheater redeeming all the channel bitcoins: the cheated party first signs the transaction and immediately redeems her / his part of the bitcoins, whereas the cheater has to wait the time lock to expire before obtaining her / his bitcoins. While the time lock is still valid, the cheated party detects that the other party tries to cheat, publishing a previous transaction, and then uses the pre-image of the previous transaction to redeem the remaining bitcoins registered in the channel, which were originally intended to the cheater.

Then, in order to avoid to be cheated, a party of the channel has to periodically check the blockchain to monitor possible cheating behaviours of the other party. This is necessary because the anti-cheating action must be taken before the time lock expires. A party of the channel can also demand the monitor of the blockchain to a third party.