UNSW COMP6451
Assignment 2 – Ethereum Programming
Total Marks: 35
Due Date: 5pm, March 28, 2024

Background

It's election year, and you have been commissioned to develop a blockchain based system for securing an election to be conducted using a "Preferential Voting" method that operates as follows.

- Voters submit their votes by ranking the candidates, in order of preference. Thus, if there are n candidates, numbered $1 \ldots n$, then a vote is a list L that is some reordering of the list $[1, \ldots, n]$. (Every element of the list $[1, \ldots, n]$ must appear in L, and no element may appear in L more than once.) The first element of L is the voters first preference, the second element is the second preference, etc.
- In the first phase of counting, we determine the number of first preferences each candidate received. If any candidate was the first preference of more than 50% of voters, then they are the winner of the election.
- If no candidate received more than 50%, we declare that the candidate with the least number of first preferences is a loser, and eliminate them from further consideration. The votes of voters who had that candidate as their first preference are then *redistributed* to the remaining candidates, using the second preferences of these votes.
- If this redistribution leads to some candidate having more than 50% of voters, they are declared the winner. If not, we again choose the candidate remaining with the least number of votes, eliminate them, and redistribute their votes to the remaining candidates. (Note that when redistributing a vote, the vote is assigned to the most prefered of the remaining candidates. For example, a vote when candidates 1 and 2 have already been eliminated, a vote [1,3,2,4] is assigned to candidate 3.)

- This process repeats until some candidate has more than 50% of the vote. (In case of a tie, the election will be repeated between just the two remaining candidates. It is not necessary for you to implement this repeated election.)

Some further requirements for the election are the following:

- Only properly registered voters should be able to vote.
- Votes should be *anonymous*, and not identified by name. Even during the counting period, the identity of voters should not be revealed. Representation of voters by public cryptographic addresses is satisfactory.
- Voters should vote independently, and not be able take into account how other people are voting when they cast their vote. To this end, no-one except the voter should know the value of any vote until the voting deadline has passed and votes are ready to be counted.
- Since election campaigns can be expensive to run, and operating an election also imposes costs on the election authority, voters are required to pay a *voting tax* when they vote. The amount of the voting tax is set by the election authority. After the election, 50% of the voting tax collected from a voter is paid to that voter's most preferred candidate as a contribution to the cost of their campaign. The other 50% is paid to the election authority to cover the costs of the election.

Part 1 (25 marks)

Develop a smart contract in Solidity implementing a voting system that uses the above preferential counting rule to determine the winner of the election. There are several types of actors in this system:

- An election authority, responsible for ensuring correct operation of the election.
- Candidates
- Voters

In part 1 of the assignment, we assume the election authority is responsible for registration of voters and candidates, and setting deadlines for the election. If necessary, they may also perform other actions required for the running of the election.

More precisely, the contract should have the following functionality:

- There are three deadlines: a "register-by" deadline, a (later) "vote-by" deadline and a (still later) "reveal-vote-by" deadline. Only the election authority may set these deadlines, using functions `set_register_by_deadline(date)` and `set_vote_by_deadline(date)` and `set_reveal_vote_by_deadline(date)`.

No registrations of voters or candidates are permitted after the "register-by" deadline. Votes may be accepted only after the "register-by" deadline, but must be rejected after the "vote-by" deadline.

- There is a function `set_voting_tax(amount)` that only the election authority may call to set the voting tax for the election. This function should have no effect if called after the "register-by" deadline.
- There should be a function `register_voter(address)` that may be called by the election authority. Here `address` is an Ethereum address controlled by the voter. This permits the address to vote in the election.
- There should be a function `register_candidate(name)` that may be called by the election authority. This adds the name to the list of candidates in the election. Candidates may also be identified by a number ≥ 1, corresponding to the order in which they were added to the list of candidates.
- After the "register-by" deadline, but before the "vote-by" deadline, voters may cast their *secret* vote using a routine `blinded-vote(data)`. Only registered voters may cast a vote. Here `data` should be information that does not reveal the actual vote, but commits the voter to an actual vote. It should not be possible for any person other than the voter to determine what the voter's vote is by means of a brute-force guessing attack using the information `data`. When calling this function, the voter should attach the appropriate amount of voting tax. (Their vote is not valid if a lesser amount is attached. Any amount greater than the voting tax should be repaid to the voter.)

  Voters may change their minds during the voting period: each voter may call `blinded-vote(data)` an arbitrary number of times, but only the last vote cast by a voter will count; earlier votes will be discarded. The voting tax needs to be paid only for the first vote.
- A voters actual vote should be a list of numbers that orders the set of all candidate numbers $\{1, \ldots, n\}$ in some way. For example, if $n = 3$ then $[1, 2, 3]$ and $[3, 1, 2]$ are valid votes. A list of numbers is not valid if it does not contain all the numbers $1 \ldots n$, if it contains any other numbers, or if it contains repeated numbers.
- After the "vote-by" deadline, but before the "reveal-vote-by" deadline, a voter may call a routine `unblind-vote(vote)`, that "opens" or "reveals" their vote for counting. It should not be possible for a voter to cheat, by revealing a vote that is different from the last blinded-vote that they submitted. The system should check at this point that `vote` satisfies the rules for being a valid vote.
- After the "reveal-vote-by" deadline, the counting process commences. The exact details of the functions involved in the counting process is not specified, but it should implement the preferential voting scheme described

above. There should be a function `winner()` that returns the number of the winning candidate, once the voting computation is complete. Exactly which agents participate in the counting process, and what functions they call in order to effect the counting process, is open to an implementation decision. (Add any new functions that your design requires.)

However, you should take into account the gas cost of the vote counting computation, and make reasonable decisions about who bears this cost. The `winner()` function should have a low cost to call, so that other smart contracts may efficiently discover the winner of the election and take actions on this basis, without having to bear the cost of the vote counting computation.

- The implementation should provide some way by which candidates and the election authority receive their share of the voting taxes collected.

Your implementation of these functions may add new arguments, if it helps to have *auxiliary data* that can be used to optimize the execution of the function in any way (without changing the meaning of the function). For example, if the function needs to check the satisfaction of some condition, you may add an argument containing an efficiently checkable "proof" that the condition is satisfied.

The project has a strict deadline, so you should attempt the Part 1 specification first. For additional marks once you have completed this, attempt the stretch goal.

Part 2 - Stretch Goal (10 Marks)

In part 1 we assumed that the election authority is responsible for registering voters. This means that the election authority is responsible for deciding who is an eligible voter, and (because voters are anonymous in the system) making sure that no voter is registered more than once. (Double voting is not allowed.)

Suppose that this power has been delegated to *registration authorities*. Rather than operating on the blockchain, these authorities issue cryptographically signed certificates stating that a particular address belongs to an eligible voter. Specifically, the certificate should be a message (effectively) stating

> I, `authority-address` assert that address `voter-address` is controlled by a voter who is eligible to vote in the election, and no other address controlled by this voter is currently registered.

The system should accept such a certificate as sufficient evidence that the voter is eligible for registration, provided it knows that the `authority-address` belongs to an approved registration authority, and the message is correctly cryptographically signed by this authority.

Implement this idea by adding the following functions to your solution from part 1.

- `add_registration_authority(address)`: this function may be called by the election authority in order to add the address of a registration authority, so that certificates signed by this authority will be accepted.
- `register_certified_voter(voter-address,authority-address,certificate)`: provided `authority-address` is the address of one of the accepted registration authorities, and `certificate` is a correctly formed and correctly signed certificate concerning `voter-address`, this should register `voter-address` as a valid voter. Any user may call this function.

Additionally, you should provide off-chain code (probably written not in Solidity, but in some other programming language, and runnable on a users device directly), that can be used to create certificates that users may submit to register to vote.

Hint: The Ethereum function **ecrecover** and frameworks such as Web3.py or Web3.js are relevant to this part. You may also wish to investigate proposals such as EIP-721 and EIP-2612.

Deliverables

Submit the following. Note that it is *not* a requirement of the assignment to develop a Graphical User Interface for this application - where code other than Solidity code is necessary, invocation using command line instructions suffices to meet the requirements.

1. (8 marks) A report (pdf format) describing your design and implementation of the overall system. The report should

   - Describe the data model, and how you have chosen to implement this design using Solidity data structures.
   - In case use of the smart contract requires off-chain computations not implemented in the smart contract, explain what this code does and how to compile and/or operate it. You are not required in this assignment to develop a fancy user interface for any such code: some simple command-line scripts suffice.
   - For each of the requirements above, briefly indicate where and how your code meets the requirement. Briefly explain each of the functions in your code. In case you identified any missing requirements or specification ambiguities in the course of your analysis of the application, state what these are and what you have done to resolve and implement them.
   - Provide an analysis of the running costs in gas and Australian dollars of the application from the point of view of the candidates, voters and election authority, and how this depends on factors such as the total number of candidates and voters. Take into account current information on the costs of transactions on the Ethereum public blockchain, and the gas costs of running your code.

- Describe any security considerations concerning the system and its operation that you consider should be pointed out to the (various classes of) users. Are there any specific traps that the user needs to avoid in using the system, and if so, what are strategies that the user can apply to avoid these traps.
- Explain how your code avoids common Solidity security vulnerabilities like reentrancy attacks.
- Reflectively discuss the overall suitability of the Ethereum platform for this application.

Proper acknowledgement of any sources of information or libraries you have used in your project is required.

2. (10 marks) Submit a directory with all Solidity and ancillary code necessary to build and run your implementation using Truffle.[1] In particular there should be a directory **contracts** containing smart contracts in Solidity for your implementation of the basic functionality. Your code should be well documented.

   If you have used any public Javascript libraries, to avoid an overly large submission file, do not include these, but ensure that there is sufficient information in your submission that these can be automatically installed. In particular, include your `package-lock.json` file.

3. (7 marks) Include a directory **test** containing test cases to validate the correctness of your smart contract implementation. Your report should describe the approach that you have taken to testing, and summarize the test scenarios that you have constructed. It should be possible to execute your tests using the Truffle testing framework. You may also test by running a Ganache instance of the Ethereum blockchain. If specific command line arguments are required to run your tests, include a script that allows the appropriate calls to be made easily by the marker. In any case, the report should contain all the information that is required to determine how to run your tests.

4. (10 marks) Once you have met the basic requirements, attempt the stretch goal. If you attempt this part, your report should contain a section describing what you have done for this part, you should include test cases for this functionality, and it should be clear from your report how to run these test cases.

[1] You may choose to do initial development in Remix, but loading your code into Remix creates additional work and inconvenience for the grader, so you should submit in a format that allows testing to be done using Truffle.

Resources and Hints for Testing

For Part 1, it will probably be possible to write tests for your code entirely using test scripts written in Solidity. Part 2 is more challenging, since it requires constructing ECDSA signatures, for which Solidity does not provide good support. (The built-in function `ecrecover` only does signature verification.) For this, it is better to write tests in Javascript. In general, JavaScript testing is the more powerful approach (better supported because off-chain application code for interacting with the Ethereum blockchain is most commonly written in JavaScript) and there are a number of JavaScript libraries that support testing. The following resources may be useful when testing your project

- Truffle Documentation `https://trufflesuite.com/docs/truffle/`. See, in particular, the section "Debug and Test".
- The truffle-assertions library `https://www.npmjs.com/package/truffle-assertions`
- Ganache time-traveller `https://www.npmjs.com/package/ganache-time-traveler` helps to test time-based properties.

Your test scenarios should not only test smart contracts with expected input/output, but also check how it handles errors and reverts. For example, you should check that your code does the right thing in situations where there is not enough money available to make a payment that should be made (or else explain why this situation cannot arise.)

Submission

This assignment is required to be your individual work. Submit your project from your CSE account using the command

```
give cs6451 assignment2 <tarfile>
```

on a CSE server.