EC544 NETWORKING THE PHYSICAL WORLD

A SECURE APPROACH TO ELECTRONIC VOTING

May 2, 2019

Jennifer de Souza
desouj@bu.edu

Contents

1 Introduction

Antiquated voting technology in the form of electronic voting machines leaves the founding principles of democracy vulnerable to attack. At the DEFCON Voting Village Hackathon some vulnerabilities of these machines included, but were not limited to, being able to hack the electoral college votes in 23 states (and subsequently control a presidential election), being able to remotely connect to a voting machine in less than two minutes in 18 states, and being able to wirelessly reprogram a voting machine so that a user could cast as many votes as they wanted [Bla18]. With the advent of Web 3.0, it is time to start to developing smarter, safer and more secure solutions to the problems that leave our elections privy to these hacks. The goal of this project is to create a secure voting machine that preserves the integrity of the vote and is easily scalable. The proposed smart contract ballot prevents altering of results and voter fraud through a cryptographic ledger copied thousands of times through Ethereum's blockchain. Protecting the vote is a matter of national security as our elections are the backbone of the American democracy.

2 Overview

Since decentralized apps are hosted on a relatively new framework, there is no set infrastructure in place for deploying them. This architecture is just one of many possibilities that are all quite involved. The main thing to note about Web 3.0 is that it puts the responsibility of data management back into the clients' hands. The server running the front-end app is only responsible for displaying this information, which it grabs from an interface defined by the compiled smart contract.

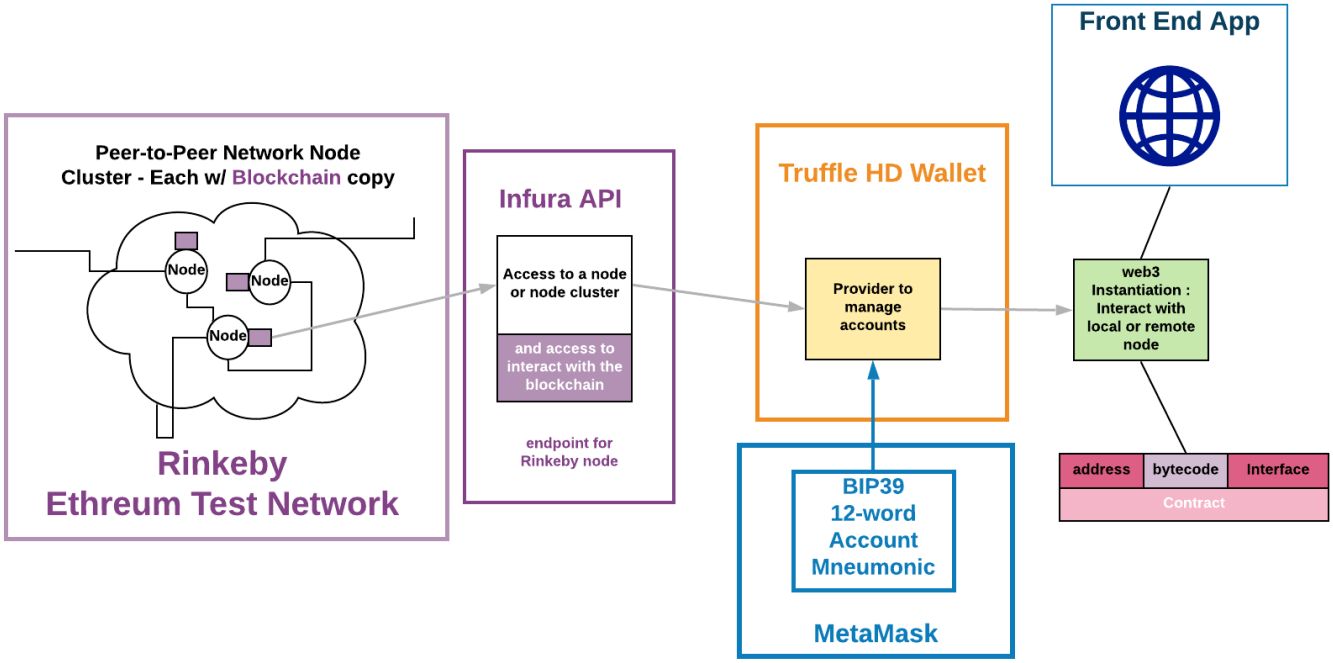

Figure 1: Final Architecture Diagram

2.1 Web3: A Portal to the Ethereum World

The web3 instantiation gives the React App programmatic access to a deployed contract on the Ethereum blockchain network [Gri19]. Web3 is a JavaScript library that provides a means to alter contract data, read contract data or send and receive ether to and from a contract.

2.2 A Solid(ity) Contract

The Ballot contract is a .sol file, which means that it is written in a language called Solidity. The Ballot.sol file is compiled by the NPM solc package to produce two outputs: the bytecode and the ABI. The bytecode gets deployed

to the Rinkeby network, and it is how others execute the smart contract. The price of execution is determined by the gas price of each type of assembly instruction in the bytecode [Gri19]. The ABI is an interface between the bytecode and the Javascript that can be thought of as a translator.

2.3 Truffle-HD Wallet Provider

In order to create a web3 instance, the application needs to pass a provider into the web3 constructor. This component provides the account information necessary for interacting with an Ethereum node or contract. The provider needs two inputs to its constructor in order to forward the necessary information to web3.

2.3.1 Account Mneumonic

The account mneumonic is a specific series of 12-words that are fed into the BIP39 algorithm to determine the public key, private key and accounts' address associated with a certain Wallet. Writing down or storing a private key can leave wallets vulnerable to attack, so the 12-word mneumonic is helpful because it a lot easier to memorize than several 64 digit hexadecimal numbers [Col19]. This mneumonic provides all the information for all the accounts that will ever be created with a certain wallet to the provider, so it is something you only need to update once. MetaMask is a Chrome browser extension used to create the 12-word mneumonic for this project.

2.3.2 Infura Endpoint

The Infura endpoint is a public API to an Infura node cluster. It is very tedious and time-consuming to host a local Rinkeby node so Infura gives you a a public API key to use one of theirs instead.

3 Ballot Smart Contract

The smart contract created for this project has all the functionality of a normal ballot while also adding the ability to autonomously calculate an election winner, set up ballot candidates in real-time and verify voting status of any constituent. The term 'ballot' in this project is not used in the traditional sense in that a Ballot is traditionally deployed on a per-constituent basis. Butterfly ballot deploys a Ballot smart contract on a per-position basis. For example, during the midterms two Ballot contracts would be deployed to the Ethereum network for each states' Senate positions. The term vote keeper is used in this context as the person who was responsible for deploying the smart contract. The deployee has special priveleges that no other constituent can obtain.

3.1 Interface

Below is a summary of each function in the file Ballot.sol, which is the smart contract written in Solidity. Next to each function is an analog for a real election, and/or an explantion of the function's context within the smart contract.

- **submitVote(uint candNum)**: User submits their choice of candidate for this Ballot's position
  Analog: Constituent votes for Al Gore for President
- **createCandidate(string name)**: Vote keeper adds candidate to the Ballot's candidate list.
  Example: Votekeeper adds Donald Trump and Barack Obama to the ballot contract for President.
- **pickWinner()**: Votes for each candidate are tallied up. The winner(s) are declared. This function's execution signifies the end of this Ballot contract. This function is able to detect a tie.
  Analog: Alexandria-Ocasio Cortez has more votes than Jim Crowley, so she wins the election for House representative.
- **getCandidate(uint candNum)**: Returns a Candidate structure consisting of the candidate's name and their current vote tally.
  Example: candNum represents "Elizabeth Warren" with "3 votes"
- **getVoters()**: Returns an array of voting account addresses. This array is used to determine if a voter has voted already.
  Use Case: If account 0xC39C..8FA0 is in the voters array, do not count their vote.
- **getWinners()**: Returns an array of winners. The value of this function is not valid until pickWinner has been called. This array will be more than one candidate long if ties are detected.
  Example: Ralph Nader has the most votes, so he wins the election for President;

3.2 Deployment

We must use the compile.js script, the Node runtime environment and the solc NPM package to compile Ballot.sol and produce two outputs: the bytecode and the ABI. The bytecode consists of all the assembly instructions for the Solidity contract. Each type of instruction and its frequency throughout the byte code determines the gas price of executing a particular contract. The ABI is a Javascript interface which will be used by any Javascript code to interact with the deployed smart contract. The ABI can be thought of as a translator between the bytecode and Javascript.
Next, we need to run deploy.js, which is slightly more involved once the bytecode and ABI has been obtained. Deploy.js is responsible for:

1. Instantiating a web3 portal to the Ethereum node. Web3 enables the Javascript code to send and deploy ether & to send and update contracts.

2. Instantiating a Wallet provider for web3 from our Infura endpoint & 12-word BIP39 mneumonic obtained from Metamask

The web3 instance is used to grab our accounts from the Metamask provider, which is injected into the browser. It is also used to instantiate the Ballot, deploy it and pay the price of gas of deployment. Once the deployment script is finished it prints out the interface and the contract's address on the Rinkeby network to the Javascript console. This address will be used with Remix to see if the contract is behaving as expected. The interface will be used by the React Web App.

4 Test Suite

4.1 Remix IDE with Javascript VM

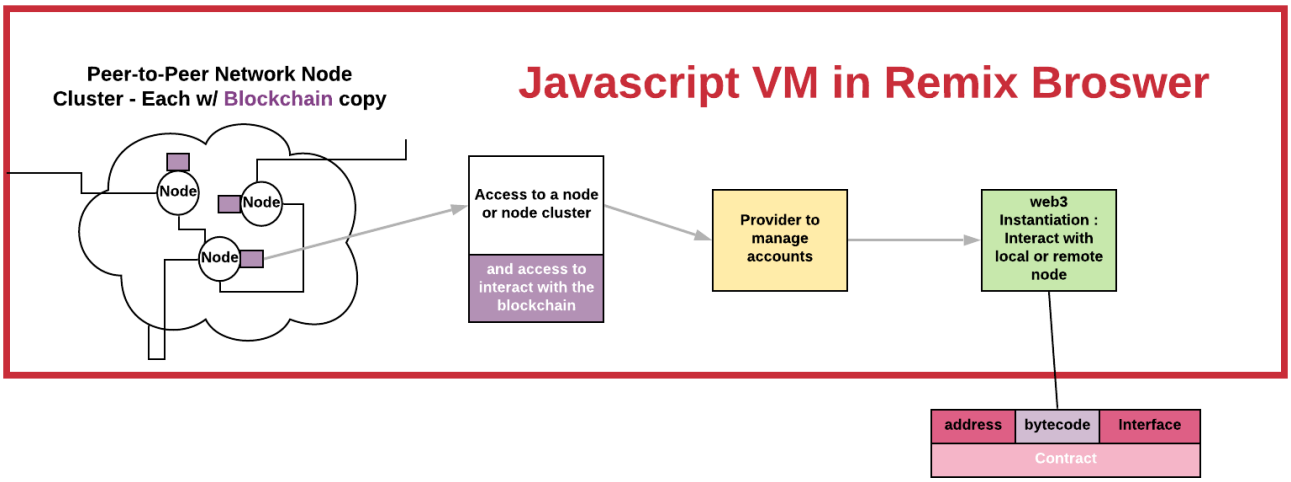

Figure 2: Step 1/3 of the Butterfly Ballot Test Suite

Development of the smart contract starts on the Remix IDE with the Javascript VM. This VM hosts an in-browser virtual Ethereum network. While deployment and contract interaction to any real Ethereum take time (proof of work), deployment to the Javascript VM is instant. This makes defining the debugging the contract seamless, as long as the time it takes for proof of work to complete is accounted for in the developers code. The work that the Javascript VM takes care of under-the-hood is shown in Figure 2.

4.2 Mocha Tests with Ganache-CLI and web3 Ethereum Portal

Once the contract is well-defined and debugged we move over to the Mocha test suite which deploys a simulated instantiation of the contract to a virtual Ethereum network. These tests are the last to run before deploying a real contract to the Rinkeby network. Mocha uses ganache-cli, which is an NPM module that simulates client behavior of

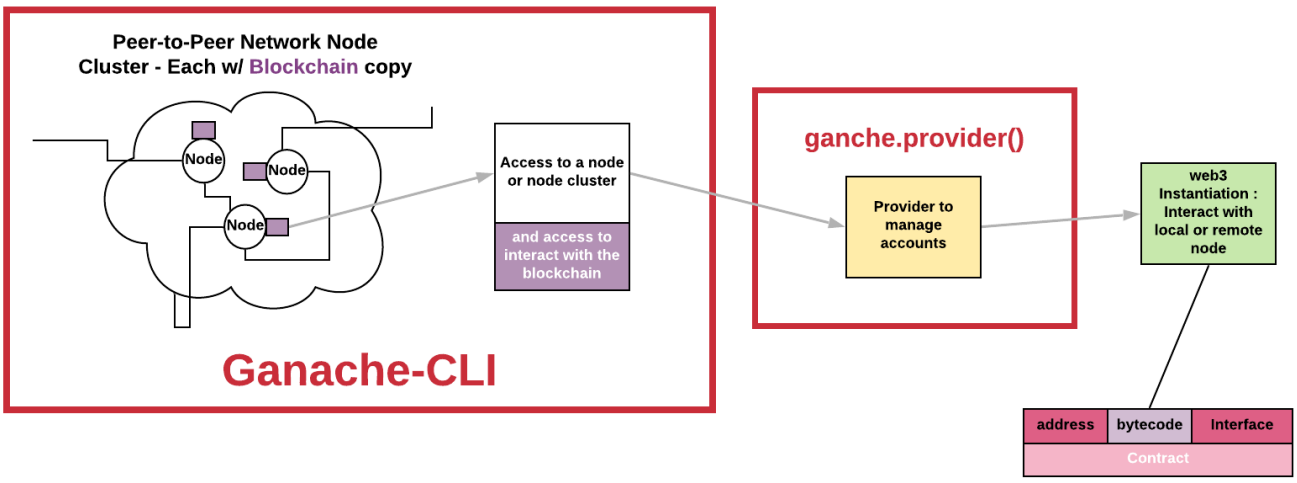

Figure 3: Step 2/3 of the Butterfly Ballot Test Suite

deployed smart contracts. Ganache-cli is very similar to Remix's Javascript VM except that it incorporates web3, which provides a taste of the time it will take to accomplish proof of work and block signing.

Mocha tests are designed to challenge the expected behavior of the contract by anticipating weaknesses and testing corner cases. One important control to note with these tests is the beforeEach function in Ballot.test.js runs the same code before every test. The beforeEach function provides each test with a list of simulated accounts as well as a freshly deployed contract. The Mocha test architecture is summarized in Figure 5 and the mocha tests are summarized in list below. Figure 4 shows what the Mocha test output looks like when run on the command line.

List of Mocha Tests Developed for Butterfly Ballot

1. **Ballot Contract** Does the contract instantiation have an address? An address signifies that it has been deployed to the Rinkeby network.
2. **Votekeeper ID**: Is the account number of the votekeeper the same as the account of number of the person who deployed the contract?
3. **Voter Array**: After a user votes, has their account address been added to the user array?
4. **Multiple Voters**: Is the contract able to store the vote of more than one voter?
5. **Voter Fraud**: Is an error thrown when an account tries to vote more than once?
6. **Candidate Adding**: Is an error thrown when an account other than vote keeper tries to add candidates to the ballot?
7. **Candidate Management #1**: When a candidate is added is the array of Candidates structures updated accordingly?
8. **Candidate Management #2**: Can multiple candidates be added to this ballot?
9. **Vote Tallying**: Are votes tallied up correctly?
10. **Picking a Winner**: Is the smart contract able to correctly identify and report a winner?
11. **Recognizing a Tie**: Is the smart contract able to correctly identify and report a tie?

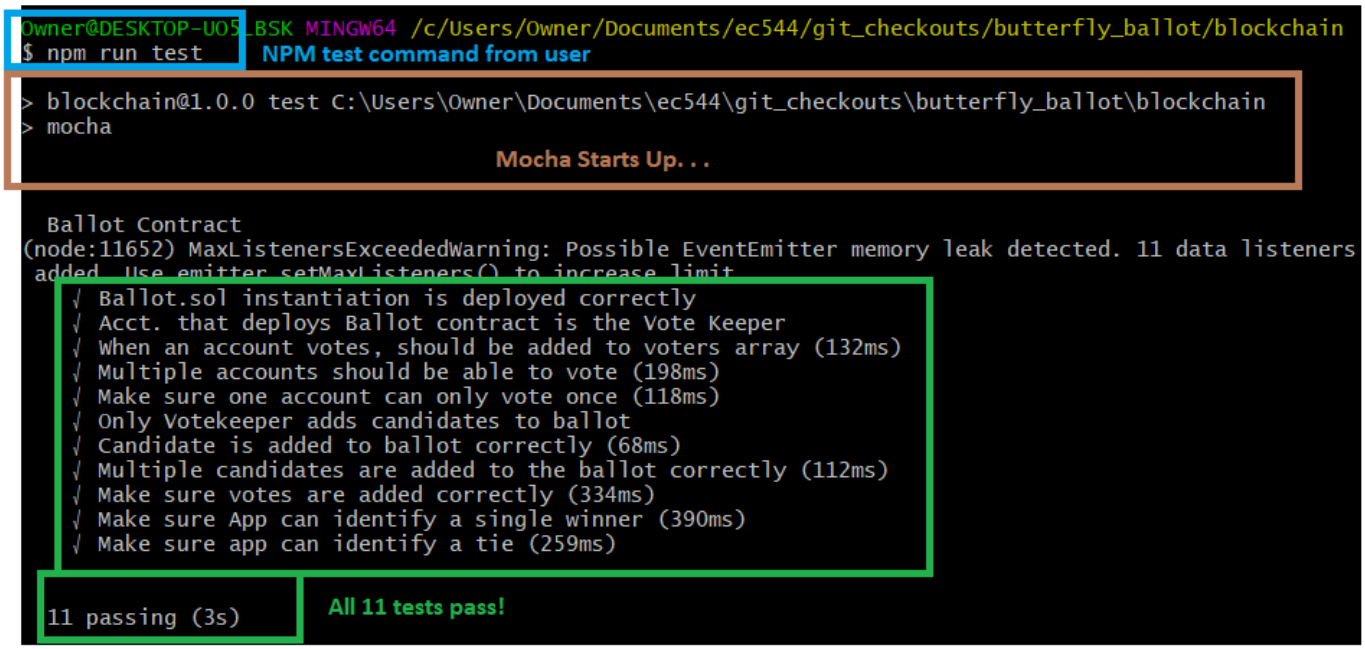

Figure 4: Mocha Output after running 'npm run test' Command

4.3 Remix IDE with Injected Web3

The final step of the test suite is to test the contract after it has been deployed. In a way the test suite goes full circle, as we end up back in the Remix IDE. Only this time, we change the run time environment from the JavaScript VM to Injected Web3. Injected Web3 allows the Metamask Chrome extension to commnuicate with remix and use the accounts that are actually on our wallet, instead of the made up ones on the Javascript VM. The Injected Web3 Setting of the Remix IDE is debug tool for if something ever happens to the contract and I need to investigate.

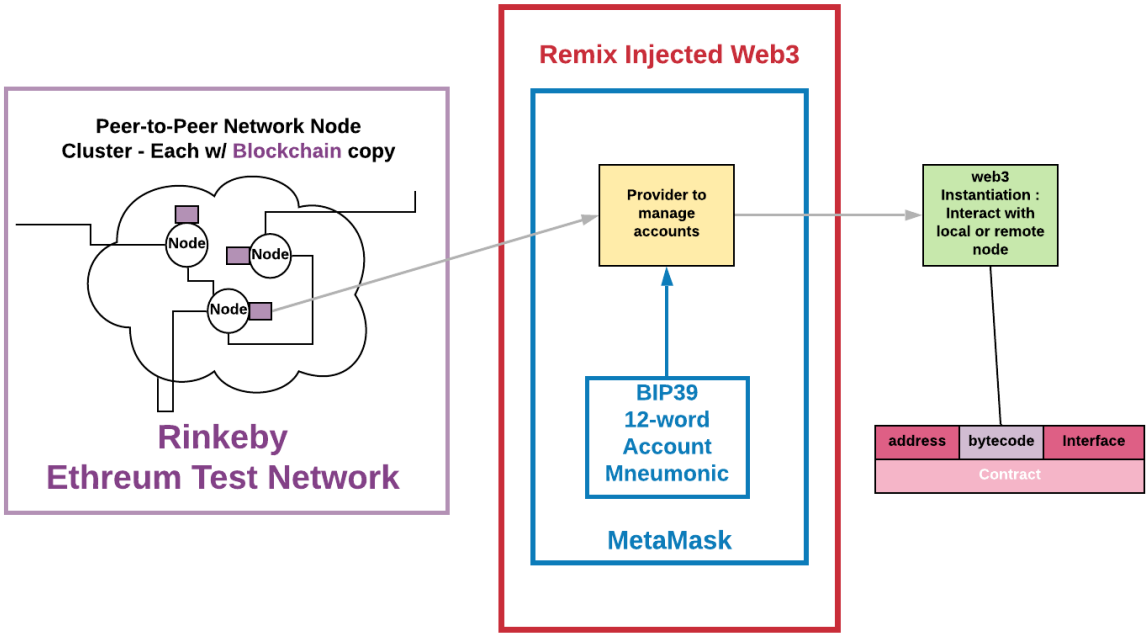

Figure 5: Step 3/3 of the Butterfly Ballot Test Suite

5 React Web Application

The React platform has created a script called create-react-app which sets up an environment to start building and deploying a React web app front-end to your Ethereum contract. The most important thing to note is that the website updates values related to the Ethereum smart contract through a structure referred to as the React app 'state.' The state structure is shown in Figure 6.

```
this.state = {
    newCandidate: '',
    statusMsg: '',
    voteMsg: '',
    addCandMsg: '',
    pickWinMsg: '',
    voteKeeper:'',
    candidateIdxs: [],
    position:'',
    items: [],
    text: '',
    vote: '',
    winner: 'no winner picked',
    ballot_addr: ''
}
```

Figure 6: Declaration of React State Struct

5.1 Voter's Console

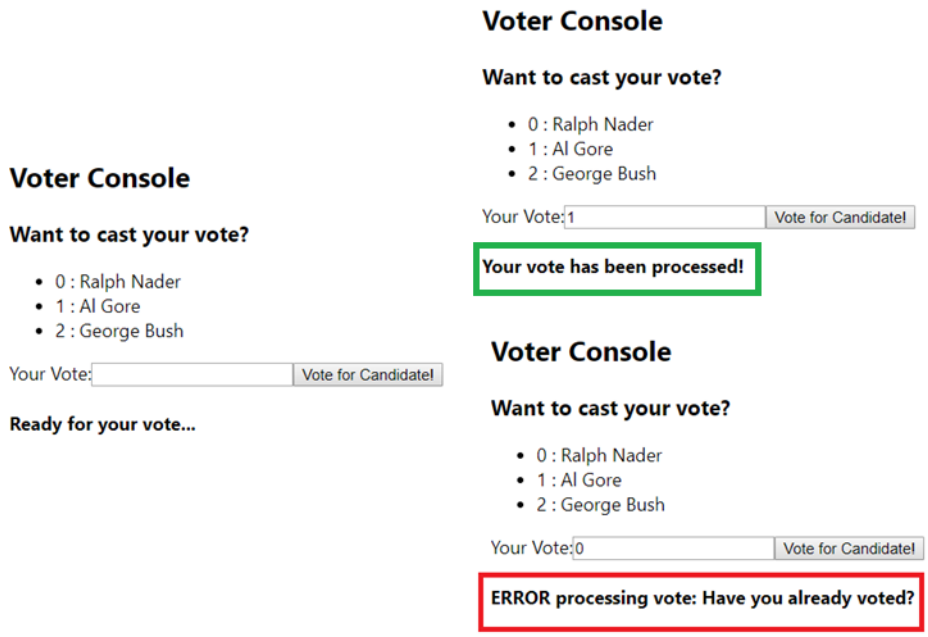

Figure 7: Initial Voting Console

The voting console lists the candidates, provides status messages and is the section of the website where the user casts their vote, as shown in Figure 7. To cast a vote the user enters the index of their preferred candidate and then presses the 'Vote for Candidate!' button. The voting console status message changes to "Processing your vote...". Since voting changes information on the smart contract (which costs ether), a MetaMask notification will ask the voter to confirm this transaction, as shown in Figure 8.

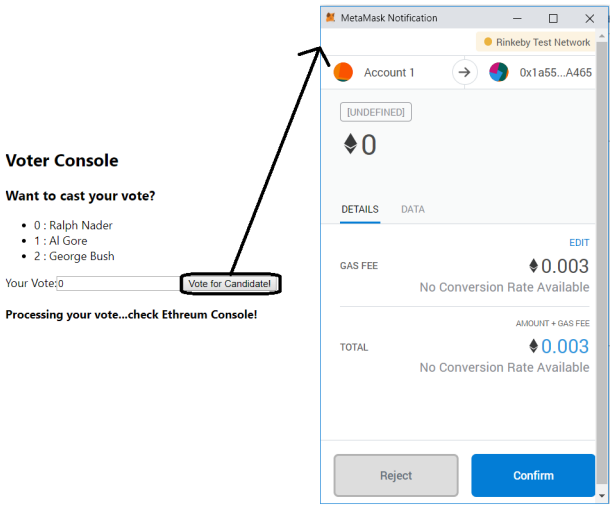

Figure 8: MetaMask Ether Transfer Voting Notification

After the transaction is complete, the status message is updated. The message circled in green (Figure 7) represents a successful transaction: the user has cast their vote and it has been counted on the deployed smart contract. The message in red identifies a potential malactor. It indicates that this account has already voted and is trying to vote again. Messages in the Ethereum status console also update the user on the status of their vote, as shown in Figure 9.

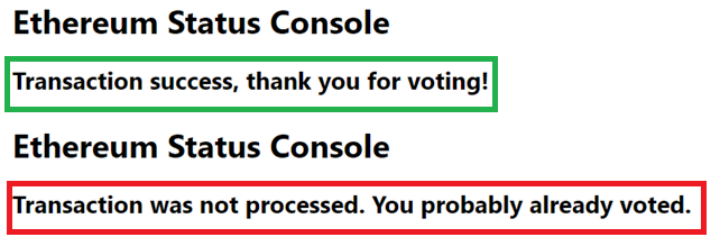

Figure 9: Voting Console and Ethereum Console Interaction

5.2 Vote Keeper's Console

The vote keeper, or rather the account that deployed the smart contract ballot, has two privileges that can change the contract: she can add a candidate and she end the election by picking a winner.

5.2.1 Adding a Candidate

Only the votekeeper has the ability to add a candidate. If an account number different from the one that deployed the contract tries to add a candidate, the web app will inform them that the transaction cannot go through, this case is shown in red in Figure 10. If the account is indeed the votekeeper, a similar transaction to that depicted in Figure 8 will occur since we have to pay ether to change information on a smart contract. Once the transaction goes through the candidate status message will change to the one marked in green in Figure 10. The candidate list will be updated in both the voter's console and the candidate console. Status messages will also appear on the Ethereum console as shown in Figure 11.

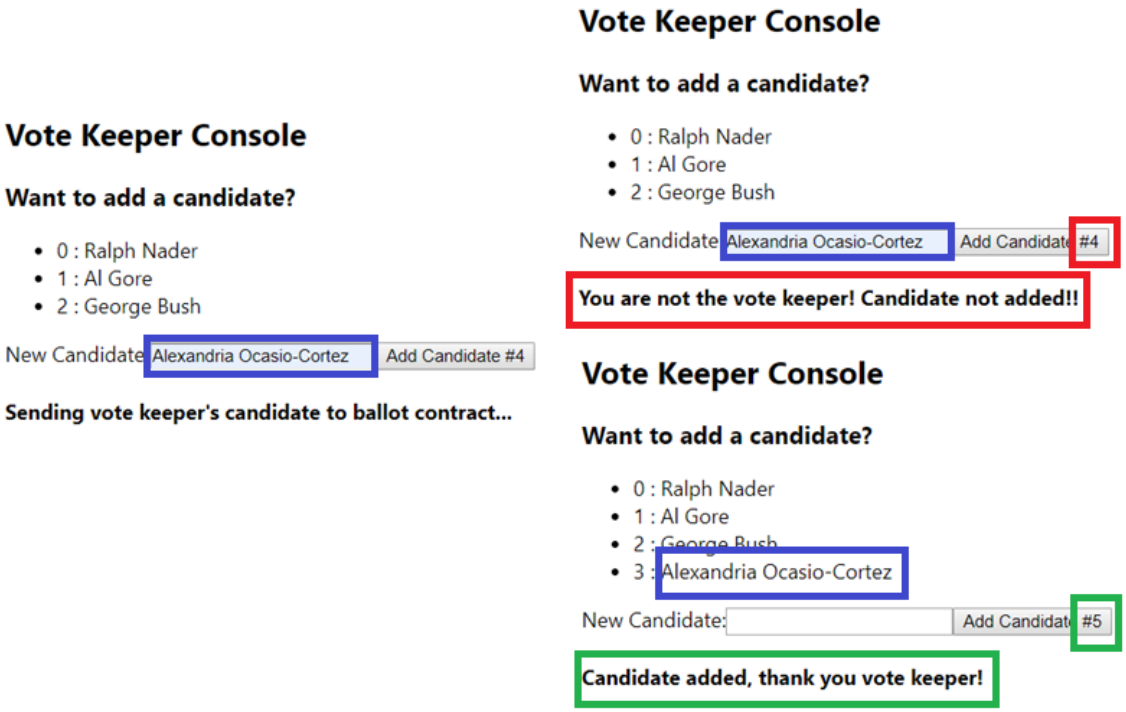

Figure 10: Candidate Console Summary

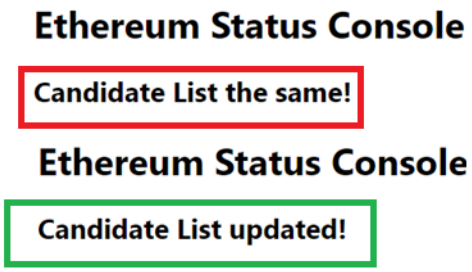

Figure 11: Candidate Information on Ethereum Console

5.2.2 Picking a Winner

Just like adding a candidate, only the vote keeper can pick a winner and, as shown in Figure 12, an error will be thrown if someone who is not the vote keeper tries to pick a winner. Picking a winner ends the election ballot. The transaction runs in a similar manner to other transactions described that update the smart contract. We must a pay a certain amount of ether through MetaMask to update the winners array on the smart contract. Figure 13 shows the corresponding message on the Ethereum console.

Figure 12: Winner Console Summary

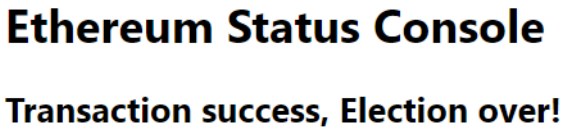

Figure 13: Election Over on Ethereum Console

5.3 End of Election

After the 'End Ballot' button has been pressed and the results are obtained it is no longer valid to interact with the contract. The status in each one of the consoles updates to reflect this information to the user, as shown in Figure 14.

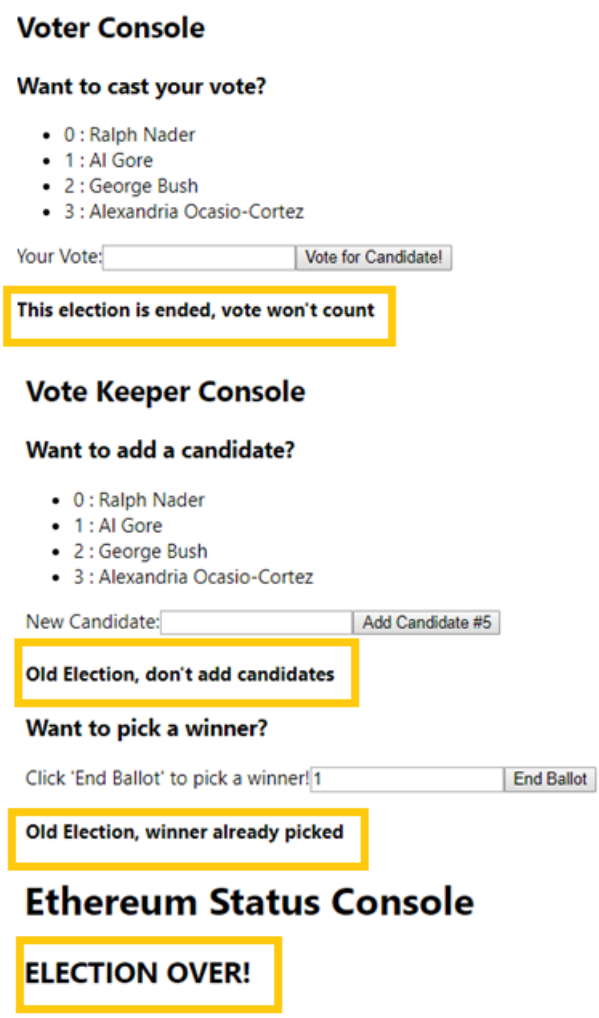

Figure 14: Election Over Page Refresh

5.4 Catching Exceptions

This app was able to catch an account trying to impersonate the vote keeper through the user of try-catch blocks on the React App which were tested on the Mocha framework. The contract code will throw an error if a function with the 'restricted' function modifier is called by a random account. The React code catches this error and uses it to update the website accordingly.

```
function createCandidate(string _name) public restricted
function pickWinner() public restricted

modifier restricted(){
    require(msg.sender == voteKeeper);
    _;
}
```

Figure 15: Protective Solidity Code

6 Software Dependencies & Influences

6.1 Butterfly Ballot Beacon

The architecture of the butterfly ballot deployed smart contract and React front-end was heavily based off of a suggested template in the Udemy course "Ethereum and Solidity: The Complete Developer's Guide" by Stephen Grider [Gri19]. Taking into account that I had never written Javascript, Solidity, JSX or React code and that the blockchain module is covered towards the end of EC544, it was important to incorporate resources from an expert. My project is based off of the knowledge I obtained from completing Sections 1-4 and Section 8 of Grider's course.

6.2 Libraries, Packages and APIs

- **Node.js and NPM**: The Node.js Javascript run-time environment, version 10.15.3, was used to compile the smart contract and deploy it to Rinkeby. It was also used to interact with the smart contract after deployment, both to read and write data. NPM, version 6.4.1, comes with comes with the Node.js download and is a registry of over 800000 Node.js packages [Sch19]. This project interfaced to NPM and Node.js through the command line client.

- **Mocha** This NPM package is used for testing the functions of a smart contract. In a .test.js file, I simulated a deployed contract as well as interactions with said contract. Tests included but were not limited to making sure a deployed contract had an address, making sure only the vote keeper could add candidates and making sure each voter could only vote once. Butterfly ballot used version 6.1.4.

- **Ganache-CLI**: Version 6.4.3 was used to simulate a local blockchain and provider in conjunction with the Mocha test suite. The ganache-cli provider supplied the Mocha test framework with fake accounts. These fake accounts were used to simulate multiple different Ethreum accounts interacting with the Ballot contract.

- **Infura API**: It is necessary to sign up for an account in order to receive a public API endpoint that connects the HD wallet provider to an already deployed node on the Rinkeby network within the Infura node cluster. Without Infura, this project would have had to host its own local Ethreum node.

- **Web3**: Web3 is a Node.js package for interfacing with a local or remote Ethereum node [web19]. This project used version 1.0.0-beta.35. Note that Web3 is a relatively new platform, so all releases are in beta. Web3 was used in conjunction with Infura and the Truffle Wallet to deploy the Ballot smart contract. It was also used in the React App to change and read contract data.

- **Truffle HD Wallet Provider**: Version 0.0.3 of this provider was used to sign keys derived from a BIP39 12-key mnemonic. This provider communicates with the Infura public API to unlock our account so that we can use ether to deploy a smart contract as well change parameters on a contract.

- **MetaMask**: Self-hosted wallet to store and send ether as well host decentralized apps. Available on the Chrome app store. Metamask also has its own version of web3 which it injects into the browser. This web3 instantiation was hi-jacked in the React web app to communicate with the deployed smart contract.

- **Remix**: In browser smart contract IDE used to develop contracts and interact with deployed contracts. The Remix IDE hosts an in-browser virtual Ethreum network. Initially, this is a great place to start for the development of Solidity code, as you can interact with your smart contract while you are writing it.

- **React**: A React template was used in this project to create a simple front end. NPM has a CLI React command (create-react-app) that makes is simple to initialize a front-end template to interact with the Ballot contract.

7 Reflection

7.1 New Skills Obtained

Coding Languages

- Solidity, JavaScript, JSX, CSS & HTML

Platforms & Compilers

- React, Node & NPM, Mocha

Conceptual Frameworks

- Coming into EC544 with no background in blockchain, I am proud of how much I was able to accomplish with this project. I feel as though I conceptually understand the blockchain and decentralized apps after completing this project. Before EC544 I remember trying to wrap my head around BitCoin and never being able fathom such a framework.
- I am now also more familiar with web coding, which I had never done before. I would like to take the skills I learned with React and Mocha to make test platforms for the embedded systems I work with at Draper.

7.2 Future Development

- **Facial Recognition**: Using OpenCV and the Logictech C920 webcam the project could emulate a real-world ballot that could use a state or federal government's database of license pictures for two-factor authentication.
- **Move Platforms to a Single Board Computer**: Originally this project tried to use a BeagleBone Black, but chose to forego this choice of hardware in lieu of a more expedient platform. In the future, it would be useful to revisit this idea as the blockchain becomes more entwined with the internet of things.

7.3 Improvements & Vulnerabilities

- **Improve Voter Identity Verification**: While Butterfly Ballot has the ability to check whether or not an account has voted twice, extra security must be added to make sure that a single person does not vote twice. As is often the case with Ethreum wallets, one person can hold several different accounts.
- **Vote Keeper's Console**: It would be magnitudes more secure to have the React App customize the web page based on the account number or voter ID number. For example, a voter should not be able to see the vote keeper's console which has the options to add a candidate as well as end the election and pick a winner. Although the smart contract won't let the voter do either of these actions, it would still be better if the voter did not know how these functions fit into the smart contract.
- **Paying to Vote**: Since elections tend to happen on Tuesdays, voter turnout is already pretty low. Requiring voters to pay ether in order to vote will probably discourage it further. This burden should be handled by the government in future iterations.

7.4 Not Implemented from Original Proposal

Due to a change in team structure halfway through the semester, some components of the project were dropped.

- Linux Kernel Modules with Custom Button Array
- Beagle Bone Capes
- OpenCV Facial Recognition
- Platform of a BeagleBone Black Rev. C

8 Acronyms

- **JS**: Javascript
- **NPM**: Node Package Manager
- **CLI**: Command Line Interface
- **VNC**: Virtual Network Computing
- **FTP**: File Transfer Protocol
- **SCP**: Secure Copy
- **API**: Application Programming Interface
- **DMV**: Department of Motor Vehicles
- **BBB**: Beagle Bone Black
- **ID**: Identification
- **VM**: Virtual Machine
- **IDE**: Integrated Development Environment

References

[Bla18] Matt Blaze. *DEFCON 26 Voting Village*. Aug. 2018. URL: https://www.defcon.org/images/defcon-26/DEF%20CON%2026%20voting%20village%20report.pdf.

[Col19] Ian Coleman. *Mneumonic Code Coverter*. 2019. URL: https://iancoleman.io/bip39/.

[Gri19] Stephen Grider. *Ethereum and Solidity: A Complete Developer's Guide*. Jan. 2019. URL: https://www.udemy.com/course/ethereum-and-solidity-the-complete-developers-guide.

[Sch19] W3 Schools. *Blockchain Demo*. 2019. URL: https://www.w3schools.com/whatis/whatis_npm.asp.

[web19] web3.js. *Ethereum Javascript API*. 2019. URL: https://web3js.readthedocs.io/en/1.0/.