Implemention and evaluation of a practical secret voting scheme for large scale elections

Balraj Gill

Minf Project
Master of Informatics
School of Informatics
University of Edinburgh

2023

Abstract

In this paper we use an existing proposal for a scalable e-voting scheme and implement it using blockchain technology. We also evaluate the finished protocol to see whether it is adequately scalable.

Acknowledgements

I would like to acknowledge my supervisor Myrto Arapinis for helping me throughout the project and guiding me to relevant sources and suggesting solutions to the problems i had in the project.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Setting up physical election can be quite expensive, for example the 2015 UK election cost £114,732,548 [36]. Costs include setting up locations and payment of staff involved throughout the process. Low voter attendance is also an issue [37], as the participation of the valid age population (VAP) turnout for the UK 2019 election was 62.55%. A lot of paper is also used for paper ballot elections, and in cases where a government is corrupt, it is possible to insert fake ballots. Evoting is an alternative to these.

1.2 What is Evoting?

Electronic voting is simply voting using electronic means to either cast or count votes. One example of this is the Electronic voting machines (EVM) used in Indian elections. The blue buttons on the ballot unit (right machine, Figure 1.1) indicate the different parties a voter can vote for. A polling officer is in charge of the machine and uses the control unit (left Figure 1.1) to register the vote and move on to the next voter. The officer clicks the ballot button on the control unit to activate the ballot machine for 1 vote. Once the last voter has voted, the officer will press the close button to end the voting session (the machine will not accept any more votes) and proceed with the next steps. Another example of this is a direct-recording electronic (DRE) voting machine. These consist of buttons, touchscreens and software to record the voting data. The voting data is stored in removable memory components to make transfer of data to a central authority easier (instead of transporting the entire machine)

1.3 E-voting Systems

E-voting systems [Figure 1.2] can be divided into different groups depending on the way the election is set up. Decentralized voting and centralized voting. Centralized can be divided further into centralized polling station vs. remote voting Decentralised voting consists of elections being run by the voters and casting votes on a distributed

Figure 1.1: E-voting Machines used in India

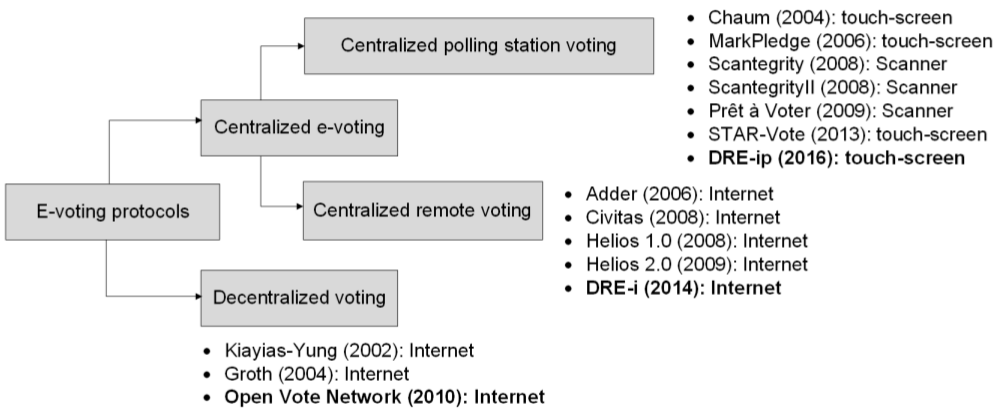

E-voting protocols
Centralized e-voting
Centralized polling station voting
Centralized remote voting
Decentralized voting
Chaum (2004): touch-screen
MarkPledge (2006): touch-screen
Scantegrity (2008): Scanner
ScantegrityII (2008): Scanner
Prêt à Voter (2009): Scanner
STAR-Vote (2013): touch-screen
DRE-ip (2016): touch-screen
Adder (2006): Internet
Civitas (2008): Internet
Helios 1.0 (2008): Internet
Helios 2.0 (2009): Internet
DRE-i (2014): Internet
Kiayias-Yung (2002): Internet
Groth (2004): Internet
Open Vote Network (2010): Internet

Figure 1.2: Categories of E-voting protocols [17]

network in several rounds, meaning voters do not cast vote at one location unlike polling stations. An example of this protocol is the Open Vote network discussed below. The requirement of multiple rounds of voting is not good for scalability as the number of voters increases. Centralised Voting is commonly used in large-scale elections overseed by administrators. An example of this is EVMs, Direct-recording electronic (DRE) machines and in remote voting cases, a web server. In the case of centralized systems, the focus is on systems that are End to End Verifiable (E2E). A system or protocol is E2E verifiable if it meets the following conditions.

1. **Cast as Intended**: The voter can verify the candidate they intend on voting on is correctly captured once they cast the vote

2. **Recorded as Cast**: Voter should be able to verify their cast vote is correctly recorded by the system

3. **Tallied as recorded**: Any observer including the voter can verify the recorded votes are correctly verified

The properties are realised through a receipt which the voter gets, the receipt allows the voter to verify integrity of the system without proving how they voted to third parties meaning voter privacy is also preserved.

1.3.1 Bulletin Board

All E2E verifiable e-voting systems make use of a bulletin board. A bulletin board is simply a system that provides services to users that consist of sharing information and data with each other instead of a central organization posting the data. In the case of e-voting the data posted on the bulletin board is the votes and voter id and users can tally the vote themselves and/or verify the recorded votes,meeting property 3 (Tallied as recorded). The bulletin board cannot be modified, and attempts at modification will be recognized by the readers of the board. Only writers allowed by the board can publish messages on the board. Implementation of a consistent and reliable bulletin board is important and problems implementing it have been highlighted by Benaloh [5] (Phd research paper on verifiable secret ballots) “implementing Bulletin Board may be a problem for itself”, problems highlighted consist of making sure the bulletin board is append only and written by the owner only.Another problem is to make sure privacy is preserved. In addition the voting information should be available for not just voters, but also any observer, to download and verify. The voter id can be encrypted to ensure voter privacy. ”On the Security Properties of e-Voting Bulletin Boards” [24] suggests 4 properties that a bulletin board should have. So we will use these four points as the standard when implementing our own bulletin board.

1. The ability to authenticate item contributors
2. Distributed operation so as to protect against attacks on availability
3. A predetermined time-span where item submission is enabled.
4. Resilience to any modification so as to facilitate verifiability.

1.4 Helios

Helios is a web-based open-audit voting system [2][3] created by Ben Adida. Anyone with access to a Web browser can set up an election, invite voters to cast a secret ballot, compute a tally, and generate a valid proof for the entire process [2]. The Helios bulletin board is implemented as a single web server, and hence it is centralized and has a single point of failure (the server). The server is assumed to be honest and provides a consistent view to all voters, for example, the same information is displayed to all users. Since the data on the server is available for anyone, anyone can audit Helios satisfying E2E verifiable property 2 and 3 (voters can check their encrypted vote on board).The board is implemented in python and using lightpd web server [6], and the data is stored using PostgreSQL database. The User interface is implemented in HTML and JavaScript. The data can be downloaded by users in JSON to be verified. Since there is a single web server, we dont meet the bulletin board property 2. In addition, the Helios paper [2] itself accepts that a compromised web server will lead to the secracy of the ballot being compromised. To counter this, the page claims that ”assuming enough

Entity: Transaction	Cost in Gas	Cost in $
A: VoteCon	3,779,963	0.83
A: CryptoCon	2,435,848	0.54
A: Eligible	2,153,461	0.47
A: Begin Signup	234,984	0.05
V: Register	763,118	0.17
A: Begin Election	3,085,449	0.68
V: Commit	70,112	0.02
V: Vote	2,490,412	0.55
A: Tally	746,485	0.16
Administrator Total	12,436,190	2.74
Voter Total	3,323,642	0.73
Election Total	145,381,858	31.98

Figure 1.3: A breakdown of the costs for 40 participants using the Open Vote Network, at the time the paper[25] was written the cost of 1 eth was $11

auditors, even a fully corrupted Helios cannot cheat the election result without a high chance of getting caught".

1.5 Open Vote Network

The open vote protocol [25] proposed by Patrick McCorry et al. is a decentralized, two-round and self-tallying internet voting protocol that uses the Ethereum Blockchain. In the first round, all voters register their intention to vote in the election, and in the second round, all voters cast their vote. The protocol uses finite cyclic group of prime-order and zero-knowledge proofs. The table in Figure 1.3 above shows the cost of running an election using Open vote network with 40 voters. As the price of ethereum goes up there is a stronger need to consider approaches that can scale better and at the same time have smaller costs.

1.6 Blockchains

In this project the Bulletin Board for the protocol will be implemented as a Smart Contract on the Ethereum Block. A blockchain is simply a distributed and immutable (append only) ledger that records transactions and movement of assets. A block is made up of transactions, and once it is added to the blockchain it is very hard to modify transactions because each block contains the hash of the previous block and changing a transaction on a block will change the hash of the next block, and so on. Since the blockchain is distributed, the modification by a hacker will result in a blockchain different from everyone else's and it will be obvious which blockchain is the true version. A majority (51%) of the blockchain require that it be altered for a hacker to succeed, and this is a difficult task to complete. This property of blockchain makes it an ideal candidate for the Bulletin Board.

Through a Smart contract we can implement a bulletin board such that all the properties

1-4 in 1.2.1 are met.

1.6.1 Ethereum

Ethereum is one example of a blockchain. Ethereum uses proof of work to add blocks to the chain. Smart contracts written in the programming language Solidity can be used to implement the bulletin board. Smart contracts are stored on the blockchain and are executed via transactions by authorised users. Metamask [7] can be used to interact with the Ethereum blockchain and to also deploy smart contracts. Remix editor is used to write solidity programs and ethereum also offers test blockchain to run tests [8]. Gas fees are required to execute transactions in ethereum and ether is used as currency for the fees. Through a Smart contract, we can implement a bulletin board such that all the properties 1-4 in 1.2.1 are met. 1 because each transaction has a a sender, 2 because the blockchain is distributed , 3 can be achieved depending on how we code the smart contract and finally 4 as we already explained its difficult to modify blocks on the blockchain. The two things that we will use to our advantage that ethereum provides is:

1. It costs gas/money to interact with the smart contracts on the blockchain, so it is not wise to spam transactions.
2. Accounts which we can use as identities instead of personal data.

1.6.2 Solana

An alternative to ethereum is solana [9]. Solana uses proof of stake to add blocks to the chain. Solana is one of the fastest blockchains in the world and has much lower gas fees than Ethereum. Scalability will be less of an issue with solana if the number of voters is higher as the cost per transaction will be lower and the transactions will be processed faster [9]. Smart contract development on Solana is done in Rust, C and C++ [10]. These programs are built and deployed on-chain and run via the Solana Runtime where they live forever. These programs can be used by anyone who knows how to communicate with them by submitting transactions with instructions to the network through the JSON RPC API [10] or any SDK built on top of this API. Other on-chain programs can also make use of the JSON RPC API. Solana is better to use for low-scale elections. But the main problem with Solana is the lack of support, as it is still new. It would be significantly more difficult to debug and build programs using Solana for now.

Chapter 2

The protocol

The base protocol [16] used is from a paper called ”A Practical Secret Voting Scheme for Large Scale Elections” by Atsushi Fujioka, Tatsuki Okamoto, and Kazuo Ohta. The paper presents the design of the protocol and proofs of some properties of privacy and Security. The protocol consists of 3 entities, the voter, the admin, and the counter. Before we proceed further, it is important to introduce some concepts that will be used in the protocol.

2.1 Components

2.1.1 Hash Functions

Hash functions simply map a text input of arbitrary size to a value of fixed size. If even a single character is changed in the input, the output is completely different. The probability of collisions (2 different inputs mapping to the same output) is so small that it is negligible. An example of a hash is the Secure Hash Algorithm (SHA) [12] made by the United States National Security Agency (NSA).

2.1.2 Digital Signatures

A digital signature is a useful mathematical technique that is used to validate the authenticity (message sent by a known sender) and integrity (message has not been modified) of a message. Typical Digital Signature schemes consist of 3 components, a key generator algorithm, a signing algorithm and a signature verification algorithm. The steps are as follows:

1. Sender generates public and private key using generator.
2. Sender creates an encrypted version of the message using private key
3. Sender sends the message and encrypted version do Receiver
4. Receiver gets message and encrypted version, uses Sender’s public key to decrypt the encryption and if message and decrypted version are the same, then the message is authentic and has not been modified.

Figure 2.1 shows the scheme in work, hash of the message is encrypted instead of the whole message to make Receiver verification faster as the length of the hash of the message is smaller than the encryption of the message.

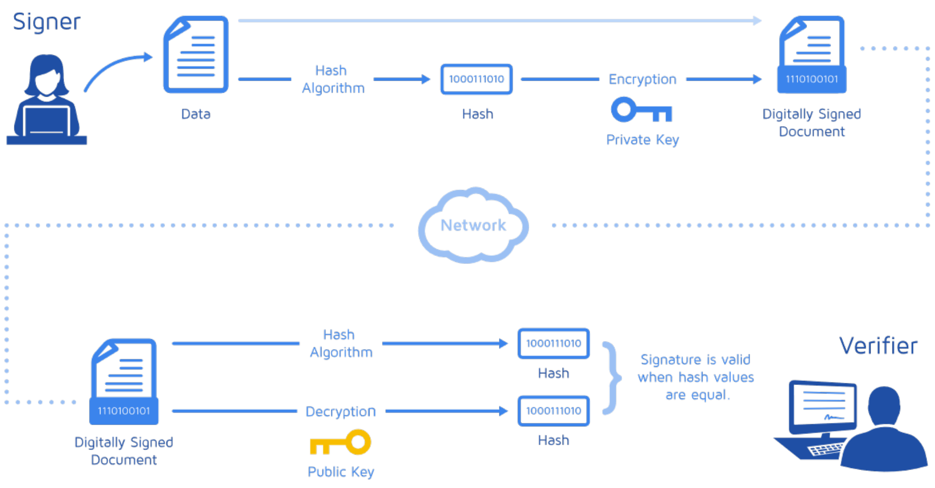


Figure 2.1: Diagram of how signatures work[13]

2.1.3 Blind Signatures

Blind Signature schemes are a key component of the protocol. David Chaum proposed blind signatures in his paper ”Blind signatures for untraceable payments” [14]. In short blind signatures are essentially variants of the digital signature schemes. The goal is to get a message verified by a signing authority (Admin) without revealing what the message was. Blind signatures consist of 3 components:

1. Blinding Algorithm that blinds the message before sending it to the Admin in 2.
2. A signing algorithm owned by the Admin that signs the blinded message and sends it back to Request
3. Unblinding algorithm used to retrieve the signature by applying the algorithm on the signed blinded message sent back by the admin.

The algorithms implemented are such that the signature of the blinded message is the same as the signature of the normal message.

2.1.4 Commitment Scheme

Commitment Schemes were first formalised in the paper Minimum disclosure proofs of knowledge [15]. A commitment scheme allows one to commit to a chosen value while keeping it hidden to others, with the ability to reveal the committed value later during a winner calculation stage so no other person can see their the value that they chose to gain an advantage. We will describe the scheme used in detail in the later chapters.

2.2 Protocol

We have gone over the base concepts that need to be understood before we go over the protocol used for this project, now we will explain the protocol in this chapter. Here are the steps taken to vote for one voter as presented in the protocol.

1. Preparation
    - Voter selects a vote v, generates a commitment x of that vote using randomly generated key k.
    - Next a blind of the commitment e is generated along with a signature s of that blinded commitment
    - a tuple (ID,e,s) is sent to Admin where ID represents a Voter
2. Administration
    - The Admin checks if the Voter has the right to vote, has not already asked for signature and the message is authentic (sent from the voter)
    - The Admin signs e to get d and sends d back to Voter
    - at the end of the this Stage Admin announces number of valid voters and publishes a list of (ID,e,s) of all the valid voters on a public bulletin Board.
3. Voting
    - Voter receives d and uses unblinding algorithm on it to retrive the signature y of x
    - Voter checks if y is indeed Admin's signature of x and not some other third party.
    - If checks pass, Voter sends x and y to the counter through the anonymous communication channel
4. Collecting
    - Counter checks the signature y of x to see it is indeed signed by the Admin. If True then appends the tuple (l,x,y) to a list. l is simply a number for voter to later specify which vote he is sending a key k for to open the commitment
    - after all voter vote, the list is published on the bulletin board
5. Opening
    - Voter will check if the number of votes are equal to number of voters and whether his vote is in list or not. If either are false the Voter will raise the issue.
    - If both true then Voter will send key k and number l to Counter through an anonymous communication channel
6. Counting

- The counter opens the commitments to get v and checks it is a valid vote for each vote.
- Then the counter tallies the votes and announces the results.

In The implementation section we will describe how this protocol was implemented using block chain as the bulletin board and show how it is End to End Verifiable.

Chapter 3

Implementation

Here, we describe the details of the implementation of the protocol such as the different components, how they were built, and how they interact with each other. For the blockchain side of the project, there were couple of frameworks that we could have used for development. But the 2 main considered were Infura and Ganache. Infura provides an api which is used to connect applications to the blockchain. Ganache on the other hand is a local ethereum simulator. It provides test accounts, eth and most importantly it can be linked to remix editor. Ganache was used because it is much simpler to use compared to infura and the testing is much faster. Most importantly we can mass produce accounts (up to a 100 ethereum accounts for testing). Python offers the Web3 library [32] that can be used to build applications that involve communicating with smart contracts on the Ethereum blockchain.

3.1 Assumptions made

For simplicity we assume that the admin through some process has come up with an eligibility list and proceed from there. The admin stores a mapping that maps an ID to a tuple of keys. The first key is a public key from the ecdsa signature scheme to verify that the message was sent by the right person. The second is a public RSA key of Voter represented by ID, used to encrypt a message sent back to Voter. Admin also holds a single Public-Private RSA key pair that is used for decrypting (using Admin's Private Key) encrypted messages (encrypted using Admin's RSA public key) sent from Voter. Details will be elaborated on in later sections.

3.2 Anonymity

After getting to the part where you have to implement an anonymous communication channel, we realized that to be part of the eligibility list, some sort of identity/or agreement needs to be made by the Admin and Voter. We assume that this has already been done.Pseudo anonymity is reached with the rest of the protocol. We consider 2 ways to do this. One is using a VPN which will hide IP address when communicating with the intenet. The other is to use a proxy in python requests. Proxies can also be used to hide IP address when communicating with Admin. While using VPN we can hide our IP while we talk to Contract.

3.3 Blind Signature

Initially Blind signature based on RSA [26] was consider but after furthur research and testing, it was dropped because the cost for verification on smart contract side was was too high. Both papers [27] and [28] are a study involving the evaluation of RSA and El gamal. Both papers agree that El gamal is faster, efficient, and sclable. Although RSA is faster for encryption, in our case we do not care as much about encryption speed, what needs to be fast and cheap is the verification on the smart contract. The Blind Signature Scheme that was built and used for this protocol was proposed in paper[18]. The paper uses the Elgamal signature [19] to make a blind Elgamal signature scheme. There are 4 cryptograhic primitives involved in this scheme. Namely Key Generation by Admin, Signing algorithm of Admin, Blinding and Unblind Algorithm of Voter.

3.3.1 Key Generation

The key generation is the same as described in the elgamal signature paper [19]. Given the following:

1. Let p be a large prime such that computing discrete logarithms modulo p is difficult.
2. Let g be a randomly chosen generator of the multiplicative group of integers modulo p

The private key of Admin is simply a random number x such that x is greater than 1 and less than (p-1). And the public key is tuple (p,g,y) where y = g^x mod p.

3.3.2 Blinding

We here have access to the public key of Admin (p,g,y) The Blinding algorithm has the following steps.

1. Choose a random number k, uniformly distributed between 1 and (p–1), such that greatest common divisor of k and (p–1) = 1.
2. r = g^x mod p.
3. Take a blinding factor h such that the greatest common divisor of h and (p–1) is 1. This h is to be kept secret.
4. Then the blind of a message m’ given a message m is simply calculated as h*m mod (p–1), we send blinded message and r to the Admin to be signed.

3.3.3 Signing

As described in the previous section, the signature of the blinded message and the message should be the same, so the signing is simply the same as the normal signature scheme. The signature for m is the pair (r,s) satisfying the equation

$$g^m \equiv y^r * r^s mod p \tag{1}$$

We know y = g^x mod p and r = g^x mod p from this we can sub in these values and rewrite (1) to obtain:

$$g^m \equiv g^{rx} * g^{ks} mod p \tag{2}$$

$$m = xr + ks \ mod(p-1) \tag{3}$$

The signature is obtained using (3) to solve (2) for s.

$$s' = (m - xr)k^{-1} \ mod(p-1) \tag{4}$$

(4) s' gets returned as the signature of the blinded message that was sent by the Voter

3.3.4 Unblinding

After we get s' from Admin we need to apply the Unblinding algorithm to get the real signature. The real signature s is:

$$s = xrk^{-1}(h^{-1} - 1) + h^{-1}s' \ mod(p-1) \tag{5}$$

3.3.5 Verification

Verificaion is simple, we simply reproduce both sides of equation (1) if they are equal then the signature is valid else it is not.

3.3.6 Our Implementation

An already built python implementation was found [35]. Because the implementation did not specify a valid source, the code and its logic had to be heavily tested. After reading the code logic it was clear that it was based on the paper that proposes the elgamal blind signature scheme. The code uses the Miller-Rabin primality test. This is a probabilistic multi-round algorithm used to test if really large numbers (128 bits or greater) are prime numbers. It is a efficient polynomial time complexity algorithm. Given a integer n, number of rounds k, the probability of n being a prime is:

$$1 - 4^{-k}$$

Given this we pick k=100 in our case which is sufficient. The text encoding (text to bytes) was also incompatible with current version of python so they also had to be changed. The code was also missing some edge cases as making sure a signature is not 0. The primality test was also slightly modified with newer libraries to make it more efficient. Following the code logic and matching the different operations with the operations in [3.2.1–5], the code was verified to be correct implementation.

3.4 Commitment Scheme

Picking a Vote is simply picking the number that represents your candidate for example picking a in Figure 3.1, we would hash 1 and send that to Admin. But we cant just simply hash the number since the admin can simply compare the hash of numbers 1 to 1000 (or arbitrary size) and see which one matches the hash sent by voter to be signed, revealing which option the voter is going to vote for. The solution to this is to generate a random key which the number is concatenated to and then this gets hashed and sent to the Admin to be signed. The key that is generated is later sent to the counter to reveal the vote. We use a 256 bit key so the probability of guessing it is 1 in

$$2^{256}$$

.

The hash algorithm used is keccek256. The main reason to use this was compatibility with solidity as solidity has efficient calculations involving this hash algorithm. 256 bit hash is also sufficient for our case and we will prove its security in the evaluation section.

3.5 Other Tools

3.5.1 AES

AES [30] was established by the U.S. National Institute of Standards and Technology (NIST) in 2001. Its a symmetric key encryption algorithm, meaning the same cryptographic key is used to for both encryption and decryption. This ended up not being used in our implementation but is still considered for a proposal in later chapter.

3.5.2 RSA

RSA (Rivest–Shamir–Adleman) [31] is a public-key cryptosystem, where you encrypt a message with a public key and decrypt with a private key. This is use full to keep data secret when communicating over a insecure channel.

3.5.3 Ecdsa

Ecdsa is an digital signature scheme based on Elliptic Curves. It was first proposed by Scott Vanstone 1992. It was a response to NIST's (National Institute of Standards and Technology) request for public comments on their first proposal for DSS (Digital Signature Scheme) [29]. This is used to sign and verify messages. The message sender will sign a message with a signing key and send the verification key to the receiver. If the signature is valid then we know with high confidence the message was send by right person (signed by signing key corresponding to verification key) and has not been modified.

3.6 Admin

The Admin entity is implemented as a Flask Webserver[20], python requests[21] are used by the Voter to send their blinded commit to the Admin. The blinded commit gets sent to a function through python request, which returns the signature of the blinded commit. The signature is generated using the signing algorithm that the admin owns. The admin also is the owner of the public and private keys used for signing.

3.7 Voter

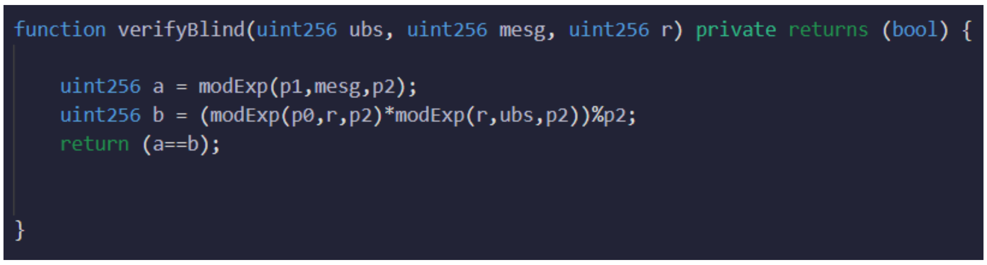
The voter entity is implemented as a Python desktop application. Kivy[22] is a Open source Python library that can be used to built applications. We use this library to build our user interface [figure 3.2 below]. We have simplified the complicated blockchain operations to simply pressing buttons. With the buttons the user can get information from the blockchain and send data to the smart contract and can confirm it using ethereum explorer.

3.8 Counter

The Counter is implemented as a Smart Contract on Ethereum Blockchain. The contract is written in the language Solidity. Different functions are implemented to verify the voters and their votes. For example [Figure 3.1 below] shows the blind signature verification. Here is a list of functions and main variables on contract.

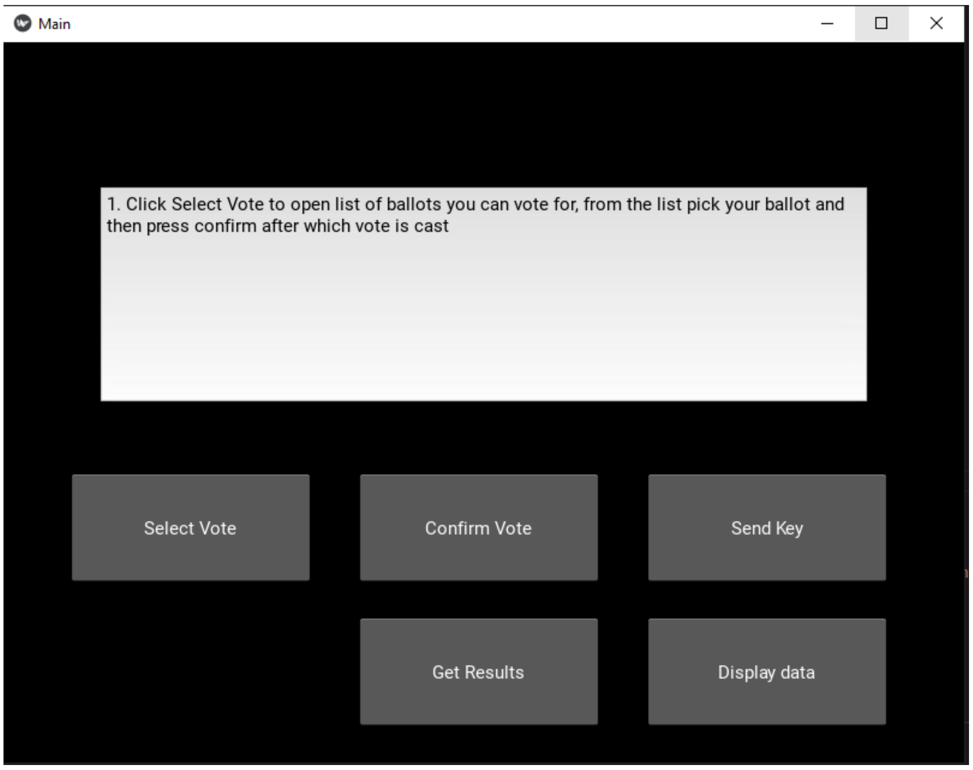
3.8.1 SendVote

This funtion is called from the Voter application when send the vote



```
function verifyBlind(uint256 ubs, uint256 mesg, uint256 r) private returns (bool) {

    uint256 a = modExp(p1,mesg,p2);
    uint256 b = (modExp(p0,r,p2)*modExp(r,ubs,p2))%p2;
    return (a==b);


}
```

Figure 3.1: code for verifying blind signatures on smart contract



1. Click Select Vote to open list of ballots you can vote for, from the list pick your ballot and then press confirm after which vote is cast

Figure 3.2: The user interface

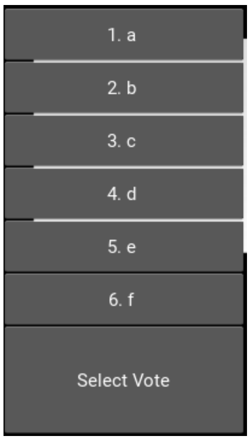


Figure 3.3: The list that shows the list of candidates

3.9 Process of one Vote

We can split the communication between entities into 2 parts. The Voter–Admin and Voter–Counter. First, we describe how Voter-Admin was implemented.

1. The Voter Runs the desktop application/program and selects the candidate they wish to vote for, then the program will produce c using keccek256 hash as mentioned in Section 3.1.2. The hash will be blinded using the elgamal blinding algorithm [3.2.2] with public key of the admin and a random integer h (which is kept secret to everyone but Voter), let this be blind message be e. Do to the

nature of Elgamal Blind scheme we need to send 2 more parameters r and k, as explained above [3.2.2]. The tuple now is (ID,e,r,k). The tuple is then signed using ecdsa signature scheme private key to obtain a signature sig, the Admin already has a public key as mentioned in assumptions. We add sig to the tuple. To Further protect this tuple while its being sent to Admin, each item in the tuple is encrypted using RSA public key of Admin. Finally what gets sent to the admin is the tuple (ID,e,r,k,sig).

2. The Admin uses its private RSA key to decrypt the data sent and gets ID .The Admin retrieves the public keys mapped to ID and uses it to verify the signature sig of the tuple. If valid then all entries in tuple are valid and indeed sent by the Voter represented by ID. Then ID is checked to see if they have already applied for a signature or not. If they have then reject message is send back, if not then admin signs blind message sbm using it's signing algorithm as described above [3.2.3]. Then sbm is encrypted using public RSA key and send back to the Voter. Admin also has a function which returns list of tuples (ID,sbm,sig).

3. Voter uses their Private RSA key to decrypt and get sbm. Then sbm is passed on to the unblinding algorithm to retrieve the actual signature y. The verification algorithm is used to check that is is indeed a signature of the blinded commit. If its valid, a tuple consisting of the admin signature y, c, r gets sent to the Counter.

Next, we discuss the communication between the Voter and Counter. Note that, Require in solidity is used to when performing validity checks. Require is essentially a built in function of the form Require(boolean expression), where if the boolean expression is false then the transaction is reverted remaining gas is refunded.

1. The counter receives the tuple in function sendVote. The sender of the message does not need to be verified directly, only the admin signature y. Again the verification algorithm is used to verify the signature. Solidity has a low level function modExp that computes efficiently equation of the form

$$b^e mod\ m$$

   where b,e,m are large. This is perfect for our use, as this is exactly how we verify the signature [3.2.3] equation (1).We can reproduce both sides with r and s sent by the Voter and the Admin's public key (p,g,y), generation specified in [3.2.1]. If the signature is valid, the tuple (i,c,y) gets added to the list, i is just a number for Voter to specify which entry they are sending the commitment reveal key k.

2. Voter can press the Display button to display all the relevant data such as blinded commit, commitment, and the list of votes mentioned above 1. And use this to audit the counter, to make sure their vote was counted and that there are not more votes then number of voters validated by Admin. At this point, the voter will send his key k used to generate the commit c and their vote to counter. If the vote is valid it's entry in the mapping gets incremented by 1. And after all voters cast their vote or a certain amount of time is passed, the counter creator can close the election to prevent any more votes. The result can be retrieved by the Voter by Clicking Get Results Button.

Chapter 4

Evaluation

Evaluation of the protocol can be divided into different aspects:

1. Security, where the security of the protocol is evaluated. We check common attacks and see how secure the cryptographic primitives used in this protocol are.
2. Cost, we calculate the cost of using this scheme in terms of ethereum gas.
3. Privacy, can we deduce who voted for what or who the voter is.

4.1 Security

Evaluation of the Cryptographic Primitives used in the protocol

4.1.1 Hash

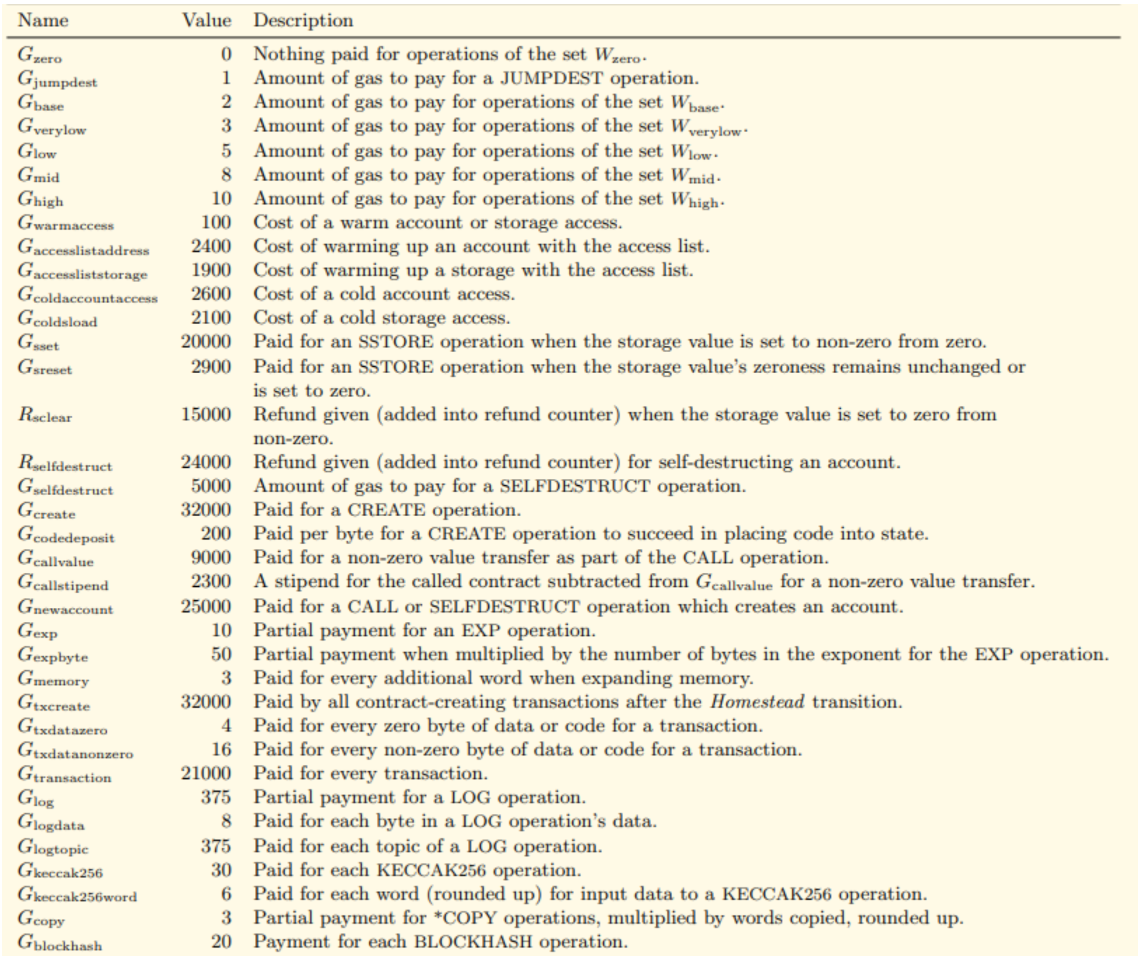
All hash algorithms must have these basic properties:

1. Collision resistance: It should be difficult to find a pair of different messages m1 and m2 such that Hash of both messages is the same.
2. Preimage resistance: Given an arbitrary n-bit value x, it should be difficult to find any message m such that H(m) = x i.e guessing the input to hash algorithm that produces the hash x.
3. Second Preimage resistance: similar to 1, given a message m1, it should be difficult to find different message m2 such that there hash is the same.

In our case and in the unlikely scenario an attacker has found a message m whose hash is same as commitment c. We are safe because in the counting phase we check if m is a valid vote. So a attack based on collisions will not work. Only choice is to find the exact message m by guessing. The probability of guessing the input which outputs one of the

$$2^{256}$$

outputs wont work for trivial reasons. An intelligent approach would be to maintain a dictionary of every value used, commonly known as a birthday attack (based on the



Name	Value	Description
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
$G_{jumpdest}$	1	Amount of gas to pay for a JUMPDEST operation.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{warmaccess}$	100	Cost of a warm account or storage access.
$G_{accesslistaddress}$	2400	Cost of warming up an account with the access list.
$G_{accessliststorage}$	1900	Cost of warming up a storage with the access list.
$G_{coldaccountaccess}$	2600	Cost of a cold account access.
$G_{coldsload}$	2100	Cost of a cold storage access.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	2900	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{selfdestruct}$	24000	Refund given (added into refund counter) for self-destructing an account.
$G_{selfdestruct}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{newaccount}$	25000	Paid for a CALL or SELFDESTRUCT operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	50	Partial payment when multiplied by the number of bytes in the exponent for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
$G_{txcreate}$	32000	Paid by all contract-creating transactions after the *Homestead* transition.
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$G_{txdatanonzero}$	16	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
$G_{logdata}$	8	Paid for each byte in a LOG operation's data.
$G_{logtopic}$	375	Paid for each topic of a LOG operation.
$G_{keccak256}$	30	Paid for each KECCAK256 operation.
$G_{keccak256word}$	6	Paid for each word (rounded up) for input data to a KECCAK256 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{blockhash}$	20	Payment for each BLOCKHASH operation.

Figure 4.1: table of gas cost of various low level functions [34]

Birthday problem). To reach a probability of 75% approximately

$$5.710^{38}$$

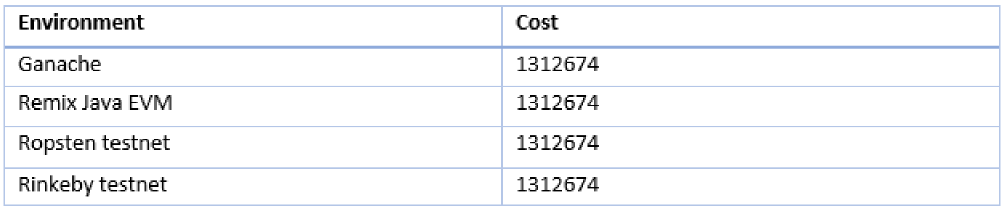
attempts have to be made. Given the storage space required for such a dictionary and the time needed, this attack is infeasible. In addition, given the time limits in our contract, it is impossible for such an attack to take place. Therefore, the hash used is safe.

4.2 Cost

Cost of the protocol is calculated in terms of ethereum gas. Gas refers to the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network [33]. The higher the gas, the more you would have to pay to trigger transactions. Table [4.1] can be used to assit in gas cost calculations. It is sufficient to calculate the gas cost of each component in the smart contract and then use the sum as the total cost of the contract. The components are: Contract creation,commitment verification, tallying, sendvote function, sendKey function. Python web3 offers a estimateGas() function that returns a very accurate gas cost estimation of any particular transaction. We use a loop with this to simulate sending hundreds of transactions to get various statistics for the cost. At the start of the script we assign a new account to send the vote from to simulate different voters sending a vote.

4.2.1 Contract Creation

The Contract created in different scenarios and the cost was as follows:



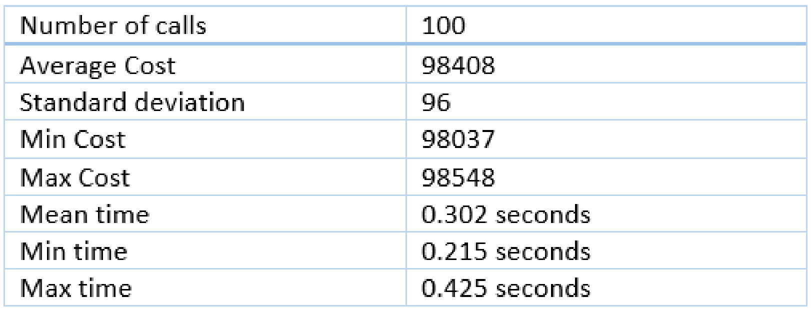
Environment	Cost
Ganache	1312674
Remix Java EVM	1312674
Ropsten testnet	1312674
Rinkeby testnet	1312674

Figure 4.2: table of contract creation fees

This is as expected since the cost of operations on ethereum is defined, so as long as the same contract is compiled on testnets, it will cost the same amount of gas. From now on, we will show results using a local chain on Ganache.

4.2.2 Send Vote functions

The send function vote receives the commit, signed commit, and r to verify the message. We used script [4.3] to get the data in table [4.2] and [4.3]

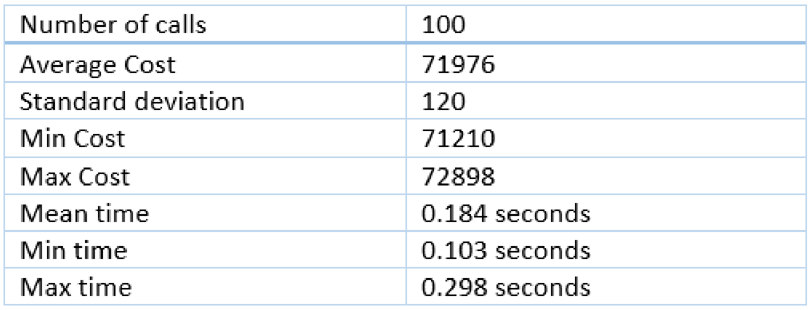


Number of calls	100
Average Cost	98408
Standard deviation	96
Min Cost	98037
Max Cost	98548
Mean time	0.302 seconds
Min time	0.215 seconds
Max time	0.425 seconds

Figure 4.3: table of contract creation fees

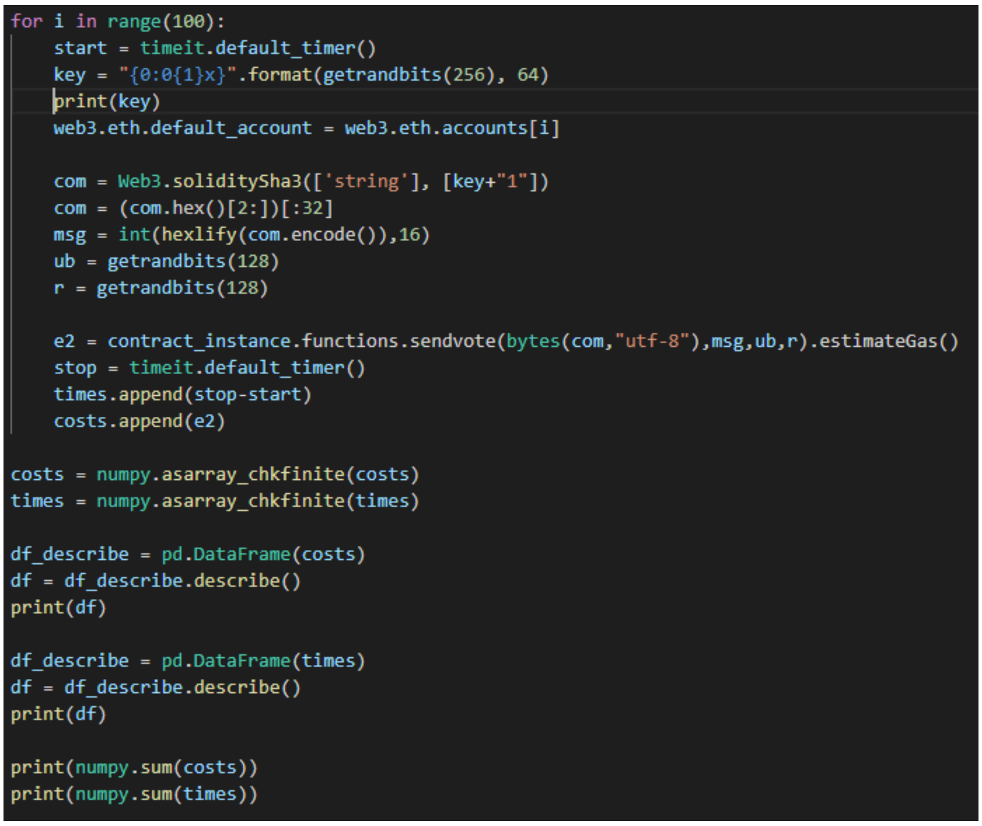
4.2.3 Send Key

In theory (based on 4.1) the cost of the keccek256 should be the same as all the commits are same number of bytes. But in practice, we have to consider the contract calls to the low-level functions and where the inputs are stored. Given this the cost can be slightly different.



Number of calls	100
Average Cost	71976
Standard deviation	120
Min Cost	71210
Max Cost	72898
Mean time	0.184 seconds
Min time	0.103 seconds
Max time	0.298 seconds

Figure 4.4: table of cost for Send key function call



```
for i in range(100):
    start = timeit.default_timer()
    key = "{0:0{1}x}".format(getrandbits(256), 64)
    print(key)
    web3.eth.default_account = web3.eth.accounts[i]

    com = Web3.soliditySha3(['string'], [key+"1"])
    com = (com.hex()[2:])[:32]
    msg = int(hexlify(com.encode()),16)
    ub = getrandbits(128)
    r = getrandbits(128)

    e2 = contract_instance.functions.sendvote(bytes(com,"utf-8"),msg,ub,r).estimateGas()
    stop = timeit.default_timer()
    times.append(stop-start)
    costs.append(e2)

costs = numpy.asarray_chkfinite(costs)
times = numpy.asarray_chkfinite(times)

df_describe = pd.DataFrame(costs)
df = df_describe.describe()
print(df)

df_describe = pd.DataFrame(times)
df = df_describe.describe()
print(df)

print(numpy.sum(costs))
print(numpy.sum(times))
```

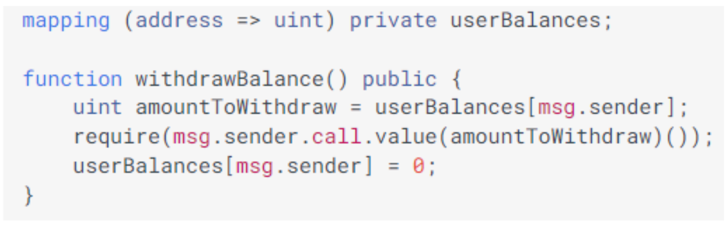
Figure 4.5: script used to calculate cost

4.2.4 Total

4.2.5 Attacks

4.2.5.1 Reentrancy

This attack involves functions that are called repeatedly before the first invocation of the function is finished. This may cause the different invocations of the function to interact in destructive ways. For example, let us consider the code in Figure 4.6. Because the user's balance is not set to 0 until the very end of the function, the second (and later) invocations will still succeed, and will withdraw the balance over and over again. We made sure in our code if there are any external function calls, they are called after all the internal changes have been made.



```
mapping (address => uint) private userBalances;

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    require(msg.sender.call.value(amountToWithdraw)());
    userBalances[msg.sender] = 0;
}
```

Figure 4.6: Insecure code

4.2.5.2 Integer Overflow and Underflow

We did not have to consider anything for these attacks as the ethereum virtual machines reverts when Over/Underflows occur, as of solidity version 0.8.0.

4.2.5.3 Transaction-Ordering Dependence (TOD) / Front Running

Before a transaction is added to a block, it resides in a place called a mempool, which can be queried. Since a transaction is in the mempool for a short while, one can know what actions will occur, before it is included in a block. We use a commitment scheme to prevent this from happening. The input is hidden so an attack or another voter cannot act upon what ever information they received from the mempool. We use block.timestamp which is used to determine when to close voting, but given that the time taken for a ethereum [ethereum.org] block to be added is 12-14 seconds , we can assume that this is the limit of time manipulation. The worst case here is that the election closes 12-14 seconds later.

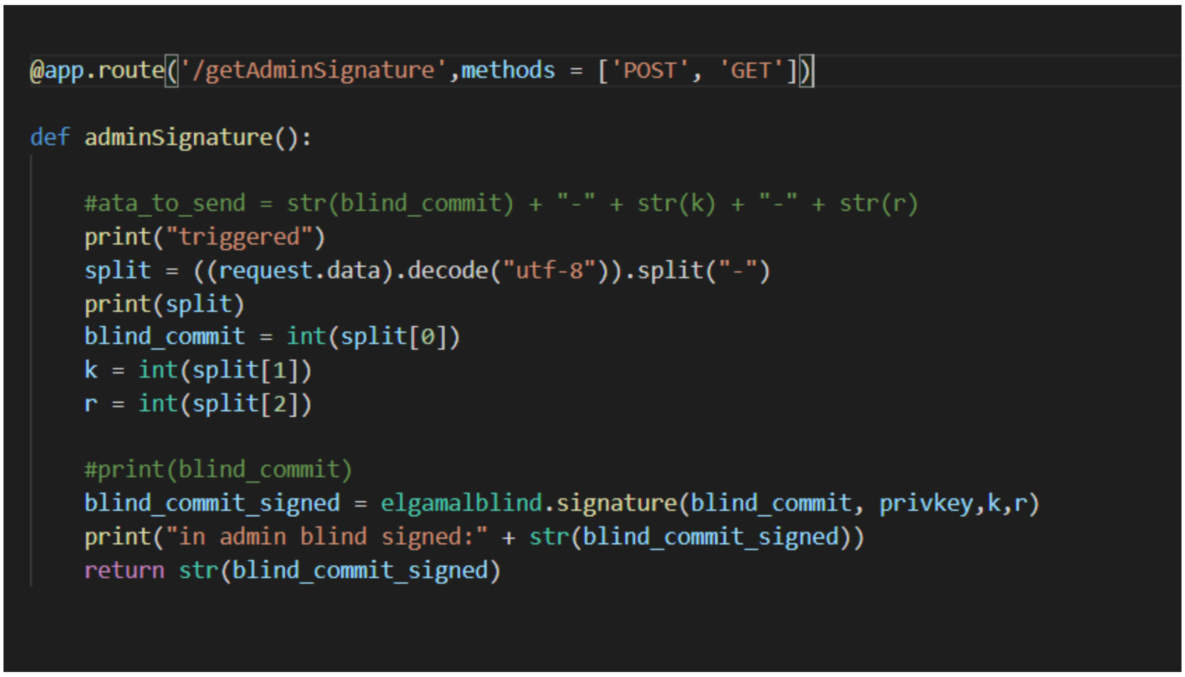
Chapter 5

Conclusions and future work

We have used an existing e-voting protocol and implemented it using blockchain. Although the protocol needs to be improved to make it truly scalable, for 40 voters it still performs better than the OpenVote network.

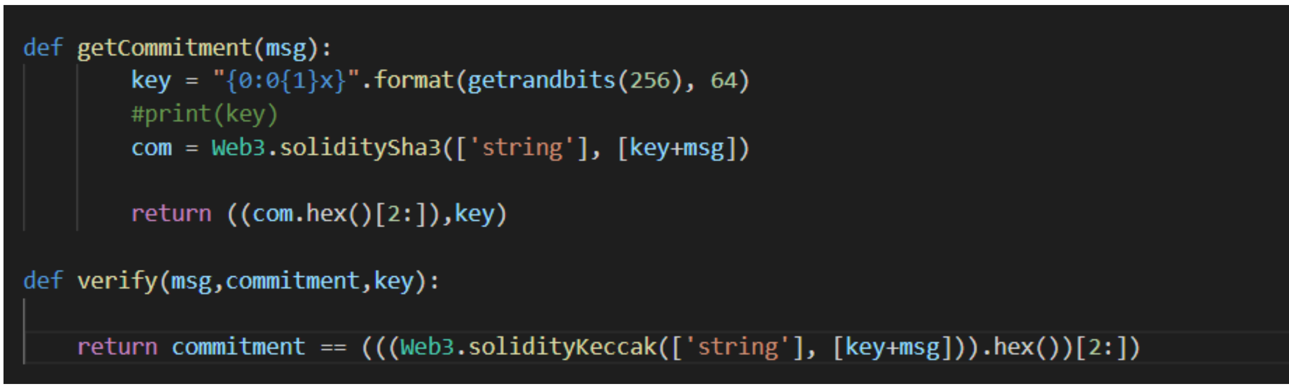
5.0.1 future work

1. Test the protocol using infura.
2. Improve User Interface



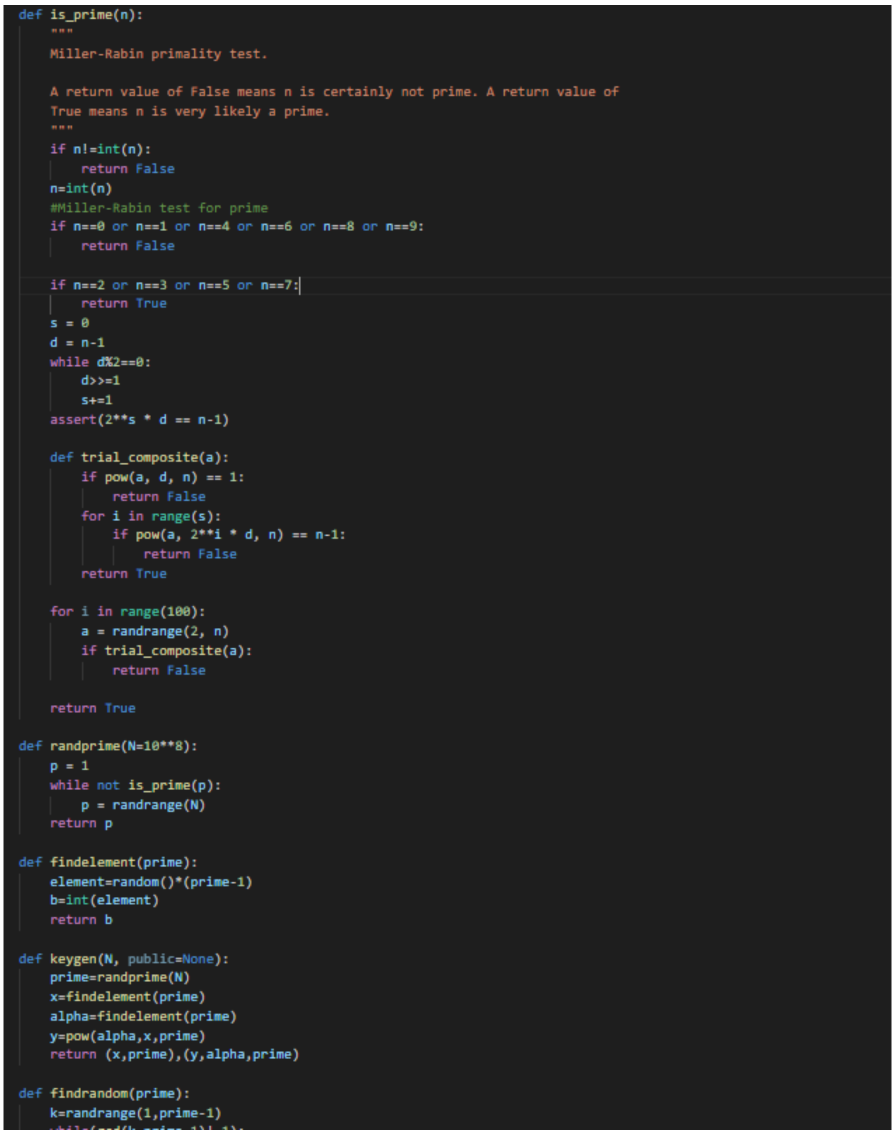
```python
@app.route('/getAdminSignature',methods = ['POST', 'GET'])

def adminSignature():

    #ata_to_send = str(blind_commit) + "-" + str(k) + "-" + str(r)
    print("triggered")
    split = ((request.data).decode("utf-8")).split("-")
    print(split)
    blind_commit = int(split[0])
    k = int(split[1])
    r = int(split[2])

    #print(blind_commit)
    blind_commit_signed = elgamalblind.signature(blind_commit, privkey,k,r)
    print("in admin blind signed:" + str(blind_commit_signed))
    return str(blind_commit_signed)
```

Figure 1: blind signing admin



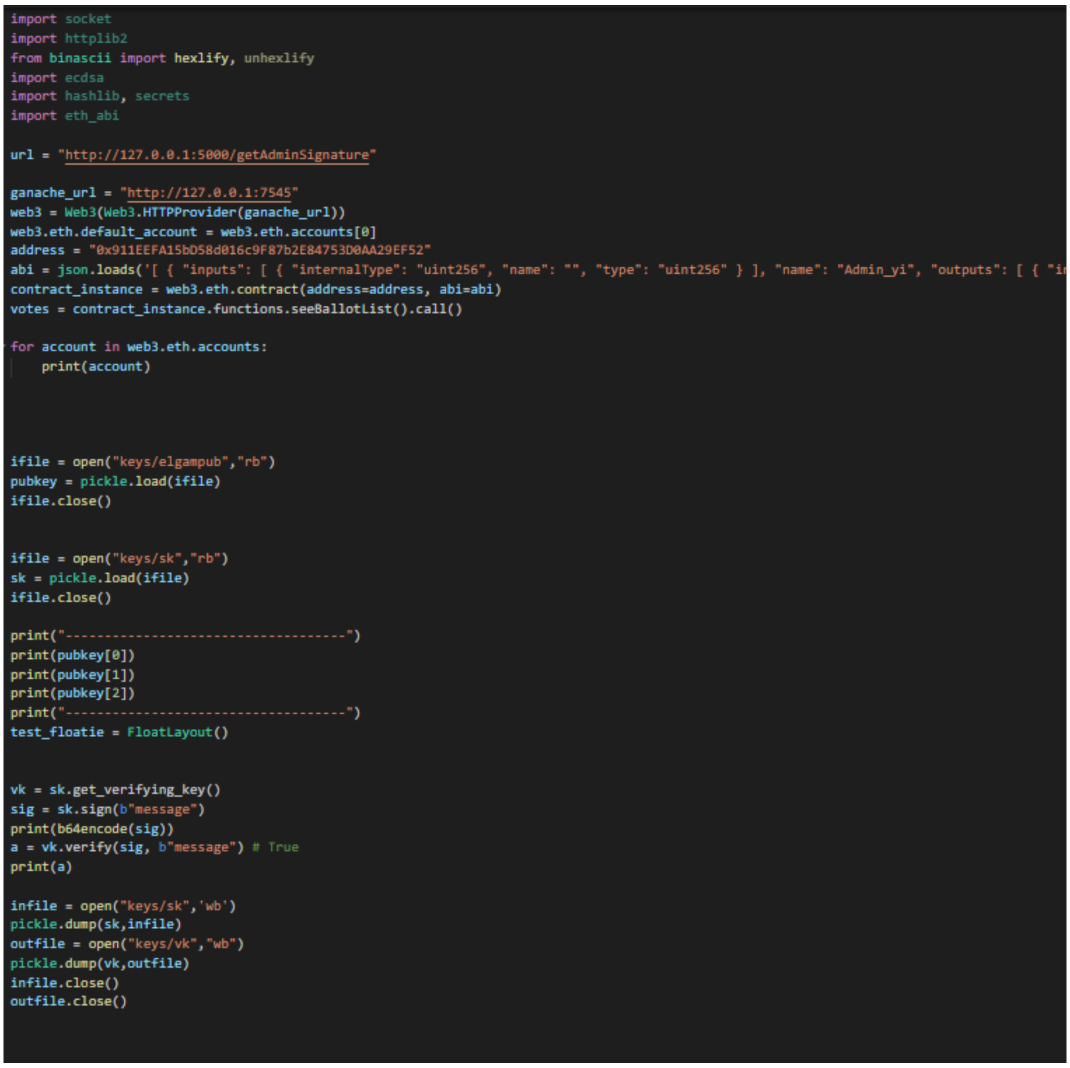
```python
def getCommitment(msg):
        key = "{0:0{1}x}".format(getrandbits(256), 64)
        #print(key)
        com = Web3.soliditySha3(['string'], [key+msg])

        return ((com.hex()[2:]),key)

def verify(msg,commitment,key):

    return commitment == (((Web3.solidityKeccak(['string'], [key+msg])).hex())[2:])
```

Figure 2



```python
def is_prime(n):
    """
    Miller-Rabin primality test.

    A return value of False means n is certainly not prime. A return value of
    True means n is very likely a prime.
    """
    if n!=int(n):
        return False
    n=int(n)
    #Miller-Rabin test for prime
    if n==0 or n==1 or n==4 or n==6 or n==8 or n==9:
        return False

    if n==2 or n==3 or n==5 or n==7:
        return True
    s = 0
    d = n-1
    while d%2==0:
        d>>=1
        s+=1
    assert(2**s * d == n-1)

    def trial_composite(a):
        if pow(a, d, n) == 1:
            return False
        for i in range(s):
            if pow(a, 2**i * d, n) == n-1:
                return False
        return True

    for i in range(100):
        a = randrange(2, n)
        if trial_composite(a):
            return False

    return True

def randprime(N=10**8):
    p = 1
    while not is_prime(p):
        p = randrange(N)
    return p

def findelement(prime):
    element=random()*(prime-1)
    b=int(element)
    return b

def keygen(N, public=None):
    prime=randprime(N)
    x=findelement(prime)
    alpha=findelement(prime)
    y=pow(alpha,x,prime)
    return (x,prime),(y,alpha,prime)

def findrandom(prime):
    k=randrange(1,prime-1)
```

Figure 3



```python
import socket
import httplib2
from binascii import hexlify, unhexlify
import ecdsa
import hashlib, secrets
import eth_abi

url = "http://127.0.0.1:5000/getAdminSignature"

ganache_url = "http://127.0.0.1:7545"
web3 = Web3(Web3.HTTPProvider(ganache_url))
web3.eth.default_account = web3.eth.accounts[0]
address = "0x911EEFA15bD58d016c9F87b2E84753D0AA29EF52"
abi = json.loads('[ { "inputs": [ { "internalType": "uint256", "name": "", "type": "uint256" } ], "name": "Admin_yi", "outputs": [ { "i
contract_instance = web3.eth.contract(address=address, abi=abi)
votes = contract_instance.functions.seeBallotList().call()

for account in web3.eth.accounts:
    print(account)



ifile = open("keys/elgampub","rb")
pubkey = pickle.load(ifile)
ifile.close()


ifile = open("keys/sk","rb")
sk = pickle.load(ifile)
ifile.close()

print("------------------------------------")
print(pubkey[0])
print(pubkey[1])
print(pubkey[2])
print("------------------------------------")
test_floatie = FloatLayout()


vk = sk.get_verifying_key()
sig = sk.sign(b"message")
print(b64encode(sig))
a = vk.verify(sig, b"message") # True
print(a)

infile = open("keys/sk",'wb')
pickle.dump(sk,infile)
outfile = open("keys/vk","wb")
pickle.dump(vk,outfile)
infile.close()
outfile.close()
```

Figure 4

Bibliography

[1] Evoting - https://en.wikipedia.org/wiki/Electronic_voting_in_India

[2] ”Helios: Web-Based Open-Audit Voting” by Ben Adida https://www.usenix.org/legacy/events/sec08/tech/full_papers/adida/adida.pdf

[3] Helios Website - https://vote.heliosvoting.org/

[4] ”On Secure E-Voting over Blockchain” by Patrick MCcorry et al. - https://www.dcs.warwick.ac.uk/~fenghao/files/main-DTRAP-final.pdf

[5] ”Veriable Secret-Ballot Elections” by Josh Daniel Cohen Benaloh https://www.microsoft.com/en-us/research/wp-content/uploads/1987/01/thesis.pdf

[6] LIGHTPD website - https://www.lighttpd.net/

[7] Metamask - https://metamask.io/

[8] Etherscan - https://ropsten.etherscan.io/

[9] Solana - https://solana.com/

[10] Json-Rpc - https://www.jsonrpc.org/

[11] Solana help - https://solana.com/news/getting-started-with-solana-development

[12] SHA-256 https://en.wikipedia.org/wiki/SHA-2

[13] Digital Signature Diagram https://www.docusign.co.uk/how-it-works/electronic-signature/digital-signature/digital-signature-faq

[14] ”Blind Signatures for untraceable payments” by David Chaum in 1983 https://sceweb.sce.uhcl.edu/yang/teaching/csci5234WebSecurityFall2011/Chaum-blind-signatures.PDF

[15] ”Minimum disclosure proofs of knowledge” by Gilles Brassard, David Chaum and Claude Crépeau in 1988 https://www.sciencedirect.com/science/article/pii/0022000088900050

[16] ”A practical secret voting scheme for large scale elections” by Atsushi Fujioka, Tatsuki Okamoto and Kazuo Ohta from NTT Laboratories https://link.springer.com/chapter/10.1007/3-540-57220-1_66

[17] ”On Secure E-Voting over Blockchain” by Patrick Mccorry et al https://www.dcs.warwick.ac.uk/~fenghao/files/main-DTRAP-final.pdf

[18] ”A blind signature scheme based on ElGamal signature” by E. Mohammed et al https://ieeexplore.ieee.org/document/874771

[19] ”A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms” by TAHER ELGAMAL in 1985 https://caislab.kaist.ac.kr/lecture/2010/spring/cs548/basic/B02.pdf

[20] ”Flask Documentation” https://flask.palletsprojects.com/en/2.1.x/

[21] Python Requests https://docs.python-requests.org/en/latest/

[22] Kivy for Python https://kivy.org/#home

[23] ”A Peered Bulletin Board for Robust Use in Verifiable Voting Systems” by Chris Culnane and Steve Schneider Department of Computing, University of Surrey https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6957110

[24] ”On the Security Properties of e-Voting Bulletin Boards” by Aggelos Kiayias, Annabell Kuldmaa, Helger Lipmaa, Janno Siim and Thomas Zacharias https://homepages.inf.ed.ac.uk/tzachari/BBs.pdf

[25] ”A Smart Contract for Boardroom Voting with Maximum Voter Privacy” by Patrick McCorry, Siamak F Shahandashti, Feng Hao https://www.dcs.warwick.ac.uk/~fenghao/files/openvotenetwork.pdf

[26] ”A Method for Obtaining Digital Signatures and Public-Key Cryptosystems” by R.L. Rivest, A. Shamir, and L. Adleman in 1978 https://people.csail.mit.edu/rivest/Rsapaper.pdf

[27] ”Comparative Analysis of RSA and ElGamal Cryptographic Public-key Algorithms” by Andysah Putera Utama Siahaan, Elviwani , Boni Oktaviana https://eudl.eu/pdf/10.4108/eai.23-4-2018.2277584

[28] ”A Review on Asymmetric Cryptography – RSA and ElGamal Algorithm” by AnnapoornaShetty, Shravya Shetty K, Krithika K

[29] Nist first proposal for a digital signature scheme https://csrc.nist.gov/publications/detail/fips/186/1/archive/1998-12-15

[30] Aes developed by NIST https://www.nist.gov/publications/advanced-encryption-standard-aes

[31] ”A Method for Obtaining Digital Signatures and Public-Key Cryptosystems” by R.L. Rivest, A. Shamir, and L. Adleman. Published in Communications of the ACM

[32] Python web3 documentation https://web3py.readthedocs.io/en/stable/

[33] Gas and fees documentation https://ethereum.org/en/developers/docs/gas/

[34] ”ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER” by Dr Gavin Wood https://ethereum.github.io/yellowpaper/paper.pdf

[35] Blind signature python coade https://github.com/iAmServerless/Blind-Signature/tree/master/Elgamal

[36] Uk General election cost https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/715422/The_Costs_of_the_2015_UK_Parliamentary_General_Election.pdf

[37] Voter data https://www.idea.int/data-tools/country-view/137/40