AnonEvote
A Project on Blockchain E-voting
Technical Guide
Supervisor: David Gray

Michael Wall
13522003

Monday 22nd May, 2017

Abstract

The AnonEvote system is a proof of concept for a distributed trustless voting system, that provides user anonymity and tamper-proof ballots. It makes use of Homomorphic cryptography and a blockchain database on a peer to peer network to achieve this functionality.

Contents

1 Introduction 4

1.1 Overview . . . 4

1.2 Glossary . . . 4

2 Motivation 5

3 Research 5

3.1 Homomorphic encryption schemes . . . 6

3.2 Blockchain structure . . . 6

3.3 Key distribution . . . 7

3.4 Key recreation . . . 7

4 Design 7

4.1 High Level Design . . . 7

4.2 Detailed design . . . 9

4.2.1 Peer synchronisation . . . 9

4.2.2 Creation of blocks . . . 9

4.2.3 Handling block updates . . . 9

4.2.4 Broadcasting key shares . . . 10

4.2.5 User voting . . . 10

5 Implementation 10

5.1 Cryptography . . . 10

5.1.1 Paillier cryptographic scheme . . . 10

5.1.2 Shamir's secret sharing . . . 14

5.1.3 Lagrange Interpolation . . . 15

5.1.4 DSA Signatures . . . 16

5.2 The blockchain . . . 17

5.2.1 Structure of the chain . . . 17

5.2.2 Main goroutines 19
5.2.3 Chain core API 23
5.2.4 Main process 26
5.3 Blocks, transactions and vote tokens 29
5.4 Ballots 31

6 Validation **32**

7 Problems solved **32**
7.1 Node communication 32
7.2 Building a blockchain from scratch 34
7.3 Consensus algorithm 34

8 Future work **34**
8.1 Distribution of vote tokens 35
8.2 Genesis block creation 35
8.3 User interface 36
8.4 Cryptographic validation 36
8.5 Zero-knowledge proofs 36
8.6 Integration with other crypto-currencies 36
8.7 More than one election per chain 36
8.8 Trustless creation of the election key 37

1 Introduction

1.1 Overview

The AnonEvote system is a distributed trustless voting system, that provides user anonymity and tamperproof ballots. Users of the system are represented as a DSA public key and a vote token. Users are authenticated to vote via their assigned vote token. This vote token is a randomly generated sequence of characters. Each user is also given the complete list of vote tokens and their corresponding public keys, so that they can verify that any given token is valid and only being spent by the authorized user. Users connect to each other via a peer to peer network. There is no centralized server to get the network started. User's can specify a list of any peers they are aware about in their config file. This will bootstrap the peer discovery process. When peers are connected, they will talk to each other about peers that they know and update their list of known peers regularly. This makes the network resistant to failure due to "core" nodes dropping out of the network. The blockchain database is built on top of this peer to peer network, and helps in providing tamper resistance to the ballots. Homomorpic cryptography is used to ensure that the tally can be calculated easily with fewer decryption operations required.

1.2 Glossary

Node A node is some machine which is running the AnonEvote client software. Any machine can be a node in the system if it is running the client software and has a working internet connection.

Peer-to-peer network A peer-to-peer network is a distributed networking architecture in which peer nodes are equal and can act as both a "client" and a "server" for other nodes on the network. It eliminates the need for a centralized server.

Blockchain The blockchain is the distributed database which maintains the ledger of ballots. It is broadcast to all participants in the system via the P2P network. Each client will maintain an up-to-date version of the ledger to the best of their knowledge. Each block in the chain is linked to the previous block by the hash of that block. The hash of the previous block is used in creating hte hash of the current block, meaning that if one of the blocks is changed, it's new hash will "break" the chain.

Ledger The ledger is a growing record of all ballots which have been cast. The ledger is comprised of a series of blocks, each block referring to the previous block to form a chain.

Block Each block contains a set number of transactions. Transactions are publicly viewable, but their contents are secured using cryptographic encryption.

Transaction Each transaction in a block contains a single ballot, which has been encrypted to provide anonymity and to prevent tampering.

Ballot A ballot is the form which a voter fills out to cast their vote.

Homomorphic encryption A homomorphic scheme is an encryption system which allows operations to be carried out on the ciphertext which, when decrypted, matches the same operations as though they were carried out on the plaintext value. We use this to our advantage to tally encrypted ciphertexts.

Vote token In this system, a Vote Token is string of charcters which acts as a token to verify a user's ability to cast a vote.

Trustless In a voting system, trust means that you are dependent on a certain number of individuals to carry out key core tasks such as collecting ballots, transporting ballots, tallying the vote, announcing results. In a trustless system, you do not depend on any one person to behave correctly, rather everyone is responsible for these tasks to some degree.

2 Motivation

I became interested in blockchains last year seeing emerging uses for the technology in sectors such as distribution, digital rights management, patent systems, smart contracts and others. I was also generally interested in voting systems at the time and interested in cryptography and security. I was initially looking to find a new application for a blockchain database, and I believed it could help to solve the issue of trust in a voting system.

3 Research

I knew that when I began this project that a large proportion of the work would be research. I understood that not alll of the research would be possible at the beginning. There were a number of areas that I could learn about however prior to the design stage of the project.

3.1 Homomorphic encryption schemes

I had not decided on an encryption scheme prior to the submission of the functional specification. Upon further research into the schemes available, I decided on the use of the Paillier cryptographic scheme.

This scheme offers a number of benefits including: Homomorphic addition of ciphertexts: this would allow the tally to be completed without requiring a decryption to be performed on individual ballots, as the ballots can be summed together in their encrypted state. The summation of ciphertexts requires fewer operations and simplifies the entire tallying process.

Randomised encryptions of plaintexts: to use the Paillier cryptosystem for our system, each user would have to encrypt their vote using the same public key. In some systems, this could be problematic, as the encryption of the same message with the same key would yeild the same ciphertext: $E(m) == C, E(m) == E(m)$ However, the Paillier cryptosystem provides a randomization element which allows for the same plaintext message to be encrypted with the same key and have a different ciphertext each time: $E(m1) == c1, E(m2) == c2, m1 == m2, c1! = c2$

3.2 Blockchain structure

I understood the theory behind how blockchains worked to an extent before beginning the project, but I did not have a deep knowledge of the inner workings of the data structure, or its consensus forming algorithm.

The blockchain is a distributed database which maintains a growing ledger of records called blocks. Each block is time stamped and linked to the previous block via its hash to create the chain. The data in a block cannot be altered without breaking the chain. Each block can contain a number of transactions. For the purposes of this system, each transaction contains the a user's ballot.

Because the blockchain is distributed, numerous versions of the ledger can exist at any one time. This presents the issue of not knowing which version of the ledger is correct. This is dealt with using a combination of a proof of work and a consensus algorithm. The consensus algorithm in essence means that nodes will agree on a certain version of the chain as being the correct version by using it as the basis for the next block that they create. Nodes decide which version to use via a longest chain decision. If a node hears about a new chain which is longer than their own, they should adopt it and use it. To prevent a node from creating an artificially long chain, proof of work is required. Proof of work entails a node finding a partial hash collision for the hash of a block. Because this operation is computationally expensive, we can take the length of the chain as meaning that a certain amount of effort or work has gone into creating this chain.

3.3 Key distribution

One of the viable schemes to accomplish the goal of key sharing I found was Shamir's Secret Sharing scheme: `https://cs.jhu.edu/~sdoshi/crypto/papers/shamirturing.pdf` It involves creating a polynomial of K degrees to represent the secret (in this case the private key of the election), and then constructing N points from the polynomial for N participants where $N >= K$. A polynomial of K degrees will require K of these points in order to retrieve the secret.

The idea behind this is that a treshold can be set in the case of one or more of the participants not being present to reconstruct the secret. This scheme allows for the private exponents of the secret key to be distributed amongst nodes for reconstruction later.

3.4 Key recreation

The Lagrange basis polynomial: `https://en.wikipedia.org/wiki/Lagrange_polynomial` can be constructed using the any subset of K points in order to get the secret value, represented by the constant a_0 in the polynomial: $f(x) = a_0 + a_1.x + a_2.x^2 + ... + a_{K-1}.x^{k-1}$

Note that the order of the points does not matter in the interpolation, only that there are enough of them available.

4 Design

4.1 High Level Design

Following my initial research, I decided upon a system architecture with the help of my supervisor. There were some of the initial goals which I had set out in my functional specification document which turned out to be impractical to implement. The final high level design which was decided upon is as follows.

Each user would be assigned a randomized unique string from a list of known strings, in such a way that no one knows who is assigned which string. Each of these tokens is also tied to a public DSA key, whose corresponding private key belongs to the user who is authorized to use the vote token. The assumption was made that this assigning would be handled by the individuals who were using the system. A single trusted individual would create a keypair for the election. While initially I intended to have a completely trustless system, it is not currently feasible to create an private key in an already sharded form such that there is no point in time at which one person has access to the private key. This does not compromise user identities, but may compromise the premature decryption of the finaly tally. The trusted individual proceeds to construct a number of shareable keys from the private key, as described in Shamir's secret sharing scheme. These keys are distributed to nodes of the system, and the original private key should be destroyed.

The public key for the election is public knowledge in the system. With this knowledge, a user can now create a vote, and encrypt it with the public election key. Each node sends both their encrypted vote, along with their unique Vote Token which verifies their eligibility to vote, to a number of nodes in the system. The node must sign their transaction with their private DSA key. This allows other nodes to validate the voter, and to prevent another node from misbehaving and using a vote token which does not belong to them.

Nodes will then collect transactions until they have enought to create a block. At this stage, the node will form the block and begin to compute the proof of work. If the node creates a valid block, they will add it to their chain and broadcast it to their peers. If however, another node creates a new valid block, the node will accept the block and begin working on the next block. If a node were to decide not to accept a block, but instead to continue to create its own, their block would likely not be accepted by the network since the amount of computing power in the network will outweigh their own. This means that the network will create new blocks faster than the misbehaving node, and so it can never create a chain that is long enough to have the entire network accept it.

Once the election is over, nodes should collaborate to reform the secret private key. They do this by collecting key shares, and broadcasting them to their peers. With an appropriate threshold set for the interpolation of the private key from the secrets, the secret key should never be discovered prematurely due to collusion from misbehaving nodes. When the key is reformed any user can then tally the results of the votes in their version of the ledger.

4.2 Detailed design

The design of each node needed to cater for the communication across the network, along with the processing of all data to ensure the successful running of the system without the need for a centralized server. Each node runs a set of core processes which allow it to function and interact with other nodes. These are the scheduling of peer synchronisation, creation of blocks, handling block updates from the network, and scheduling the broadcast and collection of key shares.

4.2.1 Peer synchronisation

For peer synchronisation to be successful, a node must be aware of at least one other node that is connected to the network. Each node will then periodically poll their known peers, and with each one they will combine their known peers together. This allows for a node to gain additional peers in case some nodes go offline, or they are unable to communicate due to network errors.

4.2.2 Creation of blocks

A node will periodically check to see if they have enough transactions available in their pool to create a block. Once they do, the process of computing the proof of work begins. This process will continue until a block is found or until the process is interrupted, at which stage the process will wait for a signal to restart the creation of blocks. Assuming a node creates a block, it will asynchronously begin to send out its new block to the peers on the network.

4.2.3 Handling block updates

This is where the core of the consensus algorithm takes place. This process will listen to a channel for updates from the network about new blocks. If a block is received which is the next in a node's current chain, it is added to the chain and a signal is sent to the block creation process to stop working. Once the update has been applied, a signal to restart work is sent to the block creation process. If a block is received which is not next in the chain, but claims to be from a longer chain, the node will attempt to retrieve this chain from the peer who created the block. If the alternative chain is in fact longer, the node will validate its contents. If valid, the node will again send a signal to stop creating blocks, and apply the new chain, restarting when the update is complete. Any updates which do not satisfy the afformentioned conditions are simply ignored. This means that shorter chains, or invalid chains are ignored, and forgotten by the network.

4.2.4 Broadcasting key shares

A node will periodically broadcast its known set of key shares to their peers. At any stage a user can attempt to interpolate a set of shares, but will be unsuccessful unless they have at least the threshold number of shares.

4.2.5 User voting

A user can cast their vote by filling out the ballot via the node. The node will encrypt and sign the ballot, and then send it to its peers for processing.

5 Implementation

The project was implemented in the Go programming language. I decided upon this language because Go has rich support for concurrency using goroutines and channels. A goroutine is a function that is capable of running concurrently with other functions. A channel provides a way for two goroutines to communicate with each other and to synchronize their execution.

I used these features of the language to my advantage to help to design the structure of a node, its processes and the intercommunication between them.

The most recent documentation for the project will always be available on the Godoc website: `https://godoc.org/github.com/CPSSD/voting/src/`

5.1 Cryptography

As with the blockchain API, the full documentation for the crypto API can be found at: `https://godoc.org/github.com/CPSSD/voting/src/crypto`

5.1.1 Paillier cryptographic scheme

I decided to implement the Paillier scheme myself in Go as there were no reliable libraries available at the time in the Go standard `crypto` library. A private key is used as the Election private key, and should be kept secret until the end of the election.

A private key is defined in the Paillier scheme as follows:

```
type PrivateKey struct {
    Lambda *big.Int
    Mu     *big.Int
    PublicKey
}

type PublicKey struct {
    N         *big.Int
    NSquared  *big.Int
    Generator *big.Int
}
```

The values `Lambda` and `Mu` are secret, and must be broken into shares to be distributed to voters in the system.

The following main operations are available under the `crypto` package for Paillier keys:

```
// Encrypt returns the ciphertext c which is created by
// encrypting the message m with the PrivateKey key. The
// encryption of a given message m is non-deterministic.
func (key *PublicKey) Encrypt(m *big.Int) (c *big.Int, err error) {

    if m == nil {
        return nil, InvalidPlaintextError
    }
    if err = key.Validate(); err != nil {
        return nil, err
    }

    r, err := rand.Int(rand.Reader, key.N)
    if err != nil {
        return nil, err
    }

    // c = ((g^m).(r^n)) mod (n^2)
    c = new(big.Int).Mod(
        new(big.Int).Mul(
            new(big.Int).Exp(key.Generator, m, key.NSquared),
            new(big.Int).Exp(r, key.N, key.NSquared)), key.NSquared)

    return c, err
}

// Decrypt returns the message m which is obtained from
// decrypting the ciphertext c using PrivateKey key. If
// a nil ciphertext is passed to this function, an
// InvalidCiphertextError will be returned along with a
// nil value for m.
// If an invalid key is used, a corresponding error will
// be returned with a nil value for m.
func (key *PrivateKey) Decrypt(c *big.Int) (m *big.Int, err error) {

    if c == nil {
        return nil, InvalidCiphertextError
    }
    if err = key.Validate(); err != nil {
        return nil, err
    }

    // m = L(c^lambda mod n^2).mu mod n
    // where L(x) = (x-1)/n
    m = new(big.Int).Exp(c, key.Lambda, key.PublicKey.NSquared)
    m = getL(m, key.PublicKey.N)
    m.Mul(m, key.Mu)
    m.Mod(m, key.PublicKey.N)

    return m, err
}

// GenerateKeyPair returns a PrivateKey struct containing the
// private components of a key-pair and the corresponding
// PublicKey struct. The value bits determines the size of
// prime numbers to be used in the generation of the key-pair.
func GenerateKeyPair(bits int) (privateKey *PrivateKey, err error) {
    n, lambda, err := generatePrimePair(bits)
    if err != nil {
        return nil, err
    }

    mu := getMu(lambda, n)
    generator := new(big.Int).Add(n, one)

    nSquared := new(big.Int).Mul(n, n)

    privateKey = &PrivateKey{
        PublicKey: PublicKey{
            N:         n,
            NSquared:  nSquared,
            Generator: generator,
        },
        Lambda: lambda,
        Mu:     mu,
    }

    err = privateKey.Validate()

    return
}
```

This scheme also allows for the Homomorphic addition of a set of ciphertexts. This process is as follows. Addition of ciphertexts only requires access to the public key, so any node can sum the ciphertexts without access to the full private key. Note that the number of operations to add two ciphertexts together is less than the number of operations to decrypt a ciphertext:

```
// AddCipherTexts accepts one or more ciphertexts
// and returns the homomorphic sums of them.
func (key *PublicKey) AddCipherTexts(ciphertexts ...*big.Int)
(total *big.Int, err error) {

    if err = key.Validate(); err != nil {
        return nil, err
    }
    // create an encryption of voting value zero to start off
    zero := new(big.Int)
    total, err = key.Encrypt(zero)
    if err != nil {
        return nil, err
    }

    // D(E(m1,r1).E(m2,r2) mod n^2) = m1 + m2 mod n
    for _, ciphertext := range ciphertexts {
        total = new(big.Int).Mul(total, ciphertext)
        total.Mod(total, key.NSquared)
    }

    return total, nil
}
```

5.1.2 Shamir's secret sharing

The private exponents of the Paillier key should not be stored anywhere. Instead we want to divide these values into a number of shares that can later be used to recreate the key's `Lambda` and `Mu` values.

To do this, we make use of Shamir's secret sharing scheme. This scheme allows us to create a number of shares `m` with a threshold value `k`. This means that we will only need `k` shares in order to recreate the value of a given secret. This feature is useful in case some nodes never contribute their secret or are unable to. It also means that at least `k` nodes would need to collude in order to compromise the secret values of the key prematurely to the end of the election. Note that having access to the private key will not allow a node to change a ballot's contents, as this is prevented by the nature of the blockchain.

The creation of a set of shares from a secret is carried out as follows:

```
// DivideSecret creates a k-threshold secret and returns
// a slice of m shares. If m < k, then k shares will be
// created. The prime modulus used to create the shares is
// also returned, as this is required in order to interpolate
// the shares into the correct polynomial.
func DivideSecret(secret *big.Int, k, m int) (shares []Share, prime *big.Int, err e

    // generate a prime P > s
    prime, _ = rand.Prime(rand.Reader, 1+secret.BitLen())

    // We need k total parts for reconstruction, so
    // we will use s as the first monomial, and k-1 extra parts.
    poly := polynomial{
        monomials: []monomial{{secret, new(big.Int)}},
    }

    for i := int64(1); i < int64(k); i++ {
        // select ai where a < P, s < P
        value, err := rand.Int(rand.Reader, prime)
        if err != nil {
            return nil, nil, err
        }
        poly.monomials = append(poly.monomials, monomial{value, big.NewInt(i)})
    }

    // Using the polynomial, construct n shares
    // of the secret. Constructing a share involves
    // soling the polynimial for x to get the point (x, y)
    for i := int64(1); i <= int64(m); i++ {
        yVal := poly.solve(big.NewInt(i))
        shares = append(shares, Share{big.NewInt(i), new(big.Int).Mod(yVal, prime)
    }

    // Return the set of shares.
    return
}
```

The slice of shares should be distributed to nodes evenly, along with the corresponding prime value which is used in the reconstruction of the secret.

5.1.3 Lagrange Interpolation

The shares created above were created using a Lagrange basis polynomial. The secret value can be reconstructed by interpolating the polynomial from a set of points (the shares that we created previously). The interpolation function is implemented as follows:

```
// Interpolate takes a Share slice and a prime modulus,
// and interpolates the shares to create a polynomial.
// It returns a secret value = f(0) for the constructed
// polynomial. If the amount of shares used is not
// greater than or equal to the original threshold for
// the polynomial, the returned secret will not be correct.
func Interpolate(points []Share, prime *big.Int) (secret *big.Int, err error) {

    secret = new(big.Int)

    // get the sum from j = 0, to k-1 of:
    // f(xj) . the product from m = 0, m != j, to k-1 of:
    // xm /( xm - xj )

    for _, j := range points {
        subProduct := calculateProduct(j, points, prime)
        subSecret := new(big.Int).Mul(j.Y, subProduct)
        //fmt.Println("(",j.Y,".",subProduct,") +",)
        secret = new(big.Int).Add(secret, subSecret)
    }

    secret = new(big.Int).Mod(secret, prime)
    return
}
```

5.1.4 DSA Signatures

The creation and verification of DSA signatures was done using the standard `crypto` library in the Golang standard library. The signing and verification process was wrapped as follows to allow easy of use:

```
type Signature struct {
    R *big.Int
    S *big.Int
}

func SignHash(privateKey *dsa.PrivateKey, hash *[32]byte) (sig *Signature) {

    r := big.NewInt(0)
    s := big.NewInt(0)

    r, s, err := dsa.Sign(rand.Reader, privateKey, hash[:])
    if err != nil {
        log.Println("Error signing the hash")
        log.Fatalln(err)
    }

    sig = &Signature{
        R: r,
        S: s,
    }

    return sig
}

func Verify(pubkey *dsa.PublicKey, hash *[32]byte, sig *Signature) (valid bool) {

    return dsa.Verify(pubkey, hash[:], sig.R, sig.S)
}
```

5.2 The blockchain

5.2.1 Structure of the chain

This is this the structure of the chain I implemented.

```
type Chain struct {
    Peers               chan map[string]bool
    TransactionPool     chan []Transaction
    TransactionsReady   chan []Transaction
    CurrentTransactions chan []Transaction
    BlockUpdate         chan BlockUpdate
    KeyShares           chan map[string]ElectionSecret
    SeenTrs             chan map[string]bool
    head                *Block
    blocks              chan []Block
    conf                Configuration
}
```

Internally the configuration of the chain is as follows.

```
type Configuration struct {
    MyAddr    string
    MyPort    string
    Peers     map[string]bool
    SyncPeers bool

    PrivateKey dsa.PrivateKey

    VoteTokens map[string]dsa.PublicKey
    MyToken    string

    ElectionFormat election.Format

    ElectionKey           crypto.PrivateKey
    ElectionKeyShare      ElectionSecret
    ElectionLambdaModulus *big.Int
    ElectionMuModulus     *big.Int
}
```

Note that the `Peers` map in the configuration is used for the initailization of the peer map in the chain itself. The values in the configuration are never written to concurrently, and so do not require management through the use of channels.

Channels are used to share memory between the goroutines of the chain. For example, when a new `BlockUpdate` is received, it is written to the BlockUpdate channel as follows:

```
func (c *Chain) ReceiveBlockUpdate(blu *BlockUpdate, _ *struct{}) (err error) {
    log.Println("Received block update, writing to respective channel")
    c.BlockUpdate <- *blu
    return
}
```

In this instance, writes to the channel will block if it is full, and reads will block until data is available to read. The corresponding consumer for this channel will wait for data as follows:

```
loop:
for {
    select {
    case <-quit:
        log.Println("Chain update process received signal to shutdown")
        ...
        break loop
    case blu := <-c.BlockUpdate:
        log.Println("Handling block update")
        ...
    }
}
```

Here, the update handling process runs a for loop, with a select statement with two cases. Note that here, both `quit` and `c.BlockUpdate` are channels. The select statement will wait until some data is written to one of the two channels. When data is available in `c.BlockUpdate`, it is written to the variable `blu` and then it is handled in the proceeding lines.

In fact, this looped `select` statement forms the basis for the main goroutines in the program.

5.2.2 Main goroutines

Peer synchronisation is handled as follows:

```
func (c *Chain)
schedulePeerSync(syncDelay int, quit chan bool, wg *sync.WaitGroup) {
    timer := time.NewTimer(time.Second)
    loop:
    for {
        select {
        case <-quit:
            log.Println("Peer syncing process received signal to shutdown")
            quit <- true
            wg.Done()
            break loop
        case <-timer.C:
            log.Println("About to sync peers")
            c.syncPeers()
            timer = time.NewTimer(time.Second * time.Duration(syncDelay))
        }
    }
}
```

When this goroutine begins, it starts a timer which will decide when to synchronise peers. If the `quit` channel receives some data, it means that we are trying to quit the program. Firstly, the goroutine will write back some data to the channel, to let the other concurrently running goroutines know that we are trying to quit. Next it will signal to the `sync.WaitGroup wg` that we have finished with our process and have quit. For the `timer`, there is a channel associated with the channel accessible through `timer.C`. When the time allocated expires, some data is written to the channel. We use this as the signal to call `c.syncPeers()`. When this is completed, we will set a new timer before we try to synchronise our peers again.

For the handling of mining, again a similar process is taking place:

```
func (c *Chain) scheduleMining(quit, stopMining, startMining,
    confirmStopped chan bool, wg *sync.WaitGroup) {

timer := time.NewTimer(time.Second)
start:

log.Println("Waiting for the signal to start mining")
_ = <-startMining
log.Println("Got the signal, about to start mining")

loop:
for {
    select {

    default:
        // By default, we wait for timer to expire, then we will check
        // to see if there are enough transactions in the pool that we
        // can create a block from.
        _ = <-timer.C

        // Get the pool and see if it is longer than the constant blockSize
        pool := <-c.TransactionPool
        if len(pool) >= blockSize {
            // if so, we will put blockSize worth of transactions into
            // the TransactionsReady channel, and replace the rest of the
            // transactions
            c.TransactionsReady <- pool[:blockSize]
            c.TransactionPool <- pool[blockSize:]
        } else {
            c.TransactionPool <- pool
        }
        // Reset the timer
        timer = time.NewTimer(time.Second * time.Duration(hashingDelay))

    case <-quit:
        log.Println("Mining process received signal to shutdown")
        quit <- true
        wg.Done()
        break loop

    case <-stopMining:
        log.Println("Mining process received signal to stop activities")
        c.CurrentTransactions <- make([]Transaction, 0)
        confirmStopped <- true
        goto start

    case blockPool := <-c.TransactionsReady:
        log.Println("We have enough transactions to create a block")
        // make a backup in case we need to stop mining
        tmpTrs := blockPool

        for _, tr := range blockPool {
            // signatures have been verified before being added to the pool
            c.head.addTransaction(&tr)
        }

        blocks := <-c.blocks
        c.blocks <- blocks

        if len(blocks) != 0 {
            c.head.Header.ParentHash = blocks[len(blocks)-1].Proof
        } else {
            c.head.Header.ParentHash = *new([32]byte)
        }

        // compute block hash until created or stopped by new longest chain
        stopped := c.head.createProof(proofDifficultyBl, stopMining)

        if stopped {

            log.Println("Mining process received signal to stop activities")
```

```
            // notify what transactions we were working with
            c.CurrentTransactions <- tmpTrs
            c.head = NewBlock()
            confirmStopped <- true

            goto start

        } else {

            log.Println("Mining process created a block")

            seenTrs := <-c.SeenTrs
            for _, tr := range c.head.Transactions {
                seenTrs[tr.Header.VoteToken] = true
            }
            c.SeenTrs <- seenTrs

            blocks := <-c.blocks
            c.blocks <- append(blocks, *c.head)

            bl := *c.head
            c.head = NewBlock()

            go c.sendBlock(&bl)
        }
    }
}
```

The steps followed by the process are described through comments and logs. In summary, the process will periodically check to see if there are enough transactions to create a block, and if so, will write a block's worth of transactions to the `c.TransactionsReady` channel. Note that the proof creation process takes in the channel `stopMining` in the we are required to stop mining because we have received a new valid update. This channel will only be written to by the block update goroutine, and only when we have accepted a new block or a new alternative chain.

These core processes all run in goroutines, and are initialized by the core function `Start()` as follows:

```
func (c *Chain) Start(delay int, quit, stop, start, confirm chan bool,
    w *sync.WaitGroup) {

    // check for new peers every "delay" seconds
    log.Println("Starting peer syncing process...")
    go c.schedulePeerSync(delay, quit, w)

    // be processing transactions aka making blocks
    log.Println("Starting mining process...")
    go c.scheduleMining(quit, stop, start, confirm, w)

    // be ready to process new blocks and consensus forming
    log.Println("Starting chain management process...")
    go c.scheduleChainUpdates(quit, stop, start, confirm, w)

    // be listening for new shares
    log.Println("Starting key share collection process...")
    go c.scheduleKeyShareBroadcasting(delay, quit, w)
}
```

This function should be called only once at the start of the program.

5.2.3 Chain core API

The full documentation for the blockchain API can be found at: `https://godoc.org/github.com/CPSSD/voting/src/blockchain` The API is described in short here:

```
func NewChain() (c *Chain, err error)
```

NewChain will return a new blank chain ready for initialization.

```
func (c *Chain) BroadcastShare()
```

BroadcastShare will add a user's share of the election key to the pool of shares which are broadcast regularly.

```
func (c *Chain) CollectBallots() *[]election.Ballot
```

CollectBallots will gather all the ballots from the current chain and return them.

```
func (c *Chain) GetChain(empty bool, altChain *[]Block) error
```

GetChain is an RPC call which will set the value of altChain to the value of this node's current set of blocks. The value of empty is unused.

```
func (c *Chain) GetElectionKey() crypto.PrivateKey
```

GetElectionKey returns the election key as currently interpolated by the node.

```
func (c *Chain) GetFormat() election.Format
```

GetFormat returns the format defined for the election.

```
func (c *Chain) GetPeers(myPeers *map[string]bool, r *map[string]bool) error
```

GetPeers is an RPC function which allows peers to combine their peer lists for syncing.

```
func (c *Chain) GetVoteToken() string
```

GetVoteToken returns the vote token of the user associated with this node.

```
func (c *Chain) Init(filename string) (err error)
```

Init will read in a configration file and set up a new chain. The RPC functions are made available during the call of this method.

```
func (c *Chain) NewTransaction(token string,
    ballot *election.Ballot) (t *Transaction)
```

NewTransaction will take a filled ballot and encrypt its contents with the election key.

```
func (c *Chain) PrintKey()
```

PrintKey prints our current interpolation of the private election key.

```
func (c *Chain) PrintPeers()
```

PrintPeers displays the list of peers known to a node

```
func (c *Chain) PrintPool()
```

PrintPool displays a list of transactions which are not yet incorporated into the chain.

```
func (c *Chain) ReceiveBlockUpdate(blu *BlockUpdate, _ *struct{}) (err error)
```

ReceiveBlockUpdate is an RPC function which allows a node to receive an update about a block for further processing.

```
func (c *Chain) ReceiveKeyShare(share *ElectionSecret, _ *struct{}) (err error)
```

ReceiveKeyShare is an RPC function which allows a node to receive a share of the private key from other nodes.

```
func (c *Chain) ReceiveTransaction(t *Transaction, _ *struct{}) (err error)
```

ReceiveTransaction is an RPC function which allows a node to recieve transactions from the network.

```
func (c *Chain) ReconstructElectionKey()
```

ReconstructElectionKey will attempt to reconstruct the election key from the shares currently available to a node.

```
func (c *Chain) SendTransaction(tr *Transaction)
```

SendTransaction will invoke the broadcasting of a transaction to peers on the network.

```
func (c *Chain) Start(delay int, quit, stop, start,
    confirm chan bool, w *sync.WaitGroup)
```

Start will begin some of the background routines required for the running of the blockchain such as searching for new peers, mining blocks, handling chain updates, and listening for new key shares.

```
func (c Chain) String() (str string)
```

String representation of a Chain.

```
func (c *Chain) ValidateSignature(t *Transaction) (valid bool)
```

ValidateSignature will allow a signature of a transaction to be validated. The result is returned in the boolean value valid.

```
type ChainUpdate struct {
    Blocks []Block
}
```

ChainUpdate contains the blocks associated with a given chain.

5.2.4 Main process

The main process function which is called to run the program is built seperately from the blockchain and cryptography packages. This means that the program's user interface can be built on top of the APIs provided by these packages. The user interface created here is just a proof of concept for how the voting system could work. This version is broken into two main sections, initialization, and user input.

For initialization we have the following:

```
func main() {

    f, err := os.OpenFile(os.Args[1]+".log", os.O_RDWR|os.O_CREATE|os.O_APPEND, 06
    if err != nil {
        panic(err)
    }
    defer f.Close()

    log.SetOutput(f)
    log.SetFlags(log.Ltime | log.Lmicroseconds | log.Lshortfile)

    c, err := blockchain.NewChain()
    if err != nil {
        panic(err)
    }

    log.Println("Setting up network config")
    filename := string(os.Args[1])

    c.Init(filename)

    // to quit entirely
    quit := make(chan bool, 1)

    // to signal to stop mining
    stop := make(chan bool, 1)

    // to signal to start mining
    start := make(chan bool, 1)

    // to signal to confirm stopped mining
    confirm := make(chan bool, 1)

    var syncDelay int = 10
    var wg sync.WaitGroup
    wg.Add(4)
    c.Start(syncDelay, quit, stop, start, confirm, &wg)
    start <- true

    fmt.Println("Welcome to voting system.")
    vt := c.GetVoteToken()
    fmt.Println("Your vote token is:", vt)

    ...
```

Here, the channels for communication are set up, and the configuration file for this instance of the program is passed in and used in the setup. Once these steps have completed, we begin to loop on user input as follows:

```
    ...

loop:
    for {
        fmt.Printf("What next? (h for help): ")
        var input string
        fmt.Scanf("%v\n", &input)

        switch input {
        case "h":
            fmt.Printf("\th\t\tPrint this help\n")
            fmt.Printf("\tpeers\t\tPrint known peers\n")
            fmt.Printf("\tpool\t\tPrint pool of transactions\n")
            fmt.Printf("\tchain\t\tPrint current chain\n")
            fmt.Printf("\tv\t\tCast a vote\n")
            fmt.Printf("\tq\t\tQuit program\n")
            fmt.Printf("\tb\t\tBroadcast share\n")
            fmt.Printf("\tr\t\tReconstruct election key\n")
            fmt.Printf("\ttally\t\tTally the votes\n")
        case "peers":
            c.PrintPeers()
        case "pool":
            c.PrintPool()
        case "chain":
            fmt.Println("Entering print chain")
            fmt.Println(c)
            fmt.Println("Exited print chain")
        case "q":
            quit <- true
            break loop
        case "b":
            fmt.Printf("Broadcasting our share of the election key\n")
            c.BroadcastShare()
        case "r":
            fmt.Printf("Attempting to reconstruct the election key\n")
            c.ReconstructElectionKey()
            c.PrintKey()
        case "v":
            token := vt

            ballot := new(election.Ballot)
            err := ballot.Fill(c.GetFormat(), tokenMsg)
            if err != nil {
                log.Printf("Error filling out the ballot")
            } else {
                tr := c.NewTransaction(token, ballot)
                go c.ReceiveTransaction(tr, nil)
            }
        case "tally":
            ballots := c.CollectBallots()
            format := c.GetFormat()
            key := c.GetElectionKey()
            fmt.Println("Calculating the tally...")
            tally, err := format.Tally(ballots, &key)
            if err != nil {
                fmt.Println("Error calculating tally")
            }
            fmt.Println(tally)
        default:
            fmt.Println(badInputMsg)
        }

    }

    fmt.Printf("%v\n", waitMsg)
    log.Printf("%v\n", waitMsg)
    wg.Wait()
    log.Println(c)
}
```

Above we can see that we are simply looping and taking user input, and using this input to call the appropriate API functions. This could easily be extended to a fully fleshed out user interface, but for the purposes of this proof of concept the above works well enough.

5.3 Blocks, transactions and vote tokens

Each block in the chain has the following structure:

```
type Block struct {
    Transactions []Transaction
    Header       BlockHeader
    Proof        [32]byte
}

// BlockHeader contains the hash of the block's transactions,
// the hash of its parent block, a timestamp and the nonce used
// in the creation of the proof of work.
type BlockHeader struct {
    MerkleHash [32]byte
    ParentHash [32]byte
    Timestamp  uint32
    Nonce      uint32
}
```

This structure allows for a chain of blocks to be created, with each one linked to the previous one in the chain by its hash value contained in `Proof`. Any changes made to the contents of `Transactions` or to the contents of the `BlockHeader` will cause the block's `Proof` to be invalidated, or else a new `Proof` will need to be calculated by a computationally expensive process. This makes changes in the history of the chain mostly infeasible.

Each block contains a series of `Transactions` as follows:

```
type Transaction struct {
    Header TransactionHeader
    Ballot election.Ballot // the encrypted vote
}

type TransactionHeader struct {
    VoteToken  string           // so that we know what token is authorizing the vo
    BallotHash [32]byte         // hash of the ballot to tie it to the header
    Signature  crypto.Signature // signature of the ballot hash
    Timestamp  uint32
```

The Ballot of each transaction contains the VoteToken associated with the transaction's voter, along with the encrypted vote. The contents of this ballot are hashed and signed using the private DSA key associated with the vote token. This will prevent the token from being used fraudulently, and also prevent the contents of the Ballot from being changed. As with changes to the block itself, any changes to the Transaction will result in a change in the hash of a Block, making it infeasible to make adjustments to another peer's block without having to recalculate the block's `Proof`.

Vote tokens are only a string of randomized characters, but because they map to a public DSA key, they can only be used by their owner, and their signatures may be verified by everyone.

5.4 Ballots

For the purposes of this proof of concept, I decided to implement a very simple voting system. A ballot is defined by a format:

```
type Format struct {
    NumSelections int
    Selections    []Selection
}
```

and the ballot is filled out accordingly:

```
type Ballot struct {
    VoteToken     string      // VT of the voter who owns ballot
    NumSelections int         // number of selections in the ballot
    Selections    []Selection // list of selections on the ballot
}
```

Each `Selection` on the ballot has a `Name` and a corresponding `Vote` as follows:

```
type Selection struct {
    Name  string   // name of this selection option
    Vote  *big.Int // value should be encrypted with PrivKE
    Proof []byte   // value used as the zero-knowledge proof
}
```

In the current implementation, the `Proof` value is not used. This would be brought into play in a future version of this project.

The format of a ballot should represent a number of candidates or options that are up for election. The voter should enter their corresponding vote as either a 1 to vote for, or a 0 to vote against a candidate. The winner will be the candidate with the most votes. It is assumed that voters are honest, and will only enter a 1 or 0 into their ballot. This assumption will be discussed later, and would be dealt with in a future version of the project.

6 Validation

Where possible, I implemented unit testing and validation tests that could be automatically run on each build of the code. This was most applicable to the cryptography of the system, and extensive testing took place in that package.

For the validation of the blockchain, there was no easy way to automate such tests. Instead, I used extensive logging throughout the code to allow me to follow the logical execution of the system, and to identify any issues. Some of the issues that this helped to solve included unsuccessful encoding of certain data structures and values, and pieces of code that would block due to poor usage of channels for communication.

When implementing the consensus algorithm, I had to manually verify that the longest chain was being accepted and that transactions were not lost in the process by creating scenarios where the implementation would be tested. I created two groups of nodes that would work on two separate versions of the chain, and then introduce a node that would connect the two groups. At this point, the group with fewer blocks in their chain converted over to the new chain and proceeded to work on bringing across the transactions from their old chain.

Using a combination of logging and error checking, I validated each commit prior to merging it with the source on GitHub. This way I could be sure that each component was integrated successfuly with the previous version of the code before moving onto the next piece of functionality.

7 Problems solved

7.1 Node communication

When deciding on how to implement communication I decided to do more research. In a project I worked on previously in Go, communication was handled by simple communication between two nodes. A node would connect to another peer, and send a message containing a number. This number would correspond to a function and the peer would perform certain communication and actions as a result. This was cumbersome to implement, and in practice very buggy. Often nodes would hang waiting for a message that would never be sent, or would send a wrong value that was handled poorly, causing a node to crash.

I decided to learn about the `rpc` package available in Go. This allows easier communication through Remote Procedure Calls. RPC calls in Go are written in a standard format:

```
func (t *T) MethodName(argType T1, replyType *T2) error
```

For a given type `T`, a public function is created. It takes two parameters of arbitrary types `T1` and `T2`, and returns with an `error`. A number of these functions were written for the `Chain` structure, including `ReceiveTransaction` and `ReceiveBlockUpdate`. To allow these RPCs to be used, a structure must handle requests by calling the following code:

```
    ...
    rpc.Register(c)
    rpc.HandleHTTP()

    ln, err := net.Listen("tcp", "localhost:8080")
    if err != nil {
        log.Fatalln(err)
    }

    go http.Serve(ln, nil)
    ...
```

The goroutine will now listen on the port at the address specified for any new requests to RPC functions. A node can make use of a function by making an RPC call to one of its peers. The process looks like this:

```
    ...
    conn, err := rpc.DialHTTP("tcp", k)
    if err != nil {
        continue
    }
    go func() {
        trCall := conn.Go("Chain.ReceiveTransaction", tr, nil, nil)
        _ = <-trCall.Done
        conn.Close()
    }()
    ...
```

We simply open a connection to the peer at the address contained in the string `k`. Once the connection is open, we can use a goroutine to perform the RPC call asynchronously, and have the connection closed automatically when the communication concludes. In the above example, the node will make a call to its peer `k` where this peer will execute the `ReceiveTransaction` function. A transaction `tr` is passed as the argument, so the peer will be able to store the transaction for processing. Since we do not care about a response from the peer, the second argument passed is `nil`.

This is the standard structure used for communication between nodes throughout the system. Each node is designed to accept communication from any other node and to make connections to other nodes to aid in the successful functioning of the system.

7.2 Building a blockchain from scratch

Having only a very high level of understanding about a blockchain, I had the problem of implementing one from the ground up. While in theory the operation of a blockchain can be relatively simply described, in practice its implementation can be quite complex.

7.3 Consensus algorithm

Deciding how to have nodes agree on what was the correct chain was a large issue. The easiest way to solve this was to have nodes verify any chain that claimed to be longer than their own. Since a node can never be sure about how many nodes are on the network at once, the system could not rely on having a vote between nodes to agree on a version of the chain. Instead nodes cast their vote for a chain by using it as the basis of the next block they create. Care was taken to avoid transactions from being lost when a new chain was accepted. Any transactions from the old chain are stored in the nodes pool of unprocessed transactions to ensure nothing is lost in the transition.

8 Future work

This project for me was a proof of concept to see if my idea was feasible. Having worked through the numerous features I initially set out in my functional specification, there are even more I would like to add in the future. Although the system does function in its current state, there is still a lot of work that would need to be done before it could be considered for use in any important government elections.

These changes and functionality are described briefly as follows.

8.1 Distribution of vote tokens

Currently for the purposes of this project, it is assumed that vote tokens are securely and fairly distributed to voters. This assumes that there are no extra tokens generated by malicious individuals, and that everyone gets to claim their own token, and only their own.

In the future I would intend to implement a scheme that allows everyone to claim a vote token anonymously, but also to be able to verify that no tokens have been claimed by fake voters and that each eligible voter has indeed received claim of a token. Ideally this process would take place inside the blockchain for the election, to keep the entire history of the election in one place.

An idea I had to solve this issue was to have each citizen create a verified DSA key pair i.e. each citizen knows the public key of each other citizen. A government body can publish a list of vote tokens somewhere publicly, with no more tokens than there are citizens eligible to vote. Each citizen would also create a shadow DSA key pair for the purposes of each election. Using this shadow key pair, they would make a claim to a single vote token by signing a token and having it published on the blockchain. Once every token is claimed, citizens would then sign a message confirming that they did in fact manage to claim a token, and also publish this to the blockchain. This seperates the claiming of tokens from the identities of the citizens. This process could be automated, and repeats until an iteration where each citizen has claimed a token. From this point on, the system can continue its election as normal.

8.2 Genesis block creation

Currently the chain's first block is the first one created by a node. In the future I would implement creation of a genesis block. This block would contain all of the important information about the election, such as the public key of the election, a list of all the vote tokens and their public DSA keys, and the format of a ballot. This would remove the requirement for this information to be stored in the configuration of each node. This genesis block's hash would be published in a number of public locations so that nodes can verify that its contents have not been changed. The genesis block would then act as block zero in the chain, thus preventing chains from being precomputed before the election.

8.3 User interface

In a future version of the project, I would implement a more informative and usable user interface. The current version serves only to demonstrate the operation of the system, and is not suitable for a real world election for a number of reasons. Voters are not given the opportunity to review their vote, and only text input through a keyboard is available. In the future, an API to handle accessible user input should be created according to local election standards.

8.4 Cryptographic validation

While in theory the cryptographic systems implemented here are secure, I am not an expert in cryptography and do not claim to be. The schemes and practices implemented would need to be verified by more knowledgeable people in the cryptography community.

8.5 Zero-knowledge proofs

Currently a voter could input a number greater than 1 for their vote for a candidate. This is not stopped by the system because an attacker could simply replace the value before encryption to throw off the vote. In the future I would implement the use of zero-knowledge proofs to allow a voter to prove that their vote is in fact only a 1 or 0 without revealing the contents of their vote to another party.

8.6 Integration with other crypto-currencies

It would be desireable to have the system integrated with another crypto-currency system in order to increase the incentive of performing mining. Alternatively, the system would implement its own crypto-currency backed by the government whom the election is for. Voters would then be incentivised to add additional nodes to help in performing computations for the network. At the end of the election, miners could be rewarded for their work in keeping the network more secure.

8.7 More than one election per chain

Ideally in the future, the system would be expanded so that a new chain is not created per election. Instead, one global ledger would be used, where any government body or individual can make use of the chain for their large or small scale election. This would increase the security of the system for smaller elections which might otherwise be vulnerable to attack.

8.8 Trustless creation of the election key

In the current implementation of the system, the election key must be created by a single individual. If possible, I would intend to remove this required trust by having the key's private exponent created by multiple nodes, so that it does not ever exist in its full state before the end of an election. To my knowledge, this is not currently possible with today's cryptographic schemes.