Building An E-Voting System: An Implementation of Scytl's Individual Verifiability Protocol

Oliver Rock 1844205
Supervisor: David Galindo
MSc Computer Science

School of Computer Science, University of Birmingham
10th September 2018

Abstract

The aim of this piece of work was to build and describe an E-Voting system that implements Individual Verification. The piece of software is based on a protocol designed by Scytl for use by Swiss Post. Individual Verification is achieved through the use of assignment codes. By assigning a code to each possible answer the voter can select during voting, the server can calculate and return the corresponding codes for confirmation without decrypting the vote that is included in the Ballot Box. The software that was built is a working E-Voting system using Java.

Introduction

The aim of this report is to describe an E-Voting system that implements Individual Verifiability. The aim of approach is to provide a mechanism to verify that each vote was cast-as-intended and recorded-as-cast. This means that the voter must be able to verify that the choice they made is the choice that is cast in the election and that this choice is stored correctly in the ballot-box. In the system that was built for this project, this was achieved through the use of assignment codes. This means that for each possible answer the voter can select during voting, the server can calculate and return the corresponding codes for confirmation without decrypting the vote. There have been a number of different approaches that achieve individual verifiability, but for this project, we looked at a high-level description of a predefined protocol and built our own implementation (Scytl, 2017).

Online voting has a number of benefits over traditional paper-based systems. It can be easier for the user if they do not have to be present in a polling station. It can be quick to set up and to count, rather than requiring a large number of people to be involved in the polling stations and in the counting. E-Voting can refer to voting over the internet or voting using specially designed machines. In this paper, we are referring only to voting over the internet.

Traditional systems also have benefits over e-voting systems. This comes from the simplicity and distribution of trust. Anyone can understand a paper-based system and the methods it uses to make be secure. This does not exist for e-voting systems, for which the cryptography is beyond most people's understanding. This can affect levels of trust in the result, even if no fault has been found. Furthermore, paper-based systems employ a distribution of trust. No one person is responsible for a significant part of the election. An individual can only count a limited number of votes and no one person can be in hundreds of polling stations at once to try and manipulate how a people vote. This means large-scale fraud requires the collusion of many people. This is often not the case in e-voting where a small group of people could be responsible for critical parts of the system.

The goal of building an e-voting system, that is genuinely better than paper-based systems, is an ongoing area of research. If it is successful, it could make elections easier and safer to carry out. One approach to making e-voting better than traditional methods is to give the voter the ability to check that their vote has been included as they intended it to be. It is this aspect that has been investigated throughout this paper.

This report will first describe the background and literature on e-voting. It will focus on the requirements of an e-voting system and the problems e-voting aims to solve. Then the report will look at three examples in more detail to give the reader a sense of the different ways these problems have been approached. After this, the report will focus in detail on the software which was built as part of this project. First, it will describe the cryptographic building blocks used, then a description of the protocol and how each building block is positioned in this. After which, we provide a description of how this protocol is implemented in our architecture and to the flow of events when a user interacts with the system. The report will then provide analysis of the software, describing its successes and drawbacks, and the methods it uses to provide individual verifiability.

Background to Electronic Voting

Building an effective e-voting system that could be used in legally binding governmental elections is a difficult problem. This is because the system must be trusted equally by specialists in cryptography and also by people who have little or no knowledge of maths, nor the cryptographic principles on which the system is built. A badly designed cryptographic protocol will leave an election open to fraud. But also a lack of general trust in the result of an election is dangerous. It can undermine the process of democracy and in its worst form push a society away from peaceful methods of governmental selection (Alvarez, Hall and Hyde, 2009). This is in a court case in Germany which ruled electronic voting to be unconstitutional. This was not because an error had been found, the election in question was still considered valid, but because it required voters to put “blind faith” in a technology they did not understand (NDI, 2013).

An electronic voting system must fulfil two requirements to be considered well made. Firstly, to be genuinely secure to the best of anyone's knowledge. And secondly, to be robust so that there are wide levels of trust in the result it produces. To understand the motivation for building, designing and using E-Voting systems, it is helpful to first look literature which defines the subject and explores the problem which electronic voting can solve. This area of the literature

gives a good understanding of the requirements and dangers of building electronic voting systems. This section of the report will first look at the theoretical requirements which an electronic voting system must provide and then analyses three different implementations of e-voting systems that have attempted to solve these problems and their varying degrees of successes.

Elections and Electoral Fraud

The International Covenant on Civil and Political Rights, a document created by the United Nations, defines 8 principles for holding elections (Tomuschat, 2008). These principles are: periodic elections, genuine elections, stand for elections, universal suffrage, voting in elections based on the right to vote, equal suffrage, secret vote, and the free expression of the will of the voters. From this list, it is clear that some of these requirements must be incorporated when designing or using electronic voting systems. There are as follows.

The election must be genuine. To ensure this, the protocol must provide some method to demonstrate that each vote is included in the way the voter intended and that the tally of the votes is the correct summation of all, and only, those votes.

The election must provide equal suffrage. There must be no way a voter is able to submit more than one vote. Furthermore, the mus be no way to interfere with the method to create the electoral role nor how each voter receives their ballot card.

The voting system must provide secrecy for the voters. To avoid coercion or vote selling, there must be no way to connect a voter to their vote.

These principles give an understanding of the theoretical aims of a successful e-voting system. It is important, that for each of these criteria, the electoral system must be able to prove that it meets these requirements. And, as mentioned earlier, not just to prove to specialists in cryptography but also in a way that the general user trusts the system. To further understand how these principles can be implemented we must provide a detailed analysis of the various methods of attack that could be used to undermine them. There are a number of academic reviews which outline possible weakness and flaws in electronic voting systems which could be manipulated to undermine a voting system (Puiggalí, Chóliz and Guasch, 2010; Mitrou *et al.,* 2003). The following section draws on these papers.

Voter impersonation. Somebody cannot impersonate another voter to submit an invalid ballot. The system must be able to detect and stop a person impersonating a different voter.

Compromising the voter privacy. This could happen at many different stages of the voting process, such as in the voting device, in the database, by the server or at the point of decryption. The system should make it impossible to connect a voter with their unencrypted vote.

Unauthorised access. The voting system must stop any unauthorised voters from casting votes and also any election authorities from having access beyond their remit to modify any election parameters.

Modifying a vote. A vote could possibly be modified from any point between when it was cast to when it is counted. A voting system should provide some assurance that this has not happened.

Vote Deletion. There must be an assurance that vote has not been deleted between the time that it was submitted and the time that it was included in the count for the election.

Leaking early results. An early release of a count for an election could influence the way that other voters chose to vote. An electronic voting system should provide a mechanism that would stop this from happening.

Ballot Stuffing. One method to change the result of an election would fill the ballot box with false ballot papers. An electronic voting system must provide a method for making sure that only the votes cast by legitimate voters are included in the final count.

Denial of service. Once an election has been initiated, the voting system must follow through with the entire process. There must be a mechanism to stop anyone from halting the process. For example, an election authority from refusing to count the election, or an attacker launching a denial of service attack on a voting server rendering it unable to receive votes.

Election logging and auditability. If there is a case where someone challenges the legality of an election, there must be sufficient evidence that the election has not been tampered with, otherwise, an attacker could merely use an unfounded doubt in the election to disrupt it.

This list shows that there are a number of different directions by which an election can be attacked or disrupted. This list is not exhaustive and with each area that can be attacked, there are many different ways that an attacker can go about it. Furthermore, often these areas are interrelated. As the security on one area is increased it can affect another area, making it hard to secure. For example, the easiest way to make an election completely auditable would be to make

it completely open, so that everything is recorded publically. This would, however, make privacy impossible. Often e-voting systems will solve some of these issues which are chosen to be important while not solving other problems. It is also important to note that these attacks can come from anyone, this could be people setting up the election, people voting, people building the software, election authorities, people auditing the election, or collusion of some of these groups. The safest approach to take is to trust no one who is involved in any way with the election.

The next section of this paper will look at different protocols which have attempted to solve some or all of these issues. Then evaluate the successes and failures of each. To do this, we will refer back to the items on the previous list and analyse the different methods that the systems incorporate to tackle these problems.

Cryptographic solutions

To demonstrate a variety of e-voting systems three implementations have been selected to show a variety of features which provided by e-voting systems. FOO (Fujioka, Okamoto and Ohta, 1992) was chosen as it was an early protocol and we can see how the systems have developed after. Helio (Adida, 2008) because it was the first one that could be used by anyone wanting to set up their own auditable election. And the case of Estonian general election because it can be seen as an end goal of the e-voting system, to be used in a general election for most of the voters.

FOO

The FOO protocol is an early design for an e-voting system (Fujioka, Okamoto and Ohta, 1992). It implements blind signatures as a way to make sure that only authorized voters can vote. Blind signatures are a method by which somebody can sign a message without knowing the content inside (Bernhard and Warinschi,2014). To vote in an election, a voter completes a ballot using a blind signature technique. Then the voter sends this to the election administrator. The election administrator can verify that the vote came from a valid voter but they are unable to see the contents of the vote. The administrator signs the vote and sends the signature back to the voter. The voter receives this signature and sends it with the vote anonymously to the counter. The counter publishes a list of all received ballots which anyone can check. At the end of the election, the voter sends their encryption key anonymously to the counter. The counter decrypts the votes and counts them, then announces the result.

The separation of the authentication stage and the counting stage, using blind signatures, provides a way to verify that only valid votes are included and maintain the privacy of the voter. When the authority receives a vote, it checks that it the sender of the vote has not yet submitted a vote. This acts as a protection against a voter submitting multiple ballots. Furthermore, as every valid vote must contain a signature from the authority, every voter must be checked by the authority. This means that the only votes in the final count should come from registered voters and only one vote from each of them. This acts to stop the possibility of ballot stuffing.

After the authentication stage, all interaction with the voter should remain anonymous. The aim of this is to provide a mechanism that protects the voters' privacy whilst not allowing false votes from being included. It is important to note that even if there was collusion between the authority and the counter, there is no way to connect a voter to their vote. This is achieved as there is a clear separation between identifiable and anonymous communication with the voter .

After a vote is submitted it is added to a public list of votes. The publication of all the votes before they are counted gives a way for voters to check that their vote has been included and not deleted. At the end of the election, each voter submits a key to decrypt the vote. This acts to stop a corrupted counting authority from decrypting the votes and release the election results early. However, this is also one of the main drawbacks of this election scheme as the voter must interact twice, once when the vote and then again when they submit their encryption key.

Helios

Helios is an online, open-audit voting system created by Ben Adida (Adida, 2008). The system does not include novel cryptographic schemes. But rather, a protocol implementing known cryptography in a way that is verifiable and provides ballot casting assurance as well as voter secrecy (Orion, 2009). Helios was the first web-based voting system that was made publicly available for anyone to use and create their own auditable elections. It is important to note that one of the purposes of the system was as an educational tool, to make electronic voting systems more accessible to the general public. This can be seen in how it implemented a "Coerce Me!" button, which emailed a complete proof of how they voted to a person of voter's choice. The aim of this was to highlight the existence of serious problems of using electronic voting systems (Adida, 2008).

The process of voting in Helios is as follows (Adida, 2008). First, the voter must authentic themselves, this could be through an email address and a password. Then the voter is taken through the voting procedure where they make their choice. The voter is able to choose if they

want to submit the ballot or to verify if the ballot was created correctly, known as the Benaloh challenge. If they choose to verify the way that the ballot was created, the voting device displays the ciphertext and the randomness used to create it, so that the voter can verify the encryption method. Alternatively, if the voter chooses to submit the ballot, the voting device discards the plaintext and the randomness used in encryption and submits the ballot. All encrypted votes are then posted to a public bulletin board. When the period of time in which people can vote has passed. Helios uses a mixnet (the Sako-Kilian protocol) to mix the votes so that a vote can no longer be connected to a voter. The mixnet produces a non-interactive proof which shows that the list of votes that went into the mixnet correspond to the list of votes it outputs, without being changed in any way. After a period of time, where the auditors can verify the shuffle, Helios decrypts the votes and provides a proof for each decryption. An auditor is able to download the entire election data and perform their own verifications of the shuffle, decryption and tally of the vote.

Helios offers three methods to verify different parts of the election process: cast-as-intended, recorded-as-cast and tallied-as-recorded (Pereira, 2016). If they all work then the system offers most of the requirements mentioned earlier that define a secure e-voting system, however, each of them also has flaws. Cast-as-intended verification is achieved in the Benaloh challenge, this is where the voter chooses if they want to submit a ballot or to verify that it was constructed correctly. This lets the voter check that the ballot preparation system is working correctly. However, if the voting client has been corrupted then this security is circumvented as the device could create two ballots but only display the correct one to the voter but submit the other. An attack similar to this has been demonstrated (Estehghari, Desmedt and Rcis, 2010).

The recorded-as-cast verification is provided by the publication of a list of all the encrypted votes that have been submitted. A voter is able to use independent software, or check themselves, using a tracking number that their vote is included. Once again there have been security flaws show in this process Namely that a voter may be subverted to connect to a false ballot tracking centre (Pereira, 2016). Any Check they made could be manipulated to make it look like their vote was included when it was not. The tallied-as-cast verification of Helios is provided by publicising most of the data the voting has finished. This public data can then be audited by anyone to check that all votes that were stored had been correctly decrypted, the mixnet was performed honestly, and the tally was done correctly.

Estonia

E-Voting has been used in Estonian elections since 2005 when it became the first country to perform legally binding elections which were carried out over the internet (BBCNews, 2005). In the most recent general election of 2017 over 30% of the electorate used the e-voting system to submit their ballots (Valimised, 2017). The system is based on an RSA encryption feature in the Estonian National Identity Card card. It contains two key sets one which is used for authentication and one for making digital signatures. We will first describe the protocol as used in 2014 (Heiberg and Willemson, 2014) and then outline some of the security measures and flaws found in the system.

The system consists of four servers. One forward facing server which all voters connect through, it acts as an intermediary to the backend servers. A vote storage server that stores all the encrypted votes and checks their authentications. A log server, to hold event logs and monitor the system. And, a vote counting server which is never connected to a network; election officials transfer the encrypted votes to a DVD to this server which then decrypts and counts the vote.

Before the election starts the election authority publishes voting client software which contains a public key for encrypting the votes and a TLS certificate for the server. On this software, the voter to authenticates themselves using their ID card and an authentication pin number. The client software verifies the server. And then the server confirms the voter's right to participate. The voter selects a choice and enters their signing key PIN number. The vote is encrypted with RSA-OAEP using the election public key and authenticated with the voter's private key. The vote is sent to the server which stores it and generates a tracking number which is sent back to the client. The client software generates a QR code containing the tracking number and the randomness used in the RSA encryption.

To verify the submitted vote, the voter uses a mobile app to take a picture of the QR. The app retrieves the associated vote without it signature. The voter makes the choice again, and then the app encrypts a second vote using the same randomness. The app then compares the two votes, the one stored on the voting server and the one generated by the app. If they are the same it is assumed the vote was generated correctly and is the same one stored on the server. The voter is allowed to vote multiple times and after each vote has 30 minutes to verify their vote.

When voting is complete the server storing all the votes re-verifies the signatures and then strips of the signatures, leaving only the valid, encrypted votes. The votes are transferred to a DVD and

taken to the counting server. The counting server holds the election private key. It decrypts all the votes and releases the result.

Springal *et al. (Springall et al., 2014)* carried out a broad security analysis of the software used in the 2014 election. This included demonstrating a number of attacks which could have been used on the election. Their main argument was that the system relied too heavily on operational procedures rather than cryptographic tools to make the election secure. For example, the election results were transferred, using a personal USB drive, from the counting server to a personal laptop where they were signed by the election authority signature. This provided an opportunity for an attacker to change the results.

Springer *et al. (Springall et al., 2014)* demonstrated three areas that could be attacked. First was an attack on client software. They claimed could be performed by any hacker with reasonable financial backing and could alter how votes were cast. Second, was an attack on the server. They argued this was within the ability of a state-level attacker or a dishonest insider. This attack could be used to change the votes stored on the database or the tally. And third was an attack an attack on voter privacy which they argued there were multiple ways to achieve. In their conclusion, they argued that it was extremely difficult to ensure the integrity of the code running on the server and that from this insecurity the system was left open to a wide possibility of attacks.

Summary of Cryptographic solutions

The purpose of going through these three examples of e-voting systems is to demonstrate the different requirements and difficulties faced in building an e-voting system. While also showing a number of different methods used to build one. Together, the three show the lifecycle of the development process of building an e-voting system. The design of FOO was almost entirely based on the protocol and the cryptographic tools needed to build it. Helios was the first online system which let general users create their own elections. It was designed for use in smaller elections where coercion or well-resourced attackers were not a threat so security flaws were accepted but kept to a minimum. And the Estonian example took place in a legal binding governmental election.

It is important to draw to attention how each stage of this process requires different analysis and focus. The aim of designing a working protocol is a highly mathematical endeavour. Whereas making an e-voting system work for the general public has more to do with working out if

people will follow security procedures. Without the connection of both of these, one can see problems arise. In FOO it is designed to require two interactions from the voter, which, in a real-world system would not be well followed. The Estonian, case on the other hand, shows that if one allows security to be carried out by people following procedures it cannot be trusted.

It is also important to point out a number of standard practices which were established across the three systems. All systems tackle to problem of authenticating that the vote came from a valid voter but keeping the privacy of the voter. FOO does this with blind signatures, Helios uses a mixnet and in Estonia discards the signatures. Furthermore, all the systems must have channels to audit the election. They all use various methods of publicising parts of the election. The rest of this paper will focus on the e-voting system that was built as part of this project and the methods used to make it secure.

Analysis and Specification

The product created for this piece of work was based of the paper *"Swiss Online Voting System: Cryptographic Proof of Individual Verifiability"* (Scytl, 2017b). The paper provides a high level description of the entire protocol and details of how the protocol meets requirements to demonstrate individual verifiability set out by the Federal Council, a body which oversees elections in Switzerland. As mentioned earlier, Individual verifiability must provide cast-as-intended and recorded-as-cast verifiability. This is within the assumed trust model that the server side of the system is trusted. But, where the client side and the communication channel between client and server is not. The voters also cannot be trusted and may attempt to send invalid votes.
It was decided to build the system in Java. The reason for this is that Java is my strongest language. This would mean that there was more time to working on the cryptography and the protocol rather than learning new programming languages. Furthermore, Java provides provides many useful cryptographic features which we used for this project. For example, Java provides a secure sockets layer which was used across all the network connections. It also has BigInteger, which allows arbitrary-size integers to be represented and methods for common mathematical operations. This was used for the cryptography. Java also provides the Java Cryptography Architecture (Oracle, 2018) which was used for some of the cryptographic primitives. We also used the library Bouncy Castle (Bouncy Castle, 2013) to provide all the other necessary cryptographic primitives.

The architecture for the system was based on the paper "*Individual Verifiability: Swiss Post E-Voting Protocol Explained*" (Scytl, 2017a). This is a paper by the company Scytl, describing

their protocol implementing Individual Verifiability. It contains a high-level description of the protocol and diagrams to explain the sequence of the communication between client and server The front end of the product is a java app build using Swing. The server handles all requests by the client, and stores any necessary information to a Postgres database.

There are some features which we decided to not include in the system. These would be necessary for a complete verifiable system. Given the time frame, it was not feasible to build a system with complete verifiability. The two main features which were omitted were a mixnet and methods to audit the server side operations. The mixnet would separate encrypted votes from anything which it identifies it to the voter before being decoded. As the server is assumed to be trusted, we also assume that the server does not release any information which would otherwise have been removed by the mixnet. There is also no way to audit the operations on the server side. As we made the assumption about the trustworthiness of the server we this feature was not paramount in the time we had.

We decided to implement certain features in the server-side operations which should only be implemented if the server is completely trustworthy. For example, the private key for decrypting the votes is stored in the database. In a completely verifiable system, it should not be built like this. If an attacker got access to the database the votes could be decrypted before the end of the election. Rather, an authority who has been trusted to keep the key should hold it separately from the encrypted votes. Given the server-side is assumed to be trusted, and the focus of the project is on the way that voters interact with the system rather than election authorities, we choose to make it more simple and store some extra data on the database.

A brief overview of how the system works is as follows. A user decides to create an election. They provide basic information about the election and a list of emails of invited participants. The server processes that information and creates some keys to be used across the election for encryption and authentication. The server then registers each of the voters by creating keys used to authenticate the voter and some codes the vote can use to check cast-as-intended and recorded-as-cast verifiability. Some of this information is sent by email to the voter and some of this information is stored in the database. For each possible vote option, the voter is given a choice code. This is provided in their email. When they submit a vote, the server returns a choice code. If that choice code matches with the one on their ballot card, the voter can assume their vote has been sent to the server correctly. Then the voter replies with a confirmation message to indicate they are satisfied with the choice codes. The server processes this confirmation message and returns a Vote Cast Code. This can, once again, be compared to one on the voter's ballot card. If they match we can assume the vote has been recorded correctly. The next few sections of this paper will go into greater detail about the cryptography needed to make this protocol work.

Design

The following section is divided into two parts. The first part addresses the cryptographic building blocks which are used in the protocol. The second part addresses the main algorithms of the protocol and describes the inputs, the functionality and the outputs of each one.

Cryptographic Building Blocks

This section contains descriptions of the cryptographic building blocks which have been implemented in this e-voting system. A description of how these building blocks interact will be described in the section after this.

Encryption Schemes

Public Key Encryption. A public key encryption scheme is defined by the implementation of three algorithms (Gen_e, Enc, Dec). The generation algorithm produces a key pair; made up of a public key *pk,* and a secret key *sk.* The encryption algorithm takes a public key and a message. From these, it produces a ciphertext. The decryption algorithm takes a ciphertext and a secret key and returns either a decrypted message or declares the ciphertext as invalid (Bernhard and Warinschi,2014).

This protocol uses ElGamal encryption (ElGamal, 1985). The key generation algorithm (Gen_e) is given the parameters (g, p, q), where g is a generator of order q of elements Z_q. And, where p is a safe prime given that $p = 2q + 1$ and q is a prime number (Scytl, 2017). The secret key sk is chosen at random from Z_q. The public key is then calculated as $pk = g^{sk} (mod\, p)$. This returns the key pair (pk, sk). The encryption algorithm (Enc) takes a message and pk as parameters. It chooses a random number $r \in Z_q$, and computes the ciphertext $c = (c_1, c_2) = (g^r, pk^r \cdot m)$. The decryption algorithm (Dec) takes the parameters c and sk and outputs either the message $m = c_2 / (c_1)^{sk}$ or $\perp$ if the case of an error. For this system, we wrote our own implementation of ElGamal.

Symmetric Key Encryption. Symmetric key encryption schemes are defined by the implementation of three algorithms ($KGen^s$, Enc^s, Dec^s). $KGen^s$ produces a symmetric key k

from a given keyspace. Enc^s takes the parameters of a message *m* and a key *k* and produces a ciphertext *c*. Dec^s takes a symmetric key and a ciphertext and returns a decrypted message *m*.

This protocol uses AES encryption in GCM mode. GCM mode provides authenticated encryption to a block cipher. To encrypt each block of data it uses a counter which is combined with an initialization vector (IV). Each block is then sent through the block cipher, in this case, AES. Then, it takes the output of the block cipher and xored it with the plaintext to produce the ciphertext. The encryption also contains a hash function which is used to produce an authentication tag. The output of encryption is the IV, the ciphertext and the authentication tag. The decryption algorithm is then able to obtain the plaintext and authenticate the message or it will return a failure. In our system we used a cryptographic library to provide this functionality.

Key Derivation Functions

A key derivation function derives a key from a given set of inputs, usually a passphrase or password. This means that a smaller or more memorable piece of data can be used to generate a more complex cryptographic key. This protocol uses PBKDF2 (δ) which takes, as parameters, a password, a salt, a hash function, and number of iterations. The salt and the password are concatenated. This concatenation is then repeatedly hashed however many times it has been specified. A benefit of PBKDF2 is that the number of iterations can be defined which can alter the computational power necessary to run it. This is useful as the computational power can be minimised so that it can be run on the client and server side quickly (Lindell and Katz, 2014). However, this also makes it susceptible to brute force attacks. The system also uses bcrypt to store users' passwords. Bcrypt has been designed to require higher levels of computing power which makes it harder to brute force (Lindell and Katz, 2014).

Signature Schemes

A signature scheme gives someone the ability to authenticate the author of a message. This is done by signing a message with their key. The other person, who receives that message is able to verify that the message they receive and the signature came from the same sender. A signature scheme is defined as comprising of three algorithms Gen_s, Sign, Verify . The generation algorithm outputs a key pair: pk_s is a public key and sk_s is a private key. The Sign algorithm takes a private key sk_s and a message *m,* and outputs a signature ψ. Verify takes a message, a signature and a public key and outputs true if the signature came from the message and false if it did not.

This protocol uses RSA Probabilistic Signature Scheme (RSA-PSS). Gen_s takes as input two primes *p, q* and computed the public key $pk_s = (n, e)$, where $n = pq$ and e is coprime with $\phi(n) = (p - 1)(q - 1)$. The private key sk_s is calculated as d, where $ed = 1 \ (mod \ \phi(n))$. The

algorithm Sign takes a message and a private key and outputs $\psi = (ME(m))^d \ mod\, n$. ME represents a hash function which outputs a value in the group Z_n. The final algorithm Verify takes the public key, the message and the signature and checks that $ME(m) = \psi^e \ mod\, n$. Verify returns true if the signature is associated with the message or false if it is not (Scytl, 2017).

Non-Interactive zero knowledge proofs of knowledge

Zero-knowledge proofs allow someone to prove that they have performed a certain cryptographic operation correctly without giving out any more information than the proof. This protocol uses the Fiat-Shamir transformation which turns interactive zero-knowledge proofs into non-interactive proofs (Fiat and Shamir, 1986). Non-interactive proofs are preferable in this situation because it means that the proof can be generated once and check by multiple agents, rather than each agent sending a verify request to the person who generates the proof. The Fiat-Shamir transformation is based on the assumption that a hash function behaves in the same way as a random oracle.

If a prover wants to prove that the know x for the equation $\phi(a; x) \rightarrow a^x$ the Interactive Zero Knowledge Proof system works as follows:

1. The prover computes $t = \phi(a; s) = a^s$ where s is selected at random from the same value space as x and sends it to the verifier.
2. The verifier submits a random challenge h.
3. The prover computes $z = s + x \cdot h$ and returns it to the verifier
4. The verifier checks that $(a^x)^h \cdot t = a^z$

This proof can be turned into a Non-Interactive Zero Knowledge Proof (NIZKPK) by exchanging the challenge h, for a hash of some of the elements in the proof. This procedure is as follows:

1. The prover computes $t = \phi(a; s) = a^s$ where s is selected at random from the same value space as x.
2. The prover computes the challenge $h \leftarrow H(a, a^x, \phi(a; x), t, aux)$.
3. The prover calculates $z = s + x \cdot h$ and sends the proof (h, z) to the verifier.
4. The verifier computes $h' = H(a, a^x, (a^x)^{-h} \cdot a^z, aux)$ and checks that $h' = h$.

The protocol uses three different variations of this NIZKPK, each one contains an algorithm to make a proof and one to verify a proof. These proofs are complete and sound. They are complete, for if a verifier receives a proof from an honest prover, it will always succeed on verification. They are also sound as in a case where the prover has been dishonest the verification will fail with overwhelming probability (Scytl, 2017). Each NIZKPK implemented for this project but use SHA256 from a cryptographic library. These proofs are as follows:

Equality of Discrete Logarithms. This is used to prove that given a set of numbers $(a_1, ..., a_n)$ and that same set raised to an exponent x $(a_1^x, ..., a_n^x)$, the prover aims to show that each number in the second set is genuinely the related number in the first set raised to the power x.

ProveEq($(a_1, ..., a_n)$, $(a_1^x, ..., a_n^x)$, x) . Takes a random value s and computes $a_1^s, ..., a_n^s$. The challenge is computed as $h = \mathrm{H}((a_1, ..., a_n), (a_1^x, ..., a_n^x), (a_1^s, ..., a_n^s))$. And calculates $z = s + x \cdot h$. It returns the proof $\pi_{\mathrm{eqDl}} = (h, z)$.

VerifyEq ($(a_1, ..., a_n)$, $(a_1^x, ..., a_n^x)$, x), π_{eqDl}). Computes $a_1^{s'} = a_1^z \cdot (a_1^x)^{-h}$ and computes the same for all values up to $a_n^{s'}$. It then checks that $h = \mathrm{H}\,\mathrm{H}((a_1, ..., a_n), (a_1^x, ..., a_n^x), (a_1^{s'}, ..., a_n^{s'}))$. If this is valid it returns true and false if it is not.

Knowledge of Encryption Exponent. This proof is used in conjunction with ElGamal encryption to prove that the person encrypting the data knows the randomness which was used in the encryption.

ProveExp $((g, c_1, c_2), r)$ Choses a random value s, and computes g^s, $h = H(g, c_1, c_2, g^s)$ and $z = s + r \cdot h$. The algorithm returns $\pi_{sch} = (h, z)$.

VerifyExp $((g, c_1, c_2), \pi_{sch})$ Computes $g^{s\prime} = g^z \cdot (c_1)^{-h}$, and checks that $h = H(g, c_1, c_2, g^{s\prime})$. If this verification is successful, it returns true otherwise it returns false.

Correct Decryption. This proof is used in conjunction with the decryption of an ElGamal ciphertext, to prove that the decryption has not altered the data and decrypted the message honestly.

ProveDec $((c, m), sk)$. Takes as input a ciphertext $c = (c_1, c_2)$ and a secret key used in its encryption. $c_1 = g^r$, and $c_2 = pk^r \cdot m$, and $pk = g^{sk}$. It chooses a random value s, and computes $h = H(c, m, (g^r)^s, g^s)$ and $z = s + sk \cdot h$. The proof it returns is $\pi_{dec} = (h, z)$.

VerifyDec $((c, m)\, \pi_{dec})$. Computes $(g^r)^{s\prime} = (c_1)^z \cdot (c_2/m)^{-h}$ and $g^{s\prime} = g^z \cdot pk^{-h}$ and checks that $h = H(c, m, (g^r)^{s\prime}, g^{s\prime})$. If the verification is valid it returns true and if not then it returns false.

Pseudo-random functions

The purpose of a pseudo-random function is to map two distinct sets and make it look like the mapping is random. In its most basic form it is essentially a lookup table where the data from one set has been assigned to a second one in a random way. This protocol uses two PRFs. First is an HMAC based on SHA256. HMAC is used to simultaneously verify the authenticity and data integrity of a message. In this implementation we have used an HMAC from the javax.crypto library. Initially we used an HMAC from the bouncy castle library but it ran about 20-25 times slower on average. The collision probability in HMAC is based on the collision probability of the underlying hash function, which is considered negligible for SHA256 (Lindell and Katz, 2014). The second PRF used in this protocol is the exponential function used in a cyclical set $g \rightarrow g^k \,(mod\, x)$ which is used in ElGamal encryption.

Representation of Voting Options

The voting options $\{v_i, \ldots, v_k\}$ are represented as small primes. The protocol uses two methods to encode and decode these primes. Product is used to encode the primes into a single number, and factorisation is used to separate them into their original set. It is the product which is encrypted and submitted as a person's vote. This gives the voter the option to choose one or more of the choices.

Cryptographic Protocol Methods

In the following section, we describe the main methods used by this protocol in this e-voting system. These methods provide the functionality to create an election, to register voters to the election, for a voter to submit a vote in the election, to confirm a vote in the ballot box, and tally an election once it is complete.

Setup (1^{λ}) is used to generate the keys for an election and receives a security parameter as input. It generates:

- An Electoral Board key pair $(EB_{pk}, EB_{sk}) \leftarrow Gen_e(1^{\lambda})$. This is used to encrypt and decrypt the votes of an election. EB_{sk} could be a composite key so that it requires many people to come together to decrypt the final tally.
- A Codes secret key $C_{sk} \in_R T$, where T is the key space for PRF function f. This is used to generate choice codes.
- A Vote Cast Code Signer key pair $(VCCs_{pk}, VCCs_{sk}) \leftarrow Gen_s(1^{\lambda})$. Which is used to sign Vote Cast Codes

The outputs $(EB_{pk}, VCCs_{pk})$ are inputs to all the next algorithms although not specified.

Register $(1^{\lambda}, C_{sk}, VCCs_{sk})$ is used to register a voter to an election. It creates the data required for their ballot card, the choice codes, and their vote cast code. It takes as input a security parameter 1^{λ} and the private keys $C_{sk}, VCCs_{sk}$. It generates:

- A start voting key $SVK_{id} \in_R A_{svk}$ where A_{svk} is 32^{20} and represented as 20 character string in Base32. This is used by the voter to identify themselves as the owner of a ballot card.
- A Voting Card ID $VC_{id} \leftarrow \delta(SVK_{id}, IDseed)$ where the IDseed is the voter's username.
- A keystore password $KSpwd_{id} \leftarrow \delta(SVK_{id}, KeySeed)$ where $KeySeed$ is a password that the user can choose.
- The Verification Card key data:
    - The Verification Card key pair $(VC_{pk}^{id}, VC_{sk}^{id}) \leftarrow Gen_e(1^{\lambda})$ which is formally an ElGamal key pair, will be used slightly differently in the creation and verification of an encrypted vote.
    - An encryption of the Verification Card private key into the keystore password $VCks_{id} \leftarrow Enc^s(VC_{sk}^{id}; KSpwd_{id})$.

- The Verification Card codes generation:
    - A Ballot Casting Key $BCK^{id} \in_R A_{bck}$, where A_{bck} is 10^8.
    - A long choice code for each voting option $CC_i^{id} = f_{C_{sk}}(v_i^{VC_{sk}^{id}})$ where v_i represents each voting option. And an associated short choice code sCC_i^{id} is selected at random from 10^4. The short choice codes are checked to be unique for the voter.

- A long Vote Cast Code $VCC^{id} = f_{C_{sk}}(((BCK^{id})^2)^{VC_{sk}^{id}})$ and an associated short Vote Cast Code $sVCC^{id}$ which is selected at random from 10^8.
- Computes the signature for the short Vote Cast Code $S_{VCC^{id}} \leftarrow Sign(sVCC^{id}, VCCs_{sk})$.
- Computes the Codes Mapping Table CM_{id} which consists of pairs of a hash of a code and the encryption of the associate short code. This is performed for the Choice Codes and for the Vote Cast Code:

$$\{[H(CC_i^{id}), Enc^s(sCC_i^{id}; CC_i^{id})]\}_{i=1}^{k}$$
$$[H(VCC^{id}), Enc^s((sVCC^{id} \mid S_{VCC^{id}}); VCC^{id})]$$

The algorithm Registar returns:
$SVK_{id}, VC_{id}, VC_{pk}^{id}, KSpwd_{id}, BCK^{id}, VCC^{id}, \{v_i, sCC_i^{id}\}_{i=1}^{k}, CM_{id}$.

GetID (SVK_{id}) is used by the voting device to obtain the voters Voting Card ID. When given a Start Voting Key it computes: $VC_{id} \leftarrow \delta(SVK_{id}, IDseed)$. And returns VC_{id}.

GetKey $(SVK_{id}, VCks_{id})$ is used by the voting device to obtain a Verification Card private key. It is given the inputs Start Voting Key and Verification Card private key keystore. It runs the following methods:

- Generate keystore password $KSpwd_{id} \leftarrow \delta(SVK_{id}, KeySeed)$ where $KeySeed$ is a password that the user has for their account.
- Get private key $VC_{sk}^{id} \leftarrow Dec^s(VCks_{id}; KSpwd_{id})$

It returns VC_{sk}^{id}.

CreateVote $(VC_{id}, \{v_1, \ldots, v_t\}, VC_{pk}^{id}, VC_{sk}^{id})$ is used by the voting device to create a vote to send to the server. It takes the voter's Voting Card ID, a set of voting options the voter has chosen, and the Verification Card Key pair. It performs the following actions:

- Compute an aggregate of the voters' selection $v = \Pi_{l=1}^{t}(v_l)$.
- Encrypts the result of this: $c = (c_1, c_2) \leftarrow Enc(v, EB_{pk})$.
- Generates a proof of knowledge of the randomness used in that encryption $\pi_{sch} \leftarrow ProveExp((g, c_1, c_2), r)$ where r is the randomness.
- Compute partial Choice Code $\{pCC_l^{id}\}_{l=1}^{t} = (v_1^{VC_{sk}^{id}}, \ldots, v_t^{VC_{sk}^{id}})$.
- Compute $(\overline{c} = (\overline{c_1}, \overline{c_2}) = (c_1^{VC_{sk}^{id}}, c_2^{VC_{sk}^{id}})$.
- Compute two NIZKPK proofs to show that the partial choice codes are derived from the same choices that are in *c*.

- $\pi_{exp} = ProveEq((g, c_1, c_2, VC_{pk}^{id}, \overline{c_1}, \overline{c_2}), VC_{sk}^{id})$ which proves that $\overline{c}$ is c raised to the exponent VC_{sk}^{id}, which corresponds to VC_{pk}^{id}.
- $\pi_{exp} = ProveEq((g, EB_{pk}, \overline{c_1}, \frac{\overline{c_2}}{\Pi_{l=1}^{t}(pCC_l^{id})}), r \cdot VC_{sk}^{id})$ which proves that $\overline{c}$ is equal to the encryption of the product of partial choice codes using the Electoral Board public key.

This returns three parts which together form the authenticated vote $V = (\alpha, \beta, \gamma)$.

- $\alpha \leftarrow c$
- $\beta \leftarrow \{pCC_l^{id}\}_{l=1}^{t}$
- $\gamma \leftarrow (\overline{c}, VC_{pk}^{id}, \pi_{sch}, \pi_{exp}, \pi_{exp})$

ProcessVote (BB, VC_{id}, V) is run by the server to check that a submitted vote is valid. It takes the bulletin board, a Voting Card ID and a vote V. It first checks that a vote has not already been submitted with this VC_{id} and that the public key VC_{pk}^{id} submitted with the vote corresponds to the one saved to the bulletin board. Then,the algorithm runs a verification function on each of the three NIZPKP proofs in the vote:

- VerifyExp $((g, c_1, c_2), \pi_{sch})$
- VerifyEq $((g, c_1, c_2, VC_{pk}^{id}, \overline{c_1}, \overline{c_2}), \pi_{exp})$
- VerifyEq $((g, EB_{pk}, \overline{c_1}, \frac{\overline{c_2}}{\Pi_{l=1}^{t}(pCC_l^{id})}), \pi_{exp})$

If any of these validations fail, the algorithm returns false, otherwise it returns true.

CreateCC (V, C_{sk}, CM_{id}) is run by the server after a vote has been validated. It takes the inputs of a vote *V,* the codes secret key C_{sk}, and a codes mapping table CM_{id}. It computes the associated long choice codes and decrypts the short choice codes which were stored in the codes mapping table. This is achieved through the following methods:

- Computes long choice codes $CC_l^{id} = f_{C_{sk}}(pCC_l^{id})$ for each of the choice codes in V.
- Checks for each CC_l^{id} that $H(CC_l^{id})$ is stored in the codes mapping table and retrieves the encrypted short choice code. It then runs Dec_s on the short choice code using CC_l^{id} as the symmetric key.

The output of the algorithm should either be a set of the short choice codes associated with the selection in V, or $\perp$ to indicate that there was a failure in obtaining the choice codes.

GetCC $(BB,\ VC_{id},\ C_{sk})$ is run by the server to retrieve the choice codes for a vote that has already been submitted. It takes as input the bulletin board *BB,* a Voting Card ID VC_{id}, and the Codes Secret key C_{sk}. It performs the following actions:

- Checks that there is an entry on the bulletin board for the VC_{id}.
- Retrieves vote V from the bulletin board which corresponds to VC_{id}, if no vote has been submitted it returns $\perp$.
- Retrieves the Code mapping table associated with the VC_{id}.
- Computes long choice codes $CC_l^{id} = f_{C_{sk}}(pCC_l^{id})$ for each of the choice codes in V.
- Checks for each CC_l^{id} that $H(CC_l^{id})$ is stored in the codes mapping table and retrieves the encrypted short choice code. It then runs Dec_s on the short choice code using CC_l^{id} as the symmetric key.

If all of these actions are successful the algorithm returns a list of the short choice codes, if there has been a failure at any point it returns $\perp$ to indicate this failure.

Confirm $(VC_{id},\ V,\ VC_{sk}^{id},\ BCK^{id})$ is run by the voting device to create a confirmation message. It receives a Voting Card ID VC_{id}, a vote *V,* the verification card secret key VC_{sk}^{id}, and the voter's ballot casting key BCK^{id}. It generates and returns a confirmation message $CM^{id} = ((BCK^{id})^2)^{VC_{sk}^{id}}$.

ProcessConfirm $(BB,\ VC_{id},\ CM^{id},\ C_{sk},\ VCCs_{pk})$ is run by the server to verify that a confirmation message is correct and confirm that a vote has been submitted correctly and should be included in the tally. It receives the bulletin board *BB,* a Voting Card ID VC_{id}, a confirmation message CM^{id}, the codes secret key C_{sk}, and the Vote Cast Code Signer public key $VCCs_{pk}$. It performs the following actions:

- Checks that a vote corresponding to VC_{id} has been saved in the bulletin board but has not yet been confirmed.
- Compute the long Vote Cast Code $VCC^{id} = f_{C_{sk}}(CM^{id})$.
- Takes the Codes Mapping table from the Bulletin Board and finds the entry : $[H(VCC^{id}),\ Enc^s((sVCC^{id} \mid S_{VCC^{id}});\ VCC^{id})]$.
- It retrieves the short Vote Cast Code and its associated signature by running Dec^s on $(sVCC^{id} \mid S_{VCC^{id}})$ using VCC^{id} as the key.
- Verifies that the short Vote Cast Code is correct and runs: Verify($VCCs_{pk},\ sVCC^{id},\ S_{VCC^{id}}$).

If all of these actions have succeeded and the verification is true it returns $(sVCC^{id}, S_{VCC^{id}})$, or $\perp$ if it has failed.

Tally (BB, EB_{sk}) is run by the server and outputs the final tally for an election. It takes the bulletin board and the Electoral Board private key EB_{sk}. It runs the following processes:

- Cleansing: to validate that all the votes stored in the bulletin board should be included it first runs *ProcessVote()* for all votes for all votes which have been labeled as valid. Then it runs $Verify(VCCs_{pk}, sVCC^{id}, S_{VCC^{id}})$ for each of those votes. Any votes that fail this are not included.
- Decryption: For all the remaining votes it runs $v_i \leftarrow Dec(c_i, EB_{sk})$ and $ProveDec((c_{i,}, v_i), EB_{sk})$ to produce a decryption proof. Fact(v_i) is run to obtain the factors of the composite parts to the vote $\{v_{il}\}_{l=1}^{t}$. These are then tested to be in Ω, the pre-defined group of voting options. If any of this fails the entire vote is discarded.

The output of this algorithm is the factorized votes, representing all the valid submitted votes to the election.

Implementation

This next section describes how each of the algorithms described in the previous section fit into the whole system. The structure of the system can be broken down into four distinct phases; configuration, registration, voting, tallying. This section will describe each of these in detail and how they fit into the architecture of the system.

Configuration: In the configuration phase, a person setting up the election enters the necessary details into the application. This includes a name of the election, a question, a set of possible answers $\{v_i, \ldots, v_k\}$ and a list of the email addresses of the voters. The server then runs the Setup algorithm this generates primes associated to each voting option and the keys for the election. The Election Board public key EB_{pk} and the Vote Cast Signer key VCC_{sk} should be made public. While the Election Board private key EB_{sk} should be kept secret by an election authority or authorities. The Codes Secret key C_{sk} and the Vote Cast Code Signer private key $VVCs_{sk}$ are provided to the registrars. The Codes Secret key C_{sk} is also provided to the bulletin board manager. In our implementation, we decided to save all these keys to the database and let the server have access whenever it needs. This could make the election insecure. If someone was

able to access the database they could take a key to decrypt votes before they should have been. However, we assume the server to be trustworthy. So we decided that this would make demonstration easier and it would be easy to change this configuration so that some of these keys were not stored but given to an election authority.

Registration: In the registration phase voters are registered to vote in an election. This means that their appropriate codes and keys are generated. The registrars are given the Codes Secret key C_{sk} and the Vote Cast Code Signer private key $VVCs_{sk}$ for the election. The server runs the Register algorithm for each voter. This creates a ballot card which is sent by email to the voter and contains the following information: SVK_{id}, BCK^{id}, $sVCC^{id}$, $\{v_i, sCC_i^{id}\}_{i=1}^{k}$. The other data generated from the Register algorithm is saved to the database VC_{id}, VC_{pk}^{id}, $VCks_{id}$, CM_{id}. To send the email we have used an inbuilt java library to connect to a Gmail account.

Voting: The Voting phase contains a series of steps, first the voter logins into the application and selects the election that they want to vote in. The next steps are as follows:

1. The Voter provides the SVK_{id} that was on their ballot card. The voting device runs GetID which derives a Voting Card ID VC_{id}. The voting device requests the Verification Card keystore $VCks_{id}$ associated with the voter.
2. The voting device runs GetKey to recover the Verification Card private key VC_{sk}^{id}. The device now has all the data required to generate an encrypted vote.
3. The voter makes their choice and the voting device runs the CreateVote algorithm producing an encrypted vote V.
4. The vote is sent to the server with the VC_{id}.
5. The server runs the ProcessVote algorithm on the vote. If the verification of the vote is successful the process continues, otherwise, the server sends an error message to the voting device and prompts the user to use a different method of voting.
6. The server runs CreateCC to derive the short choice codes associated with the contents of the vote. It takes the voter's Codes Mapping Table stored in the database and the Codes Secret key C_{sk}. If CreateCC is successful it returns the short choice codes to the voting device. If it is not successful, the server sends an error message to the voting device to notify the voter that their vote was not valid.
7. From this point on, the voter can request to check their short choice codes any number of times. To do this the voting device sends a request to the server. The server then runs GetCC and returns the short choice codes to the voting device.

8. The voter then compares the choice codes displayed on the voting device to the choice codes which were sent in the ballot card. If they match, the voter enters their Ballot Cast key BCK_{id}. The voting device runs the Confirm algorithm to produce a confirmation message CM^{id} which is then sent to the server with the VC_{id}.
9. The server runs the algorithm ProcessConfirm using the confirmation message CM^{id}. If the algorithm is successful it saves the Vote Cast Code $sVVC^{id}$ and its associated signature $S_{vcc}{}^{id}$ to the database and sends the Vote Cast Code $sVVC^{id}$ to the voting device. If ProcessConfirm is not successful an error message is sent to the voter.
10. The voter checks that the Vote Cast Code $sVVC^{id}$ they received matches the one on their ballot card. After this step, and until the end of the election, the voter is able to check their $sVVC^{id}$ by sending request to the server to retrieve it from the database.

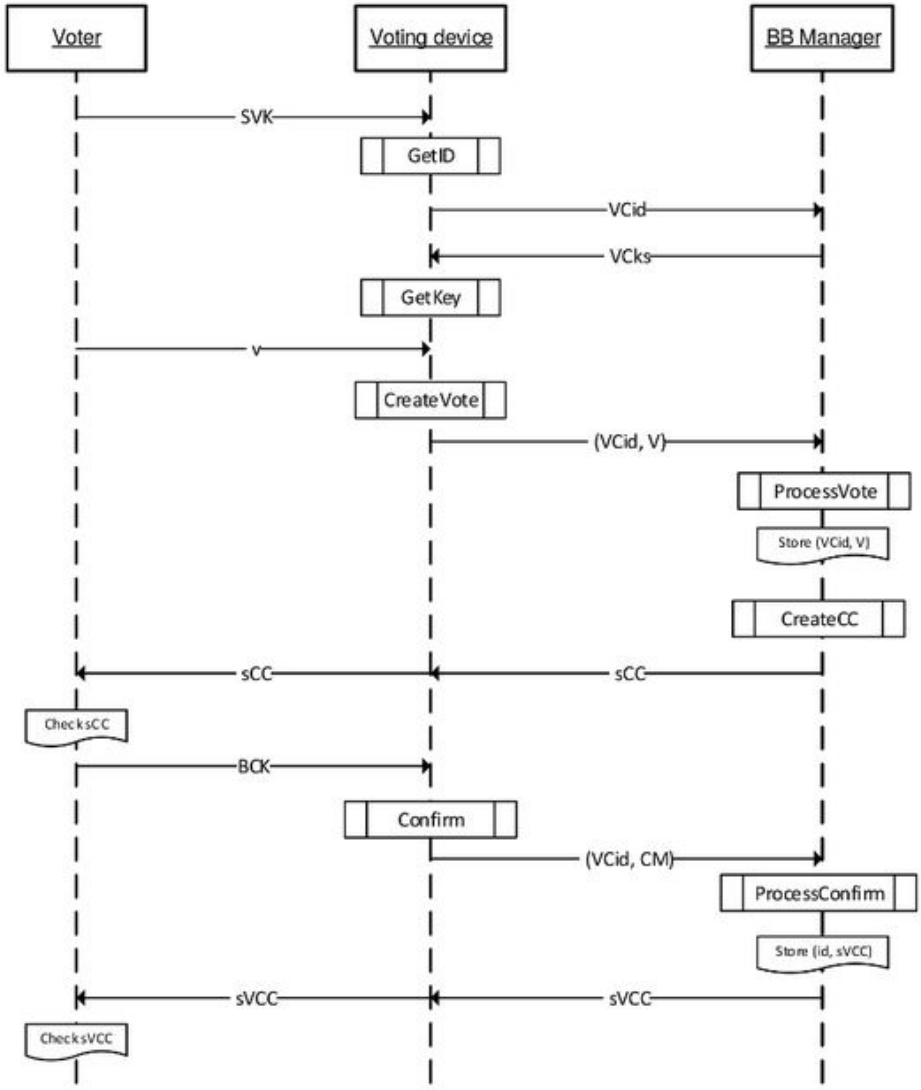


Figure 1. Work-flow of the voting phase (Scytl, 2017).

Tally: The tallying phase has been simplified for this project. The reasoning behind these changes is that, if the server is trustworthy, auditing what the server has done is unnecessary. We will first describe an ideal implementation of the phase and then describe the changes that have been made in this system.

At the end of the election the election authorities provide the Electoral Board private key EB_{sk} and the server runs the algorithm Tally. This algorithm clears the bulletin board of any invalid votes, uses a mixnet to shuffle the votes, decrypts the votes, and finally returns the results of the election with proofs of the mixnet and the decryption of each vote. All of this is then made public so that auditors can check each stage of the election.

In our implementation, we have simplified this process. Any user is able to send a request to the server to perform Tally and call an end to an election. It was chosen to be implemented in this way to make demonstration straightforward. On the backend, it is possible to give each election a time by which it will end and after this, no votes will be accepted. This feature was part-built but not included as it was deemed to be unnecessary for the scope of this project. Having a set time to end the election is important to stop any confusion for the voters about when the polls will close.

A further simplification in our implementation, is to have the Electoral Board private key EB_{sk} is saved to the database. This means that an election authority does not need to worry about keeping this key secret as we assume the server to be trustworthy and to not make this key public. The server runs the algorithm Tally. This process re-checks the validity of all submitted votes, decrypts each vote and produces a proof of decryption. In a full implementation of this voting system, the server should use a mixnet but it was decided that the implementation of this would take to long in the scope of this project. At this point, the server should also publish everything on the bulletin board. This allows others to verify the entire election process. In our implementation, all of this data is stored in the database but there is no channel by which others can access it. It was decided that this was not necessary for the same reason that we can assume that all the actions of the server are performed correctly.

Testing

Testing was a key component of the development cycle for this piece of software. Testing was broken down into four distinct areas: unit testing, integration testing, black box testing and user

feedback. Unit testing was carried out though the development cycle. The structure of the software and project management meant that unit testing was easy to perform at the same time as programming the software. Development started with creating the cryptographic building blocks to be used. Each cryptographic building block is contained within its own class in the BuildingBlock package. This structure meant that each building block could be thoroughly tested on its own before it was connected to anything else. The unit test for this can be found in the package BuildingBlockTests. Once the cryptographic building blocks were created and tested, the next stage of the development cycle moved onto building the protocol. All the functionality of the protocol is within the CryptoProtocol package and each of the main methods are in their own class. This structure gave a simple logic to the project and made unit testing easy as much of the protocol was broken down into small units. Some tests for the protocol can be found in the package CryptoProtocolTests. The other parts of the system such as the connecting to the database, the server/client connection, and the methods in the GUI, were not as simple to unit test and most of their testing came in integration testing and black box testing.

Integration testing started as soon as the protocol was connected to the database and it it was most effective when all the backend was connected to the client/server system. Integration testing generally consisted of writing methods to automate the testing of certain features of the system. For example, this could be to hard code a sequence of actions a user could make. One of these tests is as follows: a failed login, a successful login, and entering an invalid Start Voting Key for an election. Most of this testing was done in the class VotingClientAutomaticIntegrationTests. The tests were ran and depending on the printouts, you could tell if they passed.

The third type of testing was black box testing on the GUI. This consisted of two phase. The first phase was to test the functionality of the GUI while the system was being built. The second phase was carried out after the majority of the system had been built. During this phase we went back to the list of requirements to check that each of them had been met by the system.

The final testing performed was usability testing. The aim of this was to evaluate how users interact with system and if they find it meets expectations for an e-voting system. We used the System Usability Scale (SUS) to carry out usability test (usability.gov, 2013). It is a quick usability test which has become an industry standard to evaluate a wide variety of systems usability. It consists of 10 questions (*appendix b*). For each question there are 5 possible responses that range from Strongly Agree to Strongly disagree.

Usability test was carried out with 6 participants. The each used the software and then answered the 10 questions.. To get a grade which can be compared to other systems, these scores are converted to different numbers and added together. The score per user ranged from 65 to 85 and

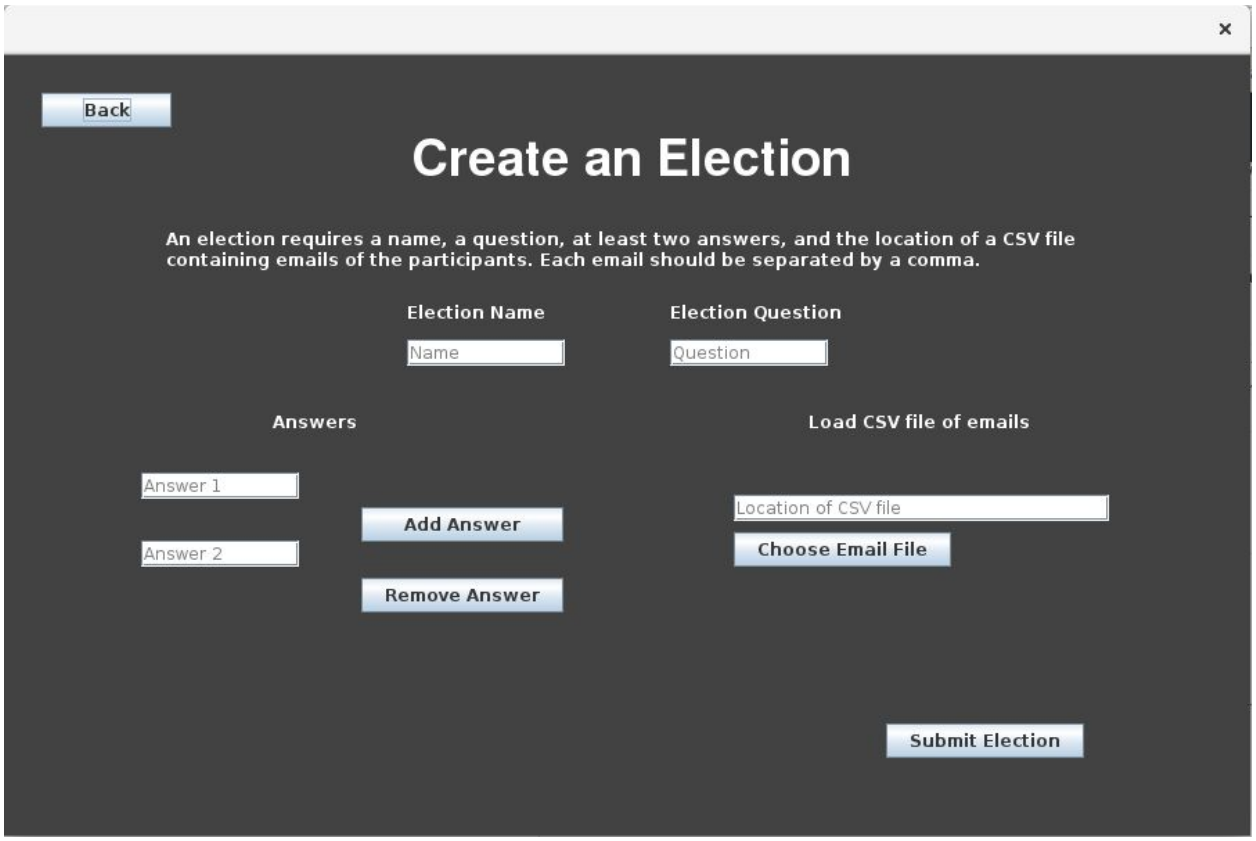
the final grade for this questionnaire was 72. To put this in perspective, anything above 68 is above average. 72 is an acceptable score but it does leave room for improvement. In spoken feedback from these questionnaires it was clear that the users felt that they were not given good direction on how the encryption was being performed. After this feedback, additional text was added to give a better explanation of the different tasks the user had to perform to confirm their vote. The results and the questions from the usability testing is contained in Appendix b.

In is important to note the limitations in this usability testing. Firstly, the small group size. Having a larger group to test the product on could have helped to identify more bugs and areas for improvement. Secondly, the lack of diversity of the participants. All of the participants were MSc students in Computer Science. This could clearly affect the feedback on a piece of software as the group only consists of people who have a specialisation in computer science. A better approach would include a wider group of participants and hopefully produce more accurate results according to the type of people who would actually use a piece of e-voting software.

User interface

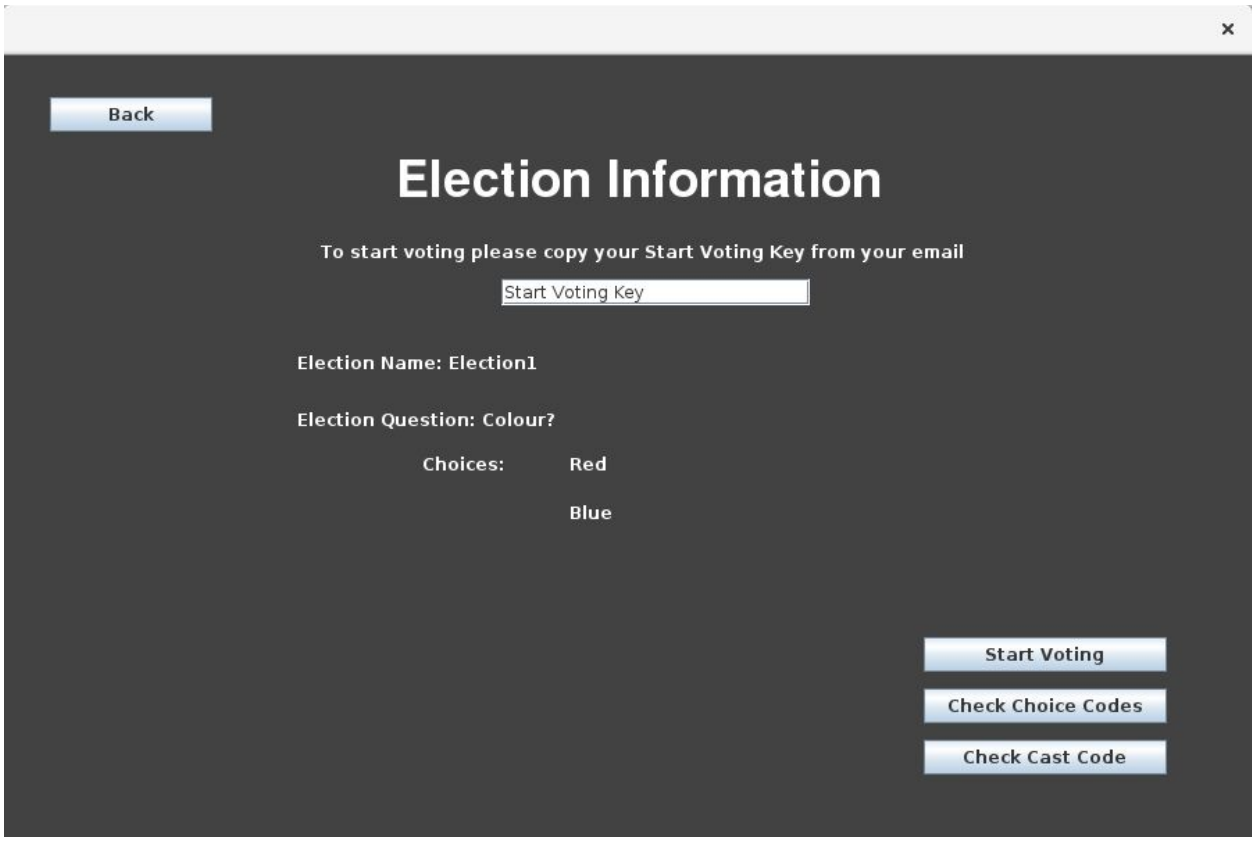
The user interface was designed using Java swing to create a desktop application. This application contains all the front-end functionality of the election system. This gives the user the ability to log-in, create an election, to vote in an election, to check their choice codes and check their ballot cast code. In designing the user interface we attempted to keep it as simple as possible with the use of a limited colour scheme and simple pages. Next, we will show two of the more complex pages to give a sense of what the App looks like.

Create Election View



This view shows the page a user sees if they request to create an election. It contains all the fields to provide the necessary information to create an election. The list of voters must be stored in a CSV file in the following format: *email1,email2, ..., emailn.*

Start Voting Key View



This view is shown to the user if they select an election for which the ballot is still open. There is a field for the Start Voting Key, which is used to generate their Voting Card ID which is used throughout the rest of the election. There are buttons to vote, to check Choice Codes and to check Vote Cast Code. The voter can only vote once but that can check their codes any number of times before the ballot closes.

Project management

The purpose of this project had three main features. First, to gain a general understanding of different e-voting systems, including the main ideas behind them and the cryptography underpinning them. Second, to understand one e-voting system in great detail, including the cryptographic features. And third, to take a high-level description of this e-voting system and build my own implementation of it. To carry out this project it has required time-management, planning and learning a number of new skills.

The project was split into four distinct phases. The first phase consisted of reading about e-voting systems and learning the core concepts surrounding them. This gave me a good background knowledge of the relative benefits of different e-voting systems. The second phase consisted of planning the piece of software and learning the new skills which would be required to build it. It was during this stage that major design decisions were made. The most significant was which platform should be used to build the system on. It was decided between either desktop app, mobile app or a website. Considering that mobile apps and websites are more widely used it would be preferable to make it in one of these. However, I do not have a background in either of these. It was decided then, that it would be better to spend more time learning about cryptography and so I chose to build a desktop app in Java. In this phase, I also had to spend significant amounts of time learning about the cryptography and the maths underpinning the system. I had a general idea about many of these features, but much of this was self-taught. When it came to the implementation of those concepts, I found many areas that I only knew superficially and require time to understand in detail. There were also features of Java which I had not used before and had to teach myself, these included crypto libraries, an email API, a secure networking library, Maven and Git. At the end of this second phase I had produced a requirements document and had knowledge of all the core features I would need to build the system.

The third phase of the project consisted of developing the software and testing the software. This phase took up the most time but I had already a robust plan of what I needed to build and all knowledge of all the necessary elements which would need to be constructed. The fourth phase consisted of turning all my writing so far into a full report and running usability tests on the software to make sure that it had met all the requirements. It was in this final phase that I could look back at the original proposal for the project and to check that all the aims of it had been met.

Results and evaluation

This next section will provide an analysis of the system that was built for this project. First, it will provide an analysis of the success of the project as an e-voting system and demonstrate how individual verifiability has been achieved. Then it will move to look at how robustness has been achieved and where it is still weak. Thirdly, it will provide a brief analysis of the performance of the software, including speed of computation. And lastly, it will provide a brief description of further developments which could be made to the project had there been more time.

Success as an e-voting system

The aim of this project was to build an e-voting system with individual verifiability. That is an e-voting system with cast-as-intended and recorded-as-cast verification. To demonstrate how this has been achieved we will go through three attacks outlined in (Scytl, 2017).

The first attack is a case where the voting device is corrupted and aims to change the choice made by the voter. The voter follows the steps of the protocol. However, to do this would require the voting device to generate the false short choice. As these codes are randomly chosen 4 digit numbers, this gives the chance of success at 1/1000.

The second attack happens if a voter makes their choice and confirms the choice codes but the device refuses to send the confirmation message. If the voter follows the protocol they should expect a vote cast code to be sent back to them. Once again this a randomly chosen number, this time of 8 digits. This means that the voting device has a 10^{-8} chance of guessing the correct vote cast code.

The third attack happens when a voter starts the voting process but decides not to complete the full process. The corrupt voting device ignores the voter and tries to confirm the vote without the voter's knowledge. To do this the voting device would have to guess the Ballot Casting Key. The probability of this is 10^{-8}.

All three of these analyses of attacks are dependant on the advantage of the attacker being negligible. That the attacker cannot see the information emailed to the voter, that the server is acting in an honest way and the attacker has no better strategy than guessing random numbers. The negligibility of the attacker has been shown in (Scytl, 2017). By achieving individual verifiability this system has also fulfilled a number of the theoretical requirements that were

outlined in the earlier section on the background of e-voting systems. Firstly, voter impersonation. As just demonstrated without knowledge of the keys and codes which are sent to a voter upon registration an attack can do no better than guess randomly when trying to false submit their ballot.

Vote secrecy has also been implemented in the use of ElGamal encryption of the votes. It is only with the Electoral Board secret key that they can be decrypted. To extend this to get complete privacy for the voter the addition of a mixnet is necessary. This would separate the decrypted vote from any indicator of who submitted the vote. In addition vote modification and vote deletion is protected against with the use of choice codes and the ability to repeatedly check them while the poles are open.

However, as was stated in the earlier section of this paper on background, to make an e-voting system good, there needs to wider security than just the protocol. For example, this protocol makes it impossible for a corrupt device to change the way a voter has voted but there is no security on the way that voters are registered. An attacker could alter a list of eligible voters without much problem. If this system was to be used it would be dependant on another system to track who is eligible to vote and only those people are included on the electoral roll.

A second area which was noted earlier in the paper was the necessity for the election be audited. If a full implementation of this system were built it must also include a method for people to audit an election. This could allow auditing of who is registered to vote, that votes had been correctly saved and correctly decrypted and the final count is accurate.

Robustness

Unit testing on the cryptographic building blocks has shown that they perform as expected when given correct data and handle invalid data as they should. Unit testing has also shown that the main components of the protocol also perform correctly with valid and invalid data. Testing has also been carried out on the database and on the client/server connection to verify that methods associated with them are robust.

As with any piece of software, we must expect that users will interact with it in unexpected ways. If a user follows the protocol as expected the software will act as expected. Black box testing and usability testing has also been carried out to remove bugs. However, there are still some improvements that should be made. The is currently no mechanism to check the number of times false passwords are attempted nor false Start Voting Keys are tried. The user is also only

given one opportunity to enter their Ballot Casting Key. The ability to try a password infinitely is a security flaw and only being able to enter a key once lowers usability. Both of these should be changed.

So far the system has been shown to be reasonably robust but it has not performed elections with more than a handful voters actually submitting votes. The next stage for testing robustness should be to use it for elections where a larger number of people participate.

Performance

There are three main stages of running the system where the time taken to perform an operation is important. The setup of an election and registration of voters is important as it requires the generation of many cryptographic keys, this could take a dangerously long time if not implemented correctly. Second is the preparation of the vote. This will be run on a voter's computer and so there is no way of knowing how powerful it will be. It will always be better to try and make this part of the process as computational quick so as to not slow a voter's computer. The third part that could possibly take a lot of computational power is the decryption and the tally of the votes. This was not rigorously tested as no election was held with a large number of voters.

The speed for setting up an election, that is running the SetUp algorithm has been tested in the class *RegistarSpeedTest* in the package *CryptoProtocolTests.* This shows that it is acceptably fast often taking between 100-150 milliseconds. A test run return this result: Time for SetUp, averaged over 20 attempts: 170.25 milliseconds.

The speed for registering a user is not as satisfactory. Tests for this are also in the *RegistarSpeedTest* class. An example result is: Time for SetUp + Register with 30 voters, averaged over 2 attempts: 50183.0 milliseconds. With average time per voter: 1672.76 milliseconds. It is important to break down where this time comes from, a large part of this is from the email API which is used to send the voter's ballot card. Speed tests for sending an email is found in *testingSpeedEmail* in the package *Email*. An example result is: Average time to generate and send an email, using 20 iterations is: 1311.9. This shows that the speed problem is in the email API rather than the cryptography. We have not yet looked into whether this can be improved. The email API also has a limit on the number of times it is used, this limits the amount it can be test. It may also limit elections being set up for large numbers of voter, this has not been tested beyond 200 voters at any one time.

The Setup and Registration of voters has been designed to be multithreaded. This was in part to make the system run more smoothly but it also shows that this stage of the creating an election is easily parallelized. This means that were you to run an election with vastly more voters, the extra work could be done in parallel rather than taking more time.

The time that it takes to encrypt a vote is generally good. The class *CreateVoteSpeedTest* in the package *CryptoProtocolTests* contains a method to test this. It usually came to around 35 milliseconds which is fast enough that the voter should not notice. In one example where 200 votes were encrypted, the printout from the test was: The average time to encrypt a vote, over 200 iterations: 35.13 milliseconds.

Future Improvements

To build a fully functioning e-voting system with complete verifiable would take considerably longer than the time allotted for this project. This, of course, would be the product to aim towards. However, there are some areas which would be important to improve on first. A significant improvement would be to implement a mixnet. This would ensure voter privacy and improve the overall security of the system dramatically.

A second area that should be developed is a system for auditing an election. Much of the data is there in the system already and stored on the database. This includes a number of proofs. But there is no channel for people to request it and to audit it themselves. To achieve this there needs to be a channel created where the server can publicise the data and methods created, which can be run by a general user, to verify this data. In addition, it may be of value to make a logging system. This would not store any sensitive data but would store data which could help should a voter claim that an error occurred. This kind of data could be numbers of login attempts or numbers of times invalid keys were entered

A third area that can be improved is usability. During usability testing, it was clear that people did not understand clearly the purpose of the different keys and codes they had to check. Some effort was made to solve this problem, but it could be taken further. Additional information in the email a voter receives when they are registered in an election would help people to understand the product they are using. This would improve the users' interaction with the software and could be achieved with a short video explaining the purpose of the choice codes.

Finally, this project was chosen to be built as a desktop app. The current trend is to move away from desktop apps and towards mobile apps or web-based systems. Moving the e-voting project to one of these would most likely be an improvement, as it would be on a platform people are more inclined to use.

Conclusion

The purpose of this project was to build an e-voting system with individual verifiability. To do this, the report looked into the requirements necessary for a complete e-voting system using three example, FOO, Helios and the Estonian system. Then, this report demonstrated the requirements that must be implemented for individual verifiability. This included, but was not limited to, voter privacy, cast-as-intended verification and recorded-as-cast verification. This project used a high-level description of a protocol from Scytl. From this high-level description, a working e-voting system was built using Java. It provided the functions to create an election, to vote in an election, tally the election and to see the results of an election. The method for achieving individual verification was in the use of choice codes. This meant that when a voter made their choice, the election server would generate a code that the voter could use to verify that their vote had been received correctly. This report described the cryptographic features that made this possible. And in conclusion, it gave an analysis of the e-voting system. It demonstrated how the system can withstand some attacks, some of the strengths of the system and some areas which could be improved upon in the future.

References

Adida, B. (2008) *Helios: Web-based Open-Audit Voting.* Available at: https://www.usenix.org/legacy/event/sec08/tech/full_papers/adida/adida.pdf (Accessed: 07.09.2018).

Alvarez, R.M., Hall, T.E. and Hyde, S.D. (2009) *Election fraud: detecting and deterring electoral manipulation.* Brookings Institution Press.

BBCNews (2005) *Estonia forges ahead with e-vote.* Available at: http://news.bbc.co.uk/1/hi/world/europe/4343374.stm (Accessed: 07.09.2018).

Bernhard, D. and Warinschi, B. (2014) 'Cryptographic Voting—A Gentle Introduction *Foundations of Security Analysis and Design VII* Springer, pp. 167-211.

Bouncy Castle (2013) *The Legion of the Bouncy Castle.* Available at: https://www.bouncycastle.org/java.html (Accessed: 7.9.2018).

ElGamal, T. (1985) 'A public key cryptosystem and a signature scheme based on discrete logarithms', *IEEE Transactions on Information Theory,* 31(4), pp. 469-472.

Estehghari, S., Desmedt, Y. and Rcis (2010) *Exploiting the Client Vulnerabilities in Internet E-voting Systems: Hacking Helios 2.0 as an Example.* Available at: https://www.usenix.org/legacy/event/evtwote10/tech/full_papers/Estehghari.pdf (Accessed: 07.09.2018).

Fiat, A. and Shamir, A. (1986) *How to prove yourself: Practical solutions to identification and signature problems.* Springer, pp. 186.

Fujioka, A., Okamoto, T. and Ohta, K. (1992) *A practical secret voting scheme for large scale elections.* Springer, Berlin, Heidelberg, pp. 244.

Heiberg, S. and Willemson, J. (2014) *Verifiable internet voting in Estonia.* Available at: https://www.researchgate.net/publication/282924965_Verifiable_internet_voting_in_Estonia (Accessed: 07.09.2018).

Lindell, Y. and Katz, J. (2014) *Introduction to modern cryptography.* Chapman and Hall/CRC.

Mitrou, L., Gritzalis, D., Katsikas, S. and Quirchmayr, G. (2003) *Electronic Voting: Constitutional and Legal Requirements, and Their Technical Implications.*

NDI (2013) *The Constitutionality of Electronic Voting in Germany.* Available at: https://www.ndi.org/e-voting-guide/examples/constitutionality-of-electronic-voting-germany (Accessed: Aug 22, 2018).

Orion. (2009) 'UW Computer Security Research and Course Blog » Security Review: Helios Online

Voting', . Available at:
https://cubist.cs.washington.edu/Security/2009/03/13/security-review-helios-online-voting/ (Accessed: Aug 28, 2018).

Pereira, O. (2016) 'Internet Voting with Helios' Floride (USA): CRC Press.

Puiggalí, J., Chóliz, J. and Guasch, S. (2010) *Best practices in internet voting.* Available at: https://www.scytl.com/wp-content/uploads/2013/05/PUIGGALI_BestPracticesInternetVoting.pdf (Accessed: 07.09.2018).

Scytl (2017a) *Individual Verifiability: Swiss Post E-Voting Protocol Explained*.

Scytl (2017b) *Swiss Online Voting System Cryptographic proof of Individual Verifiability.* Available at: https://search.credoreference.com/content/entry/heliconcwh/7_april_2017/0 (Accessed: 07.09.2018)

Springall, D., Finkenauer, T., Durumeric, Z., Kitcat, J., Hursti, H., MacAlpine, M. and Halderman, J.A. (2014) *Security analysis of the Estonian internet voting system.* Available at: https://jhalderm.com/pub/papers/ivoting-ccs14.pdf (Accessed: 07.09.2018).

Tomuschat, C. (2008) 'International covenant on civil and political rights', *United Nations Audiovisual Library of International Law, United Nations,* .
usability.gov (2013) *System Usability Scale (SUS).* Available at: /how-to-and-tools/methods/system-usability-scale.html (Accessed: Sep 5, 2018).

Valimised (2017) *Election results in rural municipalities and cities.* Available at: https://kov2017.valimised.ee/valimistulemus-vald.html (Accessed: 6.9.2018).

Appendixes

A. Location of code and instructions to run software

The software and source code for this project can be found in a GitLab repository at :

https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2017/ohr705

Two Jar files, from which the application can be run, are found in the folder ApplicationJars in this repository:

https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2017/ohr705/tree/master/ApplicationJars

To run the Server use this command, with the argument of the port number:

java -jar EVotingServer.jar *portNumber*

To run the Application use this command, with the IP address and port number as arguments:

java -jar EVotingApp.jar *IPAddress Port*

The source code can be found at, and contains the following folders:

https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2017/ohr705/tree/master/src

```
├── BuildingBlocks          (Cryptographic Building Blocks)
├── BuildingBlocksTest       (Tests for Building Blocks)
├── CryptoProtocol          (Main protocol methods)
├── CryptoProtocolTests     (Tests for the protocol methods)
├── database                (Classes for the database connection)
├── Email                   (Classes for the Email API)
├── gui                     (All the front-end)
├── NetworkProtocol         (Class to form messages between client/server)
├── NetworkProtocolTest     (Test for network messages)
├── server                  (All server classes)
├── votingClient            (All Client classes)
└── votingClientTests  (Tests for the client/server)
```

B. Usability Questionnaire and Results

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
Participants	I think that I would like to use this system frequently	I found the system unnecessarily complex	I thought the system was easy to use	I think that I would need the support of a technical person to be able to use this system	I found the various functions in this system were well integrated	I thought there was too much inconsistency in this system	I would imagine that most people would learn this system very quickly	I found this system very cumbersome to use	I felt very confident using the system	I needed to learn a lot of things before I could get going with this system
1	4	1	4	1	4	1	4	2	2	3
2	5	1	4	2	3	3	3	3	4	4
3	4	2	5	5	4	1	4	3	3	3
4	3	1	4	2	4	2	5	2	3	4
5	3	2	5	2	5	1	5	1	5	3
6	5	3	5	1	5	1	3	3	3	4