vVote Verifier – A stand-alone reference implementation of a verifier for the vVote Election System

by

JAMES RUMBLE
URN:6092746

A dissertation submitted in partial fulfilment of the requirements for the award of

MASTER OF SCIENCE IN SECURITY TECHNOLOGIES & APPLICATIONS

September 2014

Department of Computing
University of Surrey
Guildford GU2 7XH

Supervised by: Stephan Wesemeyer

I declare that this dissertation is my own work and that the work of others is acknowledged and indicated by explicit references.

James Rumble

September 2014

Abstract

In contrast to paper based systems, which are generally widely understood; electronic voting systems can often be overly complex and can require deep domain knowledge of the technologies involved. Voters have two key conflicting requirements; they want to have both anonymity in their vote and also want the election system itself to be verifiable. The problem lies in providing both verifiability and anonymity through privacy in such a way that voters are given enough information that they can prove to themselves that their vote was included as cast contributing to the final election tally whilst not giving enough information that they can prove, even if they wanted to, to another party how they voted. Verifiability of an electronic voting system is a key factor in the success of its adoption and for gaining voter trust.

The vVote System [1] is an end-to-end verifiable electronic election system based on the Prêt à Voter election scheme [2]. vVote was designed and built to be used as part of the State Election taking place in Victoria, Australia in November 2014. vVote provides three core pieces of verifiable evidence including:

- Cast-as-intended – That each vote was cast in the way it was intended.
- Counted-as-cast – That each vote was counted in the way it was cast.
- Universally verifiable tallying – That the tallying process is verifiable.

This dissertation details the implementation of a stand-alone independent reference implementation of a verifier for the vVote system – the vVote Verifier. The vVote Verifier focuses on the verification of three core areas of the vVote system; specifically:

- The Ballot Generation process – That a randomly selected set of ballots have been generated successfully.
- The Vote Packing process – That the packing together of a number of voter preferences to reduce the number of ciphertexts passed to a Mixnet has been correctly carried out as well as the way the preferences are recovered after mixing.
- The commits made to the Public WBB – That each of the commits made to the Public WBB has been correctly hashed and signed by the Public WBB meaning the data has not been tampered with.

Acknowledgements

I would first like to thank my supervisor Dr. Steve Wesemeyer for all his support and guidance provided throughout the course of the year and for constantly challenging me to do better. I would like to thank Dr. Steve Schneider for giving me the opportunity to work on one of the most interesting and demanding problems I have ever worked on and for introducing me to the area of secure electronic voting. My deepest appreciation goes to Dr. Chris Culnane for his endless patience and always finding the time to go over the finer details.

Last but not least I want to thank my family for their continued support and encouragement during my years of study.

Contents

Table of Figures

Table of Tables

Table of Code Extracts

Abbreviations

Commit Identifier (CID)	A unique identifier used for a commit during a specific voting session.
Print on Demand (PoD) Printer	Used to print ballots containing the required candidate names on demand for voters.
Victorian Electoral Commission (VEC)	Independent authority responsible for conducting Victorian State Elections.
Left Hand Side (LHS)	Generally used to describe the LHS of a ballot paper.
Right Hand Side (RHS)	Generally used to describe the RHS of a ballot paper.
Generic Ballot	A ballot containing all possible candidates which can then have candidates removed as applicable to tailor the ballot to any specific district configuration.
Legislative Assembly (LA)	
Legislative Council (LC)	
Above The Line (ATL)	
Below The Line (BTL)	
Public Web Bulletin Board (Public WBB)	
Private Web Bulletin Board (Private WBB)	
Web Bulletin Board (WBB)	
Electronic Ballot Marker (EBM)	

Glossary

PK_E	The thresholded election public key with corresponding private key SK_E
SK_E	The thresholded election private key with corresponding public key PK_E
PK_P	A PoD Printer's public key with corresponding private key SK_P
SK_P	A PoD Printer's private key with corresponding public key PK_P
n	The total number of candidates
b	The total number of ballots to be generated for any PoD Printer
G	The total number of randomness generation servers
a	The total number of ballots chosen for auditing for any PoD Printer
p	The total number of PoD Printers
s	The total number of validly submitted ballots containing voter preferences
$RGen_i$	The $i'th$ randomness generation server used during Ballot Generation. With i being between 0 and $\mathrm{G}-1$.
RT_i	$RGen_i$'s private table containing pairs of (random, witness) values
CRT_i	$RGen_i$'s public table containing corresponding commitments to the random values in RT_i. Each commitment can be opened using the witness value as the commitment opening
$SymmEnc_{sk}(m)$	Represents symmetric key encryption of message m using the symmetric key sk
$SymmDec_{sk}(m)$	Represents the symmetric key decryption of message m using the symmetric key sk
$h(m)$	A cryptographic hash of the message m.
$\mathrm{Enc}_k(m,r)$	Represents the encryption of message m with the randomness value r under the public key k.
$Dec_k(m)$	Represents the decryption of message m using the private key k.
$c(m,r)$	Represents the commitment to the message m using some randomness value r

Chapter 1 – Introduction

1.1 Introduction

The democratic election process underpins vast swathes of modern society and is the mechanism whereby the eligible decision making public exercises the power vested in them to elect a group of representatives. For such an integral part of society as a whole the voting process has not changed significantly since its early inception in Ancient Greek democracy. It could be considered surprising that in the technological age, while complex and new systems including banking and shopping to the operation of airplanes and even autonomous cars are going through rapid and broad advances, the development and improvement in voting systems has lagged far behind and remained mainly a paper based exercise having not changed significantly in hundreds of years.

Recently significant efforts have been put into developing usable electronic voting systems which could dramatically alter the way in which elections take place as well as bringing a number of potential benefits including better accuracy in the tallying phase of an election, better and more efficient coverage of remote areas, increased turnaround for the time taken to announce the election result, a possible reduction in the operational costs of elections and even making it possible to vote from any location. The security requirements of any electronic voting system are similar to that of their paper based counterparts and include:

- Correctness – The election result is accurately calculated.
- Privacy – Each voter's identity is kept hidden.
- Receipt freeness – that a voter is unable to prove to anyone how they voted, even if they want to. This provides prevention against vote selling and coercion.
- Robustness – The system needs to be robust and tolerant against certain failures both random faults as well as deliberate attempts to disrupt the election.
- Verifiability – the protocol can be provably trusted.
- Democracy – voters can provide at most a single vote.
- Fairness – No partial tallies should be revealed before the end of the voting period to prevent voters from being influenced by the current tally.

To this day there has not been a successfully implemented and widely adopted choice for providing electronic voting to the masses. The main problem seems to be the issue of voter trust in the election systems. In contrast to paper based systems, which are generally simple to explain and therefore widely understood; electronic voting systems can often only be understood by those with specific domain knowledge of the technologies involved, including that of advanced cryptography. The single biggest hurdle in the adoption of widespread use of electronic voting systems is that of voter trust. The issue of trust can be somewhat traced back to the conflicting requirements of verifiability and privacy. Voters want to have both anonymity in their vote, such that there is no way to tell how they voted, as well as having the election system be completely verifiable, so that they are able to determine that their vote is included in the tally but no one else is able to know their preferences. It should be noted that paper based election systems using the secret ballot approach [3] provide only voter privacy and do not have any mechanisms in place for verifiability of the election, meaning voting officials and the election processes must be trusted. There is little that can be done to prevent the modification of ballot papers other than trusting the procedures put in place as well as those trusted to honestly carry out them.

The privacy requirement is well understood and simple mechanisms exist which can provide anonymity to voters including the removal of voter identification from cast votes or the use of encryption. The verifiability requirement is also well understood; in normal systems we have user accounts which provide identification which then allow for authentication, accountability and non-repudiation. Normal systems also tend to produce paper trails and logs which can be used to verify that a specific action has taken place as it was intended. The problem, for voting systems, lies in providing both verifiability and privacy in such a way that voters are given enough information that they can prove to themselves that their vote was included as cast in an unmodified manner contributing to the final election tally whilst not giving enough information that they can prove, even if they wanted to, to another party how they voted. Verifiability is a key factor in the success of and adoption of an electronic election system. Verifiability of an electronic election system can be broken down into three core checks which can be used to ensure the correctness of an election:

- Cast-as-intended – That each vote was cast in the way it was intended by the voter. Each voter receives a receipt that their vote was cast in the way they intended. This receipt, however, shouldn't provide enough information that any additional details can be learnt as to how the voter voted.
- Counted-as-cast – That each vote was counted in the way it was cast. Each voter receives evidence that their individual vote was included in the unaltered tally, however, this evidence should not provide any additional details that can be used to learn how the voter voted.

- Universally verifiable tallying – That the tallying process is verifiable. Anyone, including those taking part in the election and independent parties, can check that the list of cast votes produces the tallied election outcome.

The vVote system [1] is an end-to-end verifiable electronic election system based on the Prêt à Voter election scheme [2]. vVote was designed and built to be used as part of the State Election taking place in Victoria, Australia in November 2014. The underlying protocols used in the vVote system are designed to be universally verifiable allowing anyone, both those taking part and independent parties, to carry out verification on the protocols using publicly available information. Verifiability in the vVote system was a key design aim and means that there are no trust assumptions needed for guaranteeing the integrity of the election result. Voters are provided with a receipt which can be used to verify that their vote was cast and counted as they intended. The vVote system will also provide sufficient information that the election result can be universally verified.

1.2 Project Overview

This project will, in essence, put the verifiability claims of the vVote system to the test by developing an independent reference implementation of a verification tool for the vVote system. This dissertation will outline the analysis, design and implementation of a verification tool for the vVote system specifically focusing on the portions of the code base developed at the University of Surrey.

In vVote each voter is supplied with a ballot which presents the candidates in a randomly permuted order on the left hand side with the right hand side being used to supply their preferences. The right hand side of the ballot contains the permutation of the candidates in an encrypted form. The candidate list is then removed and destroyed leaving only the voter preferences and the encrypted permutation. The right hand side of the ballot is then scanned into the system and a receipt is provided to the voter which includes their preferences as they provided. The successfully submitted votes are then shuffled together using a Mixnet which anonymises them. The Mixnet shuffling removes all mappings between voter and votes. The encrypted preferences of each vote, after shuffling, are then jointly decrypted revealing the preferences which can then be used to perform an election tally. After all the ballots have been cast the system will publish, on a public web bulletin board, all of the voter receipts as well as all of the decrypted votes. A key aspect of vVote is the use of a printed receipt provided to voters which can be used to verify that their preferences were included in the tally as they intended without revealing their actual preferences. The receipt will show the preferences chosen by the voter but without the corresponding randomly permuted candidate list.

The verification process of the vVote system will focus specifically on a number of protocols or processes carried out including:

- Ballot generation – The Ballot Generation process takes place during the pre-voting stage of the election and involves the production of generic ballots containing randomly permuted candidate names along with the candidate permutations. The random candidate permutation is constructed using combined randomness values generated and communicated by a number of independent randomness generation servers. The verification process involves the auditing of a randomly selected set of generic ballots. Each audit involves the recalculation and verification of the generic ballot.
- Vote packing – The Vote Packing process is used to reduce the amount of work carried out by the Mixnet during the tallying phase of the election. The vVote system uses a Mixnet to shuffle the voter preferences which are represented by re-encrypted candidate identifiers sorted into preference order. The vVote system uses a method of packing together multiple re-encrypted candidate identifiers before passing them to the Mixnet meaning far fewer ciphertexts require shuffling and decryption thereby producing significant efficiency benefits. The verification process is two-fold.
    1. The first check is used to verify that the ciphertexts provided to the Mixnet have been correctly packed according to the set of underlying voter preferences and that the packings match those that were input to the Mixnet.
    2. The second check is used to verify that the decryptions from the Mixnet, corresponding to packed plaintexts, correctly match the packing of the plaintext candidate identifiers using the claimed preferences from the Mixnet output.
- Commits made to the Public WBB – The Public WBB is used by the system to make available publicly relevant information which can be used in the verification process for other components such as the Vote Packing and Ballot Generation processes. The Public WBB is simply updated once per day as separate commitments. Each commitment consists of a JSON message file, an attachment ZIP file and a JSON signature file. The verification process involves a check that for each commit a signature was correctly calculated over the data. This check is used to verify that information has not been modified or implanted onto the Public WBB in an unauthorised manner.

The vVote Verifier will be designed specifically against the documentation and theoretical details of the vVote system to try to keep its implementation independent; however, in places this documentation was lacking technical details and specific functionality for certain parts and therefore reverse engineering of the existing system was also required. The main deliverable of the project was to produce a stand-alone independent reference implementation of a verifier for the vVote system. An additional deliverable was to create accompanying documentation regarding the specifics of the components being verified, as well as the verification details to aid in the understanding of the verifier produced or even to aid in the development of another implementation of a verifier for the vVote system.

1.3 Project Aims

This project has a number of aims both specifically towards the project being undertaken and also personal desirable aims and objectives which contributed to the decision to undertake this specific project. The main aim of the project is to carry out an in-depth review of the vVote electronic election system arriving at the knowledge and understanding required to carry out an analysis, design, implementation and full testing of a verification tool for the vVote system in addition to creating accompanying documentation for the tool. A number of less obvious personal aims of the project included the opportunity to expand upon the theoretical computer security knowledge gained during a Masters in Security Technologies and Applications putting the techniques and theories learnt into practice in a real world application. Another aim was to understand and use more advanced cryptographic techniques such as Elliptic curve cryptography and to understand why and how it is used in a real world application expanding upon the theories learnt, such as the basics of public key cryptography using RSA. The most exciting aim of the project was to work on and develop a large piece of open source code which upon completion would be included as part of the University of Surrey vVote delivery to the Victoria Electoral Commission.

1.4 Chapter Overview

Each chapter of this dissertation will follow on from the last, utilising the information and concepts described previously. The dissertation has the following structure:

Chapter 2– Literature Review – This chapter contains the literature review carried out for the project. It provides background information for some of the more important concepts and topic areas which will be used throughout the remaining chapters.

Chapter 3– vVote System – This chapter provides an in-depth analysis and explanation of the vVote system. Some of its core aspects are explained in detail and the techniques explained step by step. The explanation of the components of the system thereby helps with the understanding required for the verifier which was written to verify the correct working of its components.

Chapter 4– vVote Verifier Analysis – This chapter details the requirements of the problem to be solved which help with the design and implementation chapters.

Chapter 5– vVote Verifier Design – The design chapter focuses on the design of the verifier. It describes the way the design took place and the way in which the verifier is broken into a number of independent modules to help with its maintainability and code clarity.

Chapter 6– vVote Verifier Implementation – The implementation chapter details the way in which the system was developed. Some specific aspects are highlighted through the use of code snippets and examples. Complex components will be focused on in addition to a number of the challenges faced.

Chapter 7– Testing and Evaluation – The testing and evaluation chapter details the testing carried out on the system including example sample data.

Chapter 8– Conclusion and Future Work – The final chapter concludes the report detailing the results of the dissertation as a whole and provides a number of possible extensions which could be made for the verifier.

Chapter 2 – Literature Review

2.1 Introduction

When undertaking any project of significant size it can be helpful to carry out a review of the subject by delving into the work and research carried out by others. This chapter will outline and introduce a number of concepts used throughout the dissertation. The chapter will start with an overview of voting in general, move onto cryptographic primitives, then to an introduction of utilising Cryptography to tackle the problem of voting, moving onto a description of electronic voting including the description and outline of a number of existing electronic voting systems and concluding with a description of the Victorian State Election System and its peculiarities. Chapter 3 will provide an extension to the literature review and provide an overview of the electronic voting system vVote which will be used in the 2014 Victoria State election.

2.2 Voting

Democracy is a form of government whereby each and every eligible member of society has the right and obligation to participate. In democracy the real power is given to the people and may be exercised either directly or indirectly through a group of elected representatives. The election process, in a democratic society, is so integral to the philosophy of democracy that without it democracy could not exist [4]. Each eligible citizen has an equal right to express their preference towards a representative, with each voter having the same influence over the result. This preference is termed their vote. You cannot have a truly successful democratic society without proper voting procedures put in place. These voter procedures can be termed an election.

2.2.1 Secret Ballot

For a long time the election systems being used were far from democratic where voters could easily be influenced by external sources. Voter preferences were publicly known and recorded such that people knew how other people had voted. Those with power could exercise pressure on voters to vote a specific way or could pay voters to vote in any way [3]. The concept of using a secret ballot changed everything and removed the ease at which voters could be influenced. Nobody but the voter should be able to know how any voter cast their vote.

2.2.2 Voting procedure

Most typical elections follow a similar procedure consisting of 4 distinct phases as follows:

1. Set-up phase – Voting parameters are setup such as the eligibility criteria for candidates and voters. The rules by which the votes will be counted are also confirmed. These parameters are made public.
2. Registering phase – Voters register with the authorities and their eligibility is determined from the setup phase. Ineligible voters are not allowed to register and vote. A list of authorised voters is made public.
3. Voting phase – eligible registered voters cast their votes in the following format:
   - Voter authentication – The voter is authenticated against the list of authorised voters from the registration phase.
   - Voter registration – Each authorised voter receives an empty ballot. The voter may then choose their preference(s) in secret to avoid any influence by external sources.
   - Ballot casting – the ballot is cast by the voter and is anonymised so that there is no lasting mapping between the ballot and the voter.
4. Tallying phase – all cast ballots from the previous stage are processed to produce the election result.
   - Ballot collection – Ballots are collected together in a central location.
   - Ballot verification – Each ballot is checked to determine whether it has been cast validly or invalidly according to the election rules setup and published in the setup phase. Invalid ballots are removed.
   - Vote counting – Valid ballots are counted using the tallying rules. The results from any separate locations are aggregated and the election result is made public.

[5]

2.2.3 Election methods

Various elections will utilise contrasting counting and voting methods. The most important of these, for this report, include:

- Party List-PR – Voters are presented with and choose their preferences from lists of candidates. The representatives are chosen according to the party's share of the set of votes.
- First Past The Post (FPTP) – Voters simply choose a single preference corresponding to their favoured candidate and the candidate with the most votes wins with all votes for other candidates meaning nothing.
- Single transferable voting – Candidates need a known share of the votes, determined by number of voters and number of seats, rather than a majority of the votes to be elected.

Each voter can choose a number of preferences which can transfer from their first preference to lower preferences as the chosen candidate can no longer be elected.

- Borda Count – Borda Count is a form of preferential voting. Voters rank candidates in preference order with these preferences then converted to points. Candidates who are ranked last receive one point; candidates ranked next-to-last receive two points and so on. The points are then tallied and the candidate with the most points is the winner.
- Instant-runoff voting (IRV) – Voters rank as many candidates as they wish in decreasing order of preference. The tallying process is as follows: Candidates are elected if they achieve the majority of first preference votes. If there is no majority then the candidate with the least number of first preferences is removed and their votes redistributed according to the second preference.

[6]

2.2.4 Desirable Properties of a Democratic Election

There are numerous requirements for any effectively held democratic election; however, the most important of these are as follows:

- Secrecy – Nobody but the voter should be able to know how they voted; otherwise external sources could put pressure on a voter to vote a specific way.
- No Sale – It should be impossible for a voter to prove how they voted, even if they wanted to. A voter should be unable to reveal how they voted in any way other than asserting it. This means the selling of votes can only be carried out by the voter giving their word to the buying party that they voted a specific way.
- Invisible Abstention – Nobody but the voter should be able to tell whether they voted in the election; otherwise pressure could be made on voters to abstain from voting completely.
- Verifiability – Anyone should be able to verify that only those authorised to vote did so. That the authorised voters voted at most once and it was made in a valid way and that these valid authorised votes were used correctly to construct the election result. Each voter should be able to see that their vote was included, unaltered as part of the election result.

2.2.5 Why Voting is a Hard Problem

This section will briefly consider some of the difficulties and complexities involved in voting. First an example will be used to outline a number of complexities in what at first can be considered a simple problem.

We have two voters: Alice and Bob, a single coercer, Eve, wishing to influence Alice and two parties, Green and Blue, which Alice and Bob can choose between. Alice wants to cast a vote for Green whereas Bob wishes to cast a vote for Blue. Eve wants to influence Alice into choosing to cast a vote for Blue instead of Green thereby making sure the Blue party wins the election. Eve can be an external party, someone involved with the Blue party or even an election official. The problem with the above is that there is a functional conflict between the two requirements outlined above – secrecy and verifiability. On the one hand both Alice and Bob want to verify that the entire election took place correctly, including that their votes were cast as they intended, however, if in verifying the voting process either Alice or Bob can then convince Eve, the coercer, of how they voted then their vote can easily be sold. Eve can simply pay Alice to change her vote to Blue and Alice can prove to Eve how she voted prior to being given the money for voting a particular way. In this case the secrecy requirement has been broken.

One of the many difficulties is in providing voters with enough information so that they can personally verify that their vote was recorded as they intended but not enough information to prove to another party how they voted. In this example we need for Alice to be able to prove to herself that her vote was recorded as being for Green and counted accordingly without being able to prove to Eve how she voted. The coercer now has no incentive to pay for votes as she can no longer tell if her money has gone to good use and been used to actually influence how a voter would cast their vote.
[7]

It could be considered surprising that in the technological age where complex systems including everything from banking and shopping to the operation of airplanes and even autonomous cars can be understood and put into practice whilst voting has remained mainly a paper based system having not changed significantly in hundreds of years. Two points are highlighted with the aim of showing how difficult the problem of voting is:

- Incentive – The incentive for breaking a governmental election should not be underestimated. The reward of power can be considered far more valuable than that of money, which is generally the reward for those breaking traditional systems such as banking or shopping [8].
- The failure detection and recovery process for an election can be far more difficult, in some cases impossible, to recover from whereas recovery in other systems is well understood and catered for [9].

[7]

2.3 Cryptographic Primitives

This section provides a brief overview and introduction to a number of cryptographic primitives used throughout electronic voting systems in general and specifically the vVote Verifier system developed as part of this dissertation.

2.3.1 Public Key Cryptography

At the heart of public key cryptography, also termed asymmetric cryptography, is the idea that any party can encrypt and send a message to a party A, but only A can decrypt and read the original message. Public key cryptography uses two related keys – a public key, PK and a private key, SK; PK is publicly known and can be used to securely encrypt messages to party A whereas SK is secretly held by A and is used to decrypt messages. Only the holder of the private key SK can decrypt messages encrypted using the publicly known public key. Public key cryptography generally relies upon the difficulty of carrying out a task such as factoring or carrying out discrete logarithms where one operation is 'easy' such as multiplication and modular exponentiation respectively, but the inverse, finding a discrete logarithm or carrying out prime factorisation. In this way any party can carry out the 'easy' task but only the holder of the private key is able to efficiently carry out the inverse operation.

2.3.1.1 ElGamal Cipher

The ElGamal encryption system is a form of public key cryptography using the difficulty of carrying out discrete logarithms over finite fields and is based on the Diffie Hellman key exchange protocol [10].

The ElGamal encryption process of Bob sending an encrypted message to Alice uses three steps [11]:

1. Key generation:
    - Alice chooses a generator g of order q from cyclic group G.
    - Alice chooses a random number x from $\{1 \dots q-1\}$. x is retained as Alice's private key.
    - Alice computes $y = g^x$.
    - Alice publishes y along with the generator g as her public key.
2. Encryption – Bob wishes to encrypt a message m for Alice:
    - Bob chooses a random number r from $\{1 \dots q-1\}$ and calculates $c_1 = g^r$.
    - Bobs calculates the shared secret value $s = y^r$.

- The message m is converted into an element m' of G.
- Bob calculates $c_2 = m'.s$.
- Bob sends the ciphertext $(c_1, c_2) = (g^r, m'.y^r) = (g^r, m'.(g^x)^r)$ to Alice.

3. Decryption – Alice receives ciphertext (c_1, c_2) from Bob and wishes to decrypt the message with her private key x:
   - Alice calculates the shared secret value $s = {c_1}^x$.
   - Alice recovers m' by computing $m' = c_2.s^{-1}$ which is then converted back into the plaintext message m.

The strength in the ElGamal cipher is the difficulty in carrying out discrete logarithms or finding root extractions. As long as these problems remain 'hard' an ElGamal cipher, when used appropriately, is a strong encryption technique. In the above example Alice is able to easily decrypt the encryption carried out by Bob because she knows the value of the secret key x. No party, other than Alice, including Bob, can work out the message m after it has been encrypted for Alice.

2.3.1.2 Elliptic Curve Cryptography

Elliptic curve cryptography is another form of public key cryptography utilising a branch of mathematics called elliptic curves. Elliptic curve cryptography is based on the hard problem of carrying out elliptic curve discrete logarithms. For problems using numbers of the same size solving elliptic curve discrete logarithms is much harder than factoring therefore cryptographic systems based on elliptic curves are stronger than those based on factoring – such as RSA [12]. Elliptic curve cryptography can therefore use shorter key sizes whilst remaining just as secure as their factoring counterparts. Elliptic curve cryptography has high security whilst using short keys which results in better performance [13]. An elliptic curve is a plane curve over a finite field which have points satisfying the equation $y^2 = x^3 + ax + b$.

2.3.1.3 ElGamal Elliptic Curve Encryption

It is possible to extend the fundamentals of ElGamal cryptography to produce an Elliptic curve analog of the ElGamal cryptosystem [14]. An elliptic curve analog of the ElGamal cryptosystem using the multiplicative group of the finite field may produce enhanced security due to the increased difficulty of solving the analog of the discrete logarithm problem on elliptic curves. As explained previously the strength in the basic ElGamal cipher is the 'difficulty' of carrying out discrete logarithms. ElGamal Elliptic Curve Encryption can be considered a more difficult problem than that of ElGamal due to the increased difficulty of solving discrete logarithm problems on elliptic curves.

The process of carrying out an elliptic curve analog of ElGamal encryption of Bob sending an encrypted message to Alice uses three steps:

1. Shared setup steps:
   - Setup an elliptic curve E over a field F_q and a point P.
   - Setup a publicly known function $f: m \rightarrow P_m$ which maps messages m to a point P_m on the elliptic curve E. This function f should be reversible.
   - Each party chooses a secret key x from the field F_q. Publish the point $Y = xP$ as a public key.
2. Encryption – Bob wishes to encrypt a message m for Alice :
   - Choose a random r from the field F_q and calculate $c_1 = rP$ and $c_2 = rY$
   - Convert the message m to point P_m on the elliptic curve E using function f.
   - The ciphertext is the tuple $(c_1, c_2 + P_m) = (rP, rY + P_m)$.
3. Decryption – from a ciphertext (C, D):
   - Calculate $C' = xC$.
   - Retrieve the point P_m using $P_m = D - C' = (r(xP) + P_m) - x(kP)$.
   - Retrieve message m from point P_m using $f^{-1}(P_m)$.

[15]

2.3.2 Homomorphic Encryption

Homomorphic cryptosystems allow for some binary mathematical operation to take place on encrypted data which is also mirrored onto the underlying plaintext. If we have an algorithm which encrypts plaintext p to ciphertext c and we then multiply the ciphertext by 2 as $2c$ and then decrypt the ciphertext $2c$ and receive a result of $2p$ then that algorithm would be classed as homomorphic [16].

The elliptic curve analog of ElGamal cryptography is a homomorphic algorithm and provides homomorphism for either addition or multiplication but not both.

2.3.3 Re-encryption

A simple usage of the homomorphic property of any cryptographic system is that of re-encryption whereby given a ciphertext c corresponding to the encryption of a plaintext p a different ciphertext c' can be created by re-encrypting c which can still be decrypted to the same plaintext p.

In the case ElGamal cryptography we can re-encrypt an ElGamal ciphertext $E(m, r) = (c_1, c_2) = (g^r, m.y^r)$ to produce a new ElGamal ciphertext (c_1', c_2') which decrypts to the same underlying plaintext message m. To carry out the re-encryption the party must choose a new random r_1. The re-encryption is $E(E(m,r),r_1) = (c_1.g^{r_1}, c_2.y^{r_1}) = (g^{r.r_1}, m.y^{r.r_1})$. When ciphertext (c_1', c_2') is decrypted in the usual fashion both randomness values will be stripped off and the initial plaintext message m will be recovered.

2.3.4 Threshold Cryptography

In threshold cryptography a secret key is divided into a number of discrete shares which are distributed to a number of independent components. A secret S is divided into n pieces of data $D_1, \ldots, D_n$ in such a way that:

1. Knowledge of any k or more D_i pieces makes the secret S easily computable.
2. Knowledge of any $k-1$ or fewer D_i pieces leaves S completely undetermined – in such a way that all its possible values are still equally likely even if a number of the shares of S are known.

This scheme presented in [17] is called a (k,n) threshold scheme. If $k >= n$ then the secret S is easily computed.

Threshold cryptography relies upon the basis that if at least a threshold number of entities, each having partial shares of some secret S, come together in agreement with their shares then the initial secret can be reconstructed however without a threshold agreement no information is revealed about the secret. Threshold cryptography can be used to ensure full trust is not placed in any one component such that a malicious activity by one or more but less than the threshold number of components will not be accepted.

2.4 Secure Voting Schemes

Section 2.2.5 Why Voting is a Hard Problem outlined a number of problems with traditional election systems primarily highlighting the conflicting requirements of verifiability and privacy. The general approach to managing this requirement is to avoid the verifiability requirement and stick to privacy meaning voters must trust that their votes have been cast in the way in which they were submitted and that processes are in place to avoid ballots being added, removed or modified. Specifically voters must trust that the election result has been correctly calculated with regard to the votes cast by voters. Secure voting schemes tend to employ the use of cryptographic primitives to provide their security requirements with a key concept being a system's verifiability. In normal election settings voters are required to simply trust that any equipment used and those involved in the election have acted in a correct and honest manner. In secure voting systems the correct and honest behaviour of the system can be publicly verified.

Secure voting systems have a number of important requirements for providing a secure secret ballot election:

- Integrity – Only eligible citizens may cast votes, with each eligible voter only allowed to submit a single vote. Out of these votes only valid votes are counted and included in the election tally.
- Correctness – The result can be reliably obtained if all participants including voters, candidates and election officials have acted in an honest manner.
- Privacy – the mapping between any voter's identity and any corresponding vote must be kept private. Voters may therefore provide their true preferences without the possibility of any influence.
    - In voting, this anonymity requirement also depends on the total number of ballots cast as well as the total number of variations possible in a vote. The short ballot assumption requires that all ballots have low information content meaning each ballot is likely to turn up at least several times meaning a single ballot cannot be reliable tracked back to a specific voter.
- Receipt freeness – Is an enhancement of the privacy requirement and means that voters should not be able to construct a receipt proving the way in which they have voted to any other party. This requirement prevents the buying and selling of votes in addition to coercion.
- Fairness – No partial tallies should be revealed before the end of the voting period to prevent voters from being influenced by the current tally.
- Robustness – The system needs to be robust and tolerant against certain failures including both random faults as well as deliberate attempts to disrupt the election.
- Verifiability – The election result correctness as well as the processing of votes can be publicly verified by voters. An enhancement of this individual verifiability is that of universal verifiability whereby anyone including those not involved in the election can verify the correctness of the election.

The conflicting requirements of verifiability and privacy as well as the difficulty of also achieving the others show just how complex a secure election system can be to get right.

2.4.1 End-to-End Verifiability

An extension to the verifiability requirement outlined previously is that of end-to-end verifiability [18]. The requirements for such a system include:

- Cast-as-intended – That each vote was cast as intended. Each voter receives a receipt that can be used to check that their vote was cast in the way they intended.
- Counted-as-cast – That each vote was counted in the way it was cast. Each voter receives evidence that their individual vote was included in the unaltered tally.
- Universally verifiable tallying – That the tallying process is verifiable. Anyone can check that tallying process was carried out correctly.

2.4.2 Electronic Voting (E-Voting)

Electronic voting systems are an extension of secure voting schemes. It should be noted that not all secure voting schemes are necessarily electronic whereas electronic voting schemes tend to be of the secure nature. Secure voting systems tend to be very complex due to the numerous and contrasting requirements of the system and in this way the security and verifiability aspects tend to be of paramount importance. Electronic voting systems have a number of potential benefits over their paper based counterparts including:

- Higher accuracy in the tallying phase of the election due to removal of the human factor.
- Better and more efficient coverage of remote areas as ballots can be transmitted and transferred after being cast electronically.
- It may be possible to carry out the tallying phase more quickly.
- There is the potential for a reduction in the operational costs.
- It may be possible to vote from anywhere.

Electronic voting systems can be categorised into two discrete types:

- Polling site – Casting of votes can only be made inside an official polling station. This method is similar to that of the current paper based systems where physical presence is required. Polling stations contain electronic terminals which can be used to cast votes. Voters are authenticated and authorised prior to being provided permission to cast a vote. After the vote casting stage the votes are transmitted to a central location for tallying.
- Remote voting – Voters can cast their votes from any remote location. They can even cast their votes from their own home using the internet. Authentication is carried out via the use of a set of credentials provided to each voter prior to the election. The security for remote voting is considered to be low [19]. Security of the voting terminals themselves cannot easily be enforced. Public networks are also used to communicate the votes.

Electronic voting systems alleviate some of the problems with the counting and announcing of a correct election result however instead of relying on the counters for integrity and correctness we are now relying on a smaller number of programmers who can easily change the result of the election in a way that would be hard to detect. In contrast to paper based voting systems which are widely understood and accepted by the public, electronic voting systems will be too complex for the general public to understand and they will have to trust that the system is acting honestly.

2.4.3 Existing Systems

This section will be used to briefly outline a number of existing electronic voting systems. It should be noted that when researching the following systems time was also taken to research any corresponding verifiers developed for the system. It was surprising yet somewhat expected that implementations of independent verifiers were not easily available for the systems. Helios Voting provided an additional verifier along with its source; however, this verifier was developed by the author of the election system itself and shared a number of its components. This means the results produced should be scrutinised heavily before being accepted as a true verification of the system.

2.4.3.1 Prêt à Voter

Prêt à Voter was initially created by Peter Ryan in 2004 and was designed to be an end-to-end verifiable election system providing transparency to both voters and third parties whilst still keeping the privacy requirement intact. The confidence placed in Prêt à Voter is due to the verifiability and auditability Prêt à Voter lends rather than requiring trust to be placed in any one component.

2.4.3.1.1 How it works

In Prêt à Voter each voter is supplied with a paper ballot which can be split into two halves. The left hand side (LHS) contains a randomly permuted candidate list and the right hand side (RHS) contains boxes in which voters can supply their preferences and a code of some kind which contains the permutation of the candidate list on the LHS in an encrypted form. An example ballot paper can be seen in Figure 2-1. A key concept in Prêt à Voter is that the candidates shown on the LHS of a voter's ballot are randomly permuted meaning the candidate ordering varies between ballots. Two different voters may but may not have the same candidate ordering. The code containing an encryption of the permutation of the candidate ordering can only be decrypted if a threshold number of independent trusted peers come together in agreement. Once the preferences have been written on the RHS then the candidate list on the LHS is separated and destroyed leaving only the RHS containing the voter preferences and the code. The RHS is then scanned into the system and a receipt provided to the voter which includes the preferences.

After the vote casting phase of the election the system then shuffles the votes together which performs the task of anonymising them. The shuffling removes all mappings between voter and votes. The permutation of each vote, after shuffling, is then decrypted revealing the preferences which can then be used to perform an election tally.

After all ballots have been cast the system will publish on a public web bulletin board all of the voter receipts as well as all of the decrypted votes.

2.4.3.1.2 Verifiability

The Prêt à Voter system provides verifiability mechanisms for each stage of the system the votes go through from the initial production of ballot papers to the election tally.

Voters and independent parties can carry out four key checks against the Prêt à Voter system to verify the election result:

- A number of ballot papers are audited randomly before being used by voters whereby the code representing the permutation of the candidates is decrypted and the list of candidates on the LHS is checked against this permutation.
- All cast votes are included – Any voter can use their receipt to confirm that their preferences have been included in the tally by looking up their receipt in the published list of receipts on the public web bulletin board. This check also prevents against fraud whereby it can easily be detected in receipts are not included in this list.
- Votes are counted as cast – The shuffling and decryption of votes can be independently verified. It can be confirmed that the published list of decrypted preferences corresponds to the published list of receipts. This check proves that votes are counted as they were cast.
- Verifiable tallying process – it can be verified that the decrypted voter preferences are correctly counted as part of the election tally.

Prêt à Voter achieves confidence using its verifiability rather than requiring trust to be placed in any component or in election officials [2].

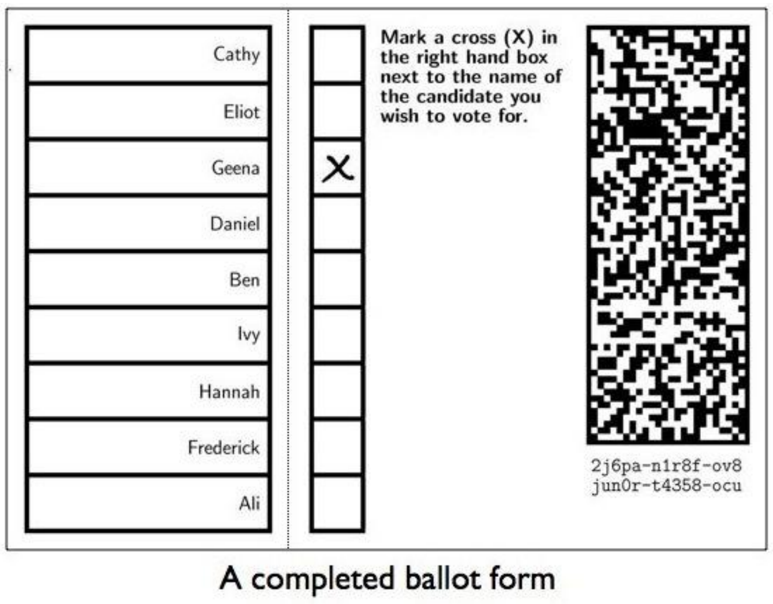


Figure 2-1 - A completed Prêt à Voter ballot form [20]

2.4.3.2 Helios voting

Helios, presented by Ben Adida in [21], is a web-based cryptographically auditable voting system which can be used to run any online election in settings where trustworthy, secret ballots are required but coercion is not a serious concern. Helios was developed specifically for low-stakes elections where coercion is not a serious issue for settings such as local clubs and student governments rather than high-stakes governmental elections where coercion can be considered a serious matter. Helios favours integrity over voter privacy.

2.4.3.2.1 How it works

The Helios election system follows closely Benaloh's Verifiable Voting Protocol detailed in [22] where the process of ballot preparation is separate from that of the vote casting stage. Voter authentication is only required at the ballot casting stage. The process of ballot preparation is split into 6 steps which are outlined using Alice as a voter and the Ballot Preparation System (BPS):

1. Alice provides an indication of which election she wishes to participate in.
2. The BPS leads Alice through the ballot questions recording her questions as she progresses.
3. Alice confirms her choices which are then encrypted, using the ElGamal cipher, and committed to by the BPS by creating a hash of the ciphertext.
4. Alice may audit the ballot which involves the BPS revealing the ciphertext and randomness used to create it. Alice can thereby verify the encryption took place correctly with her preferences. Upon auditing, the encryption will need to be re-carried out using a different randomness value.
5. After the encryption process the ballot can be sealed whereby the BPS discards randomness and plaintext information leaving the ciphertext for vote casting.

6. Alice must then authenticate herself to the system; if the authentication process is successful the encrypted vote is recorded as Alice's vote.

The full Helios protocol consists of a number of steps which will be detailed for a voter called Alice:

1. Alice follows the ballot preparation steps as above and may audit and prepare as many ballots as she wishes. Once she is satisfied with her vote she submits her encrypted vote after successfully navigating an authentication process.
2. Alice's encrypted vote is published onto a public web bulletin board allowing anyone including Alice to verify that her encrypted vote is included.
3. After the vote casting stage the encrypted votes are shuffled together also producing a non-interactive proof of the shuffling.
4. Shuffled encrypted votes are then decrypted whilst also providing a decryption proof for each. The tally is then performed on the decrypted votes.
5. The election data can be used to perform verification on the shuffling, decryptions and the tallying process.

2.4.3.2.2 Verifiability

The Helios voting system provides auditable information throughout from the ballot preparation steps and vote casting stages to the tallying process.

Helios provides the ability to carry out 5 key checks to verify the election result:

- A voter or any other party may audit a prepared ballot as many times as they wish which involves the BPS revealing the ciphertext and randomness used to create it. The voter can verify that the encryption took place correctly using the displayed randomness. An audited ballot cannot then be used for voting purposes.
- A voter or any other party can check that any specific voter's encrypted vote is included on the bulletin board displayed next to either their name or identification number. Here there is no privacy against forced abstention as an external party could apply pressure onto a voter to not cast a vote.
- In Helios the shuffling of encrypted votes takes place using the simple verifiable Sako-Kilian protocol, outlined in [23], which produces a non-interactive proof of correct shuffling.
- Helios provides proof of the decryption process for each encrypted vote using the Chaum-Pedersen protocol described in [24].
- Verifiable tallying process – it can be verified that the decrypted voter preferences are correctly counted as part of the election tally.

2.4.3.3 Scantegrity II

Scantegrity II, initially presented in [25], presents a simple novel electronic voting system through the use of confirmation codes printed on ballots using invisible ink. Voters can check that their votes have been correctly included, without revealing their preferences, using their confirmation codes which are optionally taken as a receipt.

2.4.3.3.1 How it works

Scantegrity II uses a simple voting procedure split into a number of steps:

1. An eligible voter is authenticated at the polling station and is issued with a ballot and a decoder pen.
2. The voter then chooses a single preference using the decoder pen. The decoder pen is used to mark a specific area of the ballot, termed a bubble, corresponding to the candidate choice. The ballot is cast by scanning it using an optical scanner which reads the preference but not the confirmation code.
3. A voter can optionally choose to create a receipt of their vote by writing down the confirmation code revealed in the process of marking a bubble.
4. A voter can optionally request an additional ballot for auditing. The ballot, used only for auditing, has each of its confirmation codes revealed. The voter can take this ballot paper away to be used later in a voter verification step.

2.4.3.3.2 Verifiability

Scantegrity II aims to be an end-to-end verifiable election system and provides verifiable information for each stage of the election including the use of post-election auditing of a random ballot in addition to the optional receipt created from the confirmation code revealed through the use of the decoder pen during vote casting. The method whereby votes are anonymised will also be verifiable.

2.5 Elections in the Australian State of Victoria

In the Australian state of Victoria voting is compulsory for Federal, State and local council elections, statutory elections and polls. Elections take place on a specified ‘Election Day’; however voting is also possible during an early voting process or via post. This dissertation is mainly concerned with the verifiability of an election system designed to be used in the Australian state of Victoria.

2.5.1 Victoria State Elections

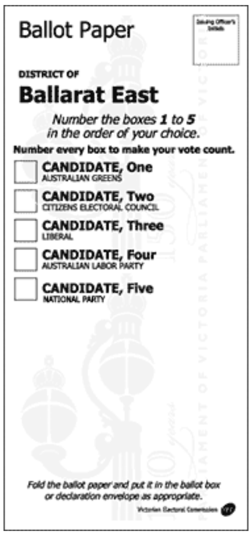
The focus of this section will be on the State elections in Victoria which are held every four years in November. In the State elections voters are selecting a single person to represent their district, which is simply a local area, in the Legislative Assembly, which is the Lower House, and five people to represent their district in the Legislative Council, which is the Upper House. The Legislative Assembly consists of 88 members, one for each district in Victoria. The party or coalition with the most seats in the Legislative Assembly, determined via elections taking place throughout Australia, forms the government. The Legislative Council consists of 40 members. The state of Victoria is divided into 8 regions with each region being represented by 5 members.

In the 2014 Victoria State election voting will also be possible using electronic assistance, for those who otherwise cannot vote, using a system called the vVote system. The vVote system is a focus of this dissertation and will be explained in detail in Chapter 3.

2.5.2 Voting in the Victoria State Elections

In the Legislative Assembly full preferential voting is employed whereby a voter enters 1 in the box of their preferred candidate, 2 in their next preferred candidate and so on for all the remaining candidates. An example ballot for the Legislative Assembly can be seen in Figure 2-2.

In the Legislative Council there are two methods whereby voters can provide their preferences: above the line and below the line. The Legislative Council ballot paper, as seen in Figure 2-3, is divided into the two corresponding sections. The top of the ballot paper contains the above the line portion of the ballot paper which includes the party names running in the race. Voters simply enter a 1 next to the group of their choice. The voter's preferences will then be counted according to the party's pre-registered candidate ordering. The bottom of the ballot paper contains the below the line portion of the ballot paper and includes all of the candidates running in the race. A voter is then required to provide at least 5 preferences against the candidates using optional preferential voting.



Ballot Paper

Issuing Officer's Initials

DISTRICT OF

Ballarat East

*Number the boxes **1** to **5** in the order of your choice.*

Number every box to make your vote count.

☐ **CANDIDATE, One** AUSTRALIAN GREENS

☐ **CANDIDATE, Two** CITIZENS ELECTORAL COUNCIL

☐ **CANDIDATE, Three** LIBERAL

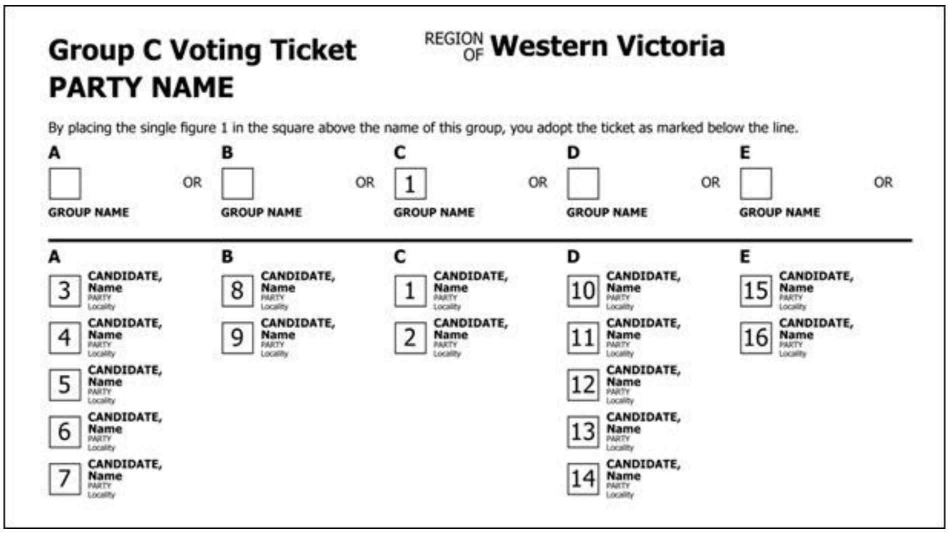
☐ **CANDIDATE, Four** AUSTRALIAN LABOR PARTY

☐ **CANDIDATE, Five** NATIONAL PARTY

Fold the ballot paper and put it in the ballot box or declaration envelope as appropriate.

Victorian Electoral Commission

Figure 2-2 - An example of a full preferential ballot paper [26]



Group C Voting Ticket
PARTY NAME

REGION OF **Western Victoria**

By placing the single figure 1 in the square above the name of this group, you adopt the ticket as marked below the line.

A		B		C		D		E	
☐	OR	☐	OR	1	OR	☐	OR	☐	OR
GROUP NAME		GROUP NAME		GROUP NAME		GROUP NAME		GROUP NAME	

A		B		C		D		E	
3	CANDIDATE, Name PARTY Locality	8	CANDIDATE, Name PARTY Locality	1	CANDIDATE, Name PARTY Locality	10	CANDIDATE, Name PARTY Locality	15	CANDIDATE, Name PARTY Locality
4	CANDIDATE, Name PARTY Locality	9	CANDIDATE, Name PARTY Locality	2	CANDIDATE, Name PARTY Locality	11	CANDIDATE, Name PARTY Locality	16	CANDIDATE, Name PARTY Locality
5	CANDIDATE, Name PARTY Locality					12	CANDIDATE, Name PARTY Locality		
6	CANDIDATE, Name PARTY Locality					13	CANDIDATE, Name PARTY Locality		
7	CANDIDATE, Name PARTY Locality					14	CANDIDATE, Name PARTY Locality		

Figure 2-3 - An example of a group voting ticket for the State Legislative Council (Upper House) [27]

2.6 Conclusion

Electronic voting is a hugely difficult problem because of the complexity of the systems required to meet their many, varied and conflicting requirements. Electronic election systems tend to provide verifiability to attempt to gain voter trust as to the correct and secure operation of the system. This dissertation is concerned with providing a verifier for the vVote system. The vVote system and the verifier itself are explained in the following chapters.

Chapter 3 – vVote System [1]

3.1 Introduction

The vVote system is an end-to-end verifiable electronic election system building upon the foundations of the Prêt à Voter election system, explained in Chapter 2, modifying it to make it usable for a Victorian Electoral Commission (VEC) election. The extensions and modifications made enable the vVote system to be applied and used in a real election and moves the underlying ideas used in Prêt à Voter out of the theoretical and into the usable.

The vVote system has been specifically tailored to handle specific traits of elections held in the Australian state of Victoria; these traits include the use of a preferential ballot (IRV) for the Legislative Assembly and a single transferable vote for the Legislative Council. The specifics of these have been described in 2.2.3 Election methods.

The vVote system will be run alongside an existing paper based ballot system and will provide an optional alternative for those who otherwise cannot vote. The use of an electronic system, which produces verifiable evidence for the actions taken throughout the system, provides a number of advantages over paper based systems including the provision for real-time electronic transfer of ballot and voting data from polling stations, including both distant domestic polling stations as well as overseas polling stations whilst still providing evidence for the data's correctness. In addition to providing real-time electronic transfer the system also provides a mechanism whereby voters can choose to cast their vote from any polling station. Tailored ballot papers will be produced for the voter's eligibility rather than requiring voters to travel to a designated polling station to then be provided with a relevant ballot paper.

The protocols underlying the vVote system are universally verifiable and do not require any trust assumptions to be made against any of its components to be able to guarantee the integrity of votes. The two most important properties of the vVote system are privacy and its end-to-end verifiability with the main principle being that the election should be completely publicly verifiable by both those taking part in the election and also interested third parties.

This chapter will provide an extension of the literature review covered in Chapter 2 and will provide comprehensive details of the overview of the vVote system. The purpose of the chapter is to provide additional details for some of the complex components of the vVote system meaning further chapters can focus specifically on the details of the vVote Verifier and the work carried out as part of the dissertation rather than the existing vVote system.

3.2 System Overview

3.2.1 Overview

Theoretical papers outlining the details of electronic voting systems can easily abstract away and provide broad assumptions under which their protocols are deemed to be secure whereas a properly implemented system must cater for and provide solutions to these assumptions to ensure the system remains secure in all working scenarios. The designed and implemented system will not require trust to be placed in it for the integrity of votes to be guaranteed and provides verifiable proofs instead.

The broad overview of the vVote system is very similar to that of Prêt à Voter where each voter is presented with a pre-generated ballot form containing of a list of randomly permuted candidates. Voters then provide their preferences using boxes adjacent to the candidate names. The list of boxes containing voter preferences is submitted to the system and also retained as a receipt whilst the candidate list is destroyed.

The major difference between vVote and Prêt à Voter is that instead of using a paper based ballot form like in Prêt à Voter, vVote uses an Electronic Ballot Marker (EBM) to present permuted candidate names to voters and is also used to enter the voters' preferences. The EBM aids in the usability of the system and assists in the proper completion of the ballot forms which in some cases may require a large number of inputs to be provided. Simply using a paper based ballot leaves a high chance of invalidating a vote. The EBM in this case must therefore be trusted for privacy and for not revealing the way in which any voter's ballot form was constructed or the preferences entered. The choice to use an EBM was made for improved usability over trusting the device for privacy and integrity because of the vast improvement in usability it gives the system.

3.2.2 Components

The vVote system utilises a number of independent communicating components for generation of ballots, printing of ballots on demand, auditing of ballots, registering votes, the cancellation of a cast vote and additionally the storage and publication of all ballots, votes, audits and other election information allowing for the entire election to remain end-to-end verifiable.

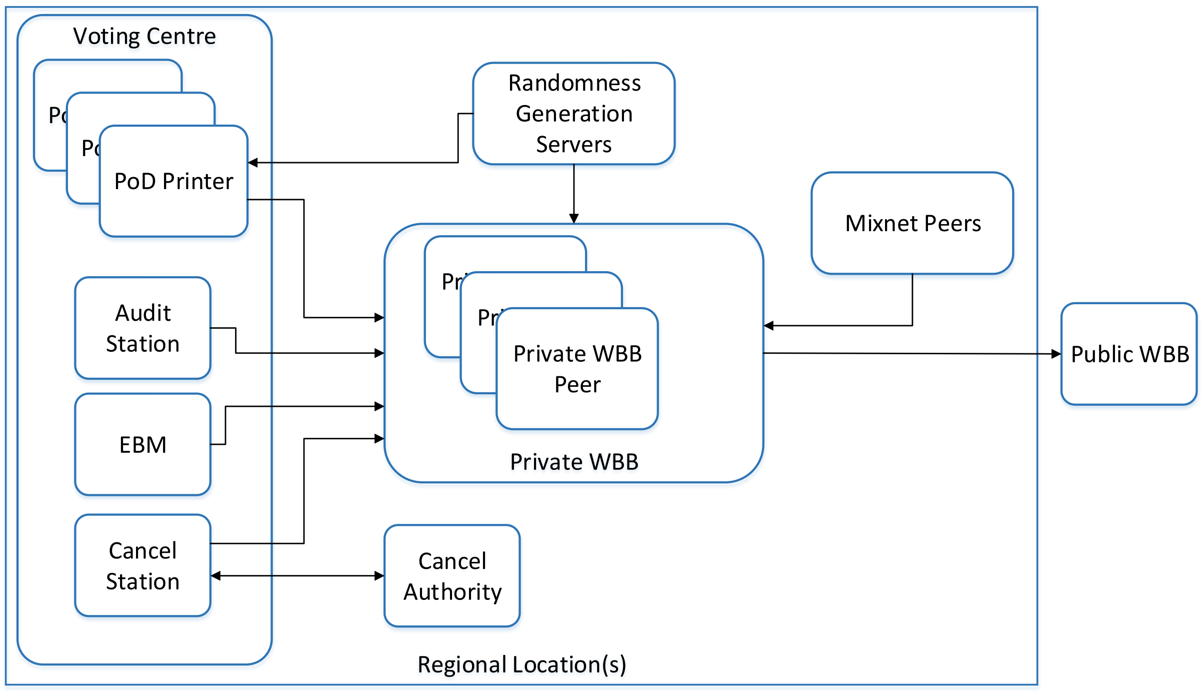


Figure 3-1 - vVote architecture [1]

3.2.2.1 Mixnet

The vVote system uses a re-encryption based Mixnet producing a non-interactive and universally verifiable proof of shuffling and decryption of the encrypted candidate identifiers sorted into preference value representing voter candidate choices. The decrypted candidate ordering and proofs are then posted to the Public WBB. The Mixnet used for the vVote system uses Randomised Partial Checking [28] with a number of modifications [29].

3.2.2.2 Public Web Bulletin Board (Public WBB)

The Public WBB's task is to hold publicly relevant information which can be used to verify certain activities have taken place correctly during the course of the election. It provides an authenticated public broadcast channel with a simple and authenticated mechanism for providing read only access to the data publicly posted. The Public WBB contains the static day-to-day public transcript for the system.

3.2.2.3 Private Web Bulletin Board (Private WBB)

The Private WBB Peers will independently receive messages, perform validity checks, perform signature verification and also sign data. It is involved in all stages from the randomness generation stage of Ballot Generation, explained in 3.5.1 Ballot Generation, up until the final commitment of data from the Mixnet to the Public WBB, explained in 3.5.5 Public WBB Commitments. The Private WBB operates in a distributed manner whereby a number of Private WBB Peers carry out the same operation and must come to a threshold consensus before any action is accepted. The Private WBB is one of the most important components in the system.

3.2.2.4 Print on Demand (PoD) Printer

The PoD Printer used in vVote is the combination of a tablet device and a printer. The tablet device may be termed the PoDTablet and is used to scan in ballots printed out as well as entering preferences. The PoD Printer combination both generates generic ballots prior to the election and prints on demand ballots to voters.

3.2.2.5 Electronic Ballot Marker (EBM)

The EBM is a device which assists voters fill in a ballot. In vVote the EBM is combined with the PoD Printer. The PoDTablet is used as the EBM in vVote.

3.2.2.6 Randomness Generation Server

A randomness generation server has the sole responsibility to produce randomness for the Ballot Generation process. The randomness generation server may be combined with the Mixnet peers.

3.2.2.7 Audit Station

The vVote system allows voters to carry out a confirmation check on the correctness of an un-voted ballot form with regard to the ordering of the candidates. This confirmation check process is explained in detail in 3.5.1.3.3 Ballot Generation Audit.

3.2.2.8 Cancel Station and Cancel Authority

The vVote system provides the ability for the supervised cancelling of a vote which has not been successfully submitted. A cancellation takes place through a Cancel Station. The Cancel Authority handles the authorisation and tracking for cancelled votes.

3.3 Election Phases

The vVote system breaks up the entire election period into three distinct stages which are **Pre-election**, which involves election setup and preparation, **Vote casting**, involving the main election activities including vote casting, audits and cancellations, and finally **Post-voting** which involves the mixing and anonymising of votes as well as the tallying of votes.

3.3.1 Pre-Election

The pre-election stage occurs before the main election takes place. This stage of the election is reserved for:

- Registration of voters and candidates.

    - The vVote system must be provided with a list of candidates running in the election as well as the number of ballots to be generated.
- Setup of voting centres, equipment and processes required for carrying out the election and announcing the election results.
    - Generation of public key pairs and digital certificates for components.
    - Generation of generic ballots with all parties and candidates listed.

3.3.2 Vote casting

The vote casting stage takes place over two phases which are the early voting phase, taking place in the two weeks prior to the Election Day, and the Election Day itself. Polling stations need to remain in an operational state for the two weeks up to the Election Day and during the Election Day itself. The process of actually casting a vote is as follows:

- A voter arrives at a polling station of their choice and confirms their name and address to a voting official.
- The voter's name will be marked showing that they have voted.
- The voter will be issued with a ballot for their voting eligibility.
- The voter casts their vote into the system and is provided with a receipt.

3.3.3 Post-Voting

After the main vote casting stage of the election the count is actually performed. Votes are anonymised by mixing votes for particular races together. After the mixing has taken place the votes are decrypted revealing the candidate preferences but not the identity of the voter. The tallying process is then carried out using an existing VEC tallying system.

3.4 End-to-end Verifiability

As described in 2.4.1 End-to-End Verifiability the end-to-end verifiability of an election system requires three core pieces of evidence. The vVote system builds upon the verifiability aspects of Prêt à Voter as follows:

- Cast-as-intended – vVote allows voters to confirm the correctness of un-voted ballots before they themselves cast a vote. This confirmed ballot cannot later be used for voting purposes:
    - Ballot generation confirmation check – each voter, after obtaining their ballot, may choose to request an audit for the ballot they are presented with. The voter will receive proof or otherwise that the ballot was correctly formed in terms of the candidate list.

  - Preference printing confirmation – each voter, after casting a vote, may check that the preferences on the receipt match those on their vote.
  - Print on Demand Confirmation check – each voter, after obtaining their ballot, can check that the printed candidate list on the ballot they receive matches the encrypted, pre-generated version stored on the Public WBB.
- Counted-as-cast verification:
  - Every voter can check that their preferences are included in a public list of all votes on the Public WBB.
- Universally verifiable tallying:
  - Anyone can check that the public list of voter preferences produces the tallied official election outcome using a public electronic proof published on the Public WBB.

One of the key aspects of vVote is the use of a printed receipt provided to voters which can be used to verify that their preferences were included in the tally as they intended without revealing their actual preferences.

3.5 Verifiable Election Stages

The following section provides in-depth descriptions of some of the main verifiable components of the vVote system including Ballot Generation, print on demand, Vote Packing, Mixnet shuffling and decryption and the Public WBB commitments. Each of the components is first described in detail as to how it functions within the vVote system and is then followed by a section detailing its verification process. Examples are provided in Appendix H showing the explicit details for the Ballot Generation and Vote Packing process.

3.5.1 Ballot Generation

3.5.1.1 Overview

Ballot Generation takes place during the pre-voting stage of the election and involves the production of randomly permuted generic ballots with each candidate being represented by an encrypted ciphertext. PoD Printers are then able to print out, in sequence, the next ballot with human readable candidate names for each voter. Each ballot produced will contain a unique set of ciphers containing the same underlying plaintexts in a permuted order. In this way it cannot be predicted in which permutation any ballot a voter will receive.

During the Ballot Generation stage a distributed set of randomness generation servers are used to generate and send randomness values to each PoD Printer. As long as at least one of the randomness generation servers remains honest the privacy security requirement will hold for all ballots generated. Each PoD Printer will receive a number of independent randomness values which are then combined and used to generate the ballots. Each generic ballot generated is then publicly published on the Public WBB.

Each of the randomness generation servers generates randomness values for each of the PoD Printers committing to the values publicly on the Public WBB. Each randomness generation server then secretly, using encryption so that the randomness values are only visible to the relevant printer, sends the randomness values to the printer where the randomness values will be used for the actual Ballot Generation.

In vVote the PoD Printers are used to carry out the expensive cryptographic operations but do not have any influence over the values used in the operations and work only on the randomness values they are provided. Ballot generation confirmation checking is used to protect against any PoD Printer attempting to mount a kleptographic attack [30], which may reveal secret information, or from having any influence over any of the generated ciphertexts for example using randomness values other than those specified in the protocol and generated by the randomness generation servers.

An auditing process is used to provide assurance that each PoD Printer has acted honestly; the auditing process relies on a sufficiently large and unpredictable fraction of the set of generic ballots being checked to provide high enough probability and unpredictability to give confidence in all ballots not checked during auditing. The unpredictable fraction of ballots to be checked is calculated using a Fiat-Shamir signature [31] which can be recalculated using publicly available information to check its authenticity.

Ballot generation includes a number of stages including:

- Pre-Ballot Generation.
- Main Ballot Generation:
    - Randomness Generation.
    - Ballot Permutation and Commitment.
- Ballot Generation Audit.

3.5.1.2 Pre-Ballot Generation

Ballot generation takes place during the pre-election phase and is one of the first steps required to take place. There are a number of activities which must take place prior to the main Ballot Generation phase; these steps are termed the pre-Ballot Generation stage:

1. The components involved in the election must jointly run a distributed key generation protocol to generate a thresholded private key SK_E and corresponding joint public key PK_E. Distributed key generation ensures that no sole party is in possession of the full private key and therefore a consensus must be achieved by a set threshold number of parties before action can be taken. Threshold cryptography is explained in detail in 2.3.4 Threshold Cryptography.
2. A list of mappings between candidate names and candidate identifiers is then generated and posted on the public WBB. The candidate identifiers are arbitrary distinct elements taken from the message space of the underlying Elliptic Curve encryption function used. An independent table, for each set of candidate identifiers, will be created for each race type in the election and therefore in the Victorian Election three tables will be produced; one for the Legislative Assembly (LA) race, Table 3-1, and two for the Legislative Council (LC) race which consists of two voting options, Above The Line (LCATL), Table 3-2, and Below The Line (LCBTL), Table 3-3.
3. For each PoD Printer used as part of the election, a list of serial numbers with the format "PrinterID:Index" is deterministically generated and posted on the Public WBB. The serial numbers are simply unique String literals and serve as row indices for computation and eventually as identifiers for the generated ballots. Example serial numbers can be seen in Table 3-4.
4. Each PoD Printer constructs a public key pair consisting of a related public key PK_P and private key SK_P. Each PoD Printer then publishes their public key on the Public WBB.

Any information to be posted publicly on the Public WBB is transferred prior to the main Ballot Generation process.

Candidate Name	**ID**
Joe Adams	*cand1*
Ben Bourne	*cand2*
…	…
Matthew May	*candn*

Table 3-1 - Legislative Assembly (LA) Candidate table

Party Name	**ID**
Labour	*cand1*
Conservative	*cand2*
…	…
Liberal Democrats	*candn*

Table 3-2 - Legislative Council (LC) Above The Line (LCATL) Candidate table

Candidate Name	**ID**
Adam Applie	*cand1*
Ben Bourne	*cand2*
…	…
Matthew May	*candn*

Table 3-3 - Legislative Council (LC) Above The Line (LCBTL) Candidate table

Serial Numbers for			
PrinterA	PrinterB	…	PrinterX
PrinterA:1	PrinterB:1	…	PrinterX:1
PrinterA:2	PrinterB:2	…	PrinterX:2
…	…	…	…
PrinterA:n	PrinterB:n	…	PrinterX:n

Table 3-4 - Example serial numbers for row indices

3.5.1.3 Main Ballot Generation

The following section outlines the details for Ballot Generation for a single PoD Printer. Each PoD Printer will carry out the same process in parallel to produce independent lists of generic ballots for use in the Print on Demand process. Each PoD Printer will interact with the same set of randomness generation servers with each server interacting with all PoD Printers involved.

3.5.1.3.1 Randomness Generation

In the randomness generation stage each randomness server $RGen_i$ produces a table RT_i containing pairs of random values consisting of a random value along with a corresponding witness value. Each pair (random, witness) is then used to produce a commitment to the random value and is included in table CRT_i. The commitments to each of the random values are publicly posted to the Public WBB after which the table of random values and corresponding witness values are sent encrypted to the PoD Printer. The tables destined for a particular PoD Printer are encrypted with their public key ensuring that only the intended PoD Printer can read them.

Table RT_i

The aim of randomness generation is for each randomness server $RGen_i$ to produce a table of randomness values for use in the generation of generic ballots. Each of the random values generated should be at least 256 bits (32 bytes) long where 32 bytes achieves an effective trade-off between security and space. For each $RGen_i$ and each secret table RT_i to construct:

- $RGen_i$ generates a random symmetric key SK_i to be used to encrypt the secret values using AES ensuring only the intended recipient can read them. The symmetric keys used are also 256 bits long.
- $RGen_i$ generates a table consisting of $b * (n + 1)$ pairs of random data with each pair comprising of a random value r and witness value R meaning each pair will be at least 512 bits (64 bytes) long. For each row of RT_i, n of the random values will be used for producing the generic ballot with the final random value being used for an additional commitment to the final permutation of the generic ballot representing the order of candidates shown.
- Each randomness pair is encrypted using the random symmetric key SK_i.

The resulting table produced RT_i is shown in Table 3-5.

Each randomness pair can be retrieved using the serial number and column index or simply using row and column indices. The first element of each pair $r_{(row,col)}$, the randomness value, will be used during the Ballot Generation phase by the recipient PoD Printer as part of a randomness combination. The second element of each pair $R_{(row,col)}$, the witness value, will be used to produce a commitment to the corresponding random value $r_{(row,col)}$ which will then be included in table CRT_i. $R_{(row,col)}$ will also be used, in the case of an audit request, to open the corresponding commitment to ensure the correct random value $r_{(row,col)}$ was used.

Serial Number	Encrypted Randomness pairs			
PrinterA:1	$SymmEnc_{SK_i}(r_{(1,1)}\|\|R_{(1,1)})$	$SymmEnc_{SK_i}(r_{(1,2)}\|\|R_{(1,2)})$	…	$SymmEnc_{SK_i}(r_{(1,n)}\|\|R_{(1,n)})$
PrinterA:2	$SymmEnc_{SK_i}(r_{(2,1)}\|\|R_{(2,1)})$	$SymmEnc_{SK_i}(r_{(2,2)}\|\|R_{(2,2)})$	…	$SymmEnc_{SK_i}(r_{(2,n)}\|\|R_{(2,n)})$
…	…	…	…	…
PrinterA:b	$SymmEnc_{SK_i}(r_{(b,1)}\|\|R_{(b,1)})$	$SymmEnc_{SK_i}(r_{(b,2)}\|\|R_{(b,2)})$	…	$SymmEnc_{SK_i}(r_{(b,n)}\|\|R_{(b,n)})$

Table 3-5 - Table RT_i, sent secretly from peer $RGen_i$ to PrinterA without publicly posting

Publicly Posted Table CRT_i

Like other components in the vVote system the generation of randomness has a built in mechanism which can be used to gain proof that the components involved have truly acted honestly. Prior to the secret sending of table RT_i to the PoD Printer each $RGen_i$ will publicly commit to the randomness values within table RT_i by producing an accompanying table which will be posted on the Public WBB. The table consists of the corresponding commitments to the randomness values $r_{(row,col)}$ using both the random value $r_{(row,col)}$ and witness value $R_{(row,col)}$. The hash-based commitment scheme used within the vVote System is described in detail in [32]. The hash-based commitment scheme simply involves hashing the concatenation of the pair of values as $\mathrm{h}(r_{(row,col)}||R_{(row,col)})$. Using this type of hash-based commitment scheme means:

- It is infeasible to find a different combination of data and witness that hash to form the same commitment value.
- The resulting hash commitment is hiding the underlying data values meaning the original random value, key to privacy in Ballot Generation, cannot be inferred from the commitment.

The resulting table $\mathrm{CRT_i}$ produced is shown in Table 3-6.

Each randomness generation sever $RGen_i$ posts each table $\mathrm{CRT_i}$ to the Public WBB and checks that all $\mathrm{CRT_i}$ tables have been posted successfully before sending the corresponding table RT_i to the intended PoD Printer. $RGen_i$ also encrypts the symmetric key SK_i used to encrypt values within the accompanying table RT_i with the corresponding PoD Printer's public key PK_P and sends the result to the PoD Printer as $esk_i = Enc_{PK_P}(SK_i, r)$.

Serial Number	Hash-based Commitment Value			
PrinterA:1	$c(r_{(1,1)}, R_{(1,1)})$	$c(r_{(1,2)}, R_{(1,2)})$	…	$c(r_{(1,n)}, R_{(1,n)})$
PrinterA:2	$c(r_{(2,1)}, R_{(2,1)})$	$c(r_{(2,2)}, R_{(2,2)})$	…	$c(r_{(2,n)}, R_{(2,n)})$
…	…	…	…	…
PrinterA:b	$c(r_{(b,1)}, R_{(b,1)})$	$c(r_{(b,2)}, R_{(b,2)})$	…	$c(r_{(b,n)}, R_{(b,n)})$

Table 3-6 – Table $\mathrm{CRT_i}$, publicly posted by peer $RGen_i$ committing to the randomness values contained in table RT_i sent to PrinterA

3.5.1.3.2 Ballot Permutation and Commitment

When the PoD Printer receives its respective table RT_i with the corresponding encrypted symmetric key esk_i it first needs to verify that the corresponding commitments to the randomness values contained within table RT_i have already been published on the Public WBB in table $\mathrm{CRT_i}$. The Public WBB needs to verify that each random value $r_{(row,col)}$ and witness value $R_{(row,col)}$ from RT_i are consistent with their corresponding commitment $\mathrm{c}(r_{(row,col)}, R_{(row,col)})$ from $\mathrm{CRT_i}$. To carry out this verification the PoD Printer must first decrypt the encrypted symmetric key esk_i recovering SK_i using $Dec_{SK_P}(esk_i)$. Using the recovered symmetric key SK_i the PoD Printer can then decrypt each pair of random values from table RT_i using $SymmDec_{SK_i}(SymmEnc_{sk}(r_{(row,col)}||R_{(row,col)}))$. The PoD Printer can then simply reconstruct $\mathrm{h}(r_{(row,col)}||R_{(row,col)})$ and check that it matches the corresponding $\mathrm{c}(r_{(row,col)}, R_{(row,col)})$ from $\mathrm{CRT_i}$. The PoD Printer is checking that the witness value $R_{(row,col)}$ opens the commitment made on the randomness value $r_{(row,col)}$ at $\mathrm{CRT_{i_{(row,col)}}}$. If the commitments are all found to be valid the PoD Printer will accept the table RT_i. If however, any of the commitments are found to be invalid the PoD Printer must challenge the randomness generation server $RGen_i$.

If the table RT_i is accepted the PoD Printer can move on to the generation of generic ballots. Each PoD Printer is provided with candidate identifiers and serial numbers shown in Table 3-1, Table 3-2, Table 3-3 and Table 3-4. Each of the candidate identifiers provided are simply elements taken from the message space of the underlying Elliptic Curve encryption function used. In the vVote System ElGamal Elliptic Curve [33] ciphertexts are used to represent the candidate identifiers and therefore they must first be transformed to this space. Initially each of the candidate identifiers are encrypted under a fixed randomness value of 1 to produce an ElGamal Elliptic Curve Point as $\mathrm{Enc}_k(cand_i, 1)$ termed the base encrypted candidate id $baseCand_i$. This fixed randomness encryption allows anyone to verify that the candidate ciphertexts are valid encryptions of the underlying candidate identifiers. For performance reasons the PoD Printer is provided with, as part of a number of other configuration files, a file containing a list of these candidate identifiers encrypted under a fixed randomness value of 1 although it would be trivial, if not somewhat computationally intensive, to recalculate these files.

G randomness generation servers will be involved in the process and will have each sent an RT_i table to the PoD Printer. In this way the PoD Printer will have received G tables each of which containing the same number of rows, corresponding to the number of ballots, and columns, corresponding to the number of candidates + 1 meaning they will all contain the same number of randomness pairs. The PoD Printer takes the first element of each pair $r_{(row,col)}$ and concatenates and hashes them together to produce a single randomness value $combinedRand_{(row,col)}$. The first n combined randomness values are taken and used to perform a re-encryption on $baseCand_i$ as $\mathrm{Enc}_k(baseCand_i, combinedRand_{(row,col)})$ which results in a different re-encryption of the same $baseCand_i$ for each ballot. The result of the re-encryption is a single list of n re-encrypted base candidate ids. The list of re-encrypted base candidates is then sorted into canonical order, by comparing the two underlying Elliptic Curve points, in a per race basis. This sorting produces a pseudo-random permutation π which is later used during Print on Demand to print the plaintext candidates in the appropriately shuffled order. The final combined randomness value is then used as a witness for a commitment made by the PoD Printer on the pseudo-random permutation π. The re-encrypted, sorted ciphertexts along with the pseudo-random permutations π are then posted on the Public WBB.

3.5.1.3.3 Ballot Generation Audit

Pre-voting Ballot Generation Audit

After the PoD Printer has publicly posted the file containing b re-encrypted, sorted ciphertexts along with their pseudo-random permutations π termed the "ciphers" file onto the Public WBB they receive a submission response from the Public WBB which includes a Fiat-Shamir heuristic [34] which is used to randomly select a percentage of generated ballots for auditing. The Fiat-Shamir signature is used as the seed for a random selection of generic ballots which should be audited for the PoD Printer. The Fiat-Shamir signature includes a number of publicly available values which allows it to be recalculated and verified. The signature includes:

- The identifier for the specific PoD Printer i.e. "PrinterA".
- The submission identifier for the message which included the ciphers file.
- The commitment time of the message.
- The actual ciphers file which contains the b generic ballots along with their permutations π.
- A threshold number of combined Boneh-Lynn-Shacham (BLS) signatures.

The PoD Printer will recalculate the Fiat-Shamir signature to verify its authenticity and will then use it to determine the serial numbers of a specific number of ballots to audit a. For each generic ballot chosen for auditing the PoD Printer will send to the Public WBB the opened randomness commitments $r_{(row,col)}, R_{(row,col)}$ from table RT_i for the specified ballot which otherwise would remain secret. Each ballot chosen for auditing cannot later be used for voting purposes and is now only available for auditing. Publicly posting the opened randomness commitments allows any party to later check that the selected ballots have been constructed honestly and that the randomness values have been combined and used appropriately in the re-encryption and sorting during the production of the generic ballot. If a large enough and unpredictable enough proportion of the generated ballots are audited, it gives confidence in the remaining un-audited generic ballots which remain private and can be used for voting anonymously. The values for the number of generic ballots produced and the number of these to audit are manually set during the pre-election phase.

The Pre-voting Ballot Generation Audit is one of the verifiable components which will be included as part of the vVote Verifier designed and implemented as part of this dissertation project explained in more detail in Chapter 4, Chapter 5 and Chapter 6.

Interactive Ballot Generation Audit – Confirmation Check

Although the Ballot Generation process is audited during the pre-voting phase of the election there is also the opportunity for voters to interactively request for an audit during the vote casting phase.

Any voter, after obtaining their ballot, may choose to request an interactive and real time audit for any ballot they are presented with called a confirmation check. The voter may repeat the ballot confirmation check as many times as they wish. The PoD Printer will print for the voter a proof that the ballot was correctly formed along with a Public WBB signature showing the proof was valid.

For any ballot chosen for auditing during the voting phase of the election the PoD Printer will submit an audit message to the Private WBB. The audit message will include the final combined randomness value which was used as a witness for a commitment made by the PoD Printer on the pseudo-random permutation π. This witness value would not otherwise be revealed and once an audit has been made against any generic ballot it cannot then be used for any other voting activities.

3.5.1.4 Verifiability

The Ballot Generation process has a number of steps which are verifiable and can be used as part of the vVote system's end-to-end verifiability. These steps include:

- The Fiat-Shamir heuristic is used to randomly select a percentage of generated ballots for auditing during the pre-voting stage of the election. This signature should be recalculated and used to verify that the correct subset of generated generic ballots was randomly selected.
- The commitments made against the randomness values for each ballot chosen for auditing should be verified by recalculating the commitments using the revealed randomness and witness value and comparing it with the corresponding commitment.
- For each ballot chosen for auditing recalculate the generic ballot and compare the resulting sorted re-encrypted base candidate identifiers.
- For each ballot chosen for auditing recalculate and check its permutation against the corresponding generic ballot.

3.5.2 Print on Demand

3.5.2.1 Overview

In the vVote system, during the Ballot Generation process, a sufficient number of generic ballots are pre-constructed. This means that during the vote casting stage of the election, running in real-time, a computationally efficient mechanism exists for providing voters with an appropriate ballot when required. This mechanism is the Print on Demand protocol. The PoD Printer will print out the next available ballot in sequence replacing the ciphertexts with human readable candidate names with only the candidates applicable for the current location.

The Print on Demand process starts as soon as a voter arrives at a polling location. A PoD Printer needs to be able to print in real-time a pre-generated ballot for the appropriate location which in a VEC election is termed a district. Varying districts may have different numbers of candidates running in each race and therefore each PoD Printer needs to be able to, upon request, tailor their generic ballots to the particular district to include only the relevant candidates.

There is a risk that a PoD Printer could print, to a voter, a completely invalid ballot which was not generated during the Ballot Generation process and could include candidates not relevant to the district, restricting voter choices or even including completely random candidates. To eliminate this risk any PoD Printer must obtain a signature from the WBB prior to printing which authenticates the ballot. The WBB provides assurance that the ciphertexts the PoD Printer is using match those previously committed to by the same PoD Printer.

3.5.2.2 Print on Demand

3.5.2.2.1 Ballot Reduction Overview

Before any election, although estimates can be made, it is not known exactly how many voters will arrive at each polling location during the election period and therefore it is unknown how many of any specialised ballot is required for any particular district. There are two primary options to overcome this issue:

1. Generate more ballots than are needed for each district which include exactly the right number of candidates for each of the districts.
2. Generate more generic ballots than are required which are then shared over all districts by the PoD Printer. Each generic ballot is then reduced down, in real-time, from the generic ballot which includes all the possible candidates to the appropriate size including only the relevant candidates running in the specific district the ballot will actually be used in.

In vVote option two was chosen for two main reasons; first the option is far more efficient in terms of the number of ballots generated and the time taken for the Ballot Generation stage of the election. The second reason option two was chosen is that it means in theory any voter can arrive at any polling location in any district and can have a ballot printed for their specific needs rather than being required to travel to a specific polling station.

For any ballot which is being printed for a district in which there are fewer candidates than in the generic ballot the ballot reduction process must be carried out. For any district with m candidates running in the district where $m \leq n$ the PoD Printer will calculate the appropriate reduced ballot and will send to the Public WBB a list of ballot reductions. The list of ballot reductions will include details for each unused candidate in the generic ballot. Three pieces of information are included in the list of ballot reductions:

- Index: This is the permuted index of the ciphertext in the specified generic ballot. This value will be between 0 and the number of candidates running in that particular race which is called the generic race size.
- Candidate Index: This is the index of the base candidate identifier that is being removed. The candidate index value ranges from the district race size + 1 to the generic race size.
- Randomness: This is the combined randomness value used for re-encrypting the base candidate identifier at “Candidate Index” to the ciphertext present in the generic ballot at “index”.

A simple example will be used to illustrate the ballot reduction process.

Race	**Candidate Identifier**	**Encryption**
LA	CandA	$\mathrm{baseCandA} = \mathrm{Enc}_{pk}(CandA, 1)$
LA	CandB	$\mathrm{baseCandB} = \mathrm{Enc}_{pk}(CandB, 1)$
LA	CandC	$\mathrm{baseCandC} = \mathrm{Enc}_{pk}(CandC, 1)$
LA	CandD	$\mathrm{baseCandD} = \mathrm{Enc}_{pk}(CandD, 1)$
LA	CandE	$\mathrm{baseCandE} = \mathrm{Enc}_{pk}(CandE, 1)$
LA	CandF	$\mathrm{baseCandF} = \mathrm{Enc}_{pk}(CandF, 1)$
LCATL	CandG	$\mathrm{baseCandG} = \mathrm{Enc}_{pk}(CandG, 1)$
LCATL	CandH	$\mathrm{baseCandH} = \mathrm{Enc}_{pk}(CandH, 1)$
LCATL	CandI	$\mathrm{baseCandI} = \mathrm{Enc}_{pk}(CandI, 1)$
LCATL	CandJ	$\mathrm{baseCandJ} = \mathrm{Enc}_{pk}(CandJ, 1)$
LCBTL	CandK	$\mathrm{baseCandK} = \mathrm{Enc}_{pk}(CandK, 1)$
LCBTL	CandL	$\mathrm{baseCandL} = \mathrm{Enc}_{pk}(CandL, 1)$

LCBTL	CandM	baseCandM $= \mathrm{Enc}_{pk}(CandM, 1)$
LCBTL	CandN	baseCandN $= \mathrm{Enc}_{pk}(CandN, 1)$
LCBTL	CandO	baseCandO $= \mathrm{Enc}_{pk}(CandO, 1)$
LCBTL	CandP	baseCandP $= \mathrm{Enc}_{pk}(CandP, 1)$
LCBTL	CandQ	baseCandQ $= \mathrm{Enc}_{pk}(CandQ, 1)$
LCBTL	CandR	baseCandR $= \mathrm{Enc}_{pk}(CandR, 1)$
LCBTL	CandS	baseCandS $= \mathrm{Enc}_{pk}(CandS, 1)$
LCBTL	CandT	baseCandT $= \mathrm{Enc}_{pk}(CandT, 1)$

Table 3-7 - Ballot reduction example - base candidate identifiers

Overall Candidate Index	**Race Candidate Index**	**Base Candidate Identifier**	**Generic Ballots**		
			Ballot:01	**Ballot:02**	**Ballot:03**
0	0	baseCandA	$\mathrm{Enc}_{pk}(baseCandF, r_{F1})$	$\mathrm{Enc}_{pk}(baseCandC, r_{C2})$	$\mathrm{Enc}_{pk}(baseCandC, r_{C3})$
1	1	baseCandB	$\mathrm{Enc}_{pk}(baseCandB, r_{B1})$	$\mathrm{Enc}_{pk}(baseCandD, r_{D2})$	$\mathrm{Enc}_{pk}(baseCandE, r_{E3})$
2	2	baseCandC	$\mathrm{Enc}_{pk}(baseCandC, r_{C1})$	$\mathrm{Enc}_{pk}(baseCandA, r_{A2})$	$\mathrm{Enc}_{pk}(baseCandD, r_{D3})$
3	3	baseCandD	$\mathrm{Enc}_{pk}(baseCandA, r_{A1})$	$\mathrm{Enc}_{pk}(baseCandE, r_{E2})$	$\mathrm{Enc}_{pk}(baseCandB, r_{B3})$
4	4	baseCandE	$\mathrm{Enc}_{pk}(baseCandE, r_{E1})$	$\mathrm{Enc}_{pk}(baseCandB, r_{B2})$	$\mathrm{Enc}_{pk}(baseCandF, r_{F3})$
5	5	baseCandF	$\mathrm{Enc}_{pk}(baseCandD, r_{D1})$	$\mathrm{Enc}_{pk}(baseCandF, r_{F2})$	$\mathrm{Enc}_{pk}(baseCandA, r_{A3})$
6	0	baseCandG	$\mathrm{Enc}_{pk}(baseCandG, r_{G1})$	$\mathrm{Enc}_{pk}(baseCandG, r_{G2})$	$\mathrm{Enc}_{pk}(baseCandH, r_{H3})$
7	1	baseCandH	$\mathrm{Enc}_{pk}(baseCandI, r_{I1})$	$\mathrm{Enc}_{pk}(baseCandI, r_{I2})$	$\mathrm{Enc}_{pk}(baseCandI, r_{I3})$
8	2	baseCandI	$\mathrm{Enc}_{pk}(baseCandH, r_{H1})$	$\mathrm{Enc}_{pk}(baseCandH, r_{H2})$	$\mathrm{Enc}_{pk}(baseCandG, r_{G3})$
9	3	baseCandJ	$\mathrm{Enc}_{pk}(baseCandJ, r_{J1})$	$\mathrm{Enc}_{pk}(baseCandJ, r_{J2})$	$\mathrm{Enc}_{pk}(baseCandJ, r_{J3})$
10	0	baseCandK	$\mathrm{Enc}_{pk}(baseCandS, r_{S1})$	$\mathrm{Enc}_{pk}(baseCandM, r_{M2})$	$\mathrm{Enc}_{pk}(baseCandO, r_{O3})$
11	1	baseCandL	$\mathrm{Enc}_{pk}(baseCandT, r_{T1})$	$\mathrm{Enc}_{pk}(baseCandL, r_{L2})$	$\mathrm{Enc}_{pk}(baseCandQ, r_{Q3})$
12	2	baseCandM	$\mathrm{Enc}_{pk}(baseCandP, r_{P1})$	$\mathrm{Enc}_{pk}(baseCandK, r_{K2})$	$\mathrm{Enc}_{pk}(baseCandR, r_{R3})$
13	3	baseCandN	$\mathrm{Enc}_{pk}(baseCandK, r_{K1})$	$\mathrm{Enc}_{pk}(baseCandN, r_{N2})$	$\mathrm{Enc}_{pk}(baseCandT, r_{T3})$
14	4	baseCandO	$\mathrm{Enc}_{pk}(baseCandL, r_{L1})$	$\mathrm{Enc}_{pk}(baseCandP, r_{P2})$	$\mathrm{Enc}_{pk}(baseCandS, r_{S3})$
15	5	baseCandP	$\mathrm{Enc}_{pk}(baseCandQ, r_{Q1})$	$\mathrm{Enc}_{pk}(baseCandR, r_{R2})$	$\mathrm{Enc}_{pk}(baseCandK, r_{K3})$
16	6	baseCandQ	$\mathrm{Enc}_{pk}(baseCandM, r_{M1})$	$\mathrm{Enc}_{pk}(baseCandQ, r_{Q2})$	$\mathrm{Enc}_{pk}(baseCandL, r_{L3})$
17	7	baseCandR	$\mathrm{Enc}_{pk}(baseCandN, r_{N1})$	$\mathrm{Enc}_{pk}(baseCandT, r_{T2})$	$\mathrm{Enc}_{pk}(baseCandP, r_{P3})$
18	8	baseCandS	$\mathrm{Enc}_{pk}(baseCandO, r_{O1})$	$\mathrm{Enc}_{pk}(baseCandS, r_{S2})$	$\mathrm{Enc}_{pk}(baseCandN, r_{N3})$
19	9	baseCandT	$\mathrm{Enc}_{pk}(baseCandR, r_{R1})$	$\mathrm{Enc}_{pk}(baseCandO, r_{O2})$	$\mathrm{Enc}_{pk}(baseCandM, r_{M3})$

Table 3-8 - Ballot reduction example - generic ballots

District Configuration			
Northcote		**Broadmeadows**	
LA	3	LA	3
LCATL	2	LCATL	3
LCBTL	6	LCBTL	7

Table 3-9 - District configurations

Table 3-8 shows a number of generated generic ballots where the generic ballot size matches Table 0-1 (6, 4, 10) shown in Appendix H meaning there are 6 candidates running in the LA race, 4 parties running in the LC ATL race and 10 candidates running in the LC BTL race. The Overall Candidate Index ranges from 0 to $n - 1$ and the Race Candidate Index ranges from 0 to the number of candidates running in each race minus 1. The three ballots generated can now be used for voting in any district where $m \leq n$.

The example will continue with the production of ballot reduction values for Ballot:01 which was printed out for the Northcote district shown in Table 3-9 with a 3,2,6 (3 LA, 2 ATL and 6 BTL) race configuration.

Race Type	**Generic Race size**	**District Race size**	**Ciphertexts to remove**	
LA	**6**	**3**	**3**	
Index	**Ciphertext at "Index" from generic ballot**	**Candidate Index**	**Base candidate identifier at "Candidate Index"**	**Randomness Value**
0	$\mathrm{Enc}_{pk}(baseCandF, r_{F1})$	5	baseCandF	r_{F1}
4	$\mathrm{Enc}_{pk}(baseCandE, r_{E1})$	4	baseCandE	r_{E1}
5	$\mathrm{Enc}_{pk}(baseCandD, r_{D1})$	3	baseCandD	r_{D1}

Table 3-10 - Ballot reduction example - LA Reductions

Race Type	**Generic Race size**	**District Race size**	**Ciphertexts to remove**	
LC ATL	**4**	**2**	**2**	
Index	**Ciphertext at "Index" from generic ballot**	**Candidate Index**	**Base candidate identifier at "Candidate Index"**	**Randomness Value**
1	$\mathrm{Enc}_{pk}(baseCandI, r_{I1})$	2	baseCandI	r_{I1}
3	$\mathrm{Enc}_{pk}(baseCandJ, r_{J1})$	3	baseCandJ	r_{J1}

Table 3-11 - Ballot reduction example - LC ATL Reductions

Race Type	**Generic Race size**	**District Race size**	**Ciphertexts to remove**	
LC BTL	**10**	**6**	**4**	
Index	**Ciphertext at "Index" from generic ballot**	**Candidate Index**	**Base candidate identifier at "Candidate Index"**	**Randomness Value**
5	$\mathrm{Enc}_{pk}(baseCandQ, r_{Q1})$	6	baseCandQ	r_{Q1}
9	$\mathrm{Enc}_{pk}(baseCandR, r_{R1})$	7	baseCandR	r_{R1}
0	$\mathrm{Enc}_{pk}(baseCandS, r_{S1})$	8	baseCandS	r_{S1}
1	$\mathrm{Enc}_{pk}(baseCandT, r_{T1})$	9	baseCandT	r_{T1}

Table 3-12 - Ballot reduction example - LC BTL Reductions

Table 3-10, Table 3-11 and Table 3-12 show the ballot reductions for the generic ballot Ballot:01 for a district with a 3,2,6 (3 LA, 2 ATL and 6 BTL) race configuration. Using the provided information the calculation of the ballot reduction process can be verified and it can be verified that the correct candidates have been removed from the generic ballot to produce the district tailored ballot. In Table 3-10 it can be seen that re-encrypting baseCandF with randomness r_{F1} results in $\mathrm{Enc}_{pk}(baseCandF, r_{F1})$.

Overall Candidate Index	Race Candidate Index	Base Candidate Identifier	Generic Ciphertext Index	Reduced Ballot:01
1	1	baseCandB	1	$\text{Enc}_{pk}(baseCandB, r_{B1})$
2	2	baseCandC	2	$\text{Enc}_{pk}(baseCandC, r_{C1})$
0	0	baseCandA	3	$\text{Enc}_{pk}(baseCandA, r_{A1})$
6	0	baseCandG	6	$\text{Enc}_{pk}(baseCandG, r_{G1})$
7	1	baseCandH	8	$\text{Enc}_{pk}(baseCandH, r_{H1})$
15	5	baseCandP	12	$\text{Enc}_{pk}(baseCandP, r_{P1})$
10	0	baseCandK	13	$\text{Enc}_{pk}(baseCandK, r_{K1})$
11	1	baseCandL	14	$\text{Enc}_{pk}(baseCandL, r_{L1})$
12	2	baseCandM	16	$\text{Enc}_{pk}(baseCandM, r_{M1})$
13	3	baseCandN	17	$\text{Enc}_{pk}(baseCandN, r_{N1})$
14	4	baseCandO	18	$\text{Enc}_{pk}(baseCandO, r_{O1})$

Table 3-13 - Ballot reduction example - Reduced ballot

Table 3-13 shows the final reduced ballot for the generic ballot Ballot:01 which would be presented to a voter with voting eligibility for a district with a 3,2,6 (3 LA, 2 ATL and 6 BTL) race configuration. The voter would not have any knowledge that originally there were more candidates in the generic ballot. The voter will only see the reduced ballot with exactly the right number of candidates listed.

3.5.2.2.2 Print on Demand Protocol

As with the other core components of the vVote system the ballot reduction and print on demand process needs to be verifiable to ensure that the reduced ballots voters are presented with are valid ballots for their voting eligibility and contain only and all of the candidates expected for the district in which they are voting. The ballot reduction method outlined produces ballots with only the candidates expected and does not need to provide any visual output or otherwise to the voter that at one point there may have been additional candidates included in a generic ballot that the reduced ballot corresponds to. To the voter it simply appears as if the ballot has been specifically generated for their district in real-time or otherwise and that the expected candidates have just been randomly permuted.

In any district there will be m candidates where $n \geq m$. Each of the generic ballots generated during Ballot Generation contain n candidates. For any printed ballot the PoD Printer will print, in a permuted order $cand_1, cand_2, \dots cand_m$.

The protocol for printing ballots on demand, after the voter arrives at the polling station, is as follows:

1. A Poll-worker authenticates the voter and sends a new print request to the PoD Printer specifying the voters voting eligibility using a district name.
2. The PoD Printer, upon receiving the print request, retrieves the next generic ballot in sequence and identifies the number of candidates.
3. The PoD Printer sends to the WBB a message containing:
   - The serial number of the generic ballot retrieved by the PoD Printer.
   - The district of the voter.
   - A list of ballot reductions, examples shown in Table 3-10, Table 3-11 and Table 3-12.
4. The WBB checks that each of the ballot reductions are valid.
   - If the list of ballot reductions are found to be valid then the WBB signs the serial number and district and sends a message in return to the PoD Printer containing this signature. In addition the ballot reductions are then posted onto the Public WBB meaning that they can be publicly verified.
   - If the list of ballot reductions are found to be invalid then an error is returned.
5. The PoD Printer verifies the WBB signature over the serial number and district and if it is found to be valid it will print the reduced ballot along with the WBB signature for the voter.
6. The voter takes the printed reduced ballot and will use it to cast their preferences.
7. The voter then shreds the left hand side (LHS) of the ballot containing the permuted candidate list.
8. The voter uses the EBM to submit their ballot preferences to the WBB.
9. The WBB accepts and signs the submitted ballot preferences only if it is accompanied by the WBB signature containing the serial number and district returned in step 4.
10. The WBB sends back the verified signature to the PoD Printer.
11. The EBM prints the WBB signature on the receipt which also contains the voter's preferences right hand side (RHS) without the permuted candidate list (LHS).
12. The voter then has an option to verify the signature which only includes data visible to the voter.
13. The voter may also later check that their vote has been included in the list of all ballots on the Public WBB using the serial number of their ballot included on their receipt.

3.5.2.2.3 Print on Demand Print Confirmation

A voter may also optionally carry out a print confirmation in which a check is made to ensure the printed candidate list on the reduced ballot matches the ciphertexts stored on the WBB after the ballot reductions have been applied to verify that the PoD Printer has not printed an invalid candidate list and the reductions have been made correctly.

During the Ballot Generation process each PoD Printer committed to the permutation of candidates for each generic ballot constructed using the final combined randomness value received from the randomness generation servers as a witness for the commitment and using the permutation itself as the random value being committed to. The commitment was then submitted to the Public WBB along with the corresponding generic ballot. Because of this commitment made to the permutation of the candidates print confirmation can be easily and computationally efficiently carried out as follows:

1. The voter requests to perform a confirmation check from the PoD Printer that originally printed their reduced ballot.
2. The PoD Printer then sends, to the WBB, the randomness pairs from each table RT_i, with the witness value being able to verify the commitments made to the randomness values in table CRT_i for the current generic ballot. The final randomness pair from each RT_i was previously combined and used to make a commitment on the permutation π.
3. In real-time the WBB will check that the provided serial number for the generic ballot has not already been used for voting purposes or used previously in a confirmation check. If the serial number is unused it will combine together the final randomness values received from the PoD Printer and use it as a witness to open the commitment made on the permutation π for the generic ballot. If the permutation check is found to be valid then the WBB creates a thresholded signature over the permutation π and sends it back to the requesting PoD Printer.
4. Later, and not in real-time, the WBB will carry out a full Ballot Generation confirmation check described in the section 3.5.1.3.3 Ballot Generation Audit.

The process above utilising both a real-time partial check and post-election full check reduces the amount of computation the WBB is required to carry out during the intensive election period.

3.5.2.3 Verifiability

The Print on Demand process has a single step which is verifiable and can be used as part of the vVote system's end-to-end verifiability. This step is the ballot reduction process. We can verify that the ballot reductions produced and contained within any PoD message for any ballot used by a voter have been created correctly and that the relevant candidates have been removed from the generic ballot to produce a district tailored ballot.

3.5.3 Vote Packing

3.5.3.1 Overview

In a typical election for the Victorian State Election in Australia there are generally around 10 parties and 38 candidates running as part of the election. As described previously in 2.5 Elections in the Australian State of Victoria the ballot form is divided into two discrete sections: Legislative Assembly (LA) which is used to submit preferences for the Legislative Assembly and Legislative Council (LC) which is further sub-divided into two more sections: Above-The-Line (ATL) and Below-The-Line (BTL) which list the parties and candidates respectively.

In the LC section voters may choose to use either the ATL or BTL part to provide their rankings. According to historical data from previous years of elections around 95% of voters choose to cast their votes using the ATL section leaving only 5% to use BTL votes. As part of the tallying process using the vVote system the voter preferences will be processed by a verifiable Mixnet. The use of a re-encryption based verifiable Mixnet can be incredibly costly due to the number of re-encryptions taking place especially when considering the large number of candidates and preferences which can be involved in a BTL vote. The vVote system was designed to be used in a real election and therefore is not just a theoretical approach and so other issues, in addition to security, have to be considered which include efficiency and usability.

Cast votes will be shuffled using the verifiable Mixnet during the vote tallying phase to provide voter anonymity whilst still providing the ability to correctly tally voter preferences together. In the vVote system the votes, prior to being shuffled, are sorted into preference order. A simple and naïve approach, shown in Figure 3-2, of producing a set of ciphertexts in preference order would be to assign a ciphertext to each candidate and then use the voter's ranking to permute the ciphertexts.

Candidate Name	Preference
Alice	2 $\mathrm{Enc}_k(A, r1)$
Bob	4 $\mathrm{Enc}_k(B, r2)$
Charlie	5 $\mathrm{Enc}_k(C, r3)$
David	3 $\mathrm{Enc}_k(D, r4)$
Eve	1 $\mathrm{Enc}_k(E, r5)$

$\longrightarrow$

Preference
2 $\mathrm{Enc}_k(A, r1)$
4 $\mathrm{Enc}_k(B, r2)$
5 $\mathrm{Enc}_k(C, r3)$
3 $\mathrm{Enc}_k(D, r4)$
1 $\mathrm{Enc}_k(E, r5)$

$\longrightarrow \{\mathrm{Enc}_k(E, r5), \mathrm{Enc}_k(A, r1), \mathrm{Enc}_k(D, r4), \mathrm{Enc}_k(B, r2), \mathrm{Enc}_k(C, r3)\}$

Figure 3-2 - Random permutation created by assigning a ciphertext to each candidate id and sorting using the voter preference

It can be seen however that for the example shown in Figure 3-2, where five candidates are running in the election, there will be a five-ciphertext tuple entered into the Mixnet. In addition to the Mixnet needing to shuffle the five-ciphertext tuple it will also need to decrypt each of the ciphertexts post shuffle. When carrying out these tasks using a verifiable Mixnet the time taken can increase dramatically as the number of candidates is increased. As mentioned previously generally around 38 candidates will be running in the election. The shuffling and decryption of a 38-ciphertext tuple will take a considerable amount of time.

In the vVote system a method of packing together multiple candidate ciphertexts before being passed to the Mixnet is used meaning far fewer ciphertexts need to be shuffled and decrypted producing significant efficiency benefits.

3.5.3.2 Vote Packing

The theory behind the Vote Packing technique is to pack together multiple ciphertexts into a single ciphertext before passing the ciphertexts to the Mixnet meaning far fewer ciphertexts are sent to the Mixnet which reduces the number of shuffles and decryptions being carried out. Vote packing relies on the use of the additive homomorphic property of exponential ElGamal encryption described in 2.3.2 Homomorphic Encryption. Even though the ciphertexts are packed together the voter’s candidate ranking can be retrieved using a meet-in-the-middle algorithm despite the discrete logarithm problem.

3.5.3.2.1 Vote Packing Pre-requisites

Vote packing occurs in the post-election phase of the election after voters have submitted their preferences for their randomly permuted reduced ballot to the Public WBB. All generic ballots have also previously been submitted to the Public WBB.

Each of the preferences and their generic ballots are looked up and divided into appropriate races; they are then sorted into preference order meaning the ciphertext that aligns with the first preference is the first ciphertext in the sorted list. Votes where s partial preferences have been provided, where s is less than the number of candidates running in the race, will only include ciphertexts from preference 1 to s.

ATL votes are not packed as only a single preference is provided which is an indication of the party the voter has nominated to vote for using the parties pre-determined candidate order. LA and BTL votes are preference votes and will be packed. LA votes can include a full partial preference whereas BTL votes must provide at least 5 preferences for the candidates running in the race.

District Configuration = Northcote (see Table 3-9) (3 LA,2 LCATL,6 LCBTL)				
LA Race	**LC ATL Race**	**LC BTL Race**	**Is valid**	**ATL or BTL**
1,2,3	1,	,,,,,	True	ATL
1,,	1,	,,,,,	True	ATL

1,,2	1,	,,,,,	True	ATL
3,2,1	1,	,,,,,	True	ATL
1,3,2	,1	,,,,,	True	ATL
1,,2	,1	,,,,,	True	ATL
1,2,3	,	1,2,3,4,5,6	True	BTL
1,,	,	1,2,3,4,5,6	True	BTL
1,3,2	,	4,5,3,2,1,6	True	BTL
,,1	,	1,2,3,4,5,6	True	BTL
,,	,	,,,,,	False – No preferences	-
1,,	1,	1,2,3,4,5,6	False – Both ATL and BTL preferences supplied	-
1,,	1,2	,,,,,	False – Only allowed single preference for ATL	-
1,,	,	1,,,,,	False – At least five preferences are required for BTL votes	-

Table 3-14 - Example race preferences pre-sorting

3.5.3.2.2 Vote Packing Pre-Mixnet

The packing size used in the vVote System is configurable for an election. Here the variable p will be used to represent the packing size. Generally the packing size used will be between 3 and 6 with the larger the packing size the greater the efficiency increase as fewer packed ciphertexts, which include larger numbers of candidate ciphertexts, will be passed to the Mixnet for shuffling and decryption. Using a packing size of p means up to p ciphertexts can be packed together into a single ciphertext.

Vote Packing Technique

To perform Vote Packing the first p ciphertexts are taken and raised to their index from 1 to p where once p is reached the index is reset to 1 again. The ciphertexts have already been sorted into preference order meaning the candidate index represents the preference -1. The process is shown in Table 3-15.

This example uses a packing size of 3			
Candidate Index	**Packing Preference**	**Candidate ciphertext**	**Candidate ciphertext raised to Packing Preference**
0	1	CandA	CandA^1
1	2	CandB	CandB^2
2	3	CandC	CandC^3
3	1	CandD	CandD^1
4	2	CandE	CandE^2

5	3	CandF	CandF^{3}
6	1	CandG	CandG^{1}
7	2	CandH	CandH^{2}

Table 3-15 - Simple Vote Packing example 1

After the ciphertexts have been raised to the power of their packing preference each set of p ciphertexts are added together to form a single ciphertext from up to p ciphertexts as shown in Table 3-16.

Candidate Index	0	1	2	3	4	5	6	7
Packing Preference	1	2	3	1	2	3	1	2
Candidate Ciphertext	CandA	CandB	CandC	CandD	CandE	CandF	CandG	CandH
Candidate ciphertext raised to Packing Preference	CandA^{1}	CandB^{2}	CandC^{3}	CandD^{1}	CandE^{2}	CandF^{3}	CandG^{1}	CandH^{2}
Packed Ciphertexts	$\mathrm{Packing1} = \mathrm{CandA}^{1} + \mathrm{CandB}^{2} + \mathrm{CandC}^{3}$			$\mathrm{Packing2} = \mathrm{CandD}^{1} + \mathrm{CandE}^{2} + \mathrm{CandF}^{3}$			$\mathrm{Packing3} = \mathrm{CandG}^{1} + \mathrm{CandH}^{2}$	

Table 3-16 - Simple Vote Packing example 2

Table 3-16 shows how using the Vote Packing technique the 8 original ciphertexts have been reduced to just three using a simple method. These ciphertexts can be kept in order and simply put through the Mixnet in the same way as if they were unpacked ciphertexts.

In the vVote System ElGamal Elliptic Curve points consisting of two Elliptic Curve points to represent ciphertexts and single Elliptic Curve points are used to represent plaintexts. In the case of plaintexts instead of raising the plaintext to the power of the packing preference from 1 to p the point is instead multiplied by the value.

Candidate Index	0	1	2	3	4	5	6	7
Packing Preference	1	2	3	1	2	3	1	2
Candidate plaintext	plainA	plainB	plainC	plainD	plainE	plainF	plainG	plainH
Candidate plaintext multiplied by the Packing Preference	$\mathrm{plainA} * 1$	$\mathrm{plainB} * 2$	$\mathrm{plainC} * 3$	$\mathrm{plainD} * 1$	$\mathrm{plainE} * 2$	$\mathrm{plainF} * 3$	$\mathrm{plainG} * 1$	$\mathrm{plainH} * 2$
Packed Plaintexts	$\mathrm{Packing1} = \mathrm{plainA} * 1 + \mathrm{plainB} * 2 + \mathrm{plainC} * 3$			$\mathrm{Packing2} = \mathrm{plainD} * 1 + \mathrm{plainE} * 2 + \mathrm{plainF} * 3$			$\mathrm{Packing3} = \mathrm{plainG} * 1 + \mathrm{plainH} * 2$	

Table 3-17 - Plaintext Vote Packing example

In the case of ciphertexts the process is again slightly different. Each ElGamal Elliptic Curve Point is made up of two Elliptic Curve points: $[alpha, beta]$. Each of the points is multiplied by the packing preference from 1 to p. The ciphertexts are then added together; each of the alpha values are added together and each of the beta values are added together. The result of this is the ElGamal Elliptic Curve Point packed value.

Candidate Index	0	1	2	3	4	5	6	7
Packing Preference	1	2	3	1	2	3	1	2
Candidate Ciphertext	$[A_{alpha}, A_{beta}]$	$[B_{alpha}, B_{beta}]$	$[C_{alpha}, C_{beta}]$	$[D_{alpha}, D_{beta}]$	$[E_{alpha}, E_{beta}]$	$[F_{alpha}, F_{beta}]$	$[G_{alpha}, G_{beta}]$	$[H_{alpha}, H_{beta}]$
Candidate ciphertext multiplied by the Packing Preference	$[A_{alpha} * 1, A_{beta} * 1]$	$[B_{alpha} * 2, B_{beta} * 2]$	$[C_{alpha} * 3, C_{beta} * 3]$	$[D_{alpha} * 1, D_{beta} * 1]$	$[E_{alpha} * 2, E_{beta} * 2]$	$[F_{alpha} * 3, F_{beta} * 3]$	$[G_{alpha} * 1, G_{beta} * 1]$	$[H_{alpha} * 2, H_{beta} * 2]$
Packed $alpha$	$Packing1_{alpha} = A_{alpha} * 1 + B_{alpha} * 2 + C_{alpha} * 3$			$Packing2_{alpha} = D_{alpha} * 1 + E_{alpha} * 2 + F_{alpha} * 3$			$Packing3_{alpha} = G_{alpha} * 1 + H_{alpha} * 2$	
Packed $beta$	$Packing1_{beta} = A_{beta} * 1 + B_{beta} * 2 + C_{beta} * 3$			$Packing2_{beta} = D_{beta} * 1 + E_{beta} * 2 + F_{beta} * 3$			$Packing3_{beta} = G_{beta} * 1 + H_{beta} * 2$	
Packed Ciphertexts	$[Packing1_{alpha}, Packing1_{beta}]$			$[Packing2_{alpha}, Packing2_{beta}]$			$[Packing3_{alpha}, Packing3_{beta}]$	

Table 3-18 - Ciphertext Vote Packing example

3.5.3.2.3 Vote Packing Post-Mixnet

The packed ciphertexts are completely normal ciphertexts and can be passed to the Mixnet as normal for shuffling and decryption. When the shuffled ciphertexts have been decrypted the result will be a packed plaintext value. In the vVote system a lookup table will be used to determine the underlying ciphertexts from the packed value. Three lookup tables will be used, one for each race containing every possible packing value calculated from the combination of p ciphers in every possible permutation.

Candidates: candA, candB, candC		Packing size = 3
Count	Permutation	Packing example
1	candA	candA^{1}
2	candB	candB^{1}
3	candC	candC^{1}
4	candA, candB	candA^{1} + candB^{2}
5	candA, candC	candA^{1} + candC^{2}
6	candA, candB, candC	candA^{1} + candB^{2}+ candC^{3}

7	candA, candC, candB	$candA^1+candC^2+candB^3$
8	candB, candA	$candB^1+candA^2$
9	candB, candC	$candB^1+candC^2$
10	candB, candA, candC	$candB^1+candA^2+candC^3$
11	candB, candC, candA	$candB^1+candC^2+candA^3$
12	candC, candA	$candC^1+candA^2$
13	candC, candB	$candC^1+candB^2$
14	candC, candA, candB	$candC^1+candA^2+candC^3$
15	candC, candB, candA	$candC^1+candB^2+candA^3$

Table 3-19 - Candidate lookup table example

Table 3-19 provides an example of a simple lookup table which contains every possible combination of 3 candidates in every possible permutation. It can be seen that with only three candidates provided the number of permutations is 15 and that as the number of candidates increases so too does the size of the candidate lookup table. The size of the candidate lookup table quickly increases and can reach the gigabyte size very easily. The candidate lookup tables will be pre-generated prior to the election taking place after the candidate identifiers have been generated.

Using the looked up values the candidate names can be mapped back and therefore the voter preferences determined. It should also be noted that from the second packed ciphertext onwards the preference values are offset by a multiple of the packing size p and therefore the preference of the first candidate returned from the lookup from the decrypted second ciphertext packing is $1 + p$ and the first candidate returned from the lookup from the decrypted third ciphertext packing is $1 + 2p$ etc.

3.5.3.3 Verifiability

The Vote Packing process has a number of steps which are verifiable and can be used as part of the vVote system's end-to-end verifiability. These steps include:

1. A check to ensure that the ciphertexts provided to the Mixnet have been correctly packed according to the set of underlying voter preferences and that the packings match those that were input to the Mixnet.
2. A check to ensure that the decryptions from the Mixnet, corresponding to packed plaintexts, correctly match the packing of the plaintext candidate identifiers using the claimed preferences from the Mixnet output. This step verifies that the values match what would have been looked up to determine the voter preferences and avoids having to perform a full recalculation of the lookup tables requiring only the specific entries (those actually used in the lookup) to be recalculated. This check relies upon the homomorphic encryption property of ElGamal Elliptic Curve encryption.

These steps correspond to the pre-Mixnet Vote Packing and post-Mixnet Vote Packing steps which will be explained in more detail in Chapter 4 and Chapter 5.

3.5.4 Mixnet

3.5.3.2.2 Vote Packing Pre-Mixnet outlines the method whereby a number of re-encrypted candidate identifiers are packed together to form ciphertext packings ready to be passed to a Mixnet. The Mixnet requires that all inputs packed at the same time are the same length. Each set of voter preferences for each race are passed to the Mixnet together meaning that all preferences relating to the LA race for a particular district are shuffled and decrypted together; equally the same happens with the LCATL and LCBTL votes. If for the LA or LCBTL race the packing size used is 3 and a voter provided 10 preferences whereas every other voter provided 8 preferences then each of the packed ciphertexts for the 8 preferences would need to be padded to the same length as that of the 10 preferences vote. Table 3-20 shows the required padding for the example provided. The value of the padding used is simply a fixed value agreed and published in advance of the election.

Packing size = 3		
Number of preferences	**Size of the set of packed ciphertexts**	**Required Padding**
10	4	0
8	3	1

Table 3-20 - Require padding for input to the Mixnet

Once each of the sets of votes have been packed and padded to the same length they can be input to the Mixnet. The Mixnet carries out a row-wise shuffling of the votes meaning that if $CipherA - \{1-6\}, CipherB - \{7-12\}, CipherC - \{13-15\}$ is submitted to the Mixnet the vote will be shuffled with other rows.

The Mixnet, after carrying out shuffling, decrypts the ciphertexts provided to it to produce packed plaintext values which can then be looked up as outlined in 3.5.3.2.3 Vote Packing Post-Mixnet. In the vVote system the ciphertexts are Elliptic Curve points and therefore a ciphertext can be homomorphically re-encrypted using a second randomness value (or third or fourth etc.). The decryption of an Elliptic Curve ciphertext will strip both randomness values and produce the original plaintext value.

$$\mathrm{Dec}_{SK}(\mathrm{Enc}_{PK}(\mathrm{Enc}_{PK}(plaintext, r1), r2)) = plaintext$$

The exact workings of the verifiable Mixnet are beyond the scope of this dissertation. It was decided that due to time constraints the Mixnet would not be chosen to be included as one of the verifiable components studied. The Mixnet implementation chosen for use in the vVote system had still not been completed at the time of writing this document. With the code in flux as well as the lack of documentation for its theoretical details it was decided that the verification of the Mixnet would not be carried out. The Mixnet was also not one of the University of Surrey deliverables. Additional details regarding this decision are included in the feasibility study within Chapter 4.

3.5.5 Public WBB Commitments

An important component of the vVote system for remaining end-to-end verifiable is the use of the Public WBB which contains the static day-to-day public transcript for the system. The public WBB is an authenticated public broadcast channel with memory with the purpose of holding publicly relevant information which can be used to verify certain activities have taken place correctly. It is simply updated once per day as separate commitments. Its functionality is relatively straightforward compared to many of the other vVote components.

Each commitment made to the Public WBB is structured the same way and consists of:

- A JSON file containing a days' worth of publicly verifiable information including commitments made by the randomness generation servers, Ballot Generation commitments, ballot audit commitments, vote messages and PoD messages. These commitments will be in the form of JSON messages included on different lines of the file.
- An accompanying ZIP file containing any additional data corresponding to a specific message within the JSON file. Each ZIP file will include an additional inner ZIP file for each set of additional data per message within the JSON file. If any of the JSON messages contained in the JSON file are a type of file message then they will have a corresponding set of data within an inner ZIP file.
- A signature JSON file containing a signed hash of the important protocol information from both the JSON and ZIP files and a key as to which JSON and ZIP file the signature corresponds to. The signing will be carried out by the Public WBB and will serve as its authenticity and can be used to verify the commitment.

The vVote system will allow for end voters to use an index file, consisting of all ballot serial numbers and their corresponding commitment, to look up and download a relevant commitment. The voter may then verify that any specific data item is present including their voting preferences corresponding to their receipt. The voter may also re-check the computed hash and verify the Public WBB signature made on the commit.

The main property of the Public WBB is that even if it has been compromised it cannot generate falsified data in a way that is undetectable.

3.5.5.1 Verifiability

The Public WBB commit process has a single step which is verifiable and can be used as part of the vVote system's end-to-end verifiability. The verification process is to recalculate the hash of the data, and to verify that the hashed data is the same data that was signed by the Public WBB to produce the signature contained in the Signature JSON file using the Public WBB's public key. The verification of the commitments posted onto the Public WBB will be explained in more detail in Chapter 4 and Chapter 5.

3.6 Conclusion

This chapter has provided an overview of the vVote system. The chapter has provided extensive additional details of the components making up the system including their communications and specific functionality. This chapter has also highlighted the main components of the vVote system requiring verification which will be included and worked on as part of this dissertation. The verifier components are the Pre-voting Ballot Generation Audit, Pre-Mixnet Vote Packing and Post-Mixnet Vote Packing and also verification of the commits made to the Public WBB.

Chapter 4 – vVote Verifier Analysis

4.1 Introduction

In this chapter the problem analysis and requirements derivation will be described. The chapter will use the literature review as well as the vVote overview provided in Chapter 3 to set out the formal requirements for a verifier for the vVote system including both its functional and non-functional requirements. The chapter will also include the inputs to the verifier and the way in which the software developed will carry out the cryptographic proofs required by the protocols underlying the vVote system.

The problem analysis chapter will form the foundation for the design and implementation stages of the project and will have a large bearing on the direction in which the development progresses as well as its successfulness.

4.2 Feasibility study

The analysis stage of any project should be accompanied by an initial feasibility study to ensure that which is proposed is viable for the timeframe and in terms of an MSc dissertation is at the right level of complexity. The feasibility study undertaken considered the areas of verifiability within the vVote system and the way in which these aspects of the system were to be verified. This study heavily utilised the vVote system's accompanying documentation which explicitly set out a number of areas which should be verified after the election to improve the trust and confidence in the election result.

Within the vVote system codebase itself, a number of built-in verification tools have been produced to improve the confidence in a number of the smaller modules and how they carry out their specific functionality. Specifically there has been a verifier developed for the Ballot Generation process as well as the building of the Vote Packing tables which are used to lookup decrypted packed values to determine voter preferences after the encrypted packed votes have been passed through the Mixnet. The problem with both of these built-in verifiers in terms of the confidence they provide is that they share significant portions of their code with the system under test and that they are developed by those developing the rest of the system. The idea behind the vVote Verifier is that it should be a stand-alone independent reference implementation which therefore meant a number of aspects of the original codebase would need to be re-implemented to ensure the stand-alone verifier is completely independent from the vVote system. A number of exceptions would be made in terms of the code being completely independent including the use of a number of third party libraries including the org.json library [35] used for reading and manipulating JSON objects as well as the use of the JAVA implementation of the Bouncy Castle Cryptographic Library [36]. Bouncy Castle was deemed to be independent of the vVote system; it is also widely used and scrutinised meaning the risk of the library being used to subvert some specific security requirement of the vVote system whilst also remaining undetected through the use of the verifier was deemed to be acceptable. A significant amount of the back end codebase from the vVote system would need to be re-implemented to ensure complete isolation from the original code. The main areas needing reimplementation were the messages, which are utilised heavily within the vVote system to carry information between components and peers from the same component, as well a number of cryptographic utilities to carry out vVote proofs. These two areas both seemed reasonable.

The accompanying documentation outlines a number of aspects of the vVote code which could be included in the verification including Ballot Generation, Vote Packing, Mixnet and Public WBB commits. Originally the aim was to carry out verification on all these components, however, after carrying out a feasibility study; the verification of the Mixnet, in addition to the other components was deemed to be infeasible in the time frame required both in terms of understanding the theoretical complexities and also producing a large amount of additional code to carry out the verification. The documentation, code and support for additionally working on the verification of the Mixnet was not deemed to be readily available. It was decided that verification of the Ballot Generation process, Vote Packing process and Public WBB commitments was sufficient for the requirements of the MSc dissertation.

The verifier tool developed also needed to be platform independent to ensure wide availability and that the platform required for running the tool does not stop any voter, or otherwise, wishing to undertake verification of the election from doing so. The JAVA programming language allows for this platform independence and a simple JAR file (JAVA Archive file) will be produced with a simple and easy user guide for running the tool requiring only the path of the directory containing the Public WBB data.

During the development of the verifier tool, and after the conclusion of the development, changes to the vVote system, whilst occurring infrequently during the development of the verifier, may require changes and maintenance to be carried out on the verifier due to the way it directly utilises the output from the system. If the vVote system's design changes in terms of how the output is formed the verifier will also need to be modified.

4.3 Requirements Analysis

After undertaking the feasibility study and problem analysis a number of components of the vVote code were identified for verification with the next stage being a full requirements analysis of those components. After analysing the vVote system in terms of its verifiable components within Chapter 3 some of the requirements were easy to deduce as they are specified explicitly in the documentation however the methodology used to actually meet these requirements as well as breaking the broad requirements down into manageable and more meaningful requirements required a significant amount of additional work.

There was no additional documentation relating specifically to the verification of the vVote system providing details as to how the verification was meant to take place in terms of the specific steps required for fully verifying that key parts of the protocols had taken place correctly. For this, numerous discussions took place to completely decide on the explicit details of the verification. Certain design decisions were completely missing from the documentation and analysis of the source code was required to verify implementation details which ensure the security requirements of the protocols remained sound.

The proposed verification tool utilises a combination of a number of component based verifiers each with their own fundamental requirements and goals.

- Ballot generation verifier – The documentation relating to the Ballot Generation audit provides explicit steps required for validating that the generation of ballots took place correctly. The Ballot Generation verifier aims to verify that the ballots chosen for auditing were correctly constructed. First the set of ballots randomly chosen for auditing must be verified by recalculating a Fiat-Shamir signature and checking that using this signature produces the same set of ballots. Next the commitments against the pairs of randomness values received from each of the randomness generation servers must be validated. A check is made that the witness from each pair opens the commitment made to the randomness value on the Public WBB. The validated randomness values are then combined together. Each of the combined randomness values is then used to re-encrypt the corresponding base candidate identifier. The list of re-encrypted base candidate identifiers is then sorted into canonical order in a per race basis with a permutation representing the order of the candidates also being committed to on the Public WBB.

- Vote packing – The documentation provided regarding Vote Packing was minimal and required a large number of discussions to determine exactly how the process was carried out and the exact way in which verification of the process should take place. It was determined that two checks would be made:
    i. A check to validate that the ciphertexts provided to the Mixnet input have been correctly packed according to the preferences provided for each reduced ballot.
    ii. A check to determine that the decryptions from the Mixnet correctly match the packing of the plaintext candidate ids using the claimed preferences.
- Commits made to the Public WBB – The documentation again does not provide explicit details as to how a hash is generated over the data provided to the Public WBB which is then signed by the Public WBB. Discussions and code analysis were the main sources for identifying the requirements. The check is simply that the signature on the hash over the data provided to the Public WBB matches what is included in a signature message for each commit.

The main aim of the project was to provide a stand-alone independent reference implementation of a verifier for the vVote system. An important requirement therefore is that the code is cleanly written with ample comments and written in a user friendly manner meaning analysis of the code can be carried out with relative ease. The implemented verifier should also be accompanied by documentation outlining the way in which the component verifiers function. The performance of the verifier is of little importance and therefore readability of the code is paramount. The code should also have a separated backend allowing the code to easily be run through a website for example offered via a trusted party for voters to verify the election more easily than downloading the verifier itself. The current progress of the verification should also be shown in real-time to an end user.

4.3.1 Inputs to the Verifier

The vVote Verifier is designed to be run after the election takes place and utilises the publicly committed to data created during the course of the election by various components recording their public activity. The data generated should provide evidence as to the correct behaviour of the system in terms of its underlying protocols. Table 4-1 shows all of the inputs important to the correct working of the vVote Verifier system.

Filename/	Description

Directory name	
/final_commits	Contains 0 or more commits made to the Public WBB. Each commit consists of: - A JSON message containing a days' worth of publicly verifiable information including commitments made by the randomness generation servers, Ballot Generation commitments, ballot audit commitments, vote messages and PoD messages. The JSON message will have the commit time in its name as commit_time.json. These commitments will be in the form of JSON messages included on different lines of the file. - An accompanying ZIP file with name commit_time_attachments.zip containing any additional data corresponding to any specific message within the JSON file which requires additional data provided. Each ZIP file will include an additional inner ZIP file for each set of additional data per relevant file message within the JSON file. - A signature JSON file with name commit_time_signature.json containing a signed hash of the important protocol information from both the JSON and ZIP files and a key as to which JSON and ZIP file the signature corresponds to.
ballot_gen_conf.json	Contains election specific information including the number of candidates running in each race, the number of ballots selected for auditing and generation for each PoD Printer.
base_encrypted_ids.json	Contains the base encrypted candidate ids. Each of the plaintext candidate ids from plaintexts_ids.json is encrypted using a fixed randomness value of 1 to produce this file.
certs.bks	Contains the public key and partial public key entries for each Private WBB peer and the public key entry for the Public WBB.
districtconf.json	Contains specific district race configuration data specifying the number of candidates running in each race for every district involved in the election.
plaintexts_ids.json	Contains the plaintext candidate identifiers for all candidates running in all races in the election.
publickey.json	Contains the election public key.
paddingpoint.json	Contains the padding used during the Vote Packing process. The padding is simply a fixed value agreed and published in advance of the election.
map.properties	Contains a number of configuration details for the Vote Packing process including the name of the Elliptic Curve to use, the padding point filename, the size of LA race packing, the size of the BTL race packing and a field detailing the races which will not be packed.
/mix/IN	Contains 0 or more .blt files which are input files ready for the Mixnet. Each of the .blt files contain packed ciphertexts ready for shuffling using the Mixnet. Each .blt file is in ASN.1 file format.
/mix/OUT	Contains a large number of different ASN.1 files as well as CSV files. Within the vVote Verifier the only files we are interested in are CSV files and .out files. The CSV files contain the output shuffled voter preferences for a specific race configuration which is included in the name of the file. The .out files contain packed plaintext candidate identifiers again for a specific race configuration

	specified in the file name.
mixrandomcommit message Attachment commitData.json	The mixrandomcommit message corresponds to a specific randomness generation server as well as a specific PoD Printer. The commitData.json file contains the commitments made against the randomness values sent to the same PoD Printer for Ballot Generation. Each line of the file corresponds to a different ballot with the serial number included.
ballotgencommit message Attachment ciphers.json	The ballotgencommit message corresponds to a specific PoD Printer. The ciphers.json file contains the generic ballots generated by the PoD Printer. Each line contains the serial number for the generic ballot, the permutation representing the random ordering of the candidates and the re-encrypted base encrypted candidate identifiers.
ballotauditcommit Attachment AuditDataFile.json	The ballotauditcommit message corresponds to a specific PoD Printer. The AuditDataFile.json file contains the ballots chosen for auditing from the list of ballots generated by the PoD Printer. Each line of file corresponds to a specific ballot chosen for auditing along with its opened and previously secret randomness pairs.
ballotauditcommit Attachment BallotSubmitResponse.json	The ballotauditcommit message corresponds to a specific PoD Printer. The BallotSubmitResponse.json file contains verifiable information relating to the Ballot Generation audit data for the PoD Printer including the Fiat-Shamir signature as well as partial WBB signatures.
race_map.json	Contains the mappings between race identifiers, included in specific filenames contained in the mix/IN and mix/OUT folders, and their districts and race configurations.

Table 4-1 - Inputs to the vVote Verifier system

As the system functions solely on the data provided to it and determines whether the vVote system has functioned correctly during the election a requirement for the tool is that the data can be successfully downloaded from the Public WBB and provided to the verifier in a complete state. Each of the verifiers again has contrasting requirements as to the data needed for successfully running the verifier.

4.3.2 Messages

The vVote system heavily relies on the use of JSON messages throughout whereby the messages are used for communication between peers and components. They are used for transporting the majority of the information throughout the system both within the system internally, from peer to peer for example between Private WBB peers and for sending messages externally for example to PoD printers or to the Public WBB. The messages are JSON formed and have a simple inheritance hierarchy. The verifier will only need to work with external messages used to send information publicly as these are the only messages which will be externally available after the election has run. The system will need to be able to read in the JSON messages, identifying and creating the correct type of JSON message on-the-fly as well as utilising their included information.

4.3.3 Data files

The vVote system uses messages for communication both internally and externally and can contain small pieces of information however for larger portions of data additional data files will be used and referenced by the messages. A substantial amount of data will be communicated by the vVote system and any publicly important information will also be stored on the Public WBB. The verifier will need to be able to utilise both the messages and the referenced data files themselves to produce the appropriate context. Examples of such data files could be the randomness values sent from randomness generation servers to the PoD Printers opened for ballots chosen for auditing contained in the AuditDataFile.json file.

4.3.4 Cryptographic Primitives

There are a number of cryptographic primitives required throughout the verifier system and are hugely important as almost every verifiable step requires some kind of cryptographic primitive from checking hash values and checking signatures to performing re-encryptions. Without the use of cryptographic primitives it would be possible to verify very little. A number of cryptographic utilities will be required for use within the verifiable components.

The bouncy castle library will be heavily utilised for the underlying cryptographic functionality. The verifier will only need to use wrappers exploiting the underlying functionality in the proper ways whilst providing specific inputs and possibly additional configuration data such as the hash algorithm or a specific Elliptic Curve to use.

The verifier will require the ability to open SHA-256 hash based commitments verifying them at the same time. A randomness value will be committed to on the Public WBB using another random witness value. The commitment will be the hash of the concatenation of the random value and the witness value together. The verifier will also need to verify signatures over certain pieces of data utilising certificate parameters, a provided signature and the data. A check will made to verify the data was signed by the appropriate entity to produce the provided signature. The verifier will also need to be able to combine a number of partial BLS signatures together to verify a threshold number of signatures has been provided as receipt of a specific action.

Finally a key cryptographic primitive used throughout the verifier is that of an ElGamal Elliptic Curve Cryptographic system. The crypto system is used to verify both encryptions and re-encryptions of plaintext Elliptic Curve points and ElGamal Elliptic Curve Point ciphertexts respectively. An ElGamal

Cryptographic system encryption is represented as $\text{Enc}_{(g,y)}(m,r) = (g^r, m * y^r)$ where (g,y) is the public key, m is the message being encrypted and r is the randomness value used. An ElGamal Elliptic Curve Cryptographic system encryption is slightly modified in that powers are swapped with multiplication and multiplication is replaced by addition to produce $\text{Enc}_{(g,y)}(m,r) = (g * r, m + y * r)$ where (g,y) is the public key, m is the message being encrypted and r is the randomness value used.

Generally the algorithms and other configuration parameters are fixed and read from library classes, however, are read in from file in certain places.

4.3.5 vVote Verifier

The vVote verification tool consists of a number of specific component focused verifiers; however, it can be run and considered as a single verifier which performs a number of election wide checks on the various underlying components. The aim of the project was to develop an independent reference implementation of a verifier with a primary focus of verifying the output produced by the portion of the vVote code developed by the University of Surrey consisting of the Ballot Generation process, Vote Packing process and commits made to the Public WBB. Initially, Mixnet verification was also considered, however the task would only be carried out if time permitted and was therefore a lower priority for the MSc project although hugely important in terms of the verifiability of the vVote election system as a whole.

Throughout this section the system as a whole will be studied in terms of its flow of data as well as providing additional details for the component specific verifiers.

The first step of the development project, after identifying the requirements, was determining how the data would flow between components and which actors would be involved in the process.

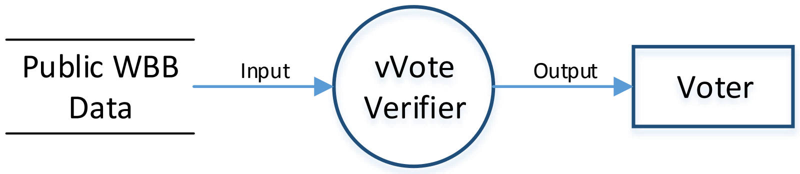


Figure 4-1 - Context level System Data Flow Diagram

Figure 4-1 shows a context level data flow diagram and shows the simplicity in which the system appears at a high level. The system has a single input which is the path to the directory containing the Public WBB data and a single output path which is to a verifying voter or even an interested party. The middle circle shows the entire system abstracted away as a single entity. The output of the system will be both as command line output and also as a corresponding text file containing the command line output. The output will contain the steps carried out during the verification process at a high level and will show the result of each stage as either a pass or fail. If a failure occurs additional information will be provided.

The verification tool is designed to be run by a voter and not an electronic voting domain expert. It will be run after the point in which the election has taken place. The event of any failures occurring is rather catastrophic showing the possibility that part of the election was not undertaken honestly.

Additional debug logs will be available showing in exact details the steps undertaken which could be used to identify how and why any possible issues have occurred. The aim is that the end user will not need to use these logs for any reason and that a simpler results file will be sufficient which outlines the steps taken and the results in easy to understand language.

The remaining parts of this section will be used to provide additional details regarding the component verifiers and provide information regarding the data each component will require and an analysis and overview of how they will undertake their verification steps.

4.3.5.1 Verifiable Components

4.3.5.1.1 Ballot Generation

Verification of the Ballot Generation process is important to the confidence end users will have in the system itself. PoD Printers are tasked with producing randomly permuted generic ballots which during the Print on Demand stage will be provided to voters. Each voter is trusting that the PoD Printer they are using is honest and that the ballot they are provided with is in fact in a randomly permuted order. In the vVote system, unlike standard versions of Prêt à Voter, in which a dishonest printer can only misalign the printed candidate names on the generated generic ballot, a dishonest PoD Printer in vVote may attempt to both generate invalid generic ballots and also perform a kleptographic attack using randomness values other than those specified by the protocol.

In vVote the PoD Printers are exploited and are used for the expensive cryptographic operations and therefore may be able to influence the generic ballots they are being used to create. Each PoD Printer should therefore be audited to ensure each has acted honestly.

The Ballot Generation Audit process provides an assumption that each PoD Printer has acted honestly. The audit does this by auditing a suitably large number of randomly and unpredictably chosen ballots, which if shown to be correctly generated, provides confidence in the accuracy of those generic ballots not chosen as part of the audit. These unchosen ballots can then be used for voting purposes with some kind of assurance that they have been reliably and honestly produced. The suitably large set of randomly and unpredictably chosen ballots is computed using the Fiat-Shamir Heuristic.

For each of the ballots randomly chosen for auditing the PoD Printer which generated it will open the commitments it made for the randomness values used during generation. During the Ballot Generation process each PoD Printer will receive $G\ RT_i$ tables containing b rows of $n+1$ pairs of random values with each pair matching a candidate identifier with one additional pair spare. For any ballot chosen for auditing the PoD Printer will find and publicly post on the Public WBB the row of previously secret pairs of random values from each of the $G\ RT_i$ tables received for only the ballot selected. This means that for each ballot chosen for auditing the Public WBB will have G sets of $n+1$ pairs of random values corresponding to a single ballot.

Each of the random values $r_{(row,col)}$ is checked to ensure it is genuine and matches what was previously committed to by each of the randomness generation servers $RGen_i$ prior to the Ballot Generation process. The corresponding witness value $R_{(row,col)}$ for each random value $r_{(row,col)}$ is used to open the commitment $c_{(row,col)}$ on the Public WBB thereby verifying the randomness value $r_{(row,col)}$ is genuine.

The corresponding random value $r_{(row,col)}$, from each of the $G\ RT_i$ tables, is then combined together to form $n+1$ combined random values in the same way as in Ballot Generation. The combined randomness values can then be used to further verify the ciphertexts are correct and in the correct permutation on the published generic ballot. This verification can be made by re-encrypting the base encrypted candidate ids, which are the plaintext candidate identifiers encrypted under the fixed randomness of 1, and sorting them into canonical order in a per race basis and then checking that they are identical to the published generic ballot and in the correctly permuted order.

The Fiat-Shamir calculation, used for determining the random selection of the ballots to audit, must also be checked to determine that a pre-selected set of ballots has not been chosen by the PoD Printer in a dishonest way. A pre-selected set of ballots would allow for modification of the unselected ballots whilst avoiding the pre-chosen set of ballots meaning any changes made will be undetected and may pass the verification steps carried out. The Fiat-Shamir heuristic must be re-calculated in the same manner from the publicly available information and checked to ensure the same ballots are selected from the re-calculated signature. The Fiat-Shamir signature includes the following values:

- The identifier for the specific PoD Printer i.e. “PrinterA”.
- The submission identifier for the message which included the generic ballots file.
- The commit time of the message.
- The actual generic ballots file containing the b generic ballots along with their permutations π.
- A threshold number of combined BLS signatures.

Inputs to the verifier

The Ballot Generation component verifier is focused mainly on the ballot generation data and therefore only requires a subset of the data to be read in and processed for its correct operation. Table 4-2 shows the inputs for the Ballot Generation component verifier.

Filename	**Purpose**
/final_commits	The /final_commits folder will be used to read in all of the Ballot Generation commitment data including mixrandomcommit, ballotgencommit and ballotauditcommit messages from each of the PoD Printers. The commits will need to be processed and organised in a PoD Printer specific manner whereby we can tell which PoD Printer is currently being audited as well as auditing only those ballots generated by them.
ballot_gen_conf.json	Used to read in election specific information including the number of candidates running in each race, the number of ballots expected for auditing and generation. The number of ballots expected for auditing and generation will be used to validate the Fiat-Shamir signature calculation and the number of candidates running in each race will be used when recalculating the generic ballots for the ballots chosen for auditing.
base_encrypted_ids.json	Used to create re-encrypted candidate identifiers for each of the generic ballots. For each of the ballots generated a combined randomness value will be used to re-encrypt the corresponding base candidate identifier. The race subsets will then be sorted producing a random permutation and the race subsets combined back together to form the generic ballot.
certs.bks	Contains the WBB peer partial public keys which will be used for validating the Fiat-Shamir signature as well as the Public WBB public key used for validating signatures against commits.
districtconf.json	Contains the district specific race size information.
plaintexts_ids.json	Contains the unencrypted plaintext candidate identifiers. These are used to verify the base encrypted candidate identifiers by

	encrypting each one with a fixed randomness value of 1.
publickey.json	The public key is a simple Elliptic Curve Point and is used as the public key in all encryptions and re-encryptions carried out.
mixrandomcommit message Attachment commitData.json	The commitData.json files contain the commitments made against randomness values sent to the PoD Printer. The commitments for ballots chosen for auditing will be used to verify the randomness values received (and used) are valid.
ballotgencommit message Attachment ciphers.json	The ciphers.json file contains the generic ballots generated during the Ballot Generation process. This file is used to lookup the generic ballots corresponding to each of the ballots chosen for auditing. Once the ballots have been re-encrypted and sorted the corresponding generic ballots will be compared against it to ensure the recalculated audited ballots match those previously generated.
ballotauditcommit Attachment AuditDataFile.json	The AuditDataFile.json file contains the serial numbers of the ballots chosen for auditing along with the randomness values received from each of the randomness generation servers which have been revealed and will be verified using hash commitments and finally combined together.
ballotauditcommit Attachment BallotSubmitResponse.json	The BallotSubmitResponse.json file contains the verifiable information relating to the selection of ballots for Ballot Generation Audit. It includes the Fiat-Shamir signature as well as partial WBB signatures both used to verify that the correct random subset of ballots was chosen for auditing.

Table 4-2 - Ballot Generation component verifier inputs

4.3.5.1.2 Vote Packing

The verification of the Vote Packing process is incredibly important in the vVote system and provides confidence that the true voter preferences are passed to the Mixnet for shuffling and decryption. During the Vote Packing process multiple ciphertexts, representing voter preferences, are packed together into a single ciphertext hiding the voter preferences. The verification process ensures that the packed ciphertexts actually represent the voter preferences. It stops pre-packed preferences, which have not been calculated from the ballots submitted during the election, being passed to the Mixnet, shuffled and decrypted and assumed to be valid preferences and used for the tallying process. Vote packing is used for performance reasons and is one of the modifications made to the original theoretical vVote protocols to ensure the system can be effectively used in a real election rather than remaining purely theoretical.

Fundamentally it provides a way of reducing the number of ciphertexts passed to the Mixnet for shuffling and decryption thereby dramatically reducing the time taken for the Mixnet process which can quickly become large. It reduces the time taken for the Mixnet process to complete by packing together multiple ciphertexts which are then passed to the Mixnet; the problem however is determining the original ciphertexts from the packed values. Vote packing utilises pre-generated lookup tables which contains the packings of every permutation of plaintext candidate identifiers.

The lookup tables are used after the mixing processes to determine the plaintext candidate identifiers from the original packed ciphertexts. Vote packing relies upon the homomorphic encryption properties of ElGamal Elliptic Curve encryption to ensure the packings of re-encrypted candidate identifiers match the packing of the same underlying plaintext candidate identifiers. Vote packing saves time during the crucial time-sensitive tallying phase of the election by pre-computing the lookup tables at an earlier time. The time taken for the generation of the tables may be significant however this can be done as soon as the candidates are confirmed and will reduce the time taken in more important stages of the election.

For verification of the Vote Packing process two checks are required:

1. A check to ensure the ciphertexts provided to the Mixnet (blt files) have been correctly packed according to the underlying voter preferences for the corresponding reduced generic ballot and that the packings match those that were input to the Mixnet. The same packing values which are input to the Mixnet should be recalculated using the reduced generic ballot and voter preferences.
2. A check to ensure the decryptions from the Mixnet, corresponding to packed plaintext identifiers (OUT files); correctly match the packing of the plaintext candidate identifiers using the claimed preferences from the Mixnet output (CSV files). This verifies that the values match what would have been looked up during the table building process for the specific packing. This check relies upon the homomorphic encryption property of ElGamal Elliptic Curve encryption. The packing skips the full recalculation of the lookup tables and requires that only specific entries (those actually used in the lookup) are recalculated.

The Vote Packing process requires two checks: one on the packed encrypted values checking the input to the Mixnet and one check on the plaintext values which checks the lookup of the output of the Mixnet.

Inputs to the Verifier

The Vote packing component verifier handles the verification of the Vote Packing and unpacking process. It validates that the packed ciphertexts were correctly packed and ready for the Mixnet as input and also verifies the Mixnet output packed plaintexts using the CSV output files and plaintext candidate identifier files. Only a subset of the data is required for the validation of the processes. Table 4-3 shows the inputs for the Vote Packing component verifier.

Filename	**Purpose**
/final_commits	The /final_commits folder will be used to read in all of the vote data

	including PoD and vote messages corresponding to cast votes made by voters. The ballotgencommit messages will also need to be stored so that the generic ballot ciphertexts can be used in the recalculation of the Vote Packing process.
base_encrypted_ids.json	Used to verify the reduction of generic ballots to district and voter specific ballots. The base candidate identifiers will be used to verify each of the reductions provided in the PoD messages for each vote. Each of the combined randomness values is used to re-encrypt the corresponding base candidate identifier and verify that it matches the corresponding ciphertext in the generic ballot.
districtconf.json	Contains the district specific race size information.
plaintexts_ids.json	Contains the unencrypted plaintext candidate identifiers. These are used to verify the base encrypted candidate identifiers by encrypting each one with a fixed randomness of 1. They are also used to verify the Mixnet output values and are used during the plaintext packing process.
publickey.json	The public key is a simple Elliptic Curve Point and is used as the public key in all encryptions and re-encryptions carried out.
ballotgencommit message Attachment ciphers.json	The ciphers.json file contains the generic ballots generated during the Ballot Generation process. The file will be used for verifying the ballot reductions as well as the Vote Packing process.
paddingpoint.json	Contains the padding used during the Vote Packing process. The padding is encrypted and then used to ensure that all rows passed to the Mixnet contain the same number of columns. The unencrypted padding point file is used to pad the plaintext packings.
map.properties	Contains a number of configuration details including the name of the Elliptic Curve to use, the padding point filename, the size of LA race packing, the size of the BTL race packing and a field detailing the races which will not be packed.
/mix/IN	Contains 0 or more .blt files which are the input files ready for the Mixnet. Each of the files contains packed ciphertexts ready for shuffling using the Mixnet. These are the input files that we are verifying by carrying out the Vote Packing process.
/mix/OUT	Contains the CSV files containing shuffled voter preferences for specific race configurations and also contains .out files which contain the packed plaintext candidate identifiers which correspond to the values looked up during the Vote Packing process. We are verifying these files.
race_map.json	Contains the mappings between race identifiers, included in specific filenames contained in the mix/IN and mix/OUT folders, and their districts and race configurations.

Table 4-3 - Vote Packing component verifier inputs

4.3.5.1.3 Public WBB Commits

Verification of the Public WBB commit data is immensely important for the verifiability and confidence of the election. If there are any disputes over the data stored on the Public WBB the entire result of the election, as calculated using the vVote system, may be thrown into complete disrepute.

The vVote system is designed to be completely end-to-end verifiable and it utilises the Public WBB to store and make available all election information which can be utilised to verify the election protocols. The information stored on the Public WBB includes all the files contained in Table 4-1 as well as additional files which can be organised into the following groups:

- Configuration data – including plaintext and base encrypted candidate identifiers, generic and district race sizes, server configuration data and public keys.
- Commits – including all receipts for data, generated candidate tables, generic ballots and randomness data.
- System state and logs – contains peer uploaded directories, the Public WBB upload directory, mongo databases, www logs and Private WBB peer logs.
- Package data – including committed vote data prepared for the Mixnet.
- Decrypted votes and proofs – includes CSV votes and various other Mixnet files.

The verifiability and all confidence in the election revolves around the security and integrity of the Public WBB data. The Public WBB utilises a signature based scheme to verify that the commitments made to the Public WBB placed in the commits folder are valid and have not been tampered with. It should be impossible to provide falsified data to the Public WBB in an undetectable manner even if the Public WBB has been compromised.

Each commitment made to the Public WBB can be broken into three parts, all of which are always present for any such commitment:

- The JSON file which contains messages on each line containing a days' worth of publicly verifiable information such as randomness generation servers, Ballot Generation commitments, ballot audit commitments, vote messages and PoD messages. Each line of the JSON file corresponds to a different JSON message.
- A ZIP file containing any additional data for any message within the JSON file. Each of the ZIP files will contain an additional inner ZIP file for each set of data corresponding to a particular message within the JSON file.
- A signature JSON file which contains a signed hash over the important protocol information from both the JSON file and ZIP files.

The verification of the Public WBB commits is relatively straightforward compared to that of the other vVote components and therefore the steps involved are minimal. The verification of the Public

WBB data checks the signed hash within the signature JSON file for each commitment made to the Public WBB and verifies falsified data has not replaced valid data and that new falsified data has not been added to the Public WBB.

The signature is on a SHA1 hash of the data that is contained within the JSON file however it only includes specific protocol important fields from inside the messages termed the internal signable content. The internal signable content can vary for each type of message. For example a vote message only includes the following fields:

- Ballot serial number
- District name
- Voter preferences
- Booth signature
- Booth identifier
- Commitment time

The commit signature is therefore determined by the types of messages within each commit. Any message which includes additional data, such as a ballotgencommit message, will require that the data provided within the corresponding ZIP corresponding to that message also be hashed and added to the previous hash of the internal signable content for the related message.

The commit signature is then a joint signature which includes the hash of the internal signable content along with its related attachments file for each message. The commit signature's format is as follows:

- The text "Commit".
- The time of the commitment – this is included in the name of the files.
- The hash detailed above.

The signature then needs to be verified using the Public WBB public key.

Inputs to the Verifier

Filename	**Purpose**
/final_commits	The /final_commits folder will be used to read in the commit data posted on the Public WBB. All of the data will be read in at a high level and stored in a per commit basis where the pieces of commits will be grouped together: JSON Message file, attachment file and JSON signature file. We need to be able to loop through the messages within the JSON message files as well as lookup corresponding

	attachment files for file messages.
certs.bks	Contains the Public WBB public key entry used for validating the signature made over the hashed data.

Table 4-4 - Public WBB Commits component verifier inputs

4.4 Functional and Non-Functional Requirements

The following section explicitly sets out the functional and non-functional requirements for the system. The requirements will be used during the testing portion of the project for determining whether the system has fully, partially or completely failed to meet each of them. These requirements will be used in part to determine the successfulness of the delivered system.

4.4.1 Functional Requirements

The following table explicitly sets out the functional requirements for the system which were described in additional detail throughout this chapter. The functional requirements relate to the operational functionality of the system.

Requirement Identifier	Requirement Description	Verifier Component	Additional notes
F 1	Verification of the Fiat-Shamir signature calculation for each PoD Printer which is used to determine the ballots chosen for auditing.	Ballot Generation	For each PoD Printer, compare a re-calculated Fiat-Shamir signature with the Fiat-Shamir signature included in the ballot submission response message corresponding to that PoD Printer's ballot audit commit message.
F 2	For each PoD Printer verify that the validated Fiat-Shamir signature was correctly used for selecting the ballots for auditing.	Ballot Generation	The Fiat-Shamir signature is used as the seed for a random selection of generic ballots which should be audited for the PoD Printer. Using the re-calculated Fiat-Shamir signature we verify that the new list of ballots chosen for auditing match those ballots included in the ballot audit commit data for the corresponding PoD Printer.
F 3	Verify that for each ballot chosen for auditing the witness value from each pair of randomness values	Ballot Generation	Each ballot chosen for auditing will have had their secret randomness values

	opens the commitment, made by the corresponding randomness generation server, to the random value on the Public WBB.		revealed and published on the Public WBB. Concatenate the random value and witness value together and hash the result using SHA256. Compare the hash with what was sent to the Public WBB during the Ballot Generation process by the corresponding randomness generation server.
F 4	For each ballot chosen for auditing, carry out the re-encryption of base candidate identifiers using a different combined randomness value for each of the candidate identifiers and sort the resulting ciphertexts in a per race basis. Verify that the sorted, re-encrypted base candidate identifiers match the list of re-encrypted base candidate identifiers contained in the matching generic ballot published on the Public WBB.	Ballot Generation	
F 5	For each ballot chosen for auditing, check that the permutation commitment contained in the corresponding generic ballot from the Public WBB can be opened by using the final combined randomness value as a witness value and the permutation constructed using the original order of the base encrypted candidate identifiers on the recalculated ballot as the random value.	Ballot Generation	Concatenate the permutation value and final combined randomness value together and hash the result using SHA256. Compare the hash with what was is included in the matching generic ballot published on the Public WBB.
F 6	Check the pre-Mixnet Vote Packing process. Check that for each of the cast votes all of the candidate ciphertexts have been correctly packed according to the preferences provided. This checks that the packed encryptions match those that were provided as input to the Mixnet.	Vote Packing	
F 7	Check the post-Mixnet Vote Packing process. Check that the decrypted packings from the Mixnet correctly match the packing of the plaintext candidate	Vote Packing	

	identifiers using the claimed preferences.		
F 8	Verify that the signatures over the commitments made to the Public WBB are valid and have been correctly calculated.	Public WBB Commits	
F 9	Verify that the base encrypted candidate identifiers are the base encryption of the public candidate identifiers using a fixed randomness of 1.	Ballot Generation/ Vote Packing	
F 10	The vVote Verifier should produce output files showing the steps carried out for the verification process and their results.		

Table 4-5 - Functional requirements

4.4.2 Non-Functional Requirements

The following table sets out the non-functional requirements for the verifier system. The non-functional requirements relate to the non-critical operational functionality including issues such as usability, performance and scalability.

Requirement Identifier	**Requirement Description**
NF 1	Ensure the code produced is cleanly written with ample comments and written in a user friendly manner allowing for easy analysis and understanding of the code.
NF 2	Produce accompanying documentation outlining the technical details for each of the component verifiers in terms of their functionality in addition to the corresponding vVote component.
NF 3	The system should have a separated backend allowing the code to easily be run in a number of different ways including command line or a website.
NF 4	The verifier should show the current progress in real-time to the end user.
NF 5	The vVote Verifier should be written in a platform independent manner using JAVA and distributed as a single JAR file.
NF 6	The vVote Verifier should show to the user in real-time the current step of the verification process being carried out.

Table 4-6 - Non-Functional requirements

4.5 Conclusion

This chapter looked at the problem analysis and requirements elicitation carried out for the vVote Verifier in general as well as its specific components. The chapter began with providing an initial brief feasibility study determining how feasible the target project would be in terms of the complexity, timeframe and amount of work required for its successful completion. The feasibility study proved helpful in identifying that the additional verification of the Mixnet on top of the Ballot

Generation process, Vote Packing process and commits made to the Public WBB could have proved to have a negative impact on the project as a whole in terms of being unable to complete the project or having to take certain shortcuts. This point by itself proved that the feasibility study was a helpful mechanism to have undertaken.

The chapter provided a more in-depth look at the verification details relating to the specific component verifiers including where necessary the details regarding why the verification should take place and what inputs each verifier will have. The section also touched on how the verification will actually take place. The chapter as a whole identified a number of functional and non-functional requirements, outlined in Table 4-5 and Table 4-6, which will be used in the design, implementation and testing chapters.

Chapter 5 – vVote Verifier Design

5.1 Introduction

The previous section provided an analysis of the proposed project as well as outlining a number of requirements for the verification tool to be developed. It provided a high-level overview of the system with an initial high-level analysis of the most important components.

This chapter will continue in the same vein and move onto the next software development lifecycle activity which is the design of the proposed system. The first part of the chapter will be used to outline the development setup used throughout the implementation stages. Next, the verifiable components analysed at a high level within 4.3.5.1 Verifiable Components, will be looked at in terms of how their requirements outlined in 4.3.5.1 Verifiable Components will be translated to algorithms which can be implemented in the final system. The system will then be looked at from a high-level in terms of its main modules and its high level architecture. Following on from the architecture design a section will be used to delve into further detail providing a low level design for the system.

5.2 Development setup

Chapter 3 and Chapter 4 and also later sections in this chapter will provide details regarding the implementation of the vVote system being verified including its requirements and technical details. This brief section will outline the development environment required to best meet these requirements in terms of libraries used, development setup and development tools.

As stated in 4.2 Feasibility study the verification tool will be developed using the JAVA programming language – chosen for its platform independence as well the numerous available external libraries which can be utilised. The use of a number of external libraries reduces the amount of code required for the development of pre-existing functions which have been developed over a large time period and scrutinised heavily. They do however require trust to be put into their correct operation. The JAVA programming language was also chosen because the vVote system itself was developed using the language and it allowed for initially sharing libraries and modules between the verifier and the original codebase to speed up development which could eventually be rewritten independently. The JAVA programming language includes a framework for working with cryptography out of the box using the Java Cryptography Architecture (JCA) in addition to the Java Cryptography Extension (JCE) which supplements this. In addition to these official standards a number of other libraries were required for more advanced cryptographic operations such as Elliptic Curve cryptography using the Bouncy Castle Crypto API, Java Pairing-Based Cryptography Library (JPBC) [37] for working with pairing based cryptographic operations and using parts of the XIMIX project [38] for threshold cryptography. In addition to the use of the JAVA based cryptographic libraries a number of additional c++ based cryptographic functions utilising OpenSSL [39] will be used through JNI for verifying a number of areas of the bouncy castle library which are shared between the vVote system and the verification tool. The c++ based functions will be compiled to both dynamic libraries (DLL) and static libraries (lib) files meaning they can be run natively on both Windows and Unix based systems and then called using the JAVA based code using JNI.

The verifier itself was developed using the Eclipse Integrated Development Environment (IDE) on a combination of Windows 8.1 Pro and Ubuntu 14.04.

5.3 Architecture Design – High-level Design

This section will be used to provide an abstracted overview of the entire verifier system. It will provide the high level structure of the system. Each of the abstracted elements will be described in brief outlining only their functionality and avoiding the detailed design for the components. Figure 5-1 shows a block level diagram for the verifier system only including the core elements of the system with their relationships.

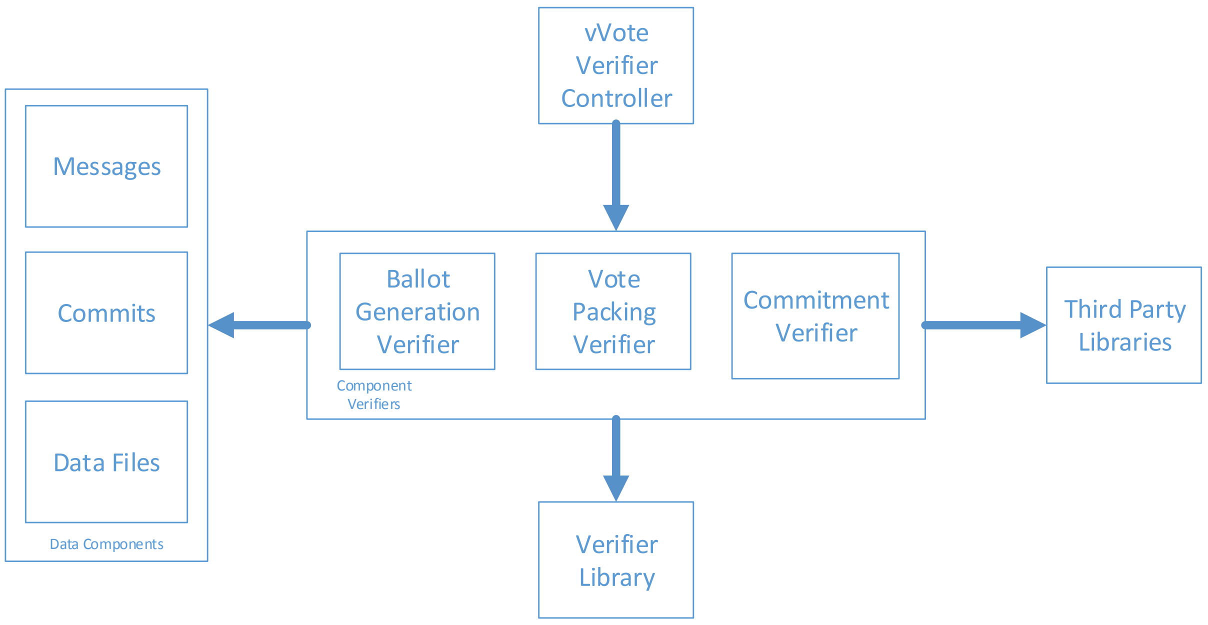


Figure 5-1 - System level block diagram

5.3.1 Messages

The verifier system needs to mirror the vVote system's heavy use of JSON messages throughout. The messages are integral to the proper and correct interaction between components and are used by the component verifiers for protocol verification. Each JSON message used in the vVote system fundamentally contains a commit time. Various types of messages will then specify additional fields. The verifier will work exclusively with the external vVote system messages which will have been submitted externally onto the Public WBB. The verifier will need to be able to read and write JSON messages. It will also need to identify and create the correct type of JSON message on-the-fly allowing for the system to completely utilise its information.

5.3.2 Data Files

Data files will be used throughout the verification tool and will contain and provide representation for any data read in from file. The data files will represent all relevant files from the Public WBB needed for the verification of the components chosen for verification. The data files will provide encapsulation and access to the data items within the files in a meaningful way. A significant portion of data will be stored on the Public WBB and will need to be easily deciphered in a way that makes verification possible.

5.3.3 Commits

Commits will provide a high level view of the data commits made to the Public WBB. All data within the final_commits folder will need to be organised together into isolated commits consisting of a JSON message file, a JSON signature file and an accompanying attachments ZIP file. Commits provide an abstracted view of the data and will provide access to it.

5.3.4 vVote Verifier Controller

The vVote Verifier controller will provide the main user interface to the end user and will take a single user provided argument which will be the path to the location of a folder containing the Public WBB data. The controller will then start up each component verifier which will independently verify their area of the vVote system utilising the same data.

5.3.5 Component Verifiers

The component verifier module will consist of the independent verifiers designed to verify the correct working of their corresponding protocols in the vVote system. Each component verifier will read the data from file independently meaning they can carry out their verification concurrently if desired. The component verifiers are the core elements of the system and carry out significant processing utilising the algorithms provided in 5.5.1 Verifiable Components.

5.3.6 Verifier Library

The verifier library will consist of all the utility functions required throughout the other aspects of the system. The library will include utilities for reading and writing files, performing cryptographic operations as well as any other necessary tasks. Any reusable functionality will be abstracted away and placed within the library to allow for easy reuse through other components.

5.3.7 Third Party Libraries

A number of third party libraries will be required for use throughout the system whereby the source code is included in full rather than simply using Java Archive files (JAR)

5.4 System Design – Low-level Design

The previous section has provided a high-level architecture view of the verifier system. It introduced the modules used and their relationships but abstracted away any concrete details as to how the modules functioned specifically.

The following section will provide an overview of the lower level design of the system. The modules introduced in the previous section will be described in further detail and will also be further divided to show how they are structured and how they function. The initial requirements set out will be revisited throughout the section.

5.4.1 Messages

As described previously the messages concept will be used heavily throughout the verifier for use in verification of protocols which utilised the use of messages for communication of protocol specific information. The messages designed in the verifier have a similar structure to that of the messages used in the vVote system, however, are slightly modified due to their changed use. Rather than being used for communication and for carrying out other specific actions, within the verification tool they are primarily used for meaningful storage of data in an accessible manner providing encapsulation for the data within.

The messages used in the verifier are simply JSON messages consisting of a pre-defined set of fields. The fields may be of any type including Boolean, Strings, Integers, JSON Objects, or JSON Arrays. At the highest level JSON messages can be typed or un-typed. Typed messages will contain within it a field “type” defining the type which can be used to construct the message on-the-fly. The majority of messages used within the system will be typed. Each different type of message will then contain specific fields and data corresponding to that unique type. Below typed messages there are two more sub hierarchies which are for vote data messages and for File messages. Vote data messages consist of all data submitted to the Public WBB during the vote casting stage of the election such as voter preferences, ballot reductions, audit messages and cancellations. File messages on the other hand consist of any message which is accompanied by additional data such as an attachment file which will also be specified in the message. File messages primarily will be used during the pre-election stage for Ballot Generation processes such as committing to random values by the randomness generation servers, committing to generated ballots and committing to ballots chosen for auditing. Figure 5-2 provides a class diagram for the Messages module which shows the structure of the class hierarchy for JSON messages used within the verifier. Appendix A shows a typical example of a PoD Message and a Vote message.

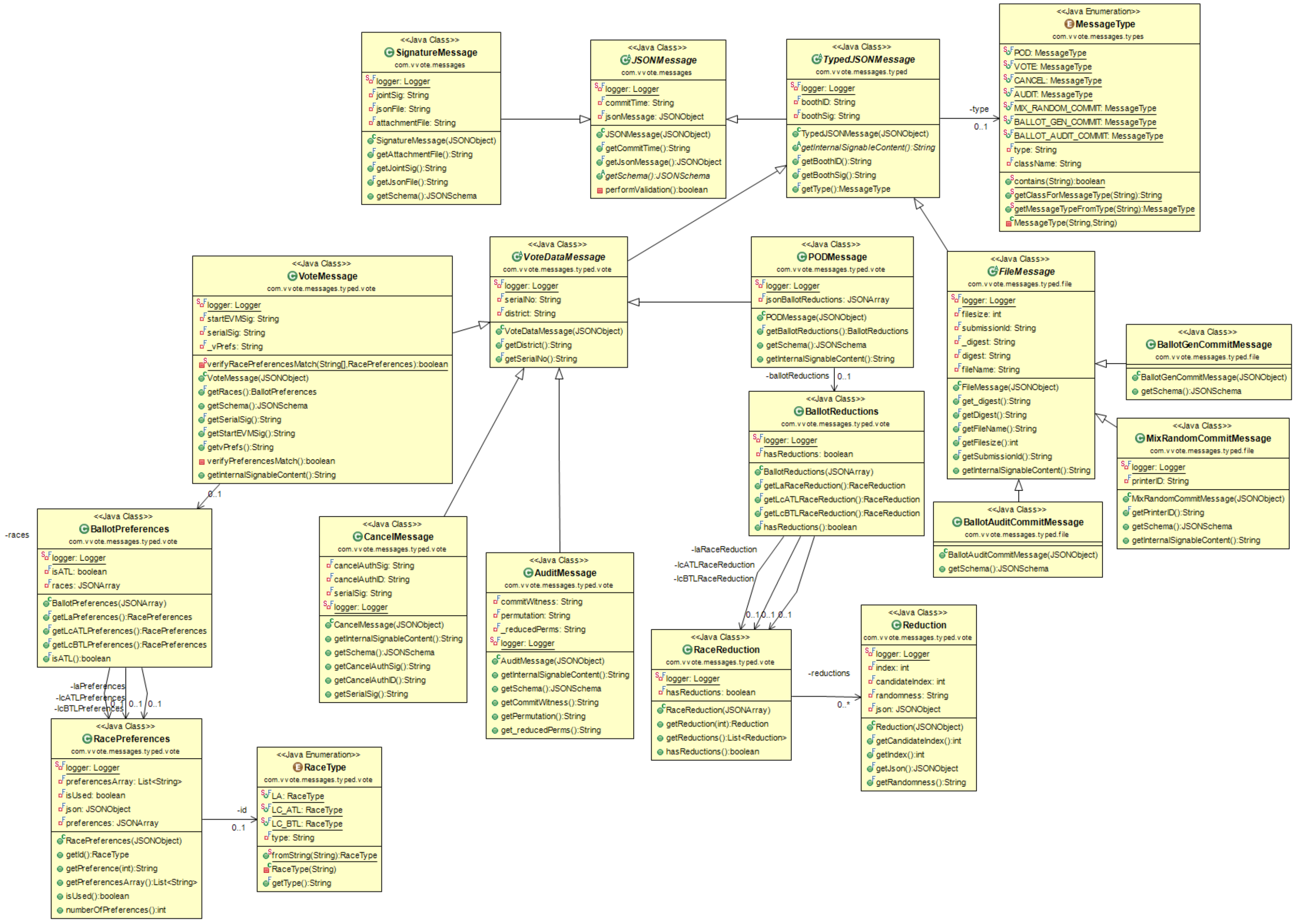


Figure 5-2 - Message class diagram

5.4.2 Data Files

The standalone verifier is designed to be run after the election has taken place and after the point where all commits have been made to the Public WBB. All the relevant information required for proving the correct operation of the specific protocols underlying the vVote system should be present on the Public WBB at the point when the verifier is started. The verifier therefore relies heavily on reading in and making sense of a large quantity of data. The data needs to be read in and stored in an efficient and meaningful way so that various parts of the data can be looked up and cross referenced using the algorithms within the verifiers.

The data provided, which requires reading in and appropriate storage, can be broken into a number of classes as follows:

- Configuration data – including plaintext and base encrypted candidate identifiers, generic and district race sizes, server configuration data and public keys.
- Commits – including all receipts for data, generated candidate tables, audit data, generic ballots and randomness data.
- Package data – including committed vote data prepared for the Mixnet
- Decrypted votes and proofs – includes CSV votes and various other Mixnet files.

The election configuration data is made up of a mix of JSON files and String files and can be read in directly. The commit data includes vote message data and ballot generation data. In 5.4.1 Messages it was stated that both vote messages and Ballot Generation messages are simply stored as messages; however, the data must be stored at a higher level than this simple message level as the messages may also include additional related data such as attachments. Vote messages must also be stored in a grouped manner whereby each Vote message is grouped together with its corresponding PoD message. The messages provide encapsulation to the simple protocol messages however other structures were required to additionally encapsulate the messages along with the additional corresponding data.

5.4.2.1 District Data

District configuration data is provided as a single file as a JSON message consisting of multiple key value pairs where the key is the district name and the value is a JSON Object containing three inner key value pairs which identify the LA, LC ATL and LC BTL race sizes. Figure 5-3 shows an example district configuration and Figure 5-4 shows the class diagram for the district configuration file.

```
{ "Gembrook" : {
    "la" : 9,
    "lc_atl" : 8,
    "lc_btl" : 42
  },
  "Northcote" : {
    "la" : 12,
    "lc_atl" : 8,
    "lc_btl" : 25
  }
}
```

Figure 5-3 - Example district configuration

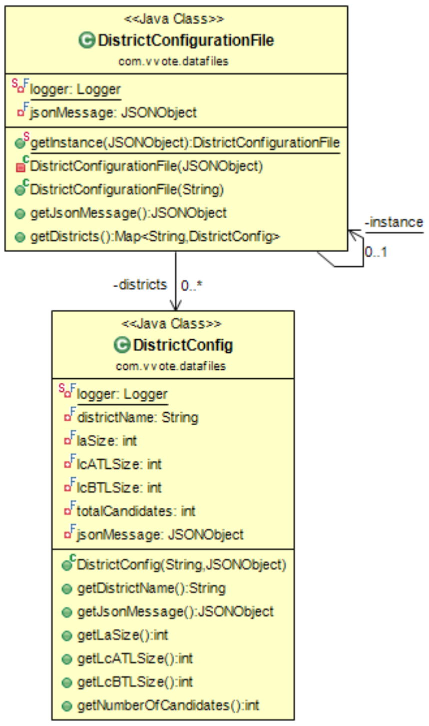


Figure 5-4 - District Configuration class diagram

5.4.2.2 Ballot Audit Commit Data

Appendix B shows an example ballot audit commit message. Within the message is a filename which specifies the name of an inner zip file within the corresponding attachment zip file for the current commitment made on the Public WBB. The inner zip file contains the audit data with each line consisting of the revealed randomness values as pairs of values used to re-encrypt the base candidate identifiers to construct the audited generic ballot with the specified serial number. An additional submission response to the audit data which includes the Fiat-Shamir signature will also be included as part of a ballot audit commit. Figure 5-5 shows the class diagram used to represent the ballot audit commit data read from the sample data.

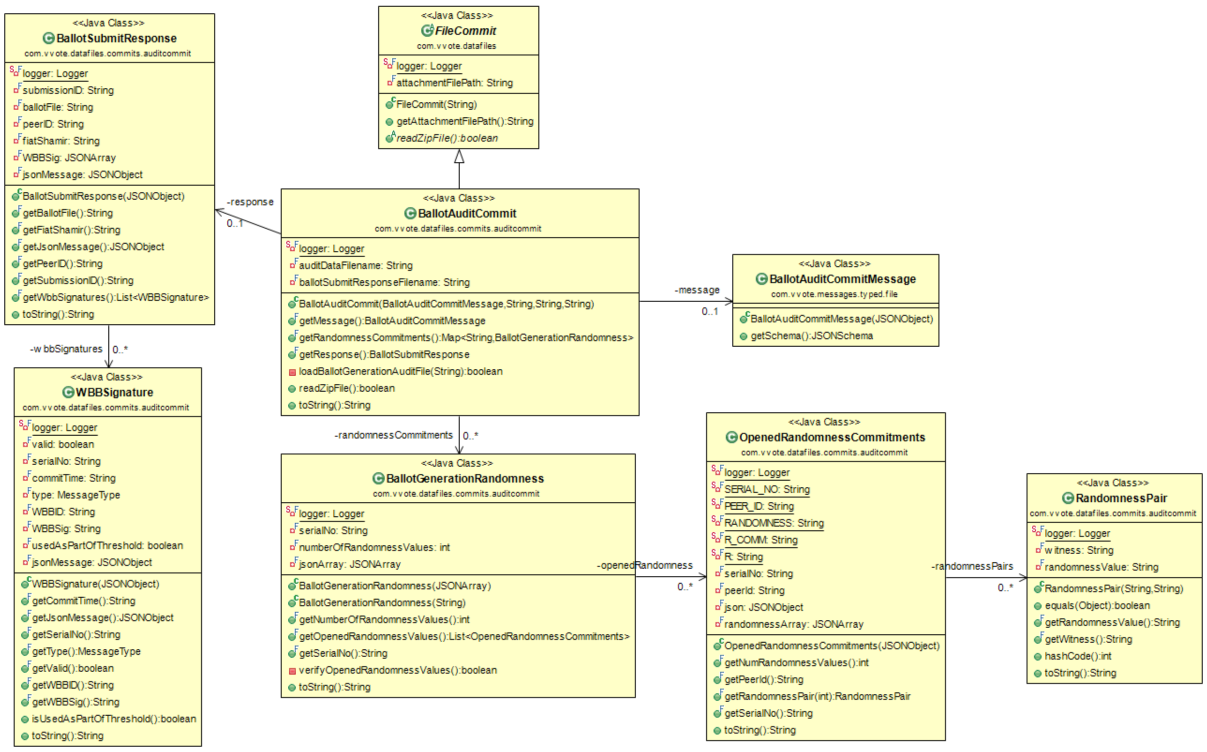


Figure 5-5 - Ballot Audit Commit data file

5.4.2.3 Ballot Generation Commit Data

Appendix B shows an example Ballot Generation commit message. Within the message is a filename which specifies the name of an inner zip file within the attachment zip file for the current commitment made on the Public WBB. The inner zip file contains the generation data which contains on each line a generic ballot which consists of a serial number, permutation and re-encrypted base candidate identifiers. Figure 5-6 shows the class diagram used to represent the Ballot Generation commit data read from the sample data.

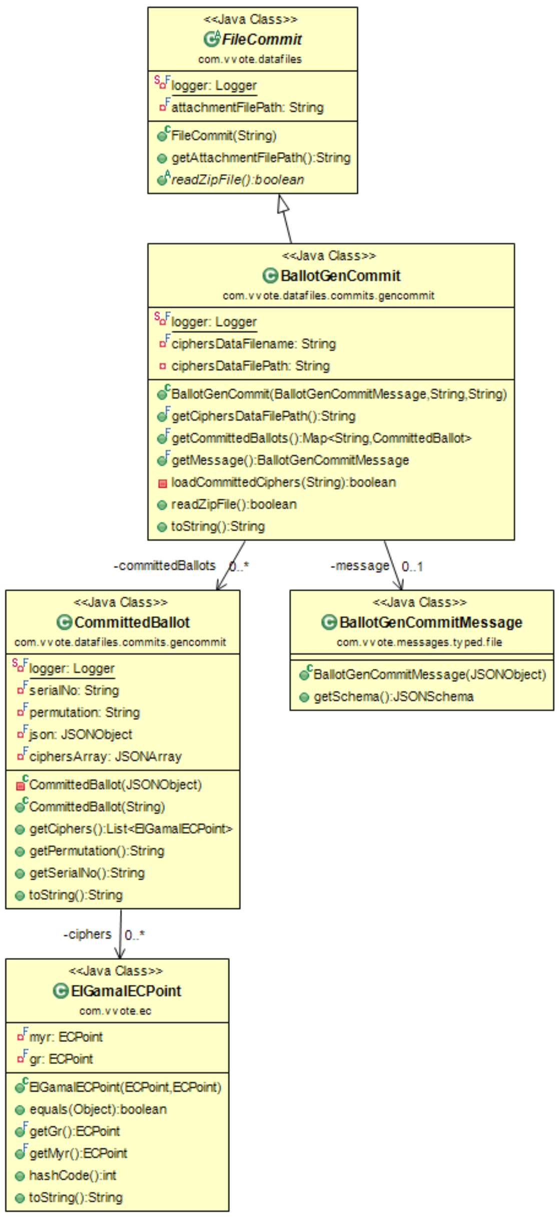
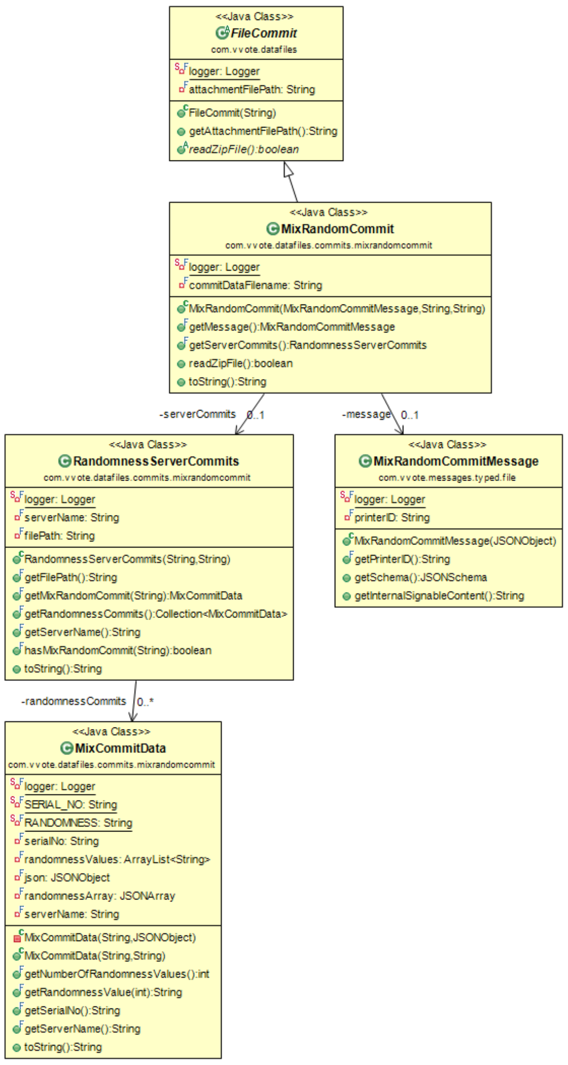


Figure 5-6 - Ballot Generation Commit data file

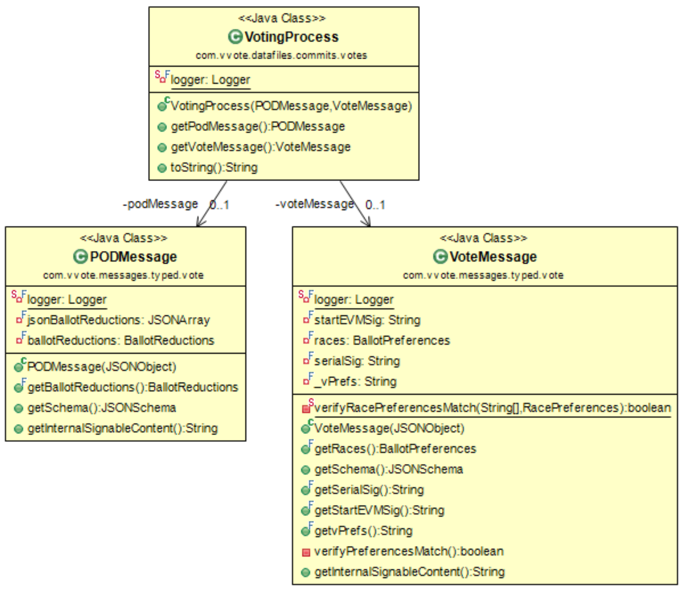
5.4.2.4 Mix Random Commit Data

Appendix B shows an example mix random commit message. Within the message is a filename which specifies the name of an inner zip file within the attachment zip file for the current commitment made on the Public WBB. The inner zip file contains the mix random commit data which contains on each line the commitments made against the randomness values generated by the randomness generation server and then sent to the specified PoD Printer for a specific ballot with the serial number provided. Figure 5-7 shows the class diagram used to represent the mix random commit data read from the sample data.


<<Java Class>>
FileCommit
com.vvote.datafiles
logger: Logger
attachmentFilePath: String
FileCommit(String)
getAttachmentFilePath():String
readZipFile():boolean
<<Java Class>>
MixRandomCommit
com.vvote.datafiles.commits.mixrandomcommit
logger: Logger
commitDataFilename: String
MixRandomCommit(MixRandomCommitMessage,String,String)
getMessage():MixRandomCommitMessage
getServerCommits():RandomnessServerCommits
readZipFile():boolean
toString():String
-serverCommits 0..1
-message 0..1
<<Java Class>>
RandomnessServerCommits
com.vvote.datafiles.commits.mixrandomcommit
logger: Logger
serverName: String
filePath: String
RandomnessServerCommits(String,String)
getFilePath():String
getMixRandomCommit(String):MixCommitData
getRandomnessCommits():Collection<MixCommitData>
getServerName():String
hasMixRandomCommit(String):boolean
toString():String
<<Java Class>>
MixRandomCommitMessage
com.vvote.messages.typed.file
logger: Logger
printerID: String
MixRandomCommitMessage(JSONObject)
getPrinterID():String
getSchema():JSONSchema
getInternalSignableContent():String
-randomnessCommits 0..*
<<Java Class>>
MixCommitData
com.vvote.datafiles.commits.mixrandomcommit
logger: Logger
SERIAL_NO: String
RANDOMNESS: String
serialNo: String
randomnessValues: ArrayList<String>
json: JSONObject
randomnessArray: JSONArray
serverName: String
MixCommitData(String,JSONObject)
MixCommitData(String,String)
getNumberOfRandomnessValues():int
getRandomnessValue(int):String
getSerialNo():String
getServerName():String
toString():String
Figure 5-7 - Mix Random Commit data file

5.4.2.5 Vote Data

Appendix A shows example voting data. The voting data consists of two related messages, a PoD message and a vote message. The two messages correspond to the same ballot sharing the serial number specified in the messages. The PoD message contains within it the ballot reductions required to reduce down the corresponding generic ballot so that it contains only the candidates applicable for the voter’s district. The vote message along with the district and serial number contains the voter’s preferences in a per race basis. Grouping together of the vote and PoD messages makes the later use of the corresponding messages easier. Figure 5-8 shows the grouping of the vote and PoD messages together into a VotingProcess.



<<Java Class>>
VotingProcess
com.vvote.datafiles.commits.votes
logger: Logger
VotingProcess(PODMessage,VoteMessage)
getPodMessage():PODMessage
getVoteMessage():VoteMessage
toString():String
-podMessage 0..1
-voteMessage 0..1
<<Java Class>>
PODMessage
com.vvote.messages.typed.vote
logger: Logger
jsonBallotReductions: JSONArray
ballotReductions: BallotReductions
PODMessage(JSONObject)
getBallotReductions():BallotReductions
getSchema():JSONSchema
getInternalSignableContent():String
<<Java Class>>
VoteMessage
com.vvote.messages.typed.vote
logger: Logger
startEVMSig: String
races: BallotPreferences
serialSig: String
_vPrefs: String
verifyRacePreferencesMatch(String[],RacePreferences):boolean
VoteMessage(JSONObject)
getRaces():BallotPreferences
getSchema():JSONSchema
getSerialSig():String
getStartEVMSig():String
getvPrefs():String
verifyPreferencesMatch():boolean
getInternalSignableContent():String

Figure 5-8 - Vote messages class diagram

5.4.2.6 Mixnet Output Data

Appendix C shows an example set of Mixnet output preferences for the district of Broadmeadows. The Mixnet output preferences correspond to the decrypted voter preferences rearranged into the original plaintext candidate identifier ordering. In the design of the verifier system each of these csv files correspond to a single Mixnet output object. Each of the Mixnet output objects then has a RaceIdentifier which is constructed using the filename or is looked up from file using the race identifier value from the race_map.json file. The filename may directly include the race identifier, race name, race type and district. The voter preferences are then stored as a list of the individual ballot preferences whereby each line for each of the files is stored as an independent CSVPreferences. The structure of the Mixnet output classes can be seen in Figure 5-9.

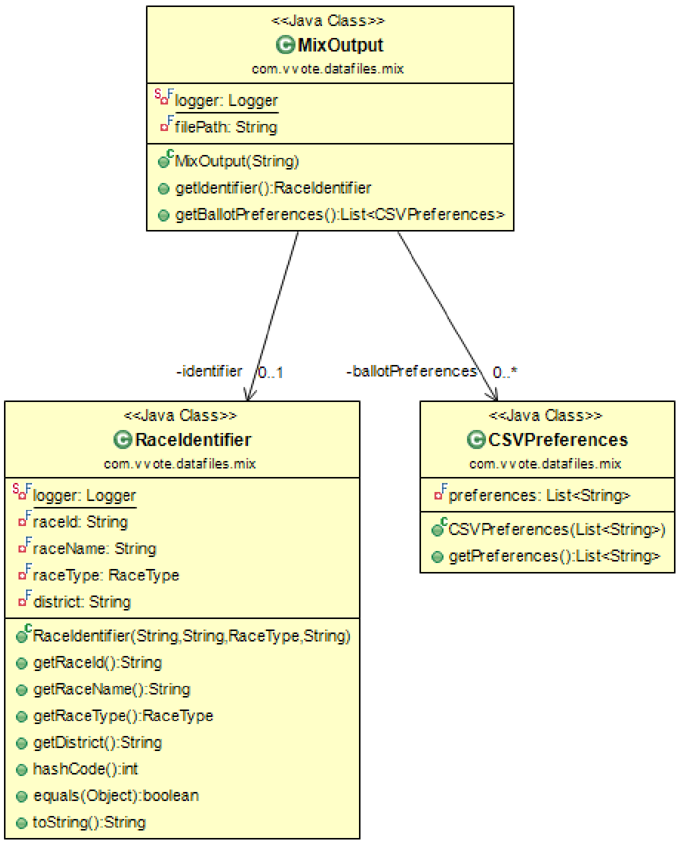


Figure 5-9 - Mixnet Output class diagram

5.4.2.7 Certificates Data

Appendix D shows an example certs.bks file which consists of the Public WBB public key entry in addition to the Private WBB peers' public key entries. Each of the Private WBB peers along with their own public key also have a partial public key which is used during certain situations to come to a threshold agreement with the other peers. The certificate entries are used throughout the project for verifying signatures and that valid threshold agreements have been made. Figure 5-10 shows the class diagram used for representing the file certs.bks.

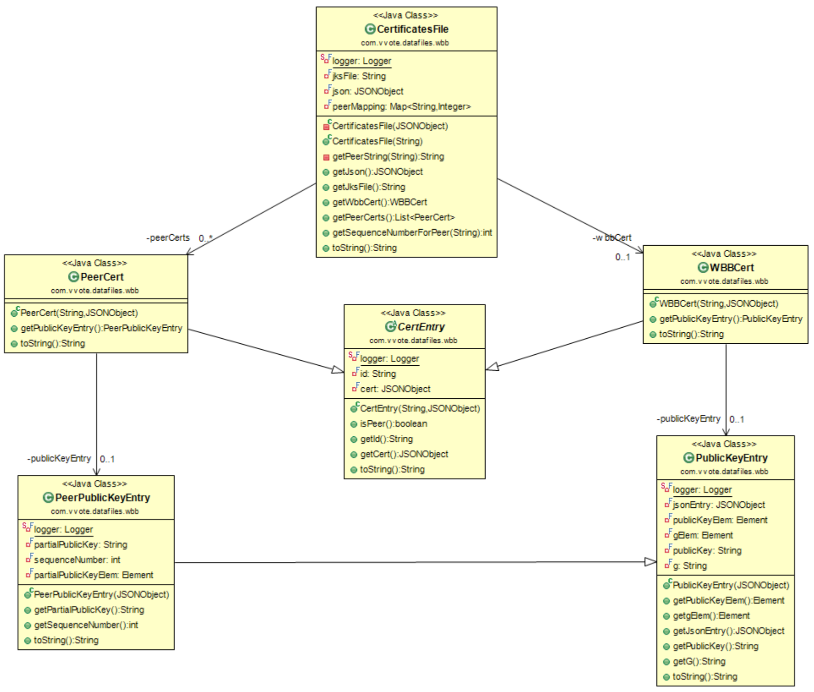


Figure 5-10 - Certificates file class diagram

5.4.3 Commits

Appendix E shows the structure of an entire commit. The commit included shows a sample set of ballot generation data and will be used to describe the commit structure. Each commit consists of a JSON message file (1403161200000.json), a signature JSON file (1403161200000_signature.json) and an accompanying attachment ZIP file (1403161200000_attachments.zip). The commit contains all the information needed to carry out the Ballot Generation verification process. The JSON message file consists of the external messages sent by the vVote system during the Ballot Generation process during a single day. If the process spanned longer than a single day the ballot generation data could have been split over multiple commits. The signature JSON file contains the joint signature which is used to verify the commit made to the Public WBB. The file also contains the names of the files over which the signature was calculated. The attachment ZIP file contains any additional data. Any external messages, in the JSON message file, which include any additional data, will specify the filename of the inner zip file in which the additional data will be present.

There will also be commits including voting data which may be split over multiple commits. The commits are made to the Public WBB at the end of each day and therefore activities may be split over multiple commits.

Figure 5-11 shows the structure for a final commitment consisting of the 3 other related files. The way in which the files are grouped together is by using the commitTime identifier within their filename along with the additional “_signature” and “_attachments.zip” identifiers.

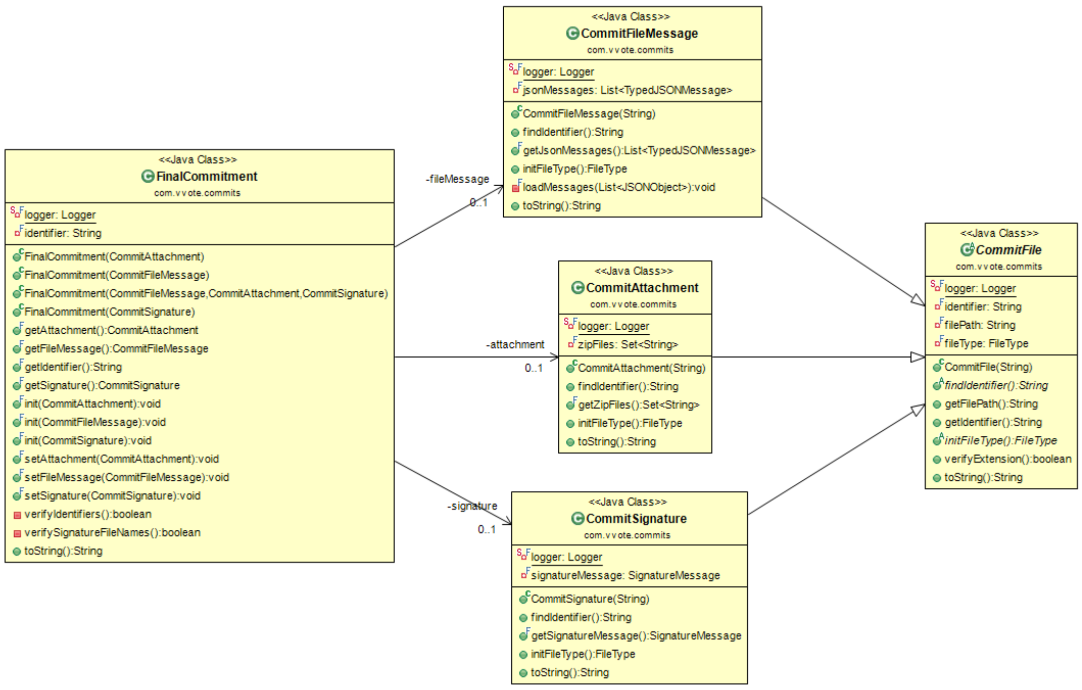


Figure 5-11 - Commit class diagram

5.4.4 Verifiers

The verifier components of the vVote Verifier system are the most important aspect of the code and carry out the algorithms which attempt to verify the correct working of the vVote system. Each verifier component has similar functionality and structure. Each of the verifier components are made up of a data store, a spec object, the main verifier and additionally an optional configuration object.

The data store's responsibility will be to read in and process the vast amount of data required for proper validation of the proper working of the vVote system. The data reading operations were extracted out of the verifiers themselves to provide better abstraction and isolation of elements.

The spec objects' responsibility is to hold component specific specification details which make reading in the data files and carrying out the verification easier by extracting out specification type details into an isolated object.

The optional configuration object may be used to isolate additional component specific related information. The configuration objects are used primarily to encapsulate the information contained within a specific configuration file such as map.properties or ballot_gen_conf.json.

Figure 5-12 shows the structure of the verifier components including the data stores, configuration files and specification objects as well as the component verifiers.

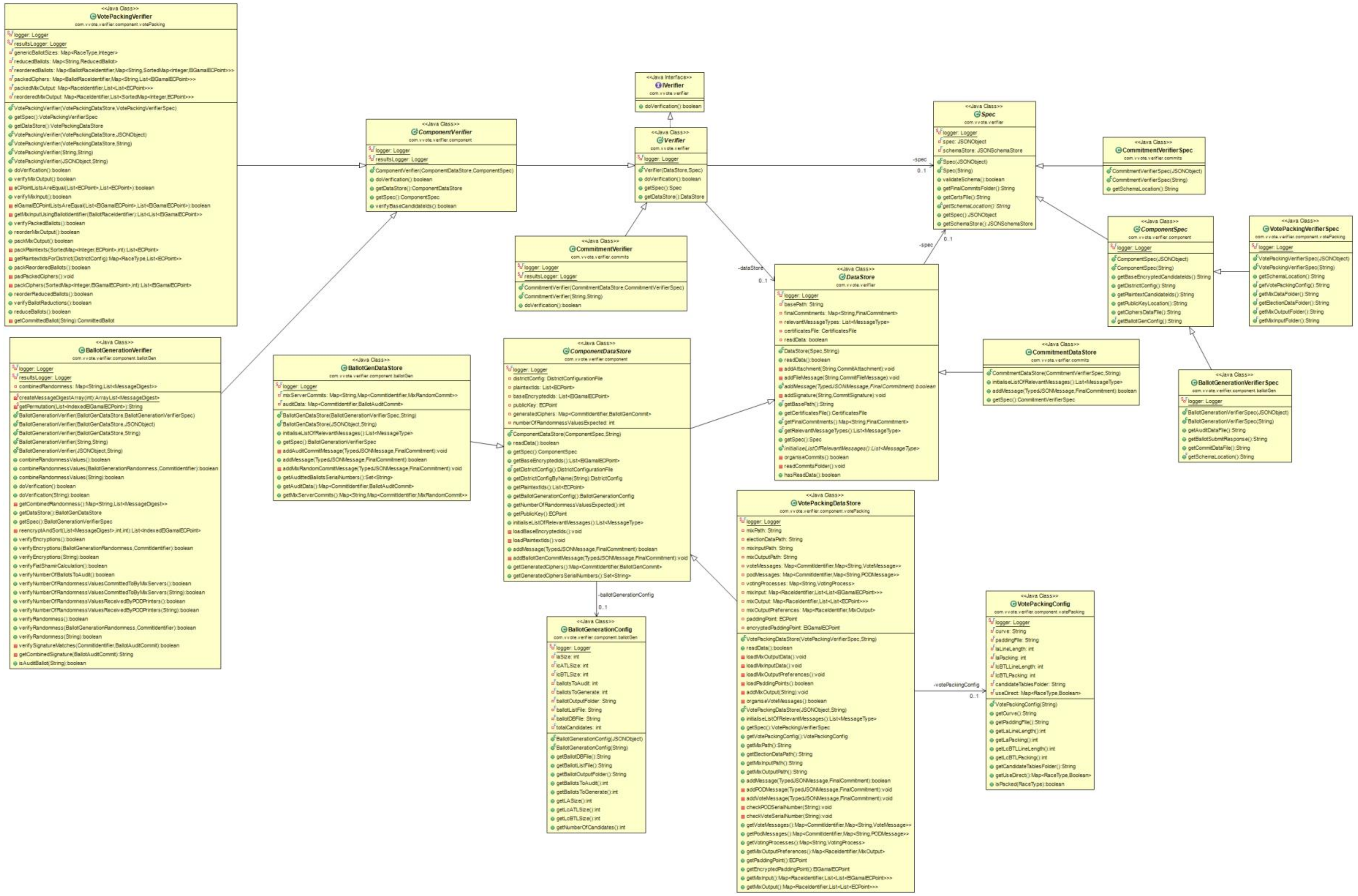

<<Java Class>>
VotePackingVerifier
com.vvote.verifier.component.votePacking
logger: Logger
resultsLogger: Logger
genericBallotSizes: Map<RaceType,Integer>
reducedBallots: Map<String,ReducedBallot>
reorderedBallots: Map<BallotRaceIdentifier,Map<String,SortedMap<Integer,ElGamalECPoint>>>
packedCiphers: Map<BallotRaceIdentifier,Map<String,List<ElGamalECPoint>>>
packedMixOutput: Map<RaceIdentifier,List<List<ECPoint>>>
reorderedMixOutput: Map<RaceIdentifier,List<SortedMap<Integer,ECPoint>>>
VotePackingVerifier(VotePackingDataStore,VotePackingVerifierSpec)
getSpec():VotePackingVerifierSpec
getDataStore():VotePackingDataStore
VotePackingVerifier(VotePackingDataStore,JSONObject)
VotePackingVerifier(VotePackingDataStore,String)
VotePackingVerifier(String,String)
VotePackingVerifier(JSONObject,String)
doVerification():boolean
verifyMixOutput():boolean
eCPointListsAreEqual(List<ECPoint>,List<ECPoint>):boolean
verifyMixInput():boolean
elGamalECPointListsAreEqual(List<ElGamalECPoint>,List<ElGamalECPoint>):boolean
getMixInputUsingBallotIdentifier(BallotRaceIdentifier):List<List<ElGamalECPoint>>
verifyPackedBallots():boolean
reorderMixOutput():boolean
packMixOutput():boolean
packPlaintexts(SortedMap<Integer,ECPoint>,int):List<ECPoint>
getPlaintextIdsForDistrict(DistrictConfig):Map<RaceType,List<ECPoint>>
packReorderedBallots():boolean
padPackedCiphers():void
packCiphers(SortedMap<Integer,ElGamalECPoint>,int):List<ElGamalECPoint>
reorderReducedBallots():boolean
verifyBallotReductions():boolean
reduceBallots():boolean
getCommittedBallot(String):CommittedBallot
<<Java Class>>
ComponentVerifier
com.vvote.verifier.component
logger: Logger
resultsLogger: Logger
ComponentVerifier(ComponentDataStore,ComponentSpec)
doVerification():boolean
getDataStore():ComponentDataStore
getSpec():ComponentSpec
verifyBaseCandidateIds():boolean
<<Java Interface>>
IVerifier
com.vvote.verifier
doVerification():boolean
<<Java Class>>
Verifier
com.vvote.verifier
logger: Logger
Verifier(DataStore,Spec)
doVerification():boolean
getSpec():Spec
getDataStore():DataStore
-spec
0..1
<<Java Class>>
Spec
com.vvote.verifier
logger: Logger
spec: JSONObject
schemaStore: JSONSchemaStore
Spec(JSONObject)
Spec(String)
validateSchema():boolean
getFinalCommitsFolder():String
getCertsFile():String
getSchemaLocation():String
getSpec():JSONObject
getSchemaStore():JSONSchemaStore
<<Java Class>>
CommitmentVerifierSpec
com.vvote.verifier.commits
CommitmentVerifierSpec(JSONObject)
CommitmentVerifierSpec(String)
getSchemaLocation():String
<<Java Class>>
ComponentSpec
com.vvote.verifier.component
logger: Logger
ComponentSpec(JSONObject)
ComponentSpec(String)
getBaseEncryptedCandidateIds():String
getDistrictConfig():String
getPlaintextCandidateIds():String
getPublicKeyLocation():String
getCiphersDataFile():String
getBallotGenConfig():String
<<Java Class>>
VotePackingVerifierSpec
com.vvote.verifier.component.votePacking
logger: Logger
VotePackingVerifierSpec(JSONObject)
VotePackingVerifierSpec(String)
getSchemaLocation():String
getVotePackingConfig():String
getMixDataFolder():String
getElectionDataFolder():String
getMixOutputFolder():String
getMixInputFolder():String
<<Java Class>>
CommitmentVerifier
com.vvote.verifier.commits
logger: Logger
resultsLogger: Logger
CommitmentVerifier(CommitmentDataStore,CommitmentVerifierSpec)
CommitmentVerifier(String,String)
doVerification():boolean
-dataStore
0..1
-spec
0..1
<<Java Class>>
DataStore
com.vvote.verifier
logger: Logger
basePath: String
finalCommitments: Map<String,FinalCommitment>
relevantMessageTypes: List<MessageType>
certificatesFile: CertificatesFile
readData: boolean
DataStore(Spec,String)
readData():boolean
addAttachment(String,CommitAttachment):void
addFileMessage(String,CommitFileMessage):void
addMessage(TypedJSONMessage,FinalCommitment):boolean
addSignature(String,CommitSignature):void
getBasePath():String
getCertificatesFile():CertificatesFile
getFinalCommitments():Map<String,FinalCommitment>
getRelevantMessageTypes():List<MessageType>
getSpec():Spec
initialiseListOfRelevantMessages():List<MessageType>
organiseCommits():boolean
readCommitsFolder():void
hasReadData():boolean
<<Java Class>>
CommitmentDataStore
com.vvote.verifier.commits
CommitmentDataStore(CommitmentVerifierSpec,String)
initialiseListOfRelevantMessages():List<MessageType>
addMessage(TypedJSONMessage,FinalCommitment):boolean
getSpec():CommitmentVerifierSpec
<<Java Class>>
BallotGenerationVerifierSpec
com.vvote.verifier.component.ballotGen
logger: Logger
BallotGenerationVerifierSpec(JSONObject)
BallotGenerationVerifierSpec(String)
getAuditDataFile():String
getBallotSubmitResponse():String
getCommitDataFile():String
getSchemaLocation():String
<<Java Class>>
BallotGenerationVerifier
com.vvote.verifier.component.ballotGen
logger: Logger
resultsLogger: Logger
combinedRandomness: Map<String,List<MessageDigest>>
createMessageDigestArray(int):ArrayList<MessageDigest>
getPermutation(List<IndexedElGamalECPoint>):String
BallotGenerationVerifier(BallotGenDataStore,BallotGenerationVerifierSpec)
BallotGenerationVerifier(BallotGenDataStore,JSONObject)
BallotGenerationVerifier(BallotGenDataStore,String)
BallotGenerationVerifier(String,String)
BallotGenerationVerifier(JSONObject,String)
combineRandomnessValues():boolean
combineRandomnessValues(BallotGenerationRandomness,CommitIdentifier):boolean
combineRandomnessValues(String):boolean
doVerification():boolean
doVerification(String):boolean
getCombinedRandomness():Map<String,List<MessageDigest>>
getDataStore():BallotGenDataStore
getSpec():BallotGenerationVerifierSpec
reencryptAndSort(List<MessageDigest>,int,int):List<IndexedElGamalECPoint>
verifyEncryptions():boolean
verifyEncryptions(BallotGenerationRandomness,CommitIdentifier):boolean
verifyEncryptions(String):boolean
verifyFiatShamirCalculation():boolean
verifyNumberOfBallotsToAudit():boolean
verifyNumberOfRandomnessValuesCommittedToByMixServers():boolean
verifyNumberOfRandomnessValuesCommittedToByMixServers(String):boolean
verifyNumberOfRandomnessValuesReceivedByPODPrinters():boolean
verifyNumberOfRandomnessValuesReceivedByPODPrinters(String):boolean
verifyRandomness():boolean
verifyRandomness(BallotGenerationRandomness,CommitIdentifier):boolean
verifyRandomness(String):boolean
verifySignatureMatches(CommitIdentifier,BallotAuditCommit):boolean
getCombinedSignature(BallotAuditCommit):String
isAuditBallot(String):boolean
<<Java Class>>
BallotGenDataStore
com.vvote.verifier.component.ballotGen
logger: Logger
mixServerCommits: Map<String,Map<CommitIdentifier,MixRandomCommit>>
auditData: Map<CommitIdentifier,BallotAuditCommit>
BallotGenDataStore(BallotGenerationVerifierSpec,String)
BallotGenDataStore(JSONObject,String)
initialiseListOfRelevantMessages():List<MessageType>
getSpec():BallotGenerationVerifierSpec
addAuditCommitMessage(TypedJSONMessage,FinalCommitment):void
addMessage(TypedJSONMessage,FinalCommitment):boolean
addMixRandomCommitMessage(TypedJSONMessage,FinalCommitment):void
getAudittedBallotsSerialNumbers():Set<String>
getAuditData():Map<CommitIdentifier,BallotAuditCommit>
getMixServerCommits():Map<String,Map<CommitIdentifier,MixRandomCommit>>
<<Java Class>>
ComponentDataStore
com.vvote.verifier.component
logger: Logger
districtConfig: DistrictConfigurationFile
plaintextIds: List<ECPoint>
baseEncryptedIds: List<ElGamalECPoint>
publicKey: ECPoint
generatedCiphers: Map<CommitIdentifier,BallotGenCommit>
numberOfRandomnessValuesExpected: int
ComponentDataStore(ComponentSpec,String)
readData():boolean
getSpec():ComponentSpec
getBaseEncryptedIds():List<ElGamalECPoint>
getDistrictConfig():DistrictConfigurationFile
getDistrictConfigByName(String):DistrictConfig
getPlaintextIds():List<ECPoint>
getBallotGenerationConfig():BallotGenerationConfig
getNumberOfRandomnessValuesExpected():int
getPublicKey():ECPoint
initialiseListOfRelevantMessages():List<MessageType>
loadBaseEncryptedIds():void
loadPlaintextIds():void
addMessage(TypedJSONMessage,FinalCommitment):boolean
addBallotGenCommitMessage(TypedJSONMessage,FinalCommitment):void
getGeneratedCiphers():Map<CommitIdentifier,BallotGenCommit>
getGeneratedCiphersSerialNumbers():Set<String>
-ballotGenerationConfig
0..1
<<Java Class>>
BallotGenerationConfig
com.vvote.verifier.component.ballotGen
logger: Logger
laSize: int
lcATLSize: int
lcBTLSize: int
ballotsToAudit: int
ballotsToGenerate: int
ballotOutputFolder: String
ballotListFile: String
ballotDBFile: String
totalCandidates: int
BallotGenerationConfig(JSONObject)
BallotGenerationConfig(String)
getBallotDBFile():String
getBallotListFile():String
getBallotOutputFolder():String
getBallotsToAudit():int
getBallotsToGenerate():int
getLASize():int
getLcATLSize():int
getLcBTLSize():int
getNumberOfCandidates():int
<<Java Class>>
VotePackingDataStore
com.vvote.verifier.component.votePacking
logger: Logger
mixPath: String
electionDataPath: String
mixInputPath: String
mixOutputPath: String
voteMessages: Map<CommitIdentifier,Map<String,VoteMessage>>
podMessages: Map<CommitIdentifier,Map<String,PODMessage>>
votingProcesses: Map<String,VotingProcess>
mixInput: Map<RaceIdentifier,List<List<ElGamalECPoint>>>
mixOutput: Map<RaceIdentifier,List<List<ECPoint>>>
mixOutputPreferences: Map<RaceIdentifier,MixOutput>
paddingPoint: ECPoint
encryptedPaddingPoint: ElGamalECPoint
VotePackingDataStore(VotePackingVerifierSpec,String)
readData():boolean
loadMixOutputData():void
loadMixInputData():void
loadMixOutputPreferences():void
loadPaddingPoints():boolean
addMixOutput(String):void
organiseVoteMessages():boolean
VotePackingDataStore(JSONObject,String)
initialiseListOfRelevantMessages():List<MessageType>
getSpec():VotePackingVerifierSpec
getVotePackingConfig():VotePackingConfig
getMixPath():String
getElectionDataPath():String
getMixInputPath():String
getMixOutputPath():String
addMessage(TypedJSONMessage,FinalCommitment):boolean
addPODMessage(TypedJSONMessage,FinalCommitment):void
addVoteMessage(TypedJSONMessage,FinalCommitment):void
checkPODSerialNumber(String):void
checkVoteSerialNumber(String):void
getVoteMessages():Map<CommitIdentifier,Map<String,VoteMessage>>
getPodMessages():Map<CommitIdentifier,Map<String,PODMessage>>
getVotingProcesses():Map<String,VotingProcess>
getMixOutputPreferences():Map<RaceIdentifier,MixOutput>
getPaddingPoint():ECPoint
getEncryptedPaddingPoint():ElGamalECPoint
getMixInput():Map<RaceIdentifier,List<List<ElGamalECPoint>>>
getMixOutput():Map<RaceIdentifier,List<List<ECPoint>>>
-votePackingConfig
0..1
<<Java Class>>
VotePackingConfig
com.vvote.verifier.component.votePacking
logger: Logger
curve: String
paddingFile: String
laLineLength: int
laPacking: int
lcBTLLineLength: int
lcBTLPacking: int
candidateTablesFolder: String
useDirect: Map<RaceType,Boolean>
VotePackingConfig(String)
getCurve():String
getPaddingFile():String
getLaLineLength():int
getLaPacking():int
getLcBTLLineLength():int
getLcBTLPacking():int
getCandidateTablesFolder():String
getUseDirect():Map<RaceType,Boolean>
isPacked(RaceType):boolean

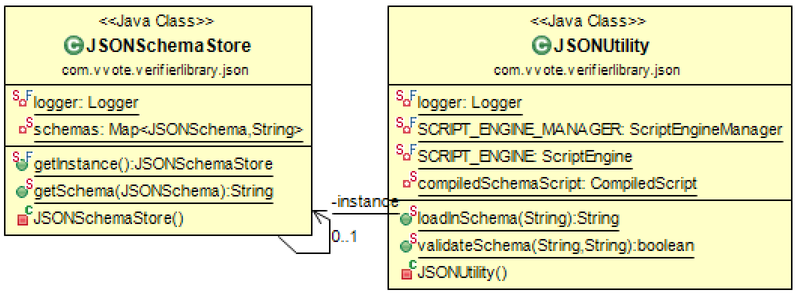
Figure 5-12 - Verifier Component Class Diagram

5.4.5 Verifier Library

The verifier library abstracts away the utility functionality required throughout other aspects of the system including reading and writing files, cryptographic operations and data type conversions.

5.4.5.1 JSON

The majority of the data read in from file is in JSON format. The successful reading in and examination of the data requires that some files have a specific format and include the required relevant fields. Functionality has therefore been produced to validate JSON messages against known schema files which specify the format and required data elements. Appendix F shows two examples of schema files which are used to validate the format for provided vote messages and Ballot Generation messages. Each of the messages shown in Figure 5-2 has a specific schema file. Also included in Appendix F is a list of all schema files along with their file path which is used for reading in the schemas. Figure 5-13 shows the JSON utility class, which provides the ability to validate JSON messages against provided schemas, along with the JSON schema store, which provides storage and access for all the JSON schemas.



<<Java Class>>
JSONSchemaStore
com.vvote.verifierlibrary.json
logger: Logger
schemas: Map<JSONSchema,String>
getInstance():JSONSchemaStore
getSchema(JSONSchema):String
JSONSchemaStore()
-instance
0..1
<<Java Class>>
JSONUtility
com.vvote.verifierlibrary.json
logger: Logger
SCRIPT_ENGINE_MANAGER: ScriptEngineManager
SCRIPT_ENGINE: ScriptEngine
compiledSchemaScript: CompiledScript
loadInSchema(String):String
validateSchema(String,String):boolean
JSONUtility()

Figure 5-13 - JSON utilities class diagram

5.4.5.2 Utilities

The verification tool relies heavily on the use of a number of utility functions which are split into a number of different areas including reading and writing files, performing cryptographic operations and processing messages.

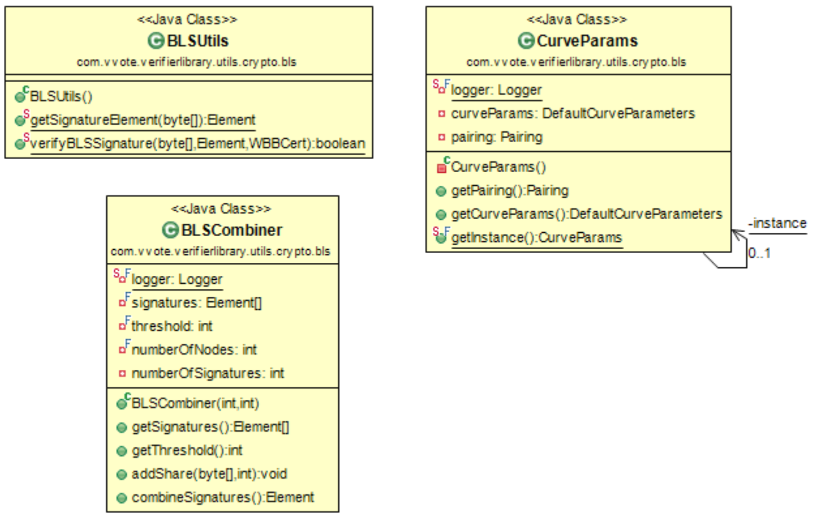
Cryptographic Primitives

The vVote Verifier system relies on the use of cryptographic primitives from verifying encryptions and re-encryptions to checking threshold signatures. The use of cryptographic primitives is integral to the successful verification of the vVote system. They are also a key area of improved understanding gained through the completion of the project where theoretical cryptographic concepts learnt throughout the MSc have been applied and put into practice.

The project required the implementation of functions providing the ability to open SHA-256 hash based commitments verifying them at the same time. For this a number of different methods were used to gain increased confidence in the commitments and remove any reliance on any one library. The initial aim was to carry this out for all cryptographic operations however the timescale did not permit this. For verifying hash commitments, three paths were created. The first method utilises Java Cryptography Architecture (JCA) simply using a Message digest and updating the random value and witness value therefore concatenating the values together and checking whether the result is what was expected. The second method utilises the bouncy castle cryptographic libraries. A HashCommitter is constructed using the witness value and a new hash digest. A Commitment is then constructed using the commitment value and the witness value. A check is then made to determine whether the witness value reveals the commitment made against the random value. The third method utilises the OpenSSL libraries. There are no OpenSSL Java bindings and therefore the Java Native Interface (JNI) was used to call compiled c++ code utilising the OpenSSL libraries to carry out the hash commitment verification. The reason for choosing to additionally use OpenSSL on top of the Java based JCA and bouncy castle libraries was due to the sharing of libraries between the verification tool and the original vVote system. The use of OpenSSL was to gain more confidence. In addition to the more complex verification of hash based commitments the system also needs to be able to carry out simpler hashing of data utilising the JCA's MessageDigest object taking as input the data in bytes. Along with hashing of simple data the system also needs to be able to hash entire files in blocks. Hashing files utilises the DigestInputStream class along with the MessageDigest and BufferedInputStream classes. Figure 5-15 shows the class diagram used for hashing files and checking hash based commitments.

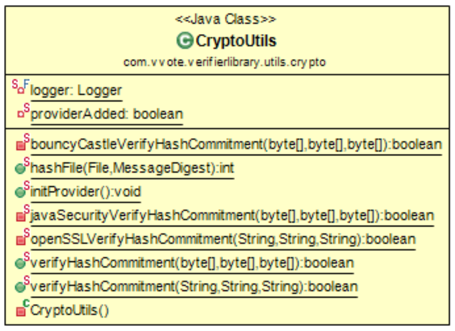
The verification tool also needs to provide the ability to perform validation of a BLS signature made over a specific set of data where the BLS signature was made using the components' private key. The verification requires the use of the same components' public key. As well as requiring the ability to perform validation of BLS signatures the system also needs to be able to combine a number of publicly known partial BLS signatures together verifying that a threshold number of signatures has been provided using a Lagrange Weight calculation along with the partial signatures [40]. Figure 5-14 shows the class diagram used for combining together BLS signatures and performing verification of a BLS signature.

Finally a key cryptographic primitive used throughout the verifier components is that of an ElGamal Elliptic Curve Cryptographic system. The crypto system is used to verify both encryptions and re-encryptions of plaintext Elliptic Curve points and ElGamal Elliptic Curve Point ciphertexts respectively. The ElGamal Elliptic Curve Cryptographic system uses two bouncy castle ECPoint objects; with the first ECPoint representing $m * y^r$ and the second representing g^r. The curve used throughout the majority of operations is P-256. In addition to constructing ElGamal Elliptic Curve points the system also needs to be able to encrypt a single ECPoint representing a plaintext producing an ElGamal Elliptic Curve points and also to re-encrypt an ElGamal Elliptic Curve point producing another ElGamal Elliptic Curve point. Figure 5-16 shows the class diagram used for providing an ElGamal Elliptic Curve Cryptographic system. The class is heavily used throughout the system.



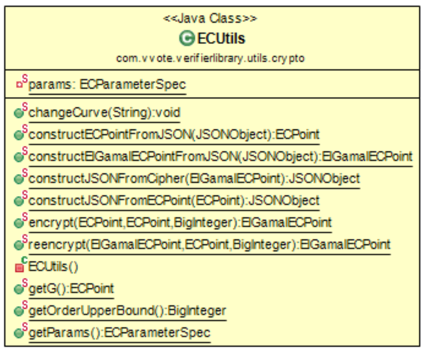
<<Java Class>>
BLSUtils
com.vvote.verifierlibrary.utils.crypto.bls
BLSUtils()
getSignatureElement(byte[]):Element
verifyBLSSignature(byte[],Element,WBBCert):boolean
<<Java Class>>
CurveParams
com.vvote.verifierlibrary.utils.crypto.bls
logger: Logger
curveParams: DefaultCurveParameters
pairing: Pairing
CurveParams()
getPairing():Pairing
getCurveParams():DefaultCurveParameters
getInstance():CurveParams
-instance
0..1
<<Java Class>>
BLSCombiner
com.vvote.verifierlibrary.utils.crypto.bls
logger: Logger
signatures: Element[]
threshold: int
numberOfNodes: int
numberOfSignatures: int
BLSCombiner(int,int)
getSignatures():Element[]
getThreshold():int
addShare(byte[],int):void
combineSignatures():Element

Figure 5-14 - BLS signature utilities class diagram



<<Java Class>>
CryptoUtils
com.vvote.verifierlibrary.utils.crypto
logger: Logger
providerAdded: boolean
bouncyCastleVerifyHashCommitment(byte[],byte[],byte[]):boolean
hashFile(File,MessageDigest):int
initProvider():void
javaSecurityVerifyHashCommitment(byte[],byte[],byte[]):boolean
openSSLVerifyHashCommitment(String,String,String):boolean
verifyHashCommitment(byte[],byte[],byte[]):boolean
verifyHashCommitment(String,String,String):boolean
CryptoUtils()

Figure 5-15 - Crypto utilities class diagram



<<Java Class>>
ECUtils
com.vvote.verifierlibrary.utils.crypto
params: ECParameterSpec
changeCurve(String):void
constructECPointFromJSON(JSONObject):ECPoint
constructElGamalECPointFromJSON(JSONObject):ElGamalECPoint
constructJSONFromCipher(ElGamalECPoint):JSONObject
constructJSONFromECPoint(ECPoint):JSONObject
encrypt(ECPoint,ECPoint,BigInteger):ElGamalECPoint
reencrypt(ElGamalECPoint,ECPoint,BigInteger):ElGamalECPoint
ECUtils()
getG():ECPoint
getOrderUpperBound():BigInteger
getParams():ECParameterSpec

Figure 5-16 - EC Utilities class diagram

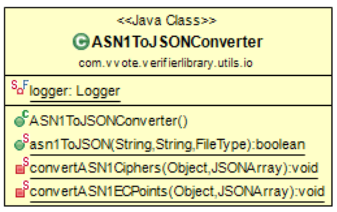
File Reading

The verifier system is designed to be run after the election has completed and after the point where all the data has been posted successfully to the Public WBB and therefore a large part of the system is the reading in of data in a meaningful way. Much of the data posted to the Public WBB is in JSON format however there are also configuration files with simple key: value pairs. The Mixnet additionally utilises the Abstract Syntax Notion One (ASN.1) file format [41]. ASN.1 is a standard data representation format designed to be independent of platform specific encoding techniques. It uses formal notation to allow validation of the data against its own specification. It provides the ability to produce a fixed format for abstract data types.

JSON file reading is carried out simply by reading a String from file and then constructing the expected format directly. ASN.1 file reading is slightly more difficult and utilises an intermediate conversion step which converts the ASN.1 file into JSON format which is then read in as usual. There are two types of ASN.1 files which require reading in for the project. The first type is ASN.1 plaintexts files containing sequences of sequences of sequences of ECPoint objects representing rows of lists of plaintext candidate identifiers. The second type is ASN.1 ciphers files containing sequences of sequences of sequences of pairs of ECPoint objects representing rows of lists of encrypted plaintext candidate identifiers. Each file will contain lists of a number of packings representing lists of voter preferences as [vote1,vote2,vote3, etc.] where vote1, vote2 and vote3 are equal to [packing1, packing2, packing3, etc.].

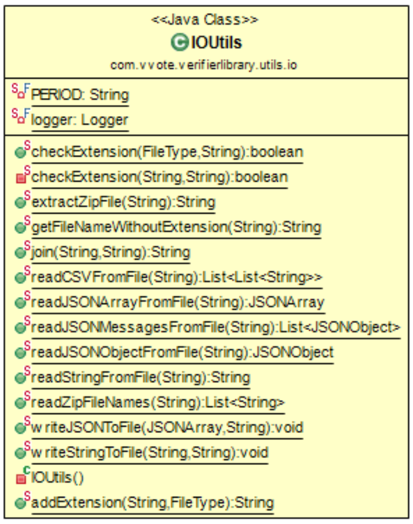
Additionally configuration files are used which are simple key: value pairs. For reading in these files in a simple and efficient manner the PropertiesConfiguration class was used from Apache Commons [42].

Figure 5-17 shows the class diagram used for the intermediate ASN.1 file to JSON file conversion step and Figure 5-18 shows the IO utility class diagram used for reading and writing JSON files in addition to extracting ZIP files.



<<Java Class>>
ASN1ToJSONConverter
com.vvote.verifierlibrary.utils.io
logger: Logger
ASN1ToJSONConverter()
asn1ToJSON(String,String,FileType):boolean
convertASN1Ciphers(Object,JSONArray):void
convertASN1ECPoints(Object,JSONArray):void

Figure 5-17 - ASN.1 to JSON conversion class diagram


<<Java Class>>
IOUtils
com.vvote.verifierlibrary.utils.io
PERIOD: String
logger: Logger
checkExtension(FileType,String):boolean
checkExtension(String,String):boolean
extractZipFile(String):String
getFileNameWithoutExtension(String):String
join(String,String):String
readCSVFromFile(String):List<List<String>>
readJSONArrayFromFile(String):JSONArray
readJSONMessagesFromFile(String):List<JSONObject>
readJSONObjectFromFile(String):JSONObject
readStringFromFile(String):String
readZipFileNames(String):List<String>
writeJSONToFile(JSONArray,String):void
writeStringToFile(String,String):void
IOUtils()
addExtension(String,FileType):String
Figure 5-18 - IO class diagram

5.5 Verification Implementation Design

In 4.3.5 vVote Verifier a high level analysis was undertaken for the entire verification system as well as for each of the component verification areas. The system was looked at from the context level, outlining the inputs and outputs of the system at the highest level. Next, each of the component verification areas were analysed and an overview was provided as to why the area of the vVote system needed verification as well giving an outline of how the verification might be carried out.

In this section a more detailed examination will take place with regard to the component verifiers and will provide a step by step design for the algorithms used for the verification process.

The vVote Verifier will be the combination of a number of independent verifiers each of which can be run in isolation when provided with the Public WBB data. The vVote Verifier will be run in such a way that by simply modifying a configuration file the component verifiers can be selected or removed. The configuration file even allows for a mechanism whereby additional component verifiers could be provided to the system if they follow the same structure as those already existing without requiring modification of the existing source code. This would, for example, allow for the addition of a component verifier for the Mixnet which would simply need to be pointed towards within the configuration file without requiring any additional modifications to source code. Currently the vVote Verifier will carry out the verification of the Ballot Generation process, Vote Packing process, both pre-mixnet and post-mixnet and finally verification of the commitments made to the Public WBB.

5.5.1 Verifiable Components

This section will provide a step by step overview of the algorithms used for the verification process for each of the chosen verifiable components. Appendix I provides additional pseudo code for each of verification algorithms which was used to help with the implementation of the algorithms.

5.5.1.1 Ballot Generation

1. **For each PoD Printer, perform validation of the Fiat-Shamir signature using the publicly available information using it to verify that the correct ballots were chosen to be part of the auditing process**:
   - The Fiat-Shamir signature is a SHA256 hash containing the following values in the specified order:
     i. The PoD Printer device Identifier. E.g. "PrinterA".
     ii. The message identifier for the file containing the generated generic ballots published on the Public WBB.
     iii. The time of the commitment to the Public WBB for the message containing the generic ballots.
     iv. A hash of the generic ballots file.
     v. The combined WBB peer signature which is a threshold combination of the WBB Peers partial BLS signatures.
   - The re-calculated Fiat-Shamir signature is then checked against the Fiat-Shamir signature included in the BallotSubmitResponseFile.json which is provided in response to the submission of the generic ballots to the Public WBB.
   - Check that by using the verified Fiat-Shamir signature the same set of ballots can be deterministically chosen for the auditing process which ensures the PoD Printer has not pre-selected any ballots for auditing and the process was completely random:
     i. Create a sorted list of the available serial numbers used as part of the Ballot Generation process.
     ii. Construct a Deterministic Random Bit Generator (DBRG) SP800 [43] using the PoD Printer device identifier as the personalisation string with the re-calculated Fiat-Shamir signature used as the source of randomness.
     iii. Shuffle the sorted list of serial numbers of all available ballots using the DBRG as the source of randomness for the shuffle.
     iv. Take the first a ballots as those chosen for auditing and compare the serial numbers against those chosen for auditing and published in the file AuditDataFile.json.

2. **Verify the randomness data stored by the PoD Printers for all ballots chosen for auditing matches the commitments made by the G randomness generation servers on the Public WBB prior to sending RT_i to the PoD Printer** – The PoD Printer, for each ballot chosen for auditing, will open their commitments made to the randomness pairs received from each of the G RT_i tables. Each of the randomness values will be checked for consistency against the commitments made by the G randomness generation servers on the Public WBB prior to sending RT_i to the PoD Printer within table CRT_i. This process can only be carried out once the commitments to the randomness values have been opened by the PoD Printer and will only happen for ballots chosen for auditing. The ballots are chosen for auditing during the Ballot Generation process randomly and cannot then be used for voting purposes:
   - For each ballot, each randomness generation server $RGen_i$ will generate enough randomness pairs for each candidate + 1.
   - Each line of the file AuditDataFile.json relates to a single ballot chosen for auditing. The line consists of the ballot serial number and G sets of $n + 1$ randomness pairs. Each set of $n + 1$ randomness pairs will originate from a different randomness generation server which will also be specified. Within each randomness pair the randomness value is identified using the key “r” and the witness value is identified using the key “rComm”.
   - For each of the G sets of $n + 1$ randomness pairs the corresponding set of commitments made by each of the randomness generation servers $RGen_i$ within their commitData.json file are verified. Each pair consisting of “r” and “rComm” are used to verify the corresponding commitment posted on the Public WBB by the current $RGen_i$ in table CRT_i. Each commitment is checked by hashing together, using the SHA256 algorithm, the random value “r” and the witness “rComm” and checking they match the corresponding commitment value for the current randomness generation server. The hashing together is a concatenation of the values with the hash algorithm applied after.

3. **Combine together each of the G randomness values “r” for each of the $n + 1$ randomness pairs**:
   - Concatenate together each of the G randomness values “r” for each of the $n + 1$ randomness pairs from the opened randomness value to form $n + 1$ combined random values for each ballot to audit.

4. **Perform re-encryption and sorting, in a per race basis, of the base encrypted candidate identifiers using n of the combined randomness values**:
   - For each of the n candidates there will be a corresponding base encrypted candidate identifier in addition to the newly combined randomness value.
   - Each of the n base encrypted candidate identifiers are re-encrypted using the corresponding combined randomness value. For each of the n base encrypted candidate identifiers and each corresponding combined randomness values:
     1. Take the current base encrypted candidate identifier represented by the two Elliptic Curve Points:
        - The first point is g^r and the second element is my^{r}.
     2. Take the public key represented by a single Elliptic Curve Point.
     3. Take the current combined randomness value.
     4. Re-encrypt the first Elliptic Curve Point of the current base encrypted candidate identifier:
        - Multiply the generator value of the underlying Elliptic Curve by the combined randomness value and add it to the first point g^r.
     5. Re-encrypt the second Elliptic Curve Point of the current base encrypted candidate identifier:
        - Multiply the public key by the current combined randomness value and add it to the second point my^{r}.
   - Sort, in a per race basis, the resulting re-encrypted candidate identifiers by comparing their underlying Elliptic Curve Points.

5. **Reconstruct the ballot permutation as shown in Table 0-6 in Appendix H which represents the final shuffling order of the generic ballot produced using the candidate identifier's original position**:
   - The recalculated permutation will be used to verify the order of ballots printed for voters.

6. **Compare the list of newly re-encrypted and sorted candidate identifiers with the matching generic ballot posted during the Ballot Generation process on the Public WBB**:

- Compare each recalculated re-encrypted candidate identifier with the corresponding re-encrypted candidate identifier previously generated within the corresponding ciphers.json file.

7. **Perform a commitment check on the stored permutation commitment on the Public WBB within the ciphers.json file. The permutation must be checked to ensure it was generated honestly as it is used when printing the correct order for a physical ballot. Assurance is needed that the permutation commitment actually matches what was submitted**:
    - The final, unused, combined randomness value is used as the witness value in the commitment check.
    - The generated permutation produced in step 5 and the witness value are hashed together to check the permutation is valid.

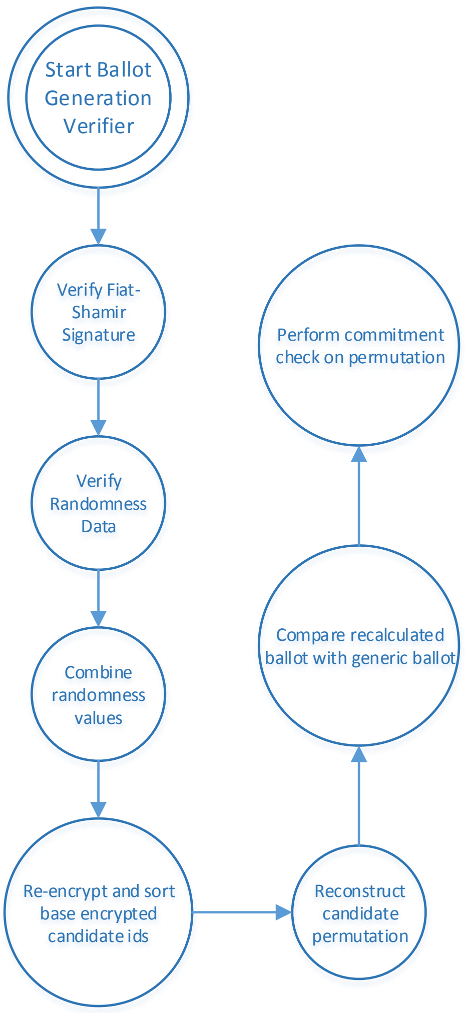

Start Ballot Generation Verifier
Verify Fiat-Shamir Signature
Verify Randomness Data
Combine randomness values
Re-encrypt and sort base encrypted candidate ids
Reconstruct candidate permutation
Compare recalculated ballot with generic ballot
Perform commitment check on permutation

Figure 5-19 - Ballot Generation Flow Diagram

5.5.1.2 Vote Packing

The following steps are carried out for each submitted vote:

1. **Get the voters' preferences for each race as LA preferences, LCATL preferences and LCBTL preferences.**

2. **Lookup the generic ballot sharing the same serial number as that of the vote message.**

3. **Carry out the necessary ballot reductions for the generic ballot for the voters district using the corresponding ballot reductions associated with the submitted vote:**
    - Verify each of the accompanying ballot reductions:
        i. Verify that the base encrypted candidate identifier at the specified "candidate index" re-encrypted using the specified randomness matches the re-encrypted candidate identifier at the "index" specified. Both the index value and the candidate index value will specified in a per generic race basis and therefore adjustments may be required when dealing with LCATL and LCBTL reductions using the sizes of the previous generic races.

4. **Reorder the reduced ballot using the preferences within the submitted vote message:**
    - The reordering is carried out on a per race basis whereby the LA preferences are used to reorder the LA candidate identifiers in the reduced ballot etc.
    - Each submitted vote may contain preferences for either LC ATL or LC BTL but cannot contain both.
    - The reordered ballots do not include preferences for the candidates where no preference has been supplied or race preferences where no preferences have been supplied.

5. **Pack the reordered ballot according to the preferences in a per race basis.**
    - The re-encrypted base encrypted candidate identifiers will have been sorted into preference order in a per race basis.
    - The re-encrypted base encrypted candidate identifiers are multiplied by the packing preference which loops from 1 to the race packing size.
    - Each set, with size equal to the race packing size, of multiplied candidate identifiers is then added together.

6. **Add padding to the packing values to ensure each column in each race contains the same number of ciphers as any other packing values in that specific race.**
   - If almost every voter provides 6 preferences for a particular race when the packing size is 3 and a single voter provides 7 preferences:
     i. For each ballot submitted with 6 preferences the result will be that there are 2 ciphertexts. This ciphertext has 2 columns.
     ii. For the single ballot submitted with 7 preferences the result will be 2 fully packed ciphertexts and another packing containing only the single extra candidate identifier multiplied by the preference 1. This ciphertext has 3 columns.
     iii. For each set of ciphertext packings with 2 columns an additional padding ciphertext will need to be added to increase the columns from 2 to 3.
   - The padding ensures that all packed ciphertexts for any specific race contains the same number of columns as is required by the Mixnet before shuffling.
   - The padding used for packing the ciphertexts should be a base encryption of the Elliptic Curve padding point contained in the paddingpoint.json file.
   - Each of the different races may contain different numbers of columns however all sets of ciphertext packings within the same race must contain the same number of columns.

7. **At this point the recalculated padded packed ciphertext values should match exactly the packings that were provided to the Mixnet during the election tallying phase.**
   - This verification step matches the first check for vote packing verification checking that the ciphertexts have been correctly packed according to the underlying votes and that the recalculated packed encryptions match those that were input to the Mixnet (blt files)
   - This step verifies the pre-mixnet operation of the vVote system.

8. **Lookup the plaintext identifiers corresponding to each output csv file with regard to its district configuration and race and reorder the plaintext identifiers by the preferences specified in the output csv files. The csv files will contain the preferences in a per race basis i.e. a single csv file will contain the preferences for a specific race in a specified district.**

- The output csv files simply contain the initial preferences supplied to the Mixnet in numerical form.
- The preferences directly map back to the original ordering of the plaintext identifiers and include blanks where no preference was supplied for a specific candidate.
- Example preferences: “,2,,3,1” for an example set of plaintext candidate identifiers: “a,b,c,d,e” will result in the following reordering according to the preference values:
    i. Reordering will be: “e,b,d”.
    ii. Again the candidates where no preferences have been supplied are not included in the reordering.

9. **Pack the plaintext candidate identifiers.**
    - The plaintext candidate identifiers will have been sorted into preference order.
    - The plaintext candidate identifiers are multiplied by the preference which loops from 1 to the race packing size.
    - Each set, with size equal to the race packing size, of multiplied candidate identifiers is then added together.

10. **Add padding to the packed plaintext identifiers in a manner similar to that described in step 6 so that each set of packed voter preferences contain the same number of columns for the same race configuration.**
    - This step mirrors step 6. Each of the original ciphertext packings will be mirrored as decrypted plaintext packings meaning the number of columns input into the Mixnet will match the number of columns of the Mixnet output values for any specific race and district configuration.

11. **At this point the packed plaintext candidate identifiers should correspond to the actual values looked up, during the election tallying phase, from the large lookup tables containing the all of the possible packings. We check that the raw decrypted value from the Mixnet equals the values calculated in step 10.**
    - This check verifies that the claimed candidate preferences were genuinely looked up using the large lookup table without requiring the whole table to be recalculated.
    - This step corresponds to the second check for the verification process checking that the decryptions from the Mixnet correctly match the claimed plaintext packing values which were looked up from the lookup tables.

- This step verifies the post-mixnet operation of the vVote system and verifies that all the preferences from the submitted ballots passed to the Mixnet are still present after the shuffling and decryption.

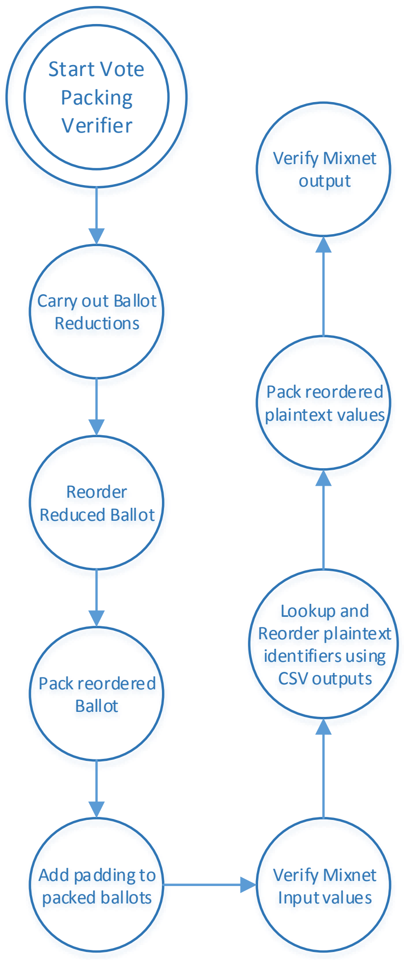

Start Vote Packing Verifier
Carry out Ballot Reductions
Reorder Reduced Ballot
Pack reordered Ballot
Add padding to packed ballots
Verify Mixnet Input values
Lookup and Reorder plaintext identifiers using CSV outputs
Pack reordered plaintext values
Verify Mixnet output

Figure 5-20 - Vote Packing Flow Diagram

5.5.1.3 Public WBB Commits

1. **The files within the commits folder need to be grouped together according to the commit time:**
    - For each JSON signature file use the file names included to group together the related parts of the commit including the JSON message file and the attachment ZIP file.

The following steps are carried out for each commitment group.

2. **Construct a new SHA1 hash object h.**

3. **For each message in the JSON message file:**

- Add the message's internal signable content to the hash h.
- If the current message is a file message then also hash the corresponding file and add it to the hash h.

4. **Construct a joint signature SHA1 hash object sig and add the following fields to it:**
   - "Commit"
   - The commit time for the current commitment group.
   - The SHA1 hash object h.

5. **Get the joint signature from the JSON signature file as $joint\_signature$.**

6. **Verify that the joint signature SHA1 hash object sig is equal to the data that was signed in the provided signature $joint\_signature$.**
   - The Public WBB public key is used for the verification.

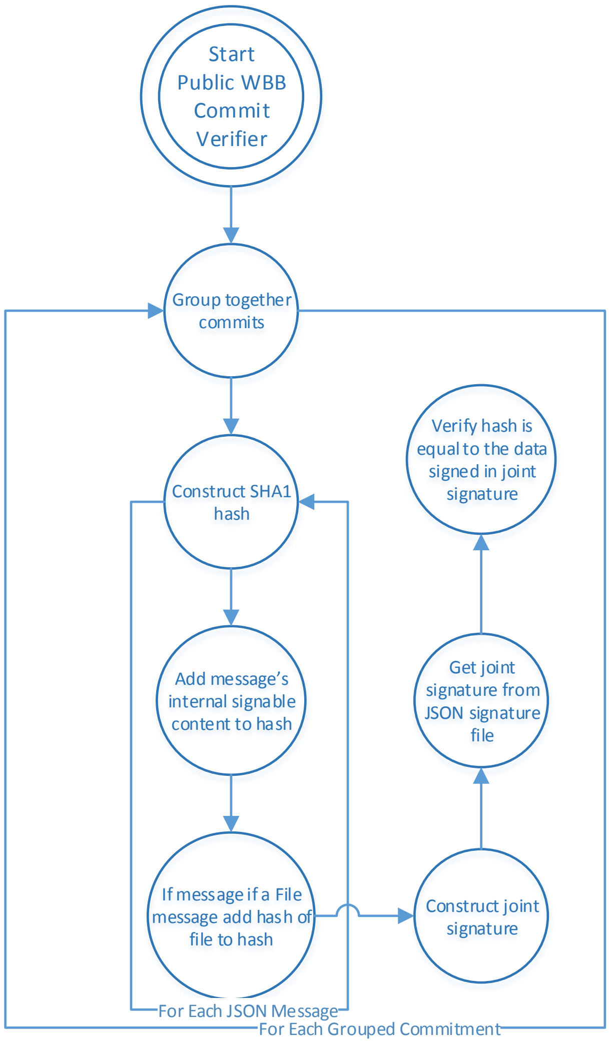

Start Public WBB Commit Verifier
Group together commits
Construct SHA1 hash
Add message's internal signable content to hash
If message if a File message add hash of file to hash
Construct joint signature
Get joint signature from JSON signature file
Verify hash is equal to the data signed in joint signature
For Each JSON Message
For Each Grouped Commitment

Figure 5-21 - Public WBB Commits Flow Diagram

5.6 Conclusion

In this chapter the design of the vVote Verifier system has been examined. The chapter outlined the development environment used in terms of operating system, tools, languages and libraries used for the system. The chapter then outlined the high level architecture to be used for the system as well as expanding on the high level architecture to provide a lower-level design of the system. Finally the chapter provided a more detailed description for the design on the verification algorithms which will be used.

The next chapter will look at the implementation of the system using the proposed design provided in this chapter.

Chapter 6 – vVote Verifier Implementation

6.1 Introduction

The previous chapter was used to provide an overview of the design for the vVote Verifier system. It provided both a high level architecture design as well as a lower layer system design. The chapter showed how the system would be split into modules and also described a number of classes which would be used. After the design portion of the project the implementation of the system took place whereby the design was used to produce a concrete implementation which would be able to meet the requirements set out in Table 4-5 and Table 4-6.

This chapter will be used to focus on the most interesting and complex areas of the development of the vVote Verifier and will also include a number of the challenges faced during its implementation.

6.2 Verifier Library Implementation

In Chapter 5 it was explained how the use of a verifier library would underpin the operations of other core aspects of the system by developing isolated reusable portions of code. The verifier library can be seen as one of the most integral parts of the entire system for providing correct verification and reliable functionality. This section will be used to outline some of the most important and complicated implementation details for the library as well as some issues faced with their development.

6.2.1 File Reading

Reading data from file is an important aspect of the system as a significant amount of data is read and processed initially. The following section outlines some of the more interesting parts of the file reading process.

6.2.1.1 ASN.1 file conversion to JSON

Even though the Mixnet was not chosen for verification the Vote Packing verifier relies on reading in input files which contain pre-processed packed re-encrypted candidate identifiers ready for the Mixnet and output files containing shuffled and decrypted packed plaintext candidate identifiers. Both of these files are in the ASN.1 file format due to the heavily utilisation of the file type throughout the Mixnet and therefore a way of reading these files was needed. The approach taken for developing an ASN.1 file reader was to use an initial conversion step taking the files from ASN.1 format to JSON format. The converter approach was chosen because of the difficulty in analysing the ASN.1 files. ASN.1 files can only be read and examined when using other tools such as OpenSSL using the asn1parse function from the command line and therefore they can’t easily be analysed like a JSON file. The conversion to JSON means that the files can be easily examined using any standard text editor which can be helpful during development.

The ASN.1 to JSON conversion function requires that the structure of the files is known beforehand. In the vVote Verifier we are only interested in two types of ASN.1 files; these are Mixnet input files corresponding to rows of lists of ElGamal Elliptic Curve Points representing packed re-encrypted candidate identifiers and Mixnet output files which correspond to ECPoint objects representing rows of lists of packed plaintext candidate identifiers. The files are read in as an ASN1InputStream which is then processed per object as ASN1Primitive’s. Each of these ASN1Primitive’s is then converted individually using a recursive function.

In the case of Mixnet output files, which in the vVote system have the file type .out, the file will be constructed using sequences of ASN1Primitive objects which represent the shuffled votes containing sequences of sequences of plaintext packed candidate identifiers representing the voter preferences for each of the ballots. A recursive function is used to process each sequence of plaintext packings with the base case being the construction of a JSON plaintext candidate identifier from a DEROctetString. This process can be seen in the ASN1toJSONCoverter class within the package com.vvote.verifierlibrary.utils.io.

Mixnet input files, which have the .blt file type, are structured in a similar way to Mixnet output files as sequences of ASN1Primitive objects each representing packings of the original voter preferences with each then containing sequences of sequences of re-encrypted packed candidate identifiers. Again a recursive function is used to process the sequences of re-encrypted packings with the base case being the construction of an ElGamal Elliptic Curve Point from a DEROctetString. The recursive case then works on each part of the sequence until the base case is met. This process can be seen in the ASN1toJSONCoverter class within the package com.vvote.verifierlibrary.utils.io.

6.2.1.2 CSV file Reading

The Mixnet shuffles ciphertext packings and then decrypts the results leaving plaintext packings of the candidate identifiers. These plaintext packings are then looked up from the lookup tables to determine the true voter preferences (this process is verified using the Vote Packing verifier). The

Mixnet reorders the preferences looked up from the tables into the original plaintext candidate identifier order. The reordering will only include the number of plaintexts required for the specific district configuration. Table 6-1, Table 6-2 and Table 6-3 show the way the CSV output files should be deciphered. Table 6-1 shows the original full list of plaintext candidate identifiers. Table 6-2 contains the district configuration for a specific district. It contains the number of candidates running in each race for the district. Table 6-3 shows an example of an output CSV file. Each of the elements in the CSV output column would be included in the file as seen in separate rows. When reducing the original list of plaintext candidate identifiers using the district specific race size the first x candidate identifiers will be chosen where x is the current race size. The vVote Verifier uses a CSVReader [44] library to reduce the amount of code required.

Candidate	**Candidate 1**	**Candidate 2**	**Candidate 3**	**Candidate 4**	**Candidate 5**	**Candidate 6**
LA Plaintexts Candidate Identifier	Plaintext1	Plaintext2	Plaintext3	Plaintext4	Plaintext5	Plaintext6

Table 6-1 - Mixnet example - original plaintexts

Race	**LA**	**LCATL**	**LCBTL**
Northcote District Race Size	4	5	6

Table 6-2 - Mixnet example - district configuration

Northcote LA Mixnet CSV output	
CSV output	**Equivalent Voter preferences**
3,1,2,	Plaintext2, Plaintext3, Plaintex1
1,3,4,2	Plaintext1, Plaintext4, Plaintext2, Plaintext3
,1,,	Plaintext2

Table 6-3 - Mixnet example - CSV output

6.2.1.3 ZIP file Extraction

The vVote system uses a method whereby each commit made to the Public WBB has an accompanying ZIP file. For each additional File message within the JSON message file the outer ZIP file will then contain an additional inner ZIP file. Java provides a standard mechanism for reading files within ZIP files using the ZipFile and ZipEntry classes however these do not work correctly for nested ZIP files and therefore the decision was made to write a utility function to fully extract a ZIP file to a folder in a similar manner to how a ZIP file would generally be extracted from a file manager. The ZIP file extraction process can be seen in the IOUtils class in the com.vvote.verifierlibrary.utils.io package.

6.2.2 Cryptographic primitives

The cryptographic primitives implemented make up part of the backbone of the verifier system and will be heavily utilised and relied upon for the successful verification of the components chosen. The following section outlines some of the more interesting parts of the cryptographic operations carried out.

6.2.2.1 Hash Commitments

As discussed in 5.4.5.2 Utilities the vVote Verifier will use the combination of three independent Hash commitment checks to gain increased confidence in their verification. The first method is a simple and reliable use of JCA. The process is carried out by concatenating together the witness value and the random value and then hashing the result. This hash will then be compared against what was published on the Public WBB. The second method uses a number of Bouncy Castle libraries specifically designed for testing hash based commitments. The process is carried out by using a HashCommitter class constructed using the witness as the secure random value. A Commitment object is then constructed using the witness and commitment values which states that the witness value should open the commitment value to reveal some randomness value. The final step is a check to determine whether the provided randomness value is revealed by this Commitment object. The final method utilises OpenSSL libraries. OpenSSL is written using the c programming language and cannot therefore be used directly by the JAVA based vVote Verifier. The method chosen for utilising the OpenSSL libraries was through the use of JNI to call natively compiled c++ code which utilises the OpenSSL libraries. The JAVA code can use the functionality of the natively compiled c++ code to carry out the hash commitment check using a completely independent library. The OpenSSL based hash commitment verification specifically uses the openssl/sha.h library giving access to a number of SHA based hash functions. A SHA256_CTX context object is initialised using the SHA256_Init function and then used to carry out the concatenation and hashing operation. The SHA256_CTX context first has the witness value added and then the randomness value by calling the SHA256_Update function; the context is then used to create the final unsigned char hash using the SHA256_Final. This hash is then compared against the commitment.

The JCA based hash commitment verification function as well as the Bouncy Castle based hash commitment verification function can be seen in the CryptoUtils class in the com.vvote.verifierlibrary.utils.crypto package. The OpenSSL implementation of the hash commitment verifier can be seen in Appendix G.

6.2.2.2 File Hashing

When verifying the commits made to the Public WBB if any of the messages within the JSON message file is a file message then the corresponding attachment file must also be hashed and added to the hash for the entire commit. Neither Java nor Bouncy Castle provided standard implementations for calculating a hash over an entire file and therefore development of a utility function to carry out this calculation was required. There is the possibility of requiring hashing of very small files (KBs) as well as very large files (GBs) and therefore only 1024 bytes are added to the hash at a time instead of trying to hash the whole file where issues could be faced. The hash function takes as arguments the file to hash as well as a pre-initialised MessageDigest into which the hash of the file is placed. Then the file is added to the hash 1024 bytes at a time. The implementation of the File hashing function can be seen in the CryptoUtils class in the com.vvote.verifierlibrary.utils.crypto package.

6.2.2.3 ElGamal Elliptic Curve Cryptographic System

The implementation of an ElGamal Elliptic Curve Cryptographic system is of paramount importance to the successful verification of the vVote system as discussed in 4.3.4 Cryptographic Primitives which culminates in the definition of the crypto system encryption as $\mathrm{Enc}_{(g,y)}(m,r) = (g * r, m + y * r)$ where (g, y) is the public key, m is the message being encrypted and r is the randomness value used. In 5.4.5.2 Utilities more details were provided as to the requirements of and the design of the crypto system including the requirement to encrypt plaintext Elliptic Curve points and ElGamal Elliptic Curve Point ciphertexts i.e. carrying out re-encryption of the ciphertexts. Re-encryption of a ciphertext is slightly different to the encryption of a plaintext and is represented by $Reenc(c,r) = (c.gr + g * r_1, c.myr + y * r_1)$ where (g, y) is the public key, c is the ciphertext originally encrypted under randomness value r with two points gr and myr which is being re-encrypted using the new randomness value r_1. The implementation of the ElGamal Elliptic Curve Cryptographic System can be seen in the ECUtils class in the com.vvote.verifierlibrary.utils.crypto package.

6.2.2.4 Threshold BLS Cryptography

The vVote system uses threshold BLS signatures to gain more confidence in the systems operations as a threshold number of Private WBB peers must come together in agreement before an action may take place and be committed to. The vVote Verifier requires the ability to both combine together partial BLS signatures and to verify that a threshold has been reached using the provided shares; it must also take an existing BLS signature and verify that the underlying signed data was signed correctly.

Verifying the Fiat-Shamir signature during the Ballot Generation verification, outlined in 5.5.1.1 Ballot Generation, requires the combination of a threshold number of WBB Peers' partial BLS signatures. A number of partial BLS signatures are obtained from a the ballot submit response message, received after the posting of the ballot generation commit message; these partial BLS signatures along with a set of peer index numbers, taken from the certs.bks file, are combined together. Only when a threshold number of peers have provided their partial signature shares can the shares be successfully combined together. A Lagrange Interpolation is used to calculate a Lagrange Weight over the provided partial signatures. These weighting values are then used to multiply the current partial BLS signature by its own weight and also multiplied by the current combined signature. In the Ballot Generation process the partial signatures are combined to form a thresholded BLS signature. This value is then added to a recalculated Fiat-Shamir signature which is then compared against one included on the Public WBB. The verification of the combination of partial BLS signatures can be seen in the BLSCombiner class in the com.vvote.verifierlibrary.utils.crypto.bls package.

In addition to combining partial signature shares the vVote Verifier must also have the ability to verify that a provided BLS signature has been calculated over the correct data set. The verification will utilise the public key of the entity upon which the signature check is being carried out which is the Public WBB. In the case of the vVote system this public key will be taken from the certs.bks file which includes the Public WBB public key in addition to Private WBB peer public keys and partial public keys. The verification of the signature is used in the Public WBB Commit verifier for verifying the authenticity of each of the commits made to the Public WBB. For each commit the joint signature is recalculated and compared against the BLS signature included in the signature JSON message for that commit. The comparison is to check that the recalculated joint signature matches the data which was signed and included in the signature JSON message. The implementation for verifying that the provided signature was used to sign the provided hash using the corresponding private key to the input public key can be seen in the BLSUtils class in the com.vvote.verifierlibrary.utils.crypto.bls package.

6.3 Verifier Component Implementation

The verifier components for the Ballot Generation, Vote Packing and Public WBB commitment processes are the core aspect of the verifier system and carry out the algorithmic steps and proofs required to gain confidence in the results produced by the vVote system. This section will not be used to outline every detail relating to the component verifiers and will instead focus on the most important and interesting aspects of the verifiers mainly focusing on the requirements listed in Table 4-5.

6.3.1 Ballot Generation

The explicit steps used for proving the correct operation of the Ballot Generation process within the vVote system can be seen in 5.5.1.1 Ballot Generation.

6.3.1.1 Functional Requirement F 1 - Verification of the Fiat-Shamir signature calculation

For each of the PoD Printers and their Ballot Generation audit commitments we need to verify that the Fiat-Shamir signature, used to select the ballots for auditing, was generated correctly. The Fiat-Shamir signature is included in a Ballot Generation ballot submit response message corresponding to the ballot generation commit message for each PoD Printer. The first step in verifying that the Fiat-Shamir signature was calculated and used correctly is recalculating the signature using publicly available information and verifying that it matches what was included in the message. The implementation used for recalculating the signature can be seen in the BallotGenerationVerifier class in the verifySignatureMatches function in the com.vvote.verifier.component.ballotGen package.

6.3.1.2 Functional Requirement F 2 - Verify that the validated Fiat-Shamir signature was correctly used for selecting the ballots for auditing

The validated Fiat-Shamir signature also needs to be checked that it was correctly used as the seed in the selection of ballots chosen for auditing. The Fiat-Shamir signature is used as the seed for a FixedSecureRandom which is then used as the randomness provider for a Deterministic Random Bit Generator (DBRG) SP800. The PoD Printer identifier is used as the personalisation string for the SP800SecureRandomBuilder which is then used to construct an SP800SecureRandom which is used to shuffle the ballot serial numbers which were generated for a particular Ballot Generation commit by the PoD Printer that is being verified. The serial numbers before this point have been sorted into alpha-numerical order. The first a ballots are then chosen and checked to ensure that these were the same ballots selected by the PoD Printer and published on the Public WBB. The implementation used for verifying that the selected ballots were correctly chosen can be seen in the BallotGenerationVerifier class in the verifyFiatShamirCalculation function in the com.vvote.verifier.component.ballotGen package.

6.3.1.3 Functional Requirement F 3 – Verify Randomness Commitments

For each of the ballots chosen for auditing the randomness values, previously secret but opened and published onto the Public WBB by the PoD Printer, must be checked against the commitments made by the randomness generation servers prior to the election. Each of the ballots chosen for auditing will have a list of corresponding randomness pairs consisting of a random value and a witness value. The witness and random value must be concatenated together and the result hashed with this hash compared against the corresponding commitment on the Public WBB. Code Extract 6-1 shows the way in which for each ballot, chosen for auditing, each of their sets of randomness pairings are checked using the hash commitments techniques explained in 6.2.2.1 Hash Commitments.

```
// loop over the randomness pairings for each of the sets of opened randomness commitments for
each of the PoD Printers
for (int i = 0; i < podOpenedRandomness.getNumRandomnessValues(); i++) {
        // get the current pair
        currentRandomPair = podOpenedRandomness.getRandomnessPair(i);
        // get commitment from the current mix server file
        commitment = currentRandomnessCommit.getRandomnessValue(i);
        // get the witness and randomness value from the current pair
        witness = currentRandomPair.getWitness();
        randomValue = currentRandomPair.getRandomnessValue();

        // check each commitment
        if (!CryptoUtils.verifyHashCommitment(commitment, witness, randomValue)) {
                return false;
        }
}
return true;
```

Code Extract 6-1 - Verifying randomness commitments

6.3.1.4 Functional Requirement F 4 – Verify Re-Encryptions and sorting

After the randomness values have been combined, as step 3 described in 5.5.1.1 Ballot Generation, the sorted and re-encrypted base candidate identifiers must be verified against the matching generic ballot published on the Public WBB. The re-encryption and sorting needs to happen independently for each different race as the sorting is carried out in a per race basis. Code Extract 6-2 shows the detailed steps as to how each base candidate identifier in the current race is encrypted using the current randomness value. An `IndexedElGamalECPoint` is then used to store the re-encrypted base candidate identifier along with its original position to be used later for calculating the permutation of the ballot. The re-encryption and sorting of each of the races and each ballot chosen for auditing can be seen in the BallotGenerationVerifier class in the verifyEncryptions function in the com.vvote.verifier.component.ballotGen package.

```
private List<IndexedElGamalECPoint> reencryptAndSort(List<MessageDigest> combinedRandomness,
int numberOfReencryptions, int currentRandomIndex) {
        List<IndexedElGamalECPoint> result = new ArrayList<IndexedElGamalECPoint>();

        for (int i = 0; i < numberOfReencryptions; i++) {
                // current random value
                BigInteger randValue = new BigInteger(1, combinedRandomness.get(i +
                currentRandomIndex).digest());

                // current base candidate identifier
                ElGamalECPoint baseCandidateId =
                this.getDataStore().getBaseEncryptedIds().get(i + currentRandomIndex);

                // current re-encrypted base candidate identifier
                ElGamalECPoint reencryptedCandidateId = ECUtils.reencrypt(baseCandidateId,
                this.getDataStore().getPublicKey(), randValue);

                // indexed re-encrypted base candidate identifier storing the original position
                IndexedElGamalECPoint indexedEC = new
                IndexedElGamalECPoint(reencryptedCandidateId, i);

                result.add(i, indexedEC);
        }
        Collections.sort(result, new ElGamalECPointComparator());
        return result;
}
```

Code Extract 6-2 - Re-encryption and sorting implementation

6.3.1.5 Functional Requirement F 5 – Verify permutation Commitment

The final check carried out by the Ballot Generation component verifier is to validate that the permutation commitment contained in the corresponding generic ballot published on the Public WBB can be opened using the final combined randomness value as a witness value and the re-calculated permutation which uses the order of the candidates on the reconstructed ballot as the randomness value. The permutation for each generic ballot is constructed using the re-encrypted candidate identifier's original index along with a "," separating each identifier and a ":" splitting each race permutation. The full permutation string is then validated using the hash commitment checks described in 6.2.2.1 Hash Commitments. The whole process can be seen in the BallotGenerationVerifier class in the verifyEncryptions function in the com.vvote.verifier.component.ballotGen package.

6.3.2 Vote Packing

The exact steps used for proving the correct operation for the Vote Packing process within the vVote system including both pre-mixnet and post-mixnet steps can be seen in 5.5.1.2 Vote Packing.

6.3.2.1 Functional Requirement F 6 – Verify Mixnet Input values

Verifying the pre-Mixnet Vote Packing process requires a number of discrete steps including validation of successful ballot reduction, reordering of the reduced ballot ciphertexts, packing of the candidate ciphertexts according to preferences and checking against the packed encryptions provided as input to the Mixnet.

6.3.2.1.1 Reducing Ballots

Each ballot used by a voter to cast a vote would have originally been a generic ballot constructed during the Ballot Generation process. Each of these generic ballots would then have been reduced so that it included only the candidates relevant for that specific voters' voting eligibility meaning the preferences they supplied will only match up with the same reduced ballot. The first step in verifying the Mixnet input process is to also reduce each of the generic ballots used for voting. Each POD message for each vote cast will contain the ballot reductions. Each ballot reduction verifies itself whereby if the base encrypted candidate identifier at the specified “candidate index” re-encrypted using the specified randomness matches the re-encrypted candidate identifier at the “index” specified then the process has been carried out correctly. Both the index value and the candidate index value will be specified in a per generic race basis and therefore adjustments may be required when dealing with LCATL and LCBTL reductions using the sizes of the previous generic races. The implementation for reducing a generic ballot can be seen in the ReducedBallot class in the com.vvote.verifier.component.votePacking package.

6.3.2.1.2 Packing ballots

After the reduced ballots have been constructed they need to be reordered into preference order as in step 4 from 5.5.1.2 Vote Packing. Once only the encryptions representing candidates the voter has provided preferences for are left, and are in preference order, these reordered ballots can be packed. Code Extract 6-3 shows the way in which the reordered ballots are packed together using the packing size provided. This process is carried out for each race and for each ballot used for voting.

```
private List<ElGamalECPoint> packCiphers(SortedMap<Integer, ElGamalECPoint> preferences, int
packingSize) {
        int packingPreference = 0;
        ElGamalECPoint currentCipher = null;
        ElGamalECPoint currentPacking = null;
        List<ElGamalECPoint> currentPackedList = null;
        currentPackedList = new ArrayList<ElGamalECPoint>();

        // loop over preferences
        for (Integer pref : preferences.keySet()) {
                // increment the packing preference which ranges from 1 to packing size
                packingPreference++;
                // gets the specific cipher
                currentCipher = preferences.get(pref);

                // multiply by the packing preference number (NOT the actual preference number)
                currentCipher.multiply(BigInteger.valueOf(packingPreference));

                // either store this current cipher
                if (currentPacking == null) {
                        currentPacking = currentCipher;
                } else {
                        // or add to the previous packing
                        currentPacking.add(currentCipher);
                }
                // packing preference needs to be reset if it equals the maximum packing size
                if (packingPreference == packingSize) {
                        packingPreference = 0;
                        currentPackedList.add(currentPacking);
                        currentPacking = null;
                }
        }
        if (currentPacking != null) {
                currentPackedList.add(currentPacking);
        }
        return currentPackedList;
}
```
Code Extract 6-3 - Vote Packing ciphertexts

6.3.2.1.3 Verify Mixnet Input

Each list of packed ciphertexts within each race is then appropriately padded as described in 6 from 5.5.1.2 Vote Packing. After this point each list of ciphertexts for each race, corresponding to the preferences made by a voter in the specified race, will contain the same number of columns and should exactly match the values as they were input into the Mixnet within the vVote system. Each of the prepared packings needs to be compared against the corresponding Mixnet input. For each set of packed (or unpacked) ciphertexts, corresponding to the voters preferences for a specific race, the corresponding Mixnet Input value prepared for the Mixnet must be found and it must be verified that these values exactly match.

6.3.2.2 Functional Requirement F 7 – Verify Mixnet Output values

The post-Mixnet Vote Packing process also requires a number of isolated steps including re-ordering the plaintext candidate identifiers according to the Mixnet output values, packing the re-ordered plaintext candidate identifiers, padding the packing values and then verifying these packings against what was output from the Mixnet.

The first step carried out in the verification of the Mixnet output is to lookup the plaintext identifiers for each output CSV using the district configuration and race specified. The plaintext identifiers are then reordered according to the preferences specified in the CSV file. The implementation for the plaintext candidate identifier reordering according to voter preferences can be seen in the class VotePackingVerifier in function reorderMixOutput in the com.vvote.verifier.component.votePacking package.

6.3.2.2.1 Packing Mixnet output

After the plaintext candidate identifiers, for each corresponding race type and district, have been reordered according to the preferences contained within the Mixnet output they must be packed together in a manner similar to that of the ciphertext packings. The plaintext candidate identifiers are packed as in step 9 from 5.5.1.2 Vote Packing. Code Extract 6-4 shows the way in which the reordered plaintexts are packed together using the packing size provided. This process is carried out for each row of each Mixnet output CSV file corresponding to a voters' preferences for a particular race type and district.

```
private List<ECPoint> packPlaintexts(SortedMap<Integer, ECPoint> preferences, int packingSize)
{
        int packingPreference = 0;
        ECPoint currentId = null;
        ECPoint currentPacking = null;
        List<ECPoint> currentPackedList = null;
        currentPackedList = new ArrayList<ECPoint>();

        // loop over preferences
        for (Integer pref : preferences.keySet()) {
                // increment the packing preference which ranges from 1 to packing size
                packingPreference++;

                // gets the specific cipher
                currentId = preferences.get(pref);

                // multiply by the packing preference number (NOT the actual preference number)
                currentId = currentId.multiply(BigInteger.valueOf(packingPreference));

                // either store this current cipher
                if (currentPacking == null) {
                        currentPacking = currentId;
                } else {
                        // or add to the previous packing
                        currentPacking = currentPacking.add(currentId);
                }
                // packing preference needs to be reset if it equals the maximum packing size
                if (packingPreference == packingSize) {
                        packingPreference = 0;
                        currentPackedList.add(currentPacking);
                        currentPacking = null;
                }
        }
        if (currentPacking != null) {
                currentPackedList.add(currentPacking);
        }
        return currentPackedList;
}
```

Code Extract 6-4 - Plaintext packing process

6.3.2.2.2 Verify Mixnet output

Each list of packed plaintext candidate identifiers for each race must then be appropriately padded as described in from step 10 from 5.5.1.2 Vote Packing. Each set of packed plaintext candidate identifiers for any specific race configuration will contain the same number of columns and should then be exact copies of the corresponding sets of values output from the Mixnet for the same race configuration. For each set of packed (or unpacked) plaintext candidate identifiers, corresponding to the shuffled voter preferences for a specific race, the corresponding Mixnet output values must be found and verified that the values exactly match.

6.3.3 Public WBB Commits

The exact details for the algorithm used for verifying the commits made to the Public WBB can be seen in 5.5.1.3 Public WBB Commits.

6.3.3.1 Functional Requirement F 8 – Verify Public WBB commits

For each of the commits made to the Public WBB the Public WBB commit component verifier validates the signature made over specific data items to ensure the commit's integrity and to ensure that which was posted is valid data. A SHA1 MessageDigest is constructed and is updated with the internal signable content from each typed JSON message within the current commit's JSON message file. If any of the typed JSON messages are file messages then the message's corresponding file is also hashed using the process outlined in 6.2.2.2 File Hashing and added to the hash. The public WBB joint signature data is then recalculated as a SHA1 hash which includes the string "Commit", the current commit time and the hash over the JSON message file and any additional file messages. The BLS signature included in the signature JSON file is then verified using the process outlined in 6.2.2.4 Threshold BLS Cryptography. The implementation for this verification can be seen in the class CommitmentVerifier in the function doVerification in the package com.vvote.verifier.commits.

6.4 Implementation Challenges

This section will be used to highlight a number of specific challenges faced during the implementation of the vVote Verifier. Some of the challenges have been briefly identified previously; however, this section will provide additional details relating to the challenge encountered.

6.4.1 OpenSSL

The main aim of this dissertation was to carry out the implementation of a stand-alone independent reference implementation of a verifier for the vVote system. The problem with this aim was whether the implementation of the verifier would be completely independent down to the libraries used or partially independent where some libraries are shared between implementations. The vVote system makes use of a number of standard JAVA libraries used for cryptography including JCA, JCE and Bouncy Castle. These libraries were chosen for the vVote system because of their completeness, including many required functions, their wide use in other projects and also due to the way in which the implementation of these libraries has been widely studied and scrutinised.

The idea of using OpenSSL was suggested due to it including a wide range of cryptographic functions as well as it being widely studied and scrutinised. The OpenSSL libraries are c based and cannot therefore be used directly by the vVote Verifier system. To successfully use the OpenSSL libraries the Java Native Interface (JNI) was required to be used whereby c++ code would be written and compiled to native code and then called directly by the JAVA code.

The use of OpenSSL along with JNI caused a number of problems. The first problem encountered was actually compiling the c++ code into native libraries which can be directly used. On Windows the libraries were compiled into DLL files and on Ubuntu the library was compiled to a shared object files (.so). There are a number of complications in compiling when using external libraries along with having support for JNI. Firstly the JAVA class which will call the native function needs to be used to create a header file to be used by the c++ class. From the top level of the project the `javah` command is used to generated a header file: `javah com.vvote.verifierlibrary.CryptoUtils`. The c++ code was written against this header file, and can be seen in Appendix G. After the code was finished it had to be compiled into native code so it could be used directly through JNI. For both Windows and Ubuntu compilation an intermediate step was used to generate an object file (.o) which is an intermediate stage between compiling and linking the library. The way in which the library was compiled for both Windows and Linux can be seen below.

Windows (tested using Windows 8.1 Pro):

- Create .o object file: `gcc -c -I"C:\Program Files\Java\jdk1.7.0_51\include" -I"C:\Program Files\Java\jdk1.7.0_51\include\win32" -I"C:\OpenSSL-Win32\include" CryptoUtils.cpp`
- Create .dll shared object: `gcc -Wl,--add-stdcall-alias -shared -o CryptoOpenSSL.dll cryptoUtils.o -lssl -lcrypto -lstdc++`

Ubuntu (tested using Ubuntu 14.04):

- Create .o object file: `gcc -c -fPIC -I/usr/lib/jvm/java-7-openjdk-amd64/include -I/usr/lib/jvm/java-7-openjdk-amd64/include/linux CryptoUtils.cpp`
- Create .so shared object: `gcc -Wl -shared -o libCryptoOpenSSL.so CryptoUtils.o -lssl -lcrypto -lstdc++`

The second problem encountered, when using OpenSSL, was that of the different data types used within JAVA, c++ code and the c code used in the OpenSSL libaries. What should have been quite a simple task of simply creating a function to check hash based commitments turned into a relatively complicated task because of the number of conversions of data types. For example when using the OpenSSL library JAVA string objects were passed to the native function. These JAVA based strings then had to be converted to c++ std:string objects. These c++ std:string objects then needed to be converted to unsigned char * types.

6.4.2 Changing Location of Data Files

During the development of the vVote Verifier a number of datasets were provided by the developers of the vVote system for use in testing the verification process. It was noticed that the location of files within the different datasets was not consistent, although the filenames were. It was decided that to be able to allow for easy testing using different datasets and not having to worry about the location of files, only requiring that the files themselves are included, with their filenames unchanged, the vVote Verifier would search for files within the datasets provided. Code Extract 6-5 shows the implementation used to find a file with the specified name within the provided folder, or file if we are checking the base case of the recursive function.

```
public static String findFile(String name, File file) {
        File[] currentFileList = file.listFiles();
        String result = null;
        if (currentFileList != null) {
                for (File currentFile : currentFileList) {
                        if (name.equalsIgnoreCase(currentFile.getName())) {
```

```
                                        return currentFile.getPath();
                                } else if (currentFile.isDirectory()) {
                                        result = findFile(name, currentFile);
                                        if (result != null) {
                                                return result;
                                        }
                                }
                        }
                }
                return null;
        }
```
Code Extract 6-5 - Searches for a file with the specified name within the file provided

6.4.3 Size of Input Data

Through the initial development a single dataset was used to test the vVote Verifier which is explained in detail in 7.4.1 Sample Dataset 1. This dataset was relatively simple and therefore did not contain very much data meaning the entire dataset was around 100MBs in size. The vVote Verifier was testing extensively against this dataset. The second set of data, explained in detail in 7.4.3 Sample Dataset 3, was received quite late on and was very much more complicated and contained far more data meaning the entire dataset was around 1.5GBs in size. The vast increase in the size of the data being processed caused problems due to the way the vVote Verifier system had been designed. The vVote Verifier initially reads all of the data and organises it into meaningful data structures. This design was fine and worked well with the smaller sample dataset however when it was run against a larger dataset the increased data set size caused the code, when running on machines with low amounts of RAM, to throw an out of memory exception: "java.lang.OutOfMemoryError: Java heap space". This problem had not been expected although it should have been considered during the analysis stage.

The solution to this problem was to initially only read in identifiers for certain parts of the data. Certain data files, such as the Ballot Generation commit files and mix random commit files, contain discrete elements on separate lines with each line containing an identifier for the data element. These files can contain significant quantities of data for example if 2000 generic ballots are generated then for each POD Printer there will be 1 line within a Ballot Generation file containing around 40,000 characters and 1 line in 5 different mix random commit data files each containing around 40,000 characters. This means that for 2000 generic ballots generated there are 6 lines with each containing significant amounts of data around 40,000 * 6 characters. Initially each of these discrete elements would be read into the program and processed and stored as a complex object. So for example using 4 POD Printers, each generating 2000 generic ballots, we have over 12000 discrete elements which were being constructed from over 12,000 * 40,000 characters of data which caused problems.

It was identified that the vVote Verifier only requires the mix random commit data and generic ballots for any ballots chosen for auditing, which can be relatively small – around 80, compared to the whole number of generic ballots meaning only 80 * 40,000 characters of data. Initially we only read in the identifiers for these 12000 discrete elements as a single String identifier such as a ballot serial number which contains around 10 characters. Later when the ballots chosen for auditing are determined we look up the specific line within the relevant files and process the entire data object. This solution works on around 12,000 * 10 + 80 * 40,000 = 3320000 characters of data rather than the original amount of data which was around 12,000 * 40,000 = 480000000 characters of data which is a saving of 476680000 characters. The solution requires that the files are searched once for each time we need to look up the required data elements which therefore takes up slightly more CPU time, however, this is definitely acceptable due to the amount of RAM storage saved.

6.5 Conclusion

This chapter has provided extensive details as to how the design was implemented to produce the working vVote Verifier system. The chapter included a breakdown of the core components used within the system including the verifier library and the actual component verifiers. To illustrate and aid in the descriptions code extracts were included showing cut down versions of the code used. The section outlining the component verifiers also outlined how the requirements in Table 4-5 were met. The chapter also provided additional details regarding a number of implementation challenges faced during the development of the vVote Verifier and how they were tackled.

The next chapter will outline and describe the testing and evaluation of the vVote Verifier which will include descriptions of the tests as well as the test data used to measure how successful the implementation has been.

Chapter 7 – Testing and Evaluation

7.1 Introduction

This chapter will be used to describe the testing and evaluation carried out on the vVote Verifier with the aim of testing the system against its initial requirements set out in Table 4-5 and Table 4-6 to determine how successful the development has been. A number of different sets of sample data were used to test the system; each of which will be explained in detail in addition to the results seen through running the verifier against each set of sample data.

7.2 Notes about Testing

The implementation of the vVote Verifier took place at the same time as the development of a number of the vVote system components. This meant that some parts of the system were not completely finished and were somewhat changeable. This meant that getting data was very difficult. The vVote Verifier uses data produced through the end-to-end running of the system through all stages of election.

The vVote Verifier requires data produced through the running of either an entire election or a somewhat involved simulation of an election. Setting up and running the vVote system was infeasible due to the large number of components involved and the substantial setup needed. This meant the data included and explained throughout this chapter was produced by and sent over from Australia by those working on and testing the vVote system itself. The production of data therefore relied upon others who themselves were busy with the development of the system.

The end-to-end testing of the vVote system did not take place until very late on in this project meaning that getting substantial meaningful sets of data was very difficult.

7.3 Sample Data Format

The vVote Verifier was designed for an ideal data set. This data set is a single unzipped folder provided as input; this folder is referred to as the base path. The base path folder then contains within it all the required data elements needed for the verification of an election run using the vVote system. The data set can be easily split up into three discrete portions:

- Commit data – consisting of groups of three files which are JSON message files, attachment files and JSON signature files. These files should be contained within a folder named final_commits.

- Configuration data – contains all election configuration specific data including district race sizes, Ballot Generation configuration data, Vote Packing configuration data, candidate identifiers, the election public key, certificates files and a Vote Packing padding point file.
- Mixnet data – contains the Mixnet specific data which should include the inputs and outputs for the Mixnet. The Input and Output folders should provide enough data for the Mixnet process to be universally verifiable. Only specific portions of the Mixnet input and output data will be used to verify the Vote Packing process.

A number of files continually changed location within the file structure and therefore a method was devised whereby the files would be 'located' and then used rather than having a fixed location. This was a number of many changes required to be made for the vVote Verifier to remain testable against the changing and in development vVote system.

7.4 Testing Data

The sample data was provided as given meaning modifications to the data or selecting subsets of the data was not possible whilst keeping the verifiable nature of the data present. The data requires that it remains in the same format and same structure. The commits made within the final_commits folder must again be processed as a whole and cannot be examined as separate commits. Completely separate commits may contain related entries which require processing together. Testing in a methodical and well-structured way was not possible and could only be performed on the data provided. This proved very difficult in that the data provided was complex from the start and mirrored complex election data rather than having simplistic data in the first set of tests with the complexity then built up in stages.

This section will be used to provide an overview of the data provided as well as their configuration details.

data_sample1, data_sample2, data_sample3 and data_sample4 can be located in the data_samples folder of the electronic submission for this report.

7.4.1 Sample Dataset 1

data_sample1 was the first data set received and resembled closely the final structure of the data which would be present on the Public WBB. Unfortunately the dataset was a combination of two completely independent sets of data and therefore certain parts of the verification would not be possible using it. Simply the Surrey code and Mixnet parts were completely independent. This meant that it was only possible to carry out the verification of the Ballot Generation process in addition to the Public WBB commits. Validation of the Vote Packing process was not possible as the Mixnet Input and Output datasets were not compatible and were run using different configuration details. In addition to the incompatible Mixnet data the plaintext identifiers provided within this dataset did not correspond with the base encrypted candidate identifiers meaning verification of the base encryption of the plaintext identifiers was not possible which otherwise would be carried out.

7.4.1.1 Dataset Configuration

data_sample1 is a simplified dataset using only a single PoD Printer meaning there is only a single full Ballot Generation process to verify and also means every single ballot generated is from the same PoD Printer. In addition to the simplification in the number of PoD Printers the dataset also nicely separates the ballot generation data from the voting data meaning that any lookup of data required for the ballot generation verification only requires checks within the same commit.

Configuration Detail Name	Configuration Value
Number of Randomness Generation Servers	5
Number of PoD Printers	1
Number of ballots for each PoD Printer to generate	1000
Number of ballots to audit for each PoD Printer	10
LA Race Size	20
LC ATL Race Size	10
LC BTL Race Size	50
Number of Districts	2
Number of votes cast	990

Table 7-1 - Dataset1 Configuration

7.4.2 Sample Dataset 2

data_sample2 was the second data set used and contained a number of modifications on the data set presented in data_sample1. This sample data was used to test the vVote Verifier against invalid data submitted to the Public WBB to simulate a basic malicious attempt at providing fraudulent data or trying to modify the data in some way to bring the election result into disrepute. data_sample2 contains a single modification of data_sample1 and other than this single modification has the same configuration and issues presented previously. The single modification is as follows:

- The jointSig contained within the Signature JSON file 1403247600000_signature.json was changed from WtfAAb7DTE3llAgICCW2756BIx8VNEWmvHEIGZgg5+/M4oDl9DxkbA== to WtfAAb7DTE3llAgICCW2756BIx8VNEWmvHEIGZgg5+/M4oD22DxkbA==.

The result of this modification is that the joint signature included within the Signature JSON file 1403247600000_signature.json will not represent the valid signature made against the remaining un-modified data contained in the commit originally made by the Public WBB.

7.4.3 Sample Dataset 3

data_sample3 was the second data set received on 12/08/2014. The sample data was created as part of the vVote system's user acceptance testing and therefore resembled closely the final structure of the Public WBB. Again unfortunately the dataset was not complete and was missing a number of files and had a number of problems including:

- Data is contained within an extra_commits folder rather than the final_commits folder. What will be seen in the real election setting will be that the data contained in extra_commits will be moved to the final_commits folder and a hash will be generated and signed by the Public WBB over the commit data to authenticate the data. The final_commits folder will then be made public. The data within extra_commits is not end of day commits and is manually requested. The data and formatting of the files is identical to what would be present in the final_commits folder and is used during the development of the system as it is more convenient to submit data manually than waiting until the end of the day before the data is moved to the final_commits folder. At the time of writing user acceptance testing was being run on the vVote system and was used to produce this set of data. The testing was being run using slightly out of date code whereby this move of data and additional hash and signature file generated is not carried out.
- As mentioned, only when the data is moved into the final_commits folder will a hash be generated and signed by the Public WBB over the commit data. This process is used to generate the Signature JSON file for each commit which is validated using the Public WBB commit component verifier which is therefore not possible using this dataset.
- The ballot_gen_conf.json file is missing from the dataset which would usually be used to determine the ballot beneration configuration data which was used to specify the size of the races, the number of ballots to generate and the number of ballots to audit. This file has been manually created and added to the sample data to avoid receiving errors of a missing file as these values are integral to the verification process.
- The file certs.bks is missing from the dataset. This file would be used to specify the Public WBB public key entry which would usually be used if the Public WBB commit component verifier was being used, however, as previously mentioned this is not possible and therefore the loss of this file has negligible impact. The file has been copied from data_sample1 again to avoid receiving errors.

- The file plaintext_ids.json was missing from the sample data however processed versions of the file which were split into separate race specific files were provided. The plaintext_ids.json was simply reconstructed from the separate files to enable testing.

The second dataset can be used to verify the Ballot Generation process as well as the Vote Packing process but cannot be used for verifying the Public WBB commits due to the issue relating to the out-of-date code being used for the user acceptance testing.

7.4.3.1 Dataset Configuration

data_sample3 is a more complex dataset using multiple PoD Printers in addition to overlapping commits meaning there are parts of ballot generation data within multiple commits and also voting data interspersed through the commit files. The dataset also includes additional messages such as Cancel Messages and Audit messages. Cancel messages needed additional processing to ensure these cancelled votes are not assumed to have been passed through the Mixnet. Although the number of votes cast has been reduced dramatically the dataset can still be considered far more complex due to the structure of the data and the number of PoD Printers in addition to the number of districts involved.

Configuration Detail Name	Configuration Value
Number of Randomness Generation Servers	5
Number of PoD Printers	4
Number of ballots for each PoD Printer to generate	2000
Number of ballots to audit for each PoD Printer	10
LA Race Size	20
LC ATL Race Size	20
LC BTL Race Size	100
Number of Districts	96
Number of votes cast	105

Table 7-2 – Dataset3 Configuration

This was the first time the vVote Verifier had been run against a complex set of data and therefore a large number of changes were made to the system to cope with the added complexity which had not been seen previously. Previously much of the system had been developed from a theoretical standpoint and therefore being able to test the system using concrete data vastly improved the functionality and correctness of the system.

7.4.4 Sample Dataset 4

data_sample4 was the fourth data set used and contained a number of modifications on the data included in data_sample3. This sample data was used to test the vVote Verifier against invalid data contained within the final_commits/extra_commits folders. This data was used to test that the vVote Verifier successfully identifies when data used for validating both the Ballot Generation process and Vote Packing process contains invalid data. data_sample4 contains a number of modifications of data_sample3, however, other than the modifications detailed the dataset has the same configuration and issues presented previously. The modifications are as follows:

Ballot Generation process modifications:

- The first randomness witness value rComm for the randomly selected ballot for auditing with serial number 2060101:603 within the AuditDataFile.json from WBBUpload7116143514599136428.zip contained in 1406876400000force3_attachments.zip was changed from 326104e0521f373fb4134d91e9d94463b48740696b9c87626c257de31b39379a to 326104e0521f373fb4134d91e9d94463b48740696b9c87626c257de31b39379e.
    - The result of this modification is that the commitment made by the corresponding randomness generation server for the modified witness value will not match the concatenation of the original randomness value and the modified witness value for the ballot with serial number 2060101:603 selected for auditing.
- The first randomness value r for the randomly selected ballot for auditing with serial number 2060101:1997 within AuditDataFile.json from WBBUpload7116143514599136428.zip contained in 1406876400000force3_attachments.zip was changed from 0bf4a91dd4ea69ea4bfe288b9fb567e344fd70e0d02dc9f892b743479f05b852 to 0bf4a91dd4ea69ea4bfe288b9fb567e3441270e0d02dc9f892b743479f05b852.
    - The result of this modification is two-fold – firstly the commitment made by the corresponding randomness generation server for the modified randomness value will not match the concatenation of the modified randomness value and the original witness value for the ballot with serial number 2060101:1997 selected for auditing. The second result will be that the recalculated generic ballot with serial number 2060101:1997 will not match the corresponding generic ballot on the Public WBB when the base candidate identifiers are re-encrypted and sorted as the modified randomness value will be combined with other randomness values and used in the encryption process.
- The first commitment made by randomness generation server with identifier MixServer3 contained within the commitData.json file in WBBUpload6470112802534051270.zip from 1406876400000force3_attachments.zip with serial number 2060101:581 was changed from

960db9f5271e0fce735e226516123c8d8c9f7cbf60daf6a4614cb74f85f79a7c to 960db9f5271e0fce735e226516123c8d8c9f7cbf60daf6a4614cb74f85f79a7e.

  - The result of this modification is that the commitment check made on the randomly selected ballot for auditing with serial number 2060101:581 carried out by concatenating the randomness value and witness value for the ballot will not match the changed commitment value.

- The re-encrypted candidate identifiers from the ciphers.json file from WBBUpload5428661629889398939.zip contained in 1406876400000force3_attachments.zip with serial number 2370101:202 at indices 0 and 1 were swapped over.
  - The result of this modification is that the recalculated generic ballot with serial number 2370101:202 will not match the corresponding generic ballot on the Public WBB when the base candidate identifiers are re-encrypted and sorted as the check on the re-encryptions at indices 0 and 1 will not match.

Vote packing process modifications:

- The candidate index within the ballot reduction at index 0 for the POD message with serial number 5150101:24 from 1407222000000force7.json was changed from 17 to 18.
  - The result of this modification will be that the verification of the ballot reduction process will not be carried out correctly for the vote.
- The randomness value within the ballot reduction at index 0 for the POD message with serial number 2370101:3 from 1407222000000force7.json was changed from Z2ANxJRM41I/SzsQfWeUdxElLuMAX0tWjzNbSKPsjzM= to EEANxJRM41I/SzsQfWeUdxElLuMAX0tWjzNbSKPsjzM=.
  - The result of this modification will be that the verification of the ballot reduction process will not be carried out correctly for the vote.
- The preferences for the LA race within the Vote message with serial number 5150101:10 from 1407222000000force7.json were changed from 2,1 to 1,2.
  - The result of this modification will be that the repacked preferences for the vote with serial number 5150101:10 will not match the previously packed value submitted to the Public WBB.

7.5 How the Verifier was run

The vVote Verifier can be run using a single command from the command line by providing a single argument. The only requirement with regard to the input is that all data be present and in the required correct format. Configuration files do not need to be specifically within any folder and will be found automatically; this is a feature added to avoid any issues with files having their locations changed during the development of the vVote system. The system takes, as input, the path to a single folder which contains all the data and configuration files required.

The implementation was initially designed to be platform independent and to have no dependencies as to the operating system or system configuration; however, due to the use of a c++ compiled OpenSSL library the vVote Verifier requires that the system used to run the tool is either Linux or Windows based. The library on Windows was compiled for 32 bit machines and on Linux was compiled for 64 bit machines.

To run the vVote Verifier JAR file the system being used must be either Ubuntu 64-bit or Windows 32-bit, must have JAVA installed and must have the OpenSSL libraries installed.

```
java -Djava.library.path=jni -jar vVoteVerifier.jar res/sample_commits/
```

Figure 7-1 - How to run the vVote Verifier

Figure 7-1 shows how to run the vVote Verifier. The results produced through the running of the verifier will be both output to command line and also saved to two log files within a “logs” folder created in the same directory as the tool being run. The first log file “results.log” will contain step by step information including the details of the verification step being carried out and the result of the verification step. The second log file “logfile.log” will contain more detailed information as to the exact steps taking during the verification. The second file will be significantly larger than the results.log file and should only be examined in the case of a failure occurring in the results.log file. A portion of a sample results.log file can be seen in Figure 7-2.

```
12:55:15.112 [main] INFO  ComponentVerifier - Successfully verified that the plaintext ids and
base candidate ids match
12:55:15.113 [main] INFO  ComponentVerifier - Verification relevant for each verifier was
carried out successfully
12:55:15.113 [main] INFO  BallotGenerationVerifier - Successfully verified that the number of
opened randomness commitments received by each PoD Printer matches the number of candidates
plus 1
12:55:15.124 [main] INFO  BallotGenerationVerifier - Successfully verified that the number of
randomness values committed to by the mix servers matches the number of candidates plus 1
12:55:15.124 [main] INFO  BallotGenerationVerifier - Successfully verified that the number of
ballots required for auditing were provided
12:55:15.669 [main] INFO  BallotGenerationVerifier - Serial numbers for auditing were checked
successfully using the Fiat shamir signature for commitment with identifier: CommitIdentifier
[identifier=1406876400000force3, printerId=2060101]
12:55:16.184 [main] INFO  BallotGenerationVerifier - Serial numbers for auditing were checked
successfully using the Fiat shamir signature for commitment with identifier: CommitIdentifier
[identifier=1406876400000force3, printerId=2370101]
12:55:16.696 [main] INFO  BallotGenerationVerifier - Serial numbers for auditing were checked
successfully using the Fiat shamir signature for commitment with identifier: CommitIdentifier
[identifier=1407049200000force5, printerId=2370201]
12:55:17.205 [main] INFO  BallotGenerationVerifier - Serial numbers for auditing were checked
successfully using the Fiat shamir signature for commitment with identifier: CommitIdentifier
[identifier=1406876400000force3, printerId=5150101]
```

```
12:55:17.205 [main] INFO  BallotGenerationVerifier - Successfully verified the Fiat Shamir
signatures were used to choose the required number of ballots for auditing
12:55:17.205 [main] INFO  BallotGenerationVerifier - Starting the verification of commitment
with identifier: CommitIdentifier [identifier=1406876400000force3, printerId=2060101]
12:55:17.453 [main] INFO  BallotGenerationVerifier - Successfully verified that the randomness
values for ballot: 2060101:581 were provided by and committed to by the mix servers
12:55:18.043 [main] INFO  BallotGenerationVerifier - Re-encryption and sorting was successfull
for ballot with serial number: '2060101:581'. The generic ballot was generated successfully by
PoD Printer: 2060101

… Steps missing

12:55:24.524 [main] INFO  BallotGenerationVerifier - Starting the verification of commitment
with identifier: CommitIdentifier [identifier=1406876400000force3, printerId=2370101]
12:55:24.693 [main] INFO  BallotGenerationVerifier - Successfully verified that the randomness
values for ballot: 2370101:886 were provided by and committed to by the mix servers
12:55:25.181 [main] INFO  BallotGenerationVerifier - Re-encryption and sorting was successfull
for ballot with serial number: '2370101:886'. The generic ballot was generated successfully by
PoD Printer: 2370101

… Steps missing

12:55:31.761 [main] INFO  BallotGenerationVerifier - Starting the verification of commitment
with identifier: CommitIdentifier [identifier=1407049200000force5, printerId=2370201]
12:55:31.958 [main] INFO  BallotGenerationVerifier - Successfully verified that the randomness
values for ballot: 2370201:856 were provided by and committed to by the mix servers
12:55:32.459 [main] INFO  BallotGenerationVerifier - Re-encryption and sorting was successfull
for ballot with serial number: '2370201:856'. The generic ballot was generated successfully by
PoD Printer: 2370201

… Steps missing

12:55:38.657 [main] INFO  BallotGenerationVerifier - Starting the verification of commitment
with identifier: CommitIdentifier [identifier=1406876400000force3, printerId=5150101]
12:55:38.814 [main] INFO  BallotGenerationVerifier - Successfully verified that the randomness
values for ballot: 5150101:1621 were provided by and committed to by the mix servers
12:55:39.301 [main] INFO  BallotGenerationVerifier - Re-encryption and sorting was successfull
for ballot with serial number: '5150101:1621'. The generic ballot was generated successfully
by PoD Printer: 5150101

… Steps missing

12:55:45.136 [main] INFO  BallotGenerationVerifier - Ballot Generation Verification was
carried out successfully
12:55:45.137 [main] INFO  CommitmentVerifier - Extra commits folder is being used so we cannot
verify signatures
12:55:45.138 [main] INFO  VotePackingVerifier - Starting Vote Packing verification
12:55:45.143 [main] INFO  ComponentVerifier - Successfully verified that the plaintext ids and
base candidate ids match
12:55:45.143 [main] INFO  ComponentVerifier - Verification relevant for each verifier was
carried out successfully
12:56:08.914 [main] INFO  VotePackingVerifier - Successfully verifyied that ballots can be
reduced successfully
12:56:08.923 [main] INFO  VotePackingVerifier - Successfully verified that the ballots have
been reduced successfully
12:56:08.925 [main] INFO  VotePackingVerifier - Successfully verifyied that the ballots have
been reordered successfully
12:56:08.949 [main] INFO  VotePackingVerifier - Successfully verifyied that the ballots have
been packed successfully
12:56:08.981 [main] INFO  VotePackingVerifier - Successfully reordered the mixnet output
values
12:56:08.994 [main] INFO  VotePackingVerifier - Successfully verifyied that the mix output has
been packed successfully
12:56:09.009 [main] INFO  VotePackingVerifier - Vote Packing Verification was carried out
successfully
```

Figure 7-2 - Sample Results.log file

7.6 Test Results

The test results for the testing carried out against the data samples explained in this chapter using the vVote Verifier will not be included in full in this document and will instead be included as part of the electronic submission. For each piece of data the output "results" logs will be included in the electronic submission.

In this chapter only a summary of the results will be provided. The testing results will be provided against each of the functional requirements described in Table 4-5. The results will be highlighted as being either correct or incorrect verification behaviour against either valid or invalid data. It will also be shown whether the verification of the requirement caused an error to be recorded to highlight that the verification process has identified an issue with the data.

7.6.1 Test 1 - Sample Dataset 1

The first testing carried out was against data_sample1 which can be considered a basic election configuration consisting of only a single PoD Printer. Each final commit was completely separated meaning the ballot generation data was confined to a single commit likewise was the voting data.

Requirement	**Valid Data (Yes / No)**	**vVote Verifier Result Correct / Incorrect**	**Error Shown**	**Notes**
F 1	Yes	Correct	No	The Fiat-Shamir signature re-calculation was carried out successfully for the single PoD Printer.
F 2	Yes	Correct	No	The correctly re-calculated Fiat-Shamir signature was successfully used to select the same subset of ballots chosen for auditing.
F 3	Yes	Correct	No	The randomness commitments made by each of the randomness generation servers was consistent with the randomness data revealed by the PoD Printer.
F 4	Yes	Correct	No	Each of the ballots chosen for auditing was successfully recalculated and matched the corresponding generic ballot.
F 5	Yes	Correct	No	The permutation commitment contained in the matching generic ballot from the Public WBB was successfully opened and verified.
F 6	No	Correct	Yes	The Mixnet input data could not be validated due to the incompatible Mixnet input and final_commit data.
F 7	No	Correct	Yes	The Mixnet output data could not be validated due to the incompatible Mixnet output and plaintext identifiers.
F 8	Yes	Correct	No	The signatures against the commits made to the Public WBB contained in the final_commits folder were successfully verified.
F 9	No	Correct	Yes	The plaintext identifiers and base encrypted candidate identifiers are incompatible and come from separate data sets.

Table 7-3 - Test results for sample dataset 1

Table 7-3 shows the results of the vVote Verifier against the functional requirements of the system. As can be seen the testing was partially successful in terms of the actual results produced however the reasons, explained in more detail in Table 7-3, show that the main issue was the incompatibility of data between the final_commits folder and the Mixnet data provided. For each requirement the vVote Verifier produced correct verification behaviour and was successful when the data was valid and showed errors when the data was invalid.

7.6.2 Test 2 - Sample Dataset 2

The second set of testing carried out was against data_sample2 which was in essence the same as data_sample1 but contained a single modification on a joint signature contained in the signature file for one of the two commits.

Requirement	**Valid Data (Yes / No)**	**vVote Verifier Result Correct / Incorrect**	**Error Shown**	**Notes**
F 1	Yes	Correct	No	The Fiat-Shamir signature re-calculation was carried out successfully for the single PoD Printer.
F 2	Yes	Correct	No	The correctly re-calculated Fiat-Shamir signature was successfully used to select the same subset of ballots chosen for auditing.
F 3	Yes	Correct	No	The randomness commitments made by each of the randomness generation servers was consistent with the randomness data revealed by the PoD Printer.
F 4	Yes	Correct	No	Each of the ballots chosen for auditing was successfully recalculated and matched the corresponding generic ballot.
F 5	Yes	Correct	No	The permutation commitment contained in the matching generic ballot from the Public WBB was successfully opened and verified.
F 6	No	Correct	Yes	The Mixnet input data could not be validated due to the incompatible Mixnet input and final_commit data.
F 7	No	Correct	Yes	The Mixnet output data could not be validated due to the incompatible Mixnet output and plaintext identifiers.
F 8	No	Correct	Yes	The signatures against the commits made to the Public WBB contained in the extra_commits folder were verified correctly and showed the presence of invalid data.
F 9	No	Correct	Yes	The plaintext identifiers and base encrypted candidate identifiers are incompatible and come from separate data sets.

Table 7-4 - Test results for sample dataset 2

Table 7-4 shows the results of the vVote Verifier against the functional requirements of the system. Functional requirement F 8 was tested against invalid data and the testing was carried out successfully by identifying that there was an issue with the Public WBB data. The log below shows the identification of an error with the joint signature for the Public WBB commit.

```
17:18:24.886 ERROR CommitmentVerifier - Verification of the joint signature for the commitment
with identifier: 1403247600000 failed. Check that the data was successfully downloaded.
```

For each requirement the vVote Verifier produced correct verification behaviour and was successful when the data was valid and showed errors when the data was invalid.

7.6.3 Test 3 - Sample Dataset 3

The third set of testing carried out was against data_sample3 which is a more complete dataset containing compatible commit data and Mixnet data. The dataset is far more complex and contains both ballot generation data and Vote Packing data for multiple PoD Printers. The ballot generation data contains generic ballots and ballots chosen for auditing for a number of different PoD Printers which must be verified independently. The Vote Packing data also consists of votes cast which were generated by different PoD Printers.

Requirement	**Valid Data (Yes / No)**	**vVote Verifier Result Correct / Incorrect**	**Error Shown**	**Notes**
F 1	Yes	Correct	No	The Fiat-Shamir signature re-calculation was carried out successfully for each of the PoD Printers.
F 2	Yes	Correct	No	The correctly re-calculated Fiat-Shamir signatures were successfully used to select the same subset of ballots chosen for auditing for each of the PoD Printers.
F 3	Yes	Correct	Warning	The randomness commitments made by each of the randomness generation servers for each of the PoD Printers were consistent with the randomness data revealed by each of the corresponding PoD Printers. A number of warnings were produced because the randomness generation server with identifier MixServer1 had sent two sets of commitments for each PoD Printer with one of the sets failing the verification and the other set producing the desired result. This issue is discussed in detail in Chapter 8.
F 4	Yes	Correct	No	Each of the ballots chosen for auditing, for each of the PoD Printers, was successfully recalculated and matched the corresponding generic ballot.
F 5	Yes	Correct	No	For each of the ballots chosen for auditing, for each of the PoD Printers, the permutation commitment contained in each of the matching generic ballots from the Public WBB was successfully opened and verified.
F 6	Yes	Correct	No	The Mixnet input data was successfully verified. Each of the packed voter preferences for each of the races has a matching set of packings contained in the Mixnet input data.
F 7	Yes	Correct	No	The Mixnet output data was successfully verified. Each of the output voter preferences contained in the CSV files was used to reorder the plaintext candidate identifiers which were then packed. Each of these plaintext packings had a matching Mixnet output packing.

F 8	No	Correct	No	The signatures against the commits made to the Public WBB contained in the extra_commits folder could not be successfully verified. The sample data was produced using an out of data version of the vVote code which did not have the functionality to produce the signature files. The signature data also will not be produced for data contained in the extra commits folder and therefore the verification of the Public WBB commits is skipped for data contained in the extra commits folder.
F 9	Yes	Correct	No	The plaintext identifiers could be successfully encrypted to match the base encrypted candidate identifiers.

Table 7-5 - Test results for sample dataset 3

Table 7-5 shows the results of the vVote Verifier against the functional requirements of the system. It can be seen that the testing was successful in terms of the results; however, a number of warnings are produced and included in the resulting log files. The warnings shown are for a number of Cancel, POD and Vote messages containing the same serial numbers. This should not happen with POD or Vote messages and therefore a warning is shown. For Cancel messages these repeated messages can be allowed if there is an issue with the cancellation process. Another warning shown is for the check on the randomness commitments made by each of the randomness generation servers for each of the PoD Printers. As explained previously, the randomness generation server with identifier MixServer1 had sent two sets of commitments for each PoD Printer with one of the sets failing the verification and the other set producing the desired result. This issue is discussed in detail in Chapter 8. These warnings indicate that questions should be asked of the election but they do not warrant an error being shown as there is probably a good reason for why this has happened.

7.6.4 Test 4 - Sample Dataset 4

The last set of testing carried out was against data_sample4 which was in essence the same as data_sample3 but contained a number of modifications on the data contained in the extra_commits/final_commits folders in a number of places to invalidate the data.

Requirement	**Valid Data (Yes / No)**	**vVote Verifier Result Correct / Incorrect**	**Error Shown**	**Notes**
F 1	No	Correct	Yes	The verification for the Fiat-Shamir signature re-calculation was carried out successfully and identified the invalid data provided.
F 2	No	Correct	Yes	The verification against the use of the re-calculated Fiat-Shamir signatures was carried out successfully and identified the invalid data provided as the expected ballots were not chosen for auditing using the invalid Fiat-Shamir signatures.
F 3	No	Correct	Yes	The verification against the randomness commitments made by each of the randomness generation servers for each of the PoD

				Printers was carried out successfully and identified the invalid data provided.
F 4	No	Correct	Yes	Each of the ballots chosen for auditing, for each of the PoD Printers, was successfully recalculated and matched the corresponding generic ballot unless there were modifications in the related data which the verifier identified.
F 5	No	Correct	Yes	For each of the ballots chosen for auditing, for each of the PoD Printers, the permutation commitment contained in each of the matching generic ballots from the Public WBB was successfully opened and verified unless there were modifications in the related data which the verifier identified.
F 6	No	Correct	Yes	The Mixnet input data was successfully verified. Each of the packed voter preferences for each of the races has a matching set of packings contained in the Mixnet input data. The verifier successfully identified the modification in voter preferences for the particular race which had been modified.
F 7	Yes	Correct	No	The Mixnet output data was successfully verified. Each of the output voter preferences contained in the CSV files was used to reorder the plaintext candidate identifiers which were then packed. Each of these plaintext packings had a matching Mixnet output packing.
F 8	No	Correct	No	The signatures against the commits made to the Public WBB contained in the extra_commits folder could not be successfully verified. The sample data was produced using an out of data version of the vVote code which did not have the functionality to produce the signature files. The signature data also will not be produced for data contained in the extra commits folder and therefore the verification of the Public WBB commits is skipped for data contained in the extra commits folder.
F 9	Yes	Correct	Yes	The plaintext identifiers could be successfully encrypted to match the base encrypted candidate identifiers.

Table 7-6 - Test results for sample dataset 4

Table 7-6 shows the results of the vVote Verifier against the functional requirements of the system. The results show that when valid data was presented to the vVote Verifier the verification took place correctly and when the verifier was presented with invalid data the verification process identified these issues correctly. A number of the issues identified by the vVote Verifier are included below.

```
17:40:03.993 ERROR VotePackingVerifier - Unable to verify Mixnet input Vote Packing process:
Could not find packing for: BallotRaceIdentifier [raceType=LA, district=Bundoora]: 5150101:10
```

The preferences for the LA race in the ballot with serial number 5150101:10 had been modified which was correctly identified.

```
17:38:07.714 ERROR BallotGenerationVerifier - Committed cipher and combined ballot cipher for
serial number: '2370101:202' with index: '0' do not match
```

The corresponding generic ballot with serial number 2370101:202 within a ciphers.json file had been modified which was correctly identified.

```
17:37:06.329 ERROR BallotGenerationVerifier - Committed cipher and combined ballot cipher for
serial number: '2060101:1997' with index: '5' do not match
```

The randomness values for the ballot with serial number 2060101:1997 chosen for auditing had been modified which was correctly identified.

```
17:36:57.892 ERROR BallotGenerationVerifier - The commitment does not match the given witness
and randomness values - Commitment with identifier: CommitIdentifier
[identifier=1406876400000force3, printerId=2060101], commitment:
960db9f5271e0fce735e226516123c8d8c9f7cbf60daf6a4614cb74f85f79a7e, witness:
b3a4bca4d220e1416d5f6daeadc5ccca9c4fd33c1eef09466e7dbe936a4fdf38, randomness:
fcdcadbf74017577fa5ee2034357827ac9d2be66d6919c13fefe5d360ba4d5b9
```

A witness value for the ballot with serial number 2060101:603 chosen for auditing had been modified which was correctly identified.

For each requirement the vVote Verifier produced correct verification behaviour and was successful when the data was valid and showed errors when the data was invalid.

7.7 Conclusion

This chapter has been used to provide comprehensive details of the sample data used to test the vVote Verifier system with the aim of evaluating and determining the successfulness of the system in terms of its initial requirements set out in Table 4-5. Also outlined was the way in which the verifier is run and an example output from the verifier was shown. The chapter finished with outlining the results produced against each of the initial requirements. The results show that the implementation of the vVote Verifier has been successful when the data provided has been complete. During the testing a number of issues were identified which will be explained in additional detail in the final chapter of this dissertation. It should be noted that the data samples containing invalid data were created naively and an attacker wishing to make modifications could do so in a far more subtle way.

Chapter 8 – Conclusion and Future Work

8.1 Introduction

This final chapter will be used to discuss the current state of the system developed throughout the course of the dissertation with regard to the initial requirements provided. It will also describe each of the stages included in this report. A list of project achievements will be used to highlight what has been carried out.

The chapter concludes with a number of possible extensions or modifications which could be made to the system which may improve its usefulness, usability and overall robustness as a verification tool had more time been available.

8.2 Conclusion

The vVote system was designed as an end-to-end verifiable electronic voting system building upon the foundations of the Prêt à Voter election system whilst making improvements and modifications to make it usable for a VEC election ready for use in a real election for the Australian state of Victoria. The key requirement for any electronic voting system is its verifiability. Verifiability aspects have been ingrained throughout the vVote system to provide verifiability to each of its core components meaning the processes and election as a whole can be verified by both those involved and any other interested party. Verifiability is the ability to check and validate that the correct operations have taken place and have been carried out in exactly the way they should. The vVote system provides a number of pieces of evidence which can be used for verifying the correctness of the election including that:

- Votes are Cast-as-intended – each voter's vote was cast in the way they intended.
- Votes are Counted-as-cast – each voter's vote is counted as they cast it.
- The tallying process is universally verifiable – anyone can check the list of encrypted votes produces the same tallied official election outcome.

This dissertation details the implementation of a stand-alone independent reference implementation of a verifier for the vVote system – the vVote Verifier. The vVote Verifier focuses on the verification of three core areas of the vVote system; specifically:

- The Ballot Generation process – That a verified randomly selected set of ballots have been generated successfully. If a large enough and unpredictable enough proportion of all those ballots generated are audited successfully it gives confidence in those ballots remaining un-audited which can then be used to vote anonymously with confidence.

- The Vote Packing process – That the performance improvements made to pack together a number of a voter preferences reducing the number of ciphertexts passed to the Mixnet have been carried out correctly. The verification includes both the pre-Mixnet validation that all the votes cast have been packed ready for the Mixnet and also the post-Mixnet validation that all the output voter preferences have been looked up using the Mixnet output packings.
- The commits made to the Public WBB – That each of the commits made to the Public WBB has been correctly hashed and signed by the Public WBB proving the information has not been modified or implanted onto the Public WBB in an unauthorised manner.

The development of the vVote Verifier system was organic and took place whilst the vVote system was still under development and therefore the changing nature of the sample election data received throughout the course of the project heavily influenced the design and implementation strategies utilised. The design of the system revolved around the increasingly complete sets of sample election data received, improving and adding features at the same time as the data became available.

Using the sample election data provided; the analysis, design, implementation and testing of the vVote Verifier has been explained step-by-step. The dissertation began with the analysis of the problem with the aim of producing a number of specific functional and non-functional requirements upon which the successfulness of the implementation could be judged. A brief feasibility study was used to determine how feasible the initial problem was in terms of its complexity, the timeframe allowed and the amount of work expected for its successful completion. The chapter also provided an in-depth look at the verification details for the chosen verifiable vVote components including the reasons why the verification should take place and how it might take place.

The next stage of the dissertation was to produce an initial design for the vVote Verifier system examined in the first chapter. The chapter provided an overall architectural design for the system including defining the high level components and their interactions. The chapter also provided a lower-level design of the system identifying the key classes and components needed for the successful implementation of the system. The chapter finished by providing the explicit details as to the algorithms which the vVote Verifier would use for carrying out the verification for each chosen component.

The implementation phase of the project looked at how the core components and classes defined in the design phase were brought into life. The chapter focused specifically on the verifier library, used throughout the system, as well as the main component verifiers. The main steps of the component verifiers were explained with regard to the corresponding requirements from Table 4-5 and how the implementation was used to meet the requirement. A section was also dedicated to a number of the challenges faced during the implementation of the vVote Verifier.

The final chapter was used to provide an overview of the testing carried out on the vVote Verifier. The sample data used was explained thoroughly as to its complexity and structure and for explaining the results produced. The results were presented alongside the initial requirements set out in Table 4-5. The results showed that the implementation of the vVote Verifier was successful when the data provided included the required information for successful validation of the vVote system. The results showed that the vVote Verifier system was able to verify the underlying verifiable protocols of the vVote system. The testing phase of the project uncovered a number of issues which required slight modifications to the vVote Verifier system.

The first issue was that in the second set of sample data (data_sample3), within the ballot generation verifiable data, there was a problem with the number of commitments made against the randomness data sent to each of the PoD Printers made by the randomness generation servers. In the sample data there are 4 PoD Printers used and 5 randomness generation servers used. Each of the PoD Printers should have received 5 sets of randomness data which would then be combined together and used to perform re-encryption. However, the first randomness generation server, with identifier MixServer1, seemed to have sent each of the PoD Printers two sets of randomness values and therefore had posted to the Public WBB two mixrandomcommit messages for each PoD Printer. This caused problems with verifying the randomness data included in the audit data for each ballot chosen for auditing. There was no way to determine which of the randomness values, corresponding to the two commitments made by the first randomness generation server, had been used in the combination. The problem was identified as being due to a failure on one of the randomness generation servers caused by a network failure or an incorrect public key. Any PoD Printer that receives multiple sets of randomness data from a single randomness generation server would indicate that the issue had occurred however this indication is not submitted to the Public WBB. The solution to this issue was that a test would be carried out for each set of randomness commitments present on the Public WBB for any randomness generation server for randomness data provided to any PoD Printer. If any of the randomness commitments was found to be valid then the check would be marked as valid. Warning messages will be shown when multiple commitments are found for a single PoD Printer originating from the same randomness generation server. This issue has been accepted as something the developers of the vVote system will address in the future to make the task of automatic verification easier.

The second issue uncovered was related to the size of the data provided to the vVote Verifier system. The first set of data received (data_ sample1) was around 100 MB whereas the second set of data received (data_sample3) was around 1.5 GB. As explained in Chapter 7 the first set of data included ballot generation data for only a single PoD Printer generating 1000 generic ballots whereas the second set of data contained ballot generation data for 4 PoD Printers each generating 2000 generic ballots. This increased data set size caused the code, when running on machines with low amounts of RAM, to throw an out of memory exception: "java.lang.OutOfMemoryError: Java heap space". This problem had not been expected although it should have been considered during the analysis stage. The system was designed in such a way that all relevant data in the current input sample data set would be read in prior to the verification taking place; this caused, on machines with low amounts of RAM, to throw the out of memory exception. The vVote Verifier, when run using machines with larger amounts of RAM, continued to function correctly. The solution to this problem was to selectively read in data as it is needed for the verification process. This solution requires that the files are repeatedly searched for specific data items, however, prevents errors from occurring. Additional details are provided for this issue in 6.4.3 Size of Input Data.

The next issue was that of providing the vVote Verifier with invalid or incomplete data sets. Because of the amount of work required and the complexity of the code there are, in places, points in the code where if a specific file is malformed or missing altogether the verification process will stop completely. The code was written in such a way that for the most part the assumption is that the data used is in the correct format. Had more time been available the vVote Verifier would have been made more robust against these kinds of issues; and attempts would have been made to handle the issues in a better way by providing additional details as to the issues faced with the current data item. The implementation, if something is very wrong with the data, will at least show to the user that there was something that went wrong and that it should be investigated rather than having the issues not being picked up at all and having that malformed data be acceptable.

One of the most troubling issues encountered throughout the dissertation was the lack of documentation for the Mixnet portion of the vVote system in addition to a number of its other components specifically missing the lack of technical details regarding the implementation details for specific components. The lack of documentation relating to the Mixnet was part of the decision taken to not additionally undertake the verification of the Mixnet functionality on top of the Surrey code base for the vVote system. The lack of documentation relating to other areas was less of an issue but still required numerous questions to be raised regarding the technical details of the vVote system components and additionally the examination of the source code for particular components which was undesirable.

8.2.1 Project Achievements

Throughout the course of the dissertation a number of goals and achievements were made:

- The design and implementation of a verifier focusing on the verifiability of the Ballot Generation process used within the vVote election system.
- The design and implementation of a verifier focusing on the verifiability of the Vote Packing process used within the vVote election system, including both the pre-Mixnet and post-Mixnet.
- The design and implementation of a verifier focusing on the verifiability of the public commits made to the Public WBB within the vVote election system.
- Additional understanding of advanced cryptographic techniques applied in a real world application. Theoretical details learnt during the taught modules of the year have been applied and put into practice.
- A partial understanding, in some places complete, of the vVote system and its functionality and how it can be used and applied as a real world electronic voting system which will be put into practice in an election in Victoria Australia in November 2014.

8.3 Future Work

Although the requirements, determined early on in the analysis stage of the project – shown in Table 4-5 and Table 4-6, were largely met a number of possible additions have been considered which could be used to improve the usefulness of the system, improve the usability and also improve its robustness. The additions and modifications come from both the testing carried out on the implementation as well as general improvements which could be made.

The first improvement which could be made would be to adapt the way in which the dataset is read and processed. Currently the entire data set is read and processed prior to the verification algorithms being carried out with the addition of a look up system to avoid reading and processing all of the data. A better and more robust system would be to utilise a database of some kind. Using a database, which does not require all the data to be held in memory concurrently, would improve the robustness of the system in that regardless of the size of the dataset the vVote Verifier would function in the same manner. The data required for the relevant steps would be looked up and loaded as and when they were needed rather than keeping a significant amount of surplus data held in memory during the entire verification process. The current implementation of the vVote Verifier is constructed using independent component verifiers which each read in the data relevant to their verification algorithms meaning the same data may be held in multiple places by independent verifiers giving a chance for inconsistency. A single database mechanism, which is filled in the correct manner a single time, would reduce the chance for inconsistencies throughout the vVote Verifier and would improve the robustness and efficiency of the system.

The second improvement identified would be the implementation of additional OpenSSL cryptographic functionality. Currently the vVote Verifier uses JCA, JCE and Bouncy Castle cryptographic libraries however these same libraries are also used throughout the vVote system which is being verified. For further confidence and independence the cryptographic functionality could be mirrored using completely independent libraries from OpenSSL. Each cryptographic operation could be carried out multiple times, once using the shared library and once using the independent OpenSSL library to provide a huge amount of additional confidence in the operations being carried out correctly. OpenSSL libraries are currently only used for verification of commitments made against randomness values however additional cryptographic operations could be implemented.

The final additional improvement to the vVote Verifier would to design and implement an additional component verifier tasked with validating the operation of the re-encryption based Mixnet. It was stated in the feasibility study, carried out in Chapter 4, that initially the verification of the Mixnet was also considered. Verification of the Mixnet was deemed to be infeasible due to the time requirements for implementing a large amount of additional code as well as taking the time to fully understand the logistics and operations of the Mixnet functionality. Most importantly the Mixnet verification was deemed to be out of scope because of the lack of available documentation and that it had not been finished and the code had not been locked down even at the time of writing this document. The Mixnet was not part of the University of Surrey delivery and therefore it was deemed to be acceptable that this component would not be part of the initial development of the vVote Verifier. The implementation of an additional component verifier tasked with the verification of the Mixnet could be carried out as a standalone project or an extension to the existing vVote Verifier.

As with any development project, had more time been available, additional time would have been spent on improving the implementation for example by making it more robust and by adding additional simple optimisations to improve the time taken for the verification process whilst not affecting the readability or the clarity of the code.

8.4 Final Statement

This project, although of substantial size – in terms of the amount of code developed – more than 24,000 lines of code, the size of the report written and the amount of research carried out, has been completed successfully when using the requirements outlined in Chapter 4 as its metric. Each of the functional and non-functional requirements has been fully achieved when the relevant data has been available. The overarching objective of the entire system was to develop verification tool to verify the behaviour of the vVote system which has been carried out successfully.

An independent standalone reference implementation for a verifier for the vVote system has been delivered to the vVote team in addition to reference documentation outlining the functionality and overview of both the components under verification as well as the verifier implementation. To date the vVote Verifier has proved to be reliable for verifying the sets of sample data provided.

I look forward, in anticipation, to testing the implementation against the data produced through the running of the system in a real election in Victoria Australia in November 2014.

Although of substantial size and taking a significant amount of time I feel the project has been thoroughly enjoyable and hugely worthwhile.

References

[1] C. Culnane, P. Y. A. Ryan, S. Schneider and V. Teague, “vVote: a Verifiable Voting System,” 24 June 2014. [Online]. Available: http://arxiv.org/abs/1404.6822. [Accessed 12 August 2014].

[2] P. Y. A. Ryan, D. Bismark, J. Heather, S. Schneider and Z. Xia, “The Pret a Voter Verifiable Election System,” 2010. [Online]. Available: http://www.pretavoter.com/publications/PretaVoter2010.pdf. [Accessed 19 August 2014].

[3] “1872 Reform Act (Secret Ballot),” spartacus-educational.com, [Online]. Available: http://spartacus-educational.com/PR1872.htm. [Accessed 18 August 2014].

[4] B. Brewster, “The Importance of Voting to Democracy,” [Online]. Available: https://www.sec.state.vt.us/kids/contest/2005/9_12_winner_2005.htm. [Accessed 18 August 2014].

[5] R. Aditya, “Secure Electronic Voting with Flexible Ballot Structure,” 2005.

[6] “Voting systems made simple,” Electoral Reform Society, [Online]. Available: http://www.electoral-reform.org.uk/voting-systems/. [Accessed 18 August 2014].

[7] B. Adida, “Advances in Cryptographic Voting Systems (2006),” 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.67.8154. [Accessed 18 August 2014].

[8] P. Hudson, “Why You Should Chase Power, Not Money,” elitedaily.com, 28 January 2014. [Online]. Available: http://elitedaily.com/money/entrepreneurship/why-you-should-chase-power-not-money/. [Accessed 18 August 2014].

[9] B. Schneier, “Internet Voting vs. Large-Value e-Commerce,” Schneier on Security, 15 February 2001. [Online]. Available: https://www.schneier.com/crypto-gram-0102.html#10. [Accessed 18 August 2014].

[10] W. Diffie and M. E. Hellman, “New directions in cryptography,” *IEEE Transactions on Information Theory,* vol. 22, no. 6, pp. 644-654, 1976.

[11] T. Elgamal, “A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms,” *IEEE Transactions on Information Theory,* vol. 31, no. 4, pp. 469-472, 1985.

[12] “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM,* pp. 120-126, February 1978.

[13] N. Sullivan, “A (relatively easy to understand) primer on elliptic curve cryptography,” http://arstechnica.com/, 24 October 2013. [Online]. Available: http://arstechnica.com/security/2013/10/a-relatively-easy-to-understand-primer-on-elliptic-curve-

cryptography/1/. [Accessed 18 August 2014].

[14] N. Koblitz, “Elliptic curve cryptosystems,” *Math. Comp,* vol. 48, pp. 203-209, 1987.

[15] figlesquidge, “ElGamal with elliptic curves,” Cryptography beta, 26 August 2013. [Online]. Available: http://crypto.stackexchange.com/questions/9987/elgamal-with-elliptic-curves. [Accessed 18 August 2014].

[16] B. Schneier, “Homomorphic Encryption Breakthrough,” Schneier on Security, 9 July 2009. [Online]. Available: https://www.schneier.com/blog/archives/2009/07/homomorphic_enc.html. [Accessed 18 August 2014].

[17] S. Adi, “How to share a secret,” *Communications of the ACM,* pp. 612-613, November 1979.

[18] D. W. Jones, “Some Problems with End-to-End Voting,” 2009. [Online]. Available: http://homepage.cs.uiowa.edu/~jones/voting/E2E2009.pdf. [Accessed 24 August 2014].

[19] “E-voting experiments end in Norway amid security fears,” bbc, 27 June 2014. [Online]. Available: http://www.bbc.co.uk/news/technology-28055678. [Accessed 19 August 2014].

[20] “About Prêt à Vote,” [Online]. Available: http://www.pretavoter.com/about.php. [Accessed 15 August 2014].

[21] B. Adida, “Helios: web-based open-audit voting,” in *SS'08 Proceedings of the 17th conference on Security symposium*, 2008.

[22] J. Benaloh, “Simple verifiable elections,” in *EVT'06 Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop 2006 on Electronic Voting Technology Workshop*, 2006.

[23] K. Sako and J. Killian, “Receipt-free mix-type voting scheme: a practical solution to the implementation of a voting booth,” in *EUROCRYPT'95 Proceedings of the 14th annual international conference on Theory and application of cryptographic techniques*, 1995.

[24] D. Chaum and T. P. Pedersen, “Wallet Databases with Observers,” in *CRYPTO '92 Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, 1992.

[25] D. Chaum, C. Richard, J. Clark, A. Essex, S. Popoveniuc, R. L. Rivest, P. Y. A. Ryan, E. Shen and A. T. Sherman, “Scantegrity II: end-to-end verifiability for optical scan election systems using invisible ink confirmation codes,” in *EVT'08 Proceedings of the conference on Electronic voting technology*, 2008.

[26] Victorian Electoral Commission, “Preferential voting,” [Online]. Available: https://www.vec.vic.gov.au/Vote/vote-about-prefvote.html. [Accessed 30 July 2014].

[27] Victorian Electoral Commission, “How to Vote Cards and Group Voting Tickets,” [Online]. Available: https://www.vec.vic.gov.au/stand/stand-howtovotecards.html. [Accessed 30 July 2014].

[28] M. Jakobsson, A. Juels and R. L. Rivest, “Making Mix Nets Robust for Electronic Voting by Randomized Partial Checking,” in *Proceedings of the 11th USENIX Security Symposium*, 2002.

[29] S. Khazaei and D. Wikström, “Randomized partial checking revisited,” in *CT-RSA'13 Proceedings of the 13th international conference on Topics in Cryptology*, 2013.

[30] M. Gogolewski, M. Klonowski, P. Kubiak, M. Kutyłowski, A. Lauks and F. Zagórski, “Kleptographic Attacks on E-Voting Schemes,” 2006. [Online]. Available: http://kutylowski.im.pwr.wroc.pl/articles/wele-folie.pdf. [Accessed 2014 August 16].

[31] M. Abdalla, J. A. Hea, M. Bellare and C. Namprempre, “From Identification to Signatures via the Fiat-Shamir Transform: Minimizing Assumptions for Security and Forward-Security,” in *Advances in Cryptology — EUROCRYPT 2002*, 2002, pp. 418-433.

[32] M. Jakobsson, A. Juels and R. L. Rivest, “Making Mix Nets Robust for Electronic Voting by Randomized Partial Checking,” San Francisco, 2002.

[33] N. Koblitz, “Elliptic curve cryptosystems,” *Mathematics of Computation,* vol. 48, no. 177, pp. 203-209, 1987.

[34] A. Fiat and S. Adi, “How to prove yourself: practical solutions to identification and signature problems,” 1986.

[35] D. Crockford, “JSON in Java,” [Online]. Available: http://json.org/java/. [Accessed 2014 January 10].

[36] “The Legion of the Bouncy Castle,” Legion of the Bouncy Castle Inc, [Online]. Available: https://www.bouncycastle.org/java.html. [Accessed 2014 January 15].

[37] “Introduction,” Java Pairing-Based Cryptography Library (JPBC), [Online]. Available: http://gas.dia.unisa.it/projects/jpbc/. [Accessed 26 August 2014].

[38] “The XIMIX Project,” The XIMIX Project, 18 June 2014. [Online]. Available: http://www.cryptoworkshop.com/ximix/doku.php. [Accessed 26 August 2014].

[39] “Welcome to the OpenSSL Project,” OpenSSL, [Online]. Available: https://www.openssl.org/. [Accessed 26 August 2014].

[40] R. Cramer, R. Gennaro and B. Schoenmakers, “A secure and optimally efficient multi-authority election scheme,” in *EUROCRYPT'97 Proceedings of the 16th annual international conference on Theory and application of cryptographic techniques*, Berlin, 1997.

[41] “ASN.1 file format,” The NCBI Structure Group, 8 January 2014. [Online]. Available: http://www.ncbi.nlm.nih.gov/Structure/asn1.html. [Accessed 8 August 2014].

[42] “Class PropertiesConfiguration,” Apache Commons, [Online]. Available: http://commons.apache.org/proper/commons-configuration/apidocs/org/apache/commons/configuration/PropertiesConfiguration.html. [Accessed

8 August 2014].

[43] E. Barker and J. Kelsey, “Recommendation for Random Number,” January 2012. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf. [Accessed 2 August 2014].

[44] G. Smith, “opencsv,” opencsv, 28 July 2011. [Online]. Available: http://opencsv.sourceforge.net/. [Accessed 23 August 23].

[45] J. Heather, C. Culnane, S. Schneider, S. Srinivasan and Z. Xia, “Solving the Discrete Logarithm Problem for Packing Candidate Preferences,” in *Security Engineering and Intelligence Informatics*, A. Cuzzocrea, C. Kittl, D. Simos E, E. Weippl and L. Xu, Eds., Regensburg, Springer Berlin Heidelberg, 2013, pp. 209-221.

[46] L. D. Kudryavtsev and M. K. Samarin, “Lagrange interpolation formula,” Encyclopedia of Mathematics, 7 February 2011. [Online]. Available: http://www.encyclopediaofmath.org/index.php?title=Lagrange_interpolation_formula&oldid=17497. [Accessed 8 August 2014].

[47] B. Felix, “Efficient cryptographic protocol design based on distributed el gamal encryption,” in *ICISC'05 Proceedings of the 8th international conference on Information Security and Cryptology*, Berlin, 2005.

[48] M. Jakobsson, A. Juels and R. L. Rivest, “Making Mix Nets Robust For Electronic Voting By,” in *Proceedings of the 11th USENIX Security Symposium*, San Francisco, 2002.

[49] J.-P. Berrut and L. N. Trefethen, “Barycentric Lagrange Interpolation,” 2004. [Online]. Available: https://people.maths.ox.ac.uk/trefethen/barycentric.pdf. [Accessed 8 August 2014].

Appendices

Appendix A – Vote Message Data

Example POD Message

```
{"boothID":"TestDeviceOne","serialNo":"TestDeviceOne:1","ballotReductions":[[{"index":18,"cand
idateIndex":18,"randomness":"LSw7m9jfUY7fdNbXxZg/Pn12hFlhAkLLIyx4PJZ/Dnw="},{"index":17,"candi
dateIndex":16,"randomness":"bGaKRktZErypzOMiAyA/EqVVRL+5b2lx6r+mjvLV0IQ="},{"index":15,"candid
ateIndex":12,"randomness":"O5VKdpmMVkabCYKow4AFurC2uc7aSff3LYqF1G9dYnY="},{"index":8,"candidat
eIndex":17,"randomness":"eR1ra95iCX3pj4eW70j7/qJeKSy2VZo3ZZ5gkIJT8hE="},{"index":6,"candidateI
ndex":13,"randomness":"rvWgOjiwen+m5GE5Fxc4jKcimhPI1n9+cB/cWGk7G8k="},{"index":5,"candidateInd
ex":14,"randomness":"fsmXtbwGS3eVsQ1aFXXjVMQQjY24Bl9KrShU4iMWhC4="},{"index":3,"candidateIndex
":15,"randomness":"d4MlpCgW4SOCEpK3gmS3AhNaybkFhr9PF2a2N6ePwUA="},{"index":1,"candidateIndex":
19,"randomness":"c+P3vAaoHR12wbhRBSoVR7u2v2luwahLkmEm1lGPPi8="}],[{"index":6,"candidateIndex":
8,"randomness":"GbsxdV2snxOAMs08/FvIQNCitpNoTp9urSzdI6LRVIo="},{"index":2,"candidateIndex":9,"
randomness":"G6TYkceCeZ0vMQqqJLa9OAu6OW83Wa6BoPxt2tBdovs="}],[{"index":46,"candidateIndex":39,
"randomness":"9DCxz+e59oiSWT7H5Gmg+6tqyc5iuUmXUIP59F6jBos="},{"index":45,"candidateIndex":30,"
randomness":"77h2NCEGPYnVGy+2nBNXJu31ctVmtYisej4F+O6gRCI="},{"index":44,"candidateIndex":42,"r
andomness":"JGLtKh5l98VTfY3hMWXkik8zexYFVgtFJPXE15bYVm0="},{"index":43,"candidateIndex":36,"ra
ndomness":"3smQYX4hzXa9sxjbLhzkXmq3tAfKx/l2Wgq5174i0/Y="},{"index":41,"candidateIndex":26,"ran
domness":"KawMfUwZc2n6pHnYlVZ+cOJaF2LLDJ3zByhF2GjPxmI="},{"index":40,"candidateIndex":48,"rand
omness":"trE2dLEKNFKWRibL3J3HkX45gH+xuTIWQWg1nQlqqSI="},{"index":38,"candidateIndex":47,"rando
mness":"hBOMg8Rk4416zgT2vXIprFU8agXtwGQzbasCP7JLrNc="},{"index":37,"candidateIndex":38,"random
ness":"tZSLi68w8eExi9ReUQTCw7hgi8jPWNGCKAD8uqWL0j8="},{"index":35,"candidateIndex":43,"randomn
ess":"doD5hWGbQRKygdSgIkGffUAZeJA3hvZk731w2vvfzKw="},{"index":31,"candidateIndex":35,"randomne
ss":"k6W5gc6+Uakq57JnKZfGMPce1SawZu+6djY+2gQFyFA="},{"index":30,"candidateIndex":44,"randomnes
s":"riLMYSONigO8lYvikOJb5FNLYOOuoQz8jFJimLgDffo="},{"index":28,"candidateIndex":28,"randomness
":"4TSXa52hYhnKnQZXv+RD7FCN9ksPkMbrCScuCTynfLg="},{"index":27,"candidateIndex":31,"randomness"
:"t0eFLwN7cG6nH4T3TvdDCkkXTQMbUi+DheQ+aRJtRqs="},{"index":24,"candidateIndex":25,"randomness":
"ZEWT8MDqBD7s744LkKX17qsRQXItTRx6HrYy6wQtzr4="},{"index":21,"candidateIndex":40,"randomness":"
cTcjA4LDIM4jx4+CDj70PEeVzIKaJrwbgqUt5XfH44s="},{"index":19,"candidateIndex":46,"randomness":"v
Jx8Qb7Eug7uf78u9QOusEilboMWCPYHZR9r+APKEPM="},{"index":17,"candidateIndex":41,"randomness":"p5
pwNCGUDbyx6qMVIV1OpscJ/vhbRKrhnkKtMkgIyIM="},{"index":15,"candidateIndex":49,"randomness":"i/J
lBX33UcsuIot3JRJGst/ZCl58g1l7mBaOHx0ceKw="},{"index":9,"candidateIndex":32,"randomness":"LJqz/
3bpSg87V1ZVeSLpH3FwBK/O77/tlWSJdoSGAi4="},{"index":8,"candidateIndex":37,"randomness":"B5c+rVP
Wgcj+1dpR/OGWeRoI76BIseZRf0LSQS8lSVE="},{"index":7,"candidateIndex":45,"randomness":"FK+xwx13U
9/f5wBw6qDoXryoZ7wbkcIrPWzrt5h7APQ="},{"index":5,"candidateIndex":27,"randomness":"34W2M6FWezL
jmtYkbj8muyAiFFsgIVfweHQF3H5WkVQ="},{"index":4,"candidateIndex":33,"randomness":"qob+DxLIXG2DM
DjYAyHA+tWk+VCBZdJ5xmbzfz2YZpY="},{"index":2,"candidateIndex":29,"randomness":"qrqudz02Elc+Amj
+PROu7yX/peNflCzQ/qVmvzY3rHM="},{"index":0,"candidateIndex":34,"randomness":"0O2NStB0KYIdb1Hzn
xjv/nXmVw3ML95WMNhB+iX3/wg="}]],"commitTime":"1403247600000","boothSig":"Maxvg0qPOAYrvs0PTbvo5
P5Q+E4uijodlRNI5X56FDttsPvzB5sFng==","district":"Northcote","type":"pod"}
```

Example Vote Message

```
{"boothID":"TestEVMOne","serialNo":"TestDeviceOne:1","startEVMSig":"GQGToO6lBsFSODqgUPAHGd4rHb
FYkU4UnDoTez7gOPzsG9qLQnKjqw==","commitTime":"1403247600000","boothSig":"G2NOtdW/mV/ys/hwlf8Wk
Aa9dtFD0iNeGMNm0k1E1Ts8wfrmYJvm8g==","races":[{"id":"LA","preferences":["2","4","6","12","8","
3","10","9","11","1","7","5"]},{"id":"LC_ATL","preferences":[" "," "," "," "," "," ","
","1"]},{"id":"LC_BTL","preferences":[" "," "," "," "," "," "," "," "," "," "," "," ","
"," "," "," "," "," "," "," "," "," ","
"]}],"serialSig":"EKbUaNQPq4kapgveFklhPeRT754M6SQ6a4eoTTSv3xWVxgIArSNxGw==","district":"Northc
ote","type":"vote","_vPrefs":"2,4,6,12,8,3,10,9,11,1,7,5: , , , , , , ,1: , , , , , , , , , ,
, , , , , , , , , , , , , , :"}
```

Appendix B – Ballot Generation Messages

Mix Random Commit Message

```
{"fileSize":3088174,"submissionID":"e2623d4b-5f0e-4515-bf2e-
d5b6ab578b39","printerID":"TestDeviceOne","boothID":"MixServer1","_digest":"kVsEqjaUXBGSxRkg0h
aDK1zo5QU=","commitTime":"1403161200000","boothSig":"W45xxdIVGo41GnhlpkdsYfXC7fYf+Y/WY8dhDOdMw
lHaKFlHRYV67w==","digest":"kVsEqjaUXBGSxRkg0haDK1zo5QU=","type":"mixrandomcommit","_fileName":
"WBBUpload4734813346612278093.zip"}
```

Ballot Generation Commit Message

```
{"fileSize":12569441,"submissionID":"48fb5db2-2762-492f-b647-
1b9f2232a2cf","boothID":"TestDeviceOne","_digest":"qO3nzIm5hus7WZtq19yin12HepE=","commitTime":
"1403161200000","boothSig":"HpoWH7esMj0oHsRnBs3YqJyAeTtQiUSbjqeis+sL55r29FFoYZ5kMw==","digest"
:"qO3nzIm5hus7WZtq19yin12HepE=","type":"ballotgencommit","_fileName":"WBBUpload635150573797729
6283.zip"}
```

Ballot Audit Commit Message

```
{"fileSize":316757,"submissionID":"e8a6d307-9bb3-4a8e-a02f-
4c8b6f249380","boothID":"TestDeviceOne","_digest":"Bn8h7oAQGMegIskslYUSwE+QSmA=","commitTime":
"1403161200000","boothSig":"TOGrGNH6p0zbh8DxfY0kSe89uHMCAiOct6PeYuIOsUjU0/p7DObUxQ==","digest"
:"Bn8h7oAQGMegIskslYUSwE+QSmA=","type":"ballotauditcommit","_fileName":"WBBUpload5182101632484
24503.zip"}
```

Appendix C – Mixnet Output Data

f6e09076-e6ca-415d-9a45-a16b6da0ea9e_LA_A.Broadmeadows.csv

```
1,6,5,4,3,2
1,5,3,2,4,6
6,5,4,3,1,2
4,3,2,1,,
4,5,6,3,2,1
```

ed4c4f1b-65ff-43d6-a2af-78da487777b0-

Broadmeadows_ATL_A.Northern Metropolitan_Broadmeadows.csv

```
,1,
1,,
,1,
```

ed4c4f1b-65ff-43d6-a2af-78da487777b0-

Broadmeadows_BTL_A.Northern Metropolitan_Broadmeadows.csv

```
6,7,8,9,1,2,3,4,5,10,11,12,13
1,2,3,4,5,6,7,,,,,,
```

Appendix D – Certificate Data

```
{
        "jksPath":"./demo/TestDeviceOne/peer.jks",
        "Peer1_SigningSK2":{
                "pubKeyEntry":{

        "publicKey":"U/yBySWaQHUC/dQwhBo4NSJPqJgQ1kN4o0r9vfO2Dme7OQspFQUhkVmnjti2dWPnctGpz/r63
K/oKHU2YuSl+ycOOVEke4hu1Rf7c1wqnR1QRgw5VJi4QBf/sa1EjpdMei7sA10qp04rXbYHHbgaqXL4FGpQCPRf",

        "g":"NIO14Z3zPIr6QbIPRMgZkAxh7cIZWgNi4a18lP41G1av/Gd4z4p91kbv0u7PNrHz01GBcXx0u+XHM0Q1M
LkLatsEExHKT3hRB8jN8Yo/920Uk3Z7ImxFdLqEVxIncVj7MgLnjCFsZmwW07H9Aj1PBPFpmplu9bWZ",

        "partialPublicKey":"ZRvW/JSKx+ML4rZ5OvZCGEZQ4tMnBh/c48aLuwYtsGh7g3gJETTcB0YgwR040ojjBW
XZSHFseBlIzMpPEZ+tbVtCcrHc84V9MXzPMEmRqlAgfcfv9MC/xgbDt5Rut2PmNQWFfkMpIRT7U0DdjFKNAc/o07p/sstQ
",
                        "sequenceNo":0
                }

        },
        "Peer5_SigningSK2":{
                "pubKeyEntry":{

        "publicKey":"U/yBySWaQHUC/dQwhBo4NSJPqJgQ1kN4o0r9vfO2Dme7OQspFQUhkVmnjti2dWPnctGpz/r63
K/oKHU2YuSl+ycOOVEke4hu1Rf7c1wqnR1QRgw5VJi4QBf/sa1EjpdMei7sA10qp04rXbYHHbgaqXL4FGpQCPRf",
        "g":"NIO14Z3zPIr6QbIPRMgZkAxh7cIZWgNi4a18lP41G1av/Gd4z4p91kbv0u7PNrHz01GBcXx0u+XHM0Q1M
LkLatsEExHKT3hRB8jN8Yo/920Uk3Z7ImxFdLqEVxIncVj7MgLnjCFsZmwW07H9Aj1PBPFpmplu9bWZ",

        "partialPublicKey":"L/4EQ4OVnxAaZMFbSA9ARAKrye0WfsmGWoK4xuW/RpqbqcKEIlTxSUTJJpHHMg/OMz
DMPZzU9dRWwSfKHZrI3yh8EQo08/z/cFwhwnisGj5AYs12LehG1CteELb7B0wWeziFAk+77AQ0FsJtntRyIn6BBUU5p+Hh
",
                        "sequenceNo":4
                }

        },
        "Peer4_SigningSK2":{
                "pubKeyEntry":{

        "publicKey":"U/yBySWaQHUC/dQwhBo4NSJPqJgQ1kN4o0r9vfO2Dme7OQspFQUhkVmnjti2dWPnctGpz/r63
K/oKHU2YuSl+ycOOVEke4hu1Rf7c1wqnR1QRgw5VJi4QBf/sa1EjpdMei7sA10qp04rXbYHHbgaqXL4FGpQCPRf",

        "g":"NIO14Z3zPIr6QbIPRMgZkAxh7cIZWgNi4a18lP41G1av/Gd4z4p91kbv0u7PNrHz01GBcXx0u+XHM0Q1M
LkLatsEExHKT3hRB8jN8Yo/920Uk3Z7ImxFdLqEVxIncVj7MgLnjCFsZmwW07H9Aj1PBPFpmplu9bWZ",

        "partialPublicKey":"bVAJ4EeecnSLCWiD9IA3jtw0I4wreLnehZx0cJXheypWn7DnCJJsSkDuKwxaMuLln5
fo/bKupY8wxVgQVGdp0ayVwE1WAMbZrNQ1xSg21n9oQtHqNyptphfZdCEyDhUprY1fHVj74zFI/dbDlkDdseqDWKcsNYXN
",
                        "sequenceNo":3
                }

        },
        "Peer3_SigningSK2":{
                "pubKeyEntry":{

        "publicKey":"U/yBySWaQHUC/dQwhBo4NSJPqJgQ1kN4o0r9vfO2Dme7OQspFQUhkVmnjti2dWPnctGpz/r63
K/oKHU2YuSl+ycOOVEke4hu1Rf7c1wqnR1QRgw5VJi4QBf/sa1EjpdMei7sA10qp04rXbYHHbgaqXL4FGpQCPRf",

        "g":"NIO14Z3zPIr6QbIPRMgZkAxh7cIZWgNi4a18lP41G1av/Gd4z4p91kbv0u7PNrHz01GBcXx0u+XHM0Q1M
LkLatsEExHKT3hRB8jN8Yo/920Uk3Z7ImxFdLqEVxIncVj7MgLnjCFsZmwW07H9Aj1PBPFpmplu9bWZ",

        "partialPublicKey":"M+Z2adfdb4IFO+GHoSR198xbBz5bW85/w7zhvBkhpD2ZPJKYC3x+MAhHY1t4WhRJ+8
3/XpWX3OdeS/NDMjCI1irfypE5Ud53ci6jseBbMWtIaS8zt4HN6OAFKJ7ulBp2Elk/I1pmO/geEOTELKd7On5/1nK02ISo
",
                        "sequenceNo":2
                }

        },
        "Peer2_SigningSK2":{
                "pubKeyEntry":{

        "publicKey":"U/yBySWaQHUC/dQwhBo4NSJPqJgQ1kN4o0r9vfO2Dme7OQspFQUhkVmnjti2dWPnctGpz/r63
K/oKHU2YuSl+ycOOVEke4hu1Rf7c1wqnR1QRgw5VJi4QBf/sa1EjpdMei7sA10qp04rXbYHHbgaqXL4FGpQCPRf",

        "g":"NIO14Z3zPIr6QbIPRMgZkAxh7cIZWgNi4a18lP41G1av/Gd4z4p91kbv0u7PNrHz01GBcXx0u+XHM0Q1M
LkLatsEExHKT3hRB8jN8Yo/920Uk3Z7ImxFdLqEVxIncVj7MgLnjCFsZmwW07H9Aj1PBPFpmplu9bWZ",

        "partialPublicKey":"XRF7sB3cswyrfsIs0dUs76HWfQMp9Xh2TIOEmc0cVdKkGZPW66cl2mkpoIOLFciFJ1
xgSWKDea723yS2JEaGFOSClwdslFZeye/1/ykXN3odTXM9ioJsWqH/B8eUEd1EXh0JmV19RfPXBDxjCLjgSwwJ8RYyWOKp
",
                        "sequenceNo":1
                }

        },
        "WBB":{
                "pubKeyEntry":{

        "publicKey":"U/yBySWaQHUC/dQwhBo4NSJPqJgQ1kN4o0r9vfO2Dme7OQspFQUhkVmnjti2dWPnctGpz/r63
K/oKHU2YuSl+ycOOVEke4hu1Rf7c1wqnR1QRgw5VJi4QBf/sa1EjpdMei7sA10qp04rXbYHHbgaqXL4FGpQCPRf",

        "g":"NIO14Z3zPIr6QbIPRMgZkAxh7cIZWgNi4a18lP41G1av/Gd4z4p91kbv0u7PNrHz01GBcXx0u+XHM0Q1M
LkLatsEExHKT3hRB8jN8Yo/920Uk3Z7ImxFdLqEVxIncVj7MgLnjCFsZmwW07H9Aj1PBPFpmplu9bWZ"
                }

        }
}
```

Appendix E – Commit Data

Ballot Generation Commit JSON File - 1403161200000.json

Mix Random Commit Message 1

{"fileSize":3088263,"submissionID":"0a6c09e1-95aa-4c69-9e87-863033905064","printerID":"TestDeviceOne","boothID":"MixServer4","_digest":"pKMp38qYQWWvGrE0a0+6YGeRNW0=","commitTime":"1403161200000","boothSig":"NgIYt59FnnYGD2NBa9EdRrLS9TJI+vloQfphDj0zq1hFLcrsoXFGhA==","digest":"pKMp38qYQWWvGrE0a0+6YGeRNW0=","type":"mixrandomcommit","_fileName":"WBBUpload4828268762118691532.zip"}

Mix Random Commit Message 2

{"fileSize":3088159,"submissionID":"38bd09ba-f836-4ad7-8e80-dfaeadecfd83","printerID":"TestDeviceOne","boothID":"MixServer3","_digest":"/4yJuyfBwBMOplNf1da/b6UthNU=","commitTime":"1403161200000","boothSig":"JshNVOGGxDRs7FdxgxQKMG8+PFEr1Mpf4ACC82bc81GyxLR+ttovQw==","digest":"/4yJuyfBwBMOplNf1da/b6UthNU=","type":"mixrandomcommit","_fileName":"WBBUpload5036147914030834553.zip"}

Mix Random Commit Message 3

{"fileSize":3088389,"submissionID":"63b07313-5967-4afc-a523-17dae69d611d","printerID":"TestDeviceOne","boothID":"MixServer2","_digest":"Rn+mwQy24PxDJlAUwuFfmLKOZHo=","commitTime":"1403161200000","boothSig":"GinNaeCn3/7auGJadgXx8TPUCExfSbIqP8kH6rJKXAulWMsrQmralg==","digest":"Rn+mwQy24PxDJlAUwuFfmLKOZHo=","type":"mixrandomcommit","_fileName":"WBBUpload4389046009419617073.zip"}

Mix Random Commit Message 4

{"fileSize":3088165,"submissionID":"95b227ce-e97c-43fe-983a-90623645f5f0","printerID":"TestDeviceOne","boothID":"MixServer5","_digest":"WTsZtTh7uVj1H1+JrQZXTOiA3Fo=","commitTime":"1403161200000","boothSig":"DKY2NnUtfZrITDRvU5hSNzMVKxUAa7GeRPNru7BsNpmCS6AW9R75BA==","digest":"WTsZtTh7uVj1H1+JrQZXTOiA3Fo=","type":"mixrandomcommit","_fileName":"WBBUpload573743525531581813.zip"}

Mix Random Commit Message 5

{"fileSize":3088174,"submissionID":"e2623d4b-5f0e-4515-bf2e-d5b6ab578b39","printerID":"TestDeviceOne","boothID":"MixServer1","_digest":"kVsEqjaUXBGSxRkg0haDK1zo5QU=","commitTime":"1403161200000","boothSig":"W45xxdIVGo41GnhlpkdsYfXC7fYf+Y/WY8dhDOdMwlHaKFlHRYV67w==","digest":"kVsEqjaUXBGSxRkg0haDK1zo5QU=","type":"mixrandomcommit","_fileName":"WBBUpload4734813346612278093.zip"}

Ballot Generation Commit Message

{"fileSize":12569441,"submissionID":"48fb5db2-2762-492f-b647-1b9f2232a2cf","boothID":"TestDeviceOne","_digest":"qO3nzIm5hus7WZtq19yin12HepE=","commitTime":"1403161200000","boothSig":"HpoWH7esMj0oHsRnBs3YqJyAeTtQiUSbjqeis+sL55r29FFoYZ5kMw==","digest":"qO3nzIm5hus7WZtq19yin12HepE=","type":"ballotgencommit","_fileName":"WBBUpload6351505737977296283.zip"}

Ballot Audit Commit Message

{"fileSize":316757,"submissionID":"e8a6d307-9bb3-4a8e-a02f-4c8b6f249380","boothID":"TestDeviceOne","_digest":"Bn8h7oAQGMegIskslYUSwE+QSmA=","commitTime":"1403161200000","boothSig":"TOGrGNH6p0zbh8DxfY0kSe89uHMCAiOct6PeYuIOsUjU0/p7DObUxQ==","digest":"Bn8h7oAQGMegIskslYUSwE+QSmA=","type":"ballotauditcommit","_fileName":"WBBUpload518210163248424503.zip"}

Ballot Generation Commit Signature File - 1403161200000_signature.json

{"jointSig":"C6USaMCw8pNJPMeGi6AJdmIsNEE0hbc86eqUv+RpSu2sQTfBvOTinw==","jsonFile":"1403161200000.json","commitTime":"1403161200000","attachmentFile":"1403161200000_attachments.zip"}

Ballot Generation Commit Attachment File (structure) - 1403161200000_attachments.zip

```
1403161200000_attachments.zip
-   WBBUpload4389046009419617073.zip
                    - commitData.json
-   WBBUpload4734813346612278093.zip
                    - commitData.json
-   WBBUpload4828268762118691532.zip
                    - commitData.json
-   WBBUpload5036147914030834553.zip
                    - commitData.json
-   WBBUpload518210163248424503.zip
                    - AuditDataFile.json
                    - BallotSubmitResponse.json
-   WBBUpload573743525531581813.zip
                    - commitData.json
-   WBBUpload6351505737977296283.zip
                    - ciphers.json
```

Appendix F – Schema data

Ballot Generation Schema – ballotGenSchema.json

```
{
        "type":"object",
        "id": "ballotGen",
        "required":["finalCommits","baseEncryptedCandidateIds","plaintextCandidateIds","public
Key","districtConfig","ballotGenConf","commitData","ciphersData","auditData","certsFile","ball
otSubmitResponse"],
        "additionalProperties":false,
        "properties":{
                "finalCommits": {
                        "type":"string",
                        "id": "finalCommits"
                },
                "baseEncryptedCandidateIds": {
                        "type":"string",
                        "id": "baseEncryptedCandidateIds"
                },
                "plaintextCandidateIds": {
                        "type":"string",
                        "id": "plaintextCandidateIds"
                },
                "publicKey": {
                        "type":"string",
                        "id": "publicKey"
                },
                "districtConfig": {
                        "type":"string",
                        "id": "districtConfig"
                },
                "ballotGenConf": {
                        "type":"string",
                        "id": "ballotGenConf"
                },
                "commitData": {
                        "type":"string",
                        "id": "commitData"
                },
                "ciphersData": {
                        "type":"string",
                        "id": "ciphersData"
                },
                "auditData": {
                        "type":"string",
                        "id": "auditData"
                },
                "certsFile": {
                        "type":"string",
                        "id": "certsFile"
                },
                "ballotSubmitResponse": {
                        "type":"string",
                        "id": "ballotSubmitResponse"
                }
        }
}
```

Vote Schema – voteschema.json

```
{
        "type":"object",
        "id": "#",
        "required":["races","boothID","boothSig","serialNo","serialSig","type","startEVMSig","
district"],
        "additionalProperties":true,
        "properties":{
                "races": {
                        "type":"array",
                        "id": "races",
                        "additionalItems": true,
                        "items":[{
                                        "type":"object",
                                        "id": "0",
                                        "required":["id","preferences"],
                                        "additionalProperties":true,
                                        "properties":{
                                                "id": {
                                                        "type":"string",
                                                        "id": "id",
                                                        "enum": ["LA"]
                                                },
                                                "preferences": {
                                                        "type":"array",
                                                        "id": "preferences",
                                                        "items":
                                                                {
                                                                        "type":"string",
                                                                        "id": "0",
                                                                        "pattern":"^[1-9 ][0-9]?$"
                                                                }


                                                }
                                        }
                                },
                                {
                                        "type":"object",
                                        "id": "1",
                                        "required":["id","preferences"],
                                        "additionalProperties":true,
                                        "properties":{
                                                "id": {
                                                        "type":"string",
                                                        "id": "id",
                                                        "enum": ["LC_ATL"]
                                                },
                                                "preferences": {
                                                        "type":"array",
                                                        "id": "preferences",
                                                        "items":
                                                                {
                                                                        "type":"string",
                                                                        "id": "0",
                                                                        "pattern":"^[1-9 ][0-9]?$"
                                                                }


                                                }
                                        }
                                },
                                {
                                        "type":"object",
                                        "id": "2",
                                        "required":["id","preferences"],
                                        "additionalProperties":true,
                                        "properties":{
                                                "id": {
                                                        "type":"string",
                                                        "id": "id",
                                                        "enum": ["LC_BTL"]
                                                },
                                                "preferences": {
                                                        "type":"array",
                                                        "id": "preferences",
                                                        "items":
                                                                {
                                                                        "type":"string",
                                                                        "id": "0",
                                                                        "pattern":"^[1-9 ][0-9]?$"
                                                                }


                                                }
                                        }
                                }]




                },
                "boothID": {
                        "type":"string",
                        "id": "boothID"
                },
                "boothSig": {
                        "type":"string",
                        "id": "boothSig",
                        "pattern":"^(?:[A-Za-z0-9+/]{4})*(?:[A-Za-z0-9+/]{2}==|[A-Za-z0-
9+/]{3}=)?$"
                },
                "serialNo": {
                        "type":"string",
                        "id": "serialNo",
                        "pattern" : "^[A-Za-z0-9]*:[0-9]{1,7}$"
                },
                "serialSig": {
                        "type":["string","array"],
                        "id": "serialSig",
                        "pattern":"^(?:[A-Za-z0-9+/]{4})*(?:[A-Za-z0-9+/]{2}==|[A-Za-z0-
9+/]{3}=)?$",
                        "items":{
                                        "type":"object",
                                        "id": "serialSigs",
                                        "required":["WBBID","WBBSig"],
                                        "additionalProperties":true,
                                        "properties": {
                                                "WBBID": {
                                                        "type":"string",
                                                        "id": "WBBID"
                                                },
                                                "WBBSig": {
                                                        "type":"string",
                                                        "id": "WBBSig",
                                                        "pattern":"^(?:[A-Za-z0-9+/]{4})*(?:[A-
Za-z0-9+/]{2}==|[A-Za-z0-9+/]{3}=)?$"
                                                }
                                        }
                                }


                },
                "type": {
                        "type":"string",
                        "id": "type",
                        "enum": ["vote"]
                },
                "startEVMSig": {
                        "type":["string","array"],
                        "id": "serialSig",
                        "pattern":"^(?:[A-Za-z0-9+/]{4})*(?:[A-Za-z0-9+/]{2}==|[A-Za-z0-
9+/]{3}=)?$",
                        "items":{
                                        "type":"object",
                                        "id": "serialSigs",
                                        "required":["WBBID","WBBSig"],
                                        "additionalProperties":true,
                                        "properties": {
                                                "WBBID": {
                                                        "type":"string",
                                                        "id": "WBBID"
                                                },
                                                "WBBSig": {
                                                        "type":"string",
                                                        "id": "WBBSig",
                                                        "pattern":"^(?:[A-Za-z0-9+/]{4})*(?:[A-
Za-z0-9+/]{2}==|[A-Za-z0-9+/]{3}=)?$"
                                                }
                                        }
                                }


                },
                "district": {
                        "type":"string",
                        "id": "district"
                },
                "_vPrefs": {
                        "type":"string",
                        "id": "district"
                }
        }
}
```

Schema list – schema_list.json

```
[
{"id":"VOTE_SCHEMA",
"schemaPath":"./schemas/voteschema.json"
},
{"id":"POD_SCHEMA",
"schemaPath":"./schemas/podschema.json"
},
{"id":"BALLOT_AUDIT_COMMIT_SCHEMA",
"schemaPath":"./schemas/ballotauditcommitschema.json"
},
{"id":"BALLOT_GEN_COMMIT_SCHEMA",
"schemaPath":"./schemas/ballotgencommitschema.json"
},
{"id":"MIX_RANDOM_COMMIT_SCHEMA",
"schemaPath":"./schemas/mixrandomcommitschema.json"
},
{"id":"BALLOT_SUBMIT_RESPONSE_SCHEMA",
"schemaPath":"./schemas/ballotsubmitresponseschema.json"
},
{"id":"BALLOT_GEN_SCHEMA",
"schemaPath":"./schemas/ballotGenSchema.json"
},
{"id":"COMMITMENT_SCHEMA",
"schemaPath":"./schemas/commitmentSchema.json"
},
{"id":"JOINT_SIGNATURE_SCHEMA",
"schemaPath":"./schemas/jointsignatureSchema.json"
},
{"id":"VOTE_PACKING_SCHEMA",
"schemaPath":"./schemas/votePackingSchema.json"
}
]
```

Appendix G – OpenSSL Hash Commitment Verification

com_vvote_verifierlibrary_utils_crypto_CryptoUtils.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_vvote_verifierlibrary_utils_crypto_CryptoUtils */

#ifndef _Included_com_vvote_verifierlibrary_utils_crypto_CryptoUtils
#define _Included_com_vvote_verifierlibrary_utils_crypto_CryptoUtils
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     com_vvote_verifierlibrary_utils_crypto_CryptoUtils
 * Method:    openSSLVerifyHashCommitment
 * Signature: (Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL
Java_com_vvote_verifierlibrary_utils_crypto_CryptoUtils_openSSLVerifyHashCommitment
  (JNIEnv *, jclass, jstring, jstring, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

CryptoUtils.cpp

```
#include <jni.h>
#include <stdio.h>
#include "com_vvote_verifierlibrary_utils_crypto_CryptoUtils.h"
#include <cstring>
#include <iostream>
#include <iomanip>
#include <sstream>
#include <string>
#include <cstdlib>

using namespace std;

#include <openssl/sha.h>

/*
 * Converts from a java string (jstring) to an std string
*/
 void GetJStringContent(JNIEnv *AEnv, jstring AStr, std::string &ARes) {
  if (!AStr) {
    ARes.clear();
    return;
  }

  const char *s = AEnv->GetStringUTFChars(AStr,NULL);
  ARes=s;
  AEnv->ReleaseStringUTFChars(AStr,s);
}

void hexStringToASCIIString(const string &inputHexString, string &outputASCIIString) {
  int input_len = inputHexString.length();
  for(int i=0; i< input_len; i+=2)
  {
    string byte = inputHexString.substr(i,2);
    char chr = (char) (int)strtol(byte.c_str(), NULL, 16);
    outputASCIIString.push_back(chr);
  }
}

/*
 * Carries out a verification on the hash commitment made using the provided witness and
randomness value
```

```
*/
JNIEXPORT jboolean JNICALL
Java_com_vvote_verifierlibrary_utils_crypto_CryptoUtils_openSSLVerifyHashCommitment
  (JNIEnv *env, jclass jClass, jstring commitment, jstring witness, jstring random){

  // convert from jstring to std string
  // these values will contain the hexadecimal strings
  string commitValue;
  string witnessValue;
  string randomValue;
  GetJStringContent(env, commitment, commitValue);
  GetJStringContent(env, witness, witnessValue);
  GetJStringContent(env, random, randomValue);

  // convert from hex string to ascii string
  std::string witnessValueString;
  hexStringToASCIIString(witnessValue, witnessValueString);

  std::string randomValueString;
  hexStringToASCIIString(randomValue, randomValueString);

  // convert from std string to unsigned char *
  const unsigned char * wit = reinterpret_cast<const unsigned char *>
(witnessValueString.c_str());
  const unsigned char * rand = reinterpret_cast<const unsigned char *>
(randomValueString.c_str());

  char combined[(SHA256_DIGEST_LENGTH * 2) + 1];
  unsigned char md[SHA256_DIGEST_LENGTH];
  unsigned char modifiedRand[SHA256_DIGEST_LENGTH];
  const unsigned char * m_rand;

  // initialise
  SHA256_CTX context;
  if(!SHA256_Init(&context)){
    return false;
  }

  // make sure the random value length is less than or equal to 32 otherwise hash it first
  if(randomValueString.length() > 32){
       if(!SHA256_Update(&context, rand, randomValueString.length())){
              return false;
       }
       if(!SHA256_Final(modifiedRand, &context)){
              return false;
       }
       // reinitialise the context
       if(!SHA256_Init(&context)){
              printf("3");
              return false;
       }
       m_rand = modifiedRand;
  }else{
         m_rand = rand;
  }

  // first update with the witness
  if(!SHA256_Update(&context, wit, SHA256_DIGEST_LENGTH)){
    return false;
  }

  // second update with the random value
  if(!SHA256_Update(&context, m_rand, SHA256_DIGEST_LENGTH)){
    return false;
  }

  // perform the hash calculation
  if(!SHA256_Final(md, &context)){
    return false;
  }

  // convert from ascii string to hex string
  for(int i = 0; i < SHA256_DIGEST_LENGTH; i++)
  {
    sprintf(combined + (i * 2), "%02x", md[i]);
  }
  combined[SHA256_DIGEST_LENGTH * 2] = 0;
  jboolean result = false;

  // check that the combined hash value matches the commitment value
  if(combined == commitValue){
        result = true;
  }else{
        // check whether we need to convert to uppercase before failing
        char c;
        int i=0;
        while (combined[i]){
               c=combined[i];
               combined[i] = toupper(c);
               i++;
        }
        if(combined == commitValue){
               result = true;
        }
  }

  return result;
  }
```

Appendix H – vVote Verifiable Component Examples

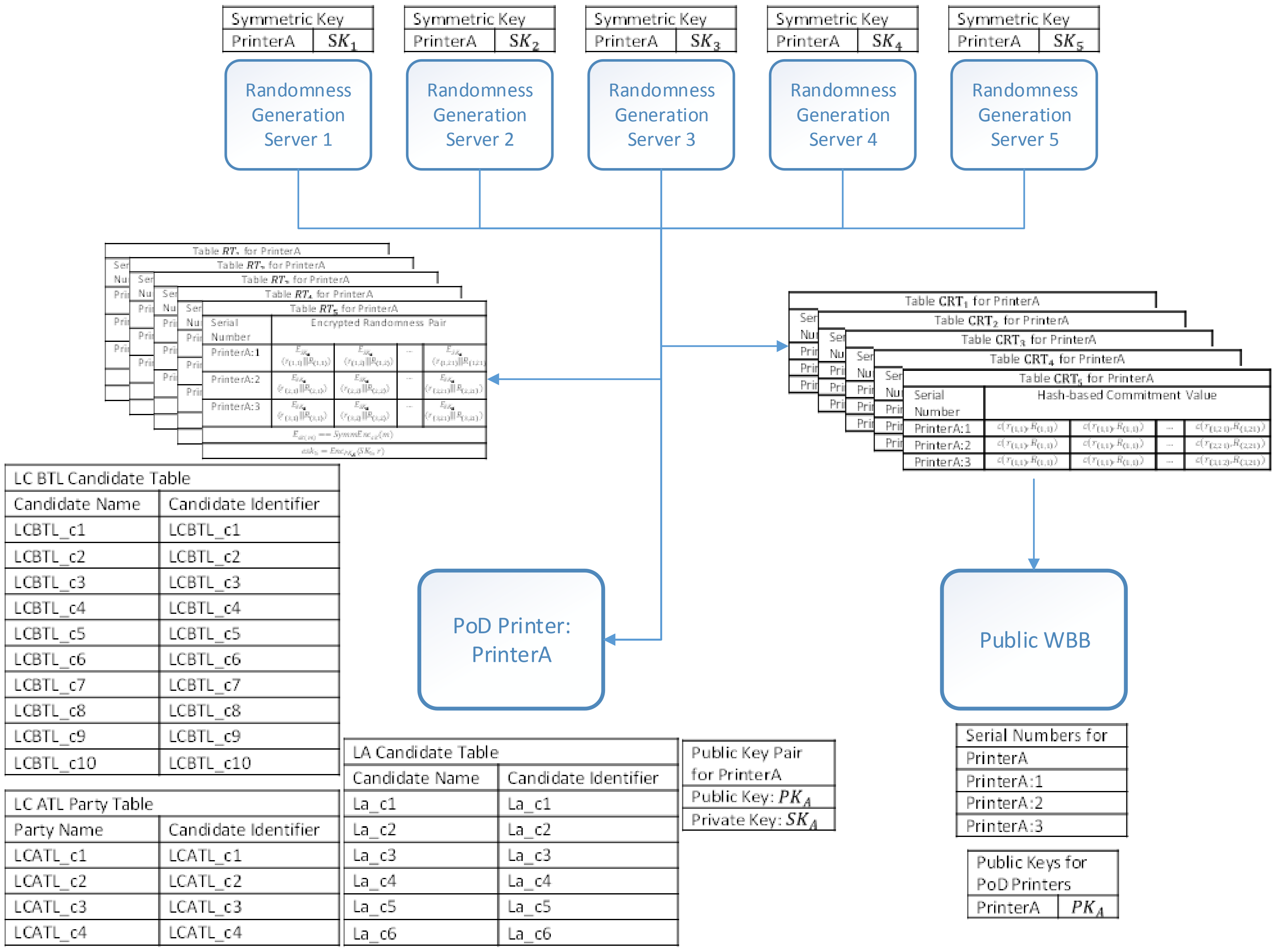
Ballot Generation Example

The following section provides an example of the Ballot Generation process for a single PoD Printer. Five randomness generation servers $RGen_i$, where i is from 1 to 5, will be used for sources of randomness. In the example three ballots will be generated; one of these ballots will be used for auditing leaving another two for voting.

Generic ballot configuration	
LA	6
LCATL	4
LCBTL	10

Table 0-1 - Generic ballot configuration

Table 0-1 shows a generic ballot configuration for an election whereby each of the generic ballots constructed during the Ballot Generation process by each PoD Printer will contain the required number of candidate ciphers for each race.



Randomness Generation Server 1
Randomness Generation Server 2
Randomness Generation Server 3
Randomness Generation Server 4
Randomness Generation Server 5
PoD Printer: PrinterA
Public WBB
Table RT_1 for PrinterA
Table RT_2 for PrinterA
Table RT_3 for PrinterA
Table RT_4 for PrinterA
Table $\mathbf{CRT}_1$ for PrinterA
Table $\mathbf{CRT}_2$ for PrinterA
Table $\mathbf{CRT}_3$ for PrinterA
Table $\mathbf{CRT}_4$ for PrinterA

Symmetric Key	
PrinterA	SK_1

Symmetric Key	
PrinterA	SK_2

Symmetric Key	
PrinterA	SK_3

Symmetric Key	
PrinterA	SK_4

Symmetric Key	
PrinterA	SK_5

Table RT_5 for PrinterA

Serial Number	Encrypted Randomness Pair			
PrinterA:1	$E_{SK_5}(r_{(1,1)}\|R_{(1,1)})$	$E_{SK_5}(r_{(1,2)}\|R_{(1,2)})$	...	$E_{SK_5}(r_{(1,21)}\|R_{(1,21)})$
PrinterA:2	$E_{SK_5}(r_{(2,1)}\|R_{(2,1)})$	$E_{SK_5}(r_{(2,2)}\|R_{(2,2)})$	...	$E_{SK_5}(r_{(2,21)}\|R_{(2,21)})$
PrinterA:3	$E_{SK_5}(r_{(3,1)}\|R_{(3,1)})$	$E_{SK_5}(r_{(3,2)}\|R_{(3,2)})$	...	$E_{SK_5}(r_{(3,21)}\|R_{(3,21)})$
$E_{sk(m)} = SymmEnc_{sk}(m)$				
$esk_5 = Enc_{PK_A}(SK_5, r)$				

Table $\mathbf{CRT}_5$ for PrinterA

Serial Number	Hash-based Commitment Value			
PrinterA:1	$c(r_{(1,1)}, R_{(1,1)})$	$c(r_{(1,1)}, R_{(1,1)})$	...	$c(r_{(1,21)}, R_{(1,21)})$
PrinterA:2	$c(r_{(1,1)}, R_{(1,1)})$	$c(r_{(1,1)}, R_{(1,1)})$	...	$c(r_{(2,21)}, R_{(2,21)})$
PrinterA:3	$c(r_{(1,1)}, R_{(1,1)})$	$c(r_{(1,1)}, R_{(1,1)})$	...	$c(r_{(3,12)}, R_{(3,21)})$

LC BTL Candidate Table	
Candidate Name	Candidate Identifier
LCBTL_c1	LCBTL_c1
LCBTL_c2	LCBTL_c2
LCBTL_c3	LCBTL_c3
LCBTL_c4	LCBTL_c4
LCBTL_c5	LCBTL_c5
LCBTL_c6	LCBTL_c6
LCBTL_c7	LCBTL_c7
LCBTL_c8	LCBTL_c8
LCBTL_c9	LCBTL_c9
LCBTL_c10	LCBTL_c10

LC ATL Party Table	
Party Name	Candidate Identifier
LCATL_c1	LCATL_c1
LCATL_c2	LCATL_c2
LCATL_c3	LCATL_c3
LCATL_c4	LCATL_c4

LA Candidate Table	
Candidate Name	Candidate Identifier
La_c1	La_c1
La_c2	La_c2
La_c3	La_c3
La_c4	La_c4
La_c5	La_c5
La_c6	La_c6

Public Key Pair for PrinterA
Public Key: PK_A
Private Key: SK_A

Serial Numbers for PrinterA
PrinterA:1
PrinterA:2
PrinterA:3

Public Keys for PoD Printers	
PrinterA	PK_A

Figure 0-1 - Example Ballot Generation process

Each randomness generation server $RGen_i$ constructs two tables, RT_i and $\mathrm{CRT_i}$. The pairs of randomness values in table RT_i are encrypted using a randomly generated symmetric key SK_i which $RGen_i$ also encrypts with the PoD Printer's public key PK_A and sends along with the table. Table RT_i is sent to PoD PrinterA and table $\mathrm{CRT_i}$ is publicly posted on the Public WBB. Upon receiving all RT_i tables the PoD Printer decrypts the pairs and verifies the commitments publicly posted on the Public WBB. The PoD Printer then combines together each corresponding randomness value to form a single random value per candidate identifier as shown in Table 0-2. In the example in Figure 0-1 each RT_i table contains 8 randomness pairs. 7 of the random values are used for re-encrypting candidate identifiers leaving the final random value as a witness in a commitment against the generated candidate identifier permutation.

Serial number	**Combined randomness value**			
PrinterA:1	$combined_{(1,1)} = RT_1.r_{(1,1)} + RT_2.r_{(1,1)} + RT_3.r_{(1,1)} + RT_4.r_{(1,1)} + RT_5.r_{(1,1)}$	$combined_{(1,2)} = RT_1.r_{(1,2)} + RT_2.r_{(1,2)} + RT_3.r_{(1,2)} + RT_4.r_{(1,2)} + RT_5.r_{(1,2)}$	…	$combined_{(1,21)} = RT_1.r_{(1,21)} + RT_2.r_{(1,21)} + RT_3.r_{(1,21)} + RT_4.r_{(1,21)} + RT_5.r_{(1,21)}$
PrinterA:2	$combined_{(2,1)} = RT_1.r_{(2,1)} + RT_2.r_{(2,1)} + RT_3.r_{(2,1)} + RT_4.r_{(2,1)} + RT_5.r_{(2,1)}$	$combined_{(2,2)} = RT_1.r_{(2,2)} + RT_2.r_{(2,2)} + RT_3.r_{(2,2)} + RT_4.r_{(2,2)} + RT_5.r_{(2,2)}$	…	$combined_{(2,21)} = RT_1.r_{(2,21)} + RT_2.r_{(2,21)} + RT_3.r_{(2,21)} + RT_4.r_{(2,21)} + RT_5.r_{(2,21)}$
PrinterA:3	$combined_{(3,1)} = RT_1.r_{(3,1)} + RT_2.r_{(3,1)} + RT_3.r_{(3,1)} + RT_4.r_{(3,1)} + RT_5.r_{(3,1)}$	$combined_{(3,2)} = RT_1.r_{(3,2)} + RT_2.r_{(3,2)} + RT_3.r_{(3,2)} + RT_4.r_{(3,2)} + RT_5.r_{(3,2)}$	…	$combined_{(3,21)} = RT_1.r_{(3,21)} + RT_2.r_{(3,21)} + RT_3.r_{(3,21)} + RT_4.r_{(3,21)} + RT_5.r_{(3,21)}$

Table 0-2 - Combining randomness values

Each of the candidate identifiers found in the Candidate tables are encrypted under a fixed randomness value of 1 to produce an ElGamal Elliptic Curve Point as $\mathrm{Enc}_k(cand_i, 1)$ termed the base encrypted candidate identifier as shown in Table 0-3.

Race	**Candidate/Party Name**	**Candidate Identifier**	**Encryption**
LA	La_c1	La_c1	$\mathrm{LA\_baseCand_1} = \mathrm{Enc}_{PK_E}(\mathrm{LA\_c1,1})$
LA	La_c2	La_c2	$\mathrm{LA\_baseCand_2} = \mathrm{Enc}_{PK_E}(\mathrm{LA\_c2,1})$
LA	La_c3	La_c3	$\mathrm{LA\_baseCand_3} = \mathrm{Enc}_{PK_E}(\mathrm{LA\_c3,1})$
LA	La_c4	La_c4	$\mathrm{LA\_baseCand_4} = \mathrm{Enc}_{PK_E}(\mathrm{LA\_c4,1})$
LA	La_c5	La_c5	$\mathrm{LA\_baseCand_5} = \mathrm{Enc}_{PK_E}(\mathrm{LA\_c5,1})$
LA	La_c6	La_c6	$\mathrm{LA\_baseCand_6} = \mathrm{Enc}_{PK_E}(\mathrm{LA\_c6,1})$
LCATL	LCATL_c1	LCATL_c1	$\mathrm{LCATL\_baseCand_1} = \mathrm{Enc}_{PK_E}(\mathrm{LCATL\_c1,1})$

LCATL	LCATL_c2	LCATL_c2	$\mathrm{LCATL\_baseCand}_1 = \mathrm{Enc}_{PK_E}(\mathrm{LCATL\_c2,1})$
LCATL	LCATL_c3	LCATL_c3	$\mathrm{LCATL\_baseCand}_1 = \mathrm{Enc}_{PK_E}(\mathrm{LCATL\_c3,1})$
LCATL	LCATL_c4	LCATL_c4	$\mathrm{LCATL\_baseCand}_2 = \mathrm{Enc}_{PK_E}(\mathrm{LCATL\_c4,1})$
LCBTL	LCBTL_c1	LCBTL_c1	$\mathrm{LCBTL\_baseCand}_1 = \mathrm{Enc}_{PK_E}(\mathrm{LCBTL\_c1,1})$
LCBTL	LCBTL_c2	LCBTL_c2	$\mathrm{LCBTL\_baseCand}_2 = \mathrm{Enc}_{PK_E}(\mathrm{LCBTL\_c2,1})$
LCBTL	LCBTL_c3	LCBTL_c3	$\mathrm{LCBTL\_baseCand}_3 = \mathrm{Enc}_{PK_E}(\mathrm{LCBTL\_c3,1})$
LCBTL	LCBTL_c4	LCBTL_c4	$\mathrm{LCBTL\_baseCand}_4 = \mathrm{Enc}_{PK_E}(\mathrm{LCBTL\_c4,1})$
LCBTL	LCBTL_c5	LCBTL_c5	$\mathrm{LCBTL\_baseCand}_5 = \mathrm{Enc}_{PK_E}(\mathrm{LCBTL\_c5,1})$
LCBTL	LCBTL_c6	LCBTL_c6	$\mathrm{LCBTL\_baseCand}_6 = \mathrm{Enc}_{PK_E}(\mathrm{LCBTL\_c6,1})$
LCBTL	LCBTL_c7	LCBTL_c7	$\mathrm{LCBTL\_baseCand}_7 = \mathrm{Enc}_{PK_E}(\mathrm{LCBTL\_c7,1})$
LCBTL	LCBTL_c8	LCBTL_c8	$\mathrm{LCBTL\_baseCand}_8 = \mathrm{Enc}_{PK_E}(\mathrm{LCBTL\_c8,1})$
LCBTL	LCBTL_c9	LCBTL_c9	$\mathrm{LCBTL\_baseCand}_9 = \mathrm{Enc}_{PK_E}(\mathrm{LCBTL\_c9,1})$
LCBTL	LCBTL_c10	LCBTL_c10	$\mathrm{LCBTL\_baseCand}_{10} = \mathrm{Enc}_{PK_E}(\mathrm{LCBTL\_c10,1})$

Table 0-3 - Constructing base candidate identifiers

The combined randomness values are then used to re-encrypt the base candidate identifiers. After the re-encryption has taken place a race-wise sort is carried out meaning the re-encrypted base candidate identifiers for the LA race are sorted and then the sorted set of LCATL re-encrypted base candidate identifiers are added to the generic ballot and finally the sorted set of LCBTL re-encrypted base candidate identifiers are added. A permutation is then constructed representing the shuffling of the generic ballot which is then committed to using the final combined random value. This process is shown in Table 0-4, Table 0-5 and Table 0-6. The final combined randomness value will be used as the witness value in a final commitment on the permutation computed for each of the generic ballots shown in Table 0-5. The final step of Ballot Generation is for the generic ballots to be submitted to the Public WBB.

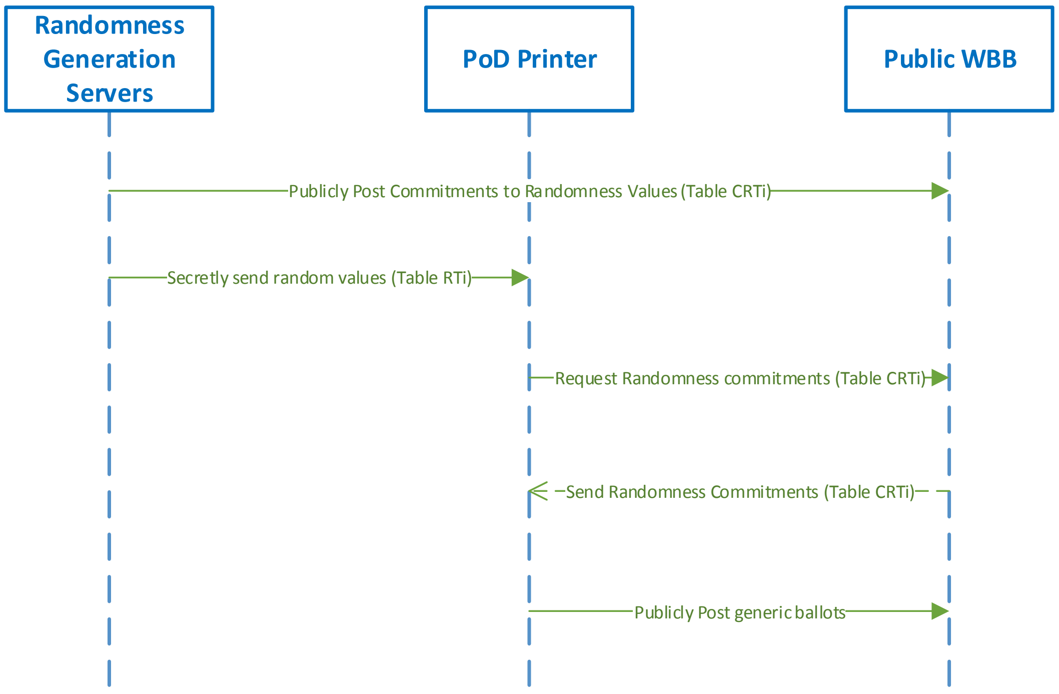

Randomness Generation Servers
PoD Printer
Public WBB
Publicly Post Commitments to Randomness Values (Table CRTi)
Secretly send random values (Table RTi)
Request Randomness commitments (Table CRTi)
Send Randomness Commitments (Table CRTi)
Publicly Post generic ballots

Figure 0-2 - Ballot Generation Sequence Diagram

Serial Number	**Re-encrypted base candidate identifier**			
PrinterA:1	$\text{cipher}_{(1,1)}(\text{56AB}) = \text{Enc}_{PK_E}(LA\_baseCand_1, combined_{(1,1)})$	$\text{cipher}_{(1,2)}(\text{37AC}) = \text{Enc}_{PK_E}(LA\_baseCand_2, combined_{(1,2)})$	…	$\text{cipher}_{(1,20)}(\text{12EE}) = \text{Enc}_{PK_E}(LCBTL\_baseCand_{10}, combined_{(1,20)})$
PrinterA:2	$\text{cipher}_{(2,1)}(\text{95DB}) = \text{Enc}_{PK_E}(LA\_baseCand_1, combined_{(2,1)})$	$\text{cipher}_{(2,2)}(\text{98CB}) = \text{Enc}_{PK_E}(LA\_baseCand_2, combined_{(2,2)})$	…	$\text{cipher}_{(2,20)}(\text{11AF}) = \text{Enc}_{PK_E}(LCBTL\_baseCand_{10}, combined_{(2,20)})$
PrinterA:3	$\text{cipher}_{(3,1)}(\text{76EA}) = \text{Enc}_{PK_E}(LA\_baseCand_1, combined_{(3,1)})$	$\text{cipher}_{(3,2)}(\text{66EB}) = \text{Enc}_{PK_E}(LA\_baseCand_2, combined_{(3,2)})$	…	$\text{cipher}_{(3,20)}(\text{99FB}) = \text{Enc}_{PK_E}(LCBTL\_baseCand_{10}, combined_{(3,20)})$

Table 0-4 - Example Re-encrypted base candidate identifiers

Candidate Identifier	**Re-encrypted ciphers**		
	Serial Numbers		
	PrinterA:1	**PrinterA:2**	**PrinterA:3**
LA_baseCand1	56AB	95DB	11AC
LA_baseCand2	37AC	98CB	23FC
LA_baseCand3	12BC	45FB	23CD
LA_baseCand4	48AF	32AD	65AB
LA_baseCand5	11AD	12ED	21DC
LA_baseCand6	12FC	46FD	65DE
LCATL_baseCand1	36CE	36CC	01FB
LCATL_baseCand2	55AA	87BA	29DD
LCATL_baseCand3	46FD	76FD	87AD
LCATL_baseCand4	34FB	48CD	50DE
LCBTL_baseCand1	32FC	47AB	22FB
LCBTL_baseCand2	56EE	33FC	76DA
LCBTL_baseCand3	12EE	11AF	33AD
LCBTL_baseCand4	22BB	00DC	01DF
LCBTL_baseCand5	23FD	81DC	30AB
LCBTL_baseCand6	12BC	09CD	46CD
LCBTL_baseCand7	98FD	45FB	49DA
LCBTL_baseCand8	87DE	40DF	10FD
LCBTL_baseCand9	12AF	33DD	77FD
LCBTL_baseCand10	91FD	55DD	65DC

Table 0-5 - Example re-encrypted ciphertexts

	Race-wise sorted re-encrypted ciphers. Here the sorting is simply an alphanumerical sort whereas with the ElGamal Elliptic Curve points comparisons would be made as to the g^r and $m * y^r$ points.				
	Serial Numbers				
Candidate Identifier (PrinterA:1)	**PrinterA:1**	**Candidate Identifier (PrinterA:2)**	**PrinterA: 2**	**Candidate Identifier (PrinterA:3)**	**PrinterA: 3**
LA_ baseCand5	11AD	LA_ baseCand5	12ED	LA_ baseCand1	11AC
LA_ baseCand3	12BC	LA_ baseCand4	32AD	LA_ baseCand5	21DC
LA_ baseCand6	12FC	LA_ baseCand3	45FB	LA_ baseCand3	23CD
LA_ baseCand2	37AC	LA_ baseCand6	46FD	LA_ baseCand2	23FC

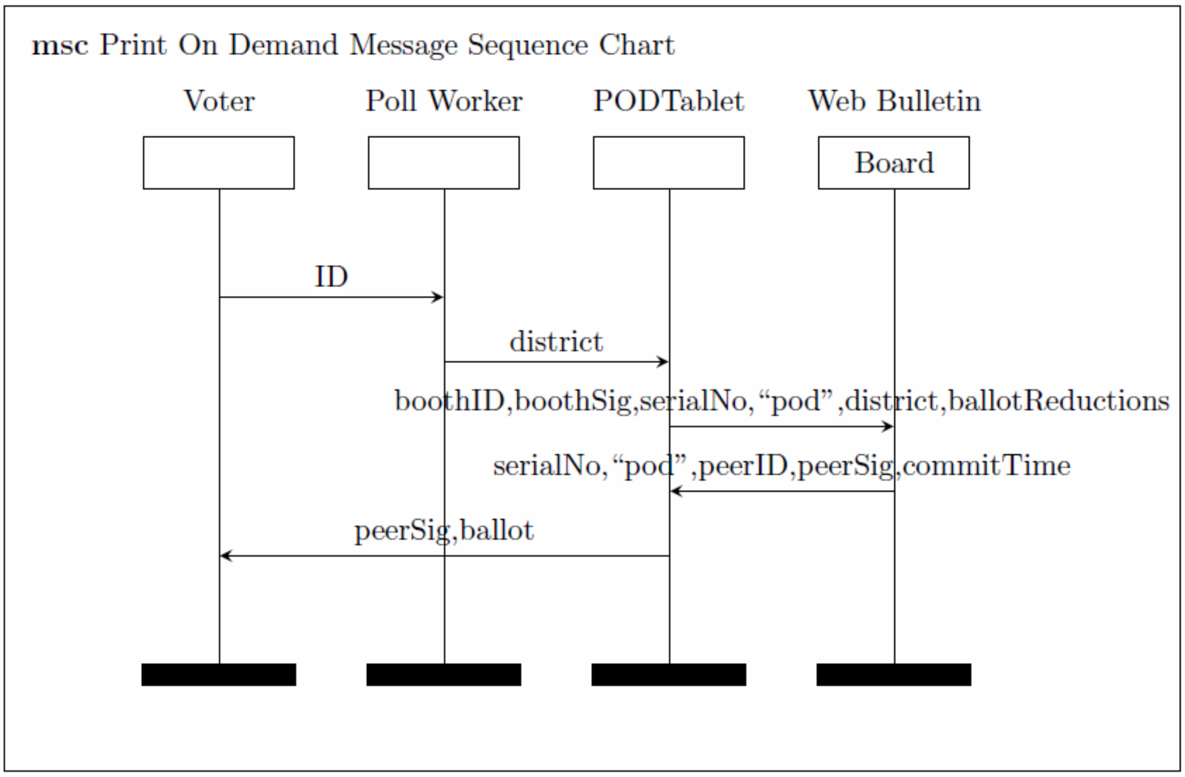
LA_ baseCand4	48AF	LA_ baseCand1	95DB	LA_ baseCand4	65AB
LA_ baseCand1	56AB	LA_ baseCand2	98CB	LA_ baseCand6	65DE
LCATL_ baseCand4	34FB	LCATL_ baseCand1	36CC	LCATL_ baseCand1	01FB
LCATL_ baseCand1	36CE	LCATL_ baseCand4	48CD	LCATL_ baseCand2	29DD
LCATL_ baseCand3	46FD	LCATL_ baseCand3	76FD	LCATL_ baseCand4	50DE
LCATL_ baseCand2	55AA	LCATL_ baseCand2	87BA	LCATL_ baseCand3	87AD
LCBTL_ baseCand9	12AF	LCBTL_ baseCand4	00DC	LCBTL_ baseCand4	01DF
LCBTL_ baseCand6	12BC	LCBTL_ baseCand6	09CD	LCBTL_ baseCand8	10FD
LCBTL_ baseCand3	12EE	LCBTL_ baseCand3	11AF	LCBTL_ baseCand1	22FB
LCBTL_ baseCand4	22BB	LCBTL_ baseCand9	33DD	LCBTL_ baseCand5	30AB
LCBTL_ baseCand5	23FD	LCBTL_ baseCand2	33FC	LCBTL_ baseCand3	33AD
LCBTL_ baseCand1	32FC	LCBTL_ baseCand8	40DF	LCBTL_ baseCand6	46CD
LCBTL_ baseCand2	56EE	LCBTL_ baseCand7	45FB	LCBTL_ baseCand7	49DA
LCBTL_ baseCand8	87DE	LCBTL_ baseCand1	47AB	LCBTL_ baseCand10	65DC
LCBTL_ baseCand10	91FD	LCBTL_ baseCand10	55DD	LCBTL_ baseCand2	76DA
LCBTL_ baseCand7	98FD	LCBTL_ baseCand5	81DC	LCBTL_ baseCand9	77FD
Permutation		**Permutation**		**Permutation**	
5,3,6,2,4,1:4,1,3,2:9,6,3,4,5,1,2,8,10,7		5,4,3,6,1,2:1,4,3,2:4,6,3,9,2,8,7,1,10,5		1,5,3,2,4,6:1,2,4,3:4,8,1,5,3,6,7,10,2,9	

Table 0-6 - Example sorting of ciphertexts to create generic ballots

Print on Demand Example

This section provides a brief example of the Print on Demand process utilising message sequence charts to aid the explanation.

Step 1 to 5 in the protocol described in 3.5.2.2.2 Print on Demand Protocol describe the protocol from the point where the voter arrives at the polling station to the point they are provided with a reduced ballot using which they can cast their vote.


msc Print On Demand Message Sequence Chart
Voter
Poll Worker
PODTablet
Web Bulletin
Board
ID
district
boothID,boothSig,serialNo,“pod”,district,ballotReductions
serialNo,“pod”,peerID,peerSig,commitTime
peerSig,ballot

- boothSig : $\text{Sign}_{PODTablet}\{\text{serialNo,district}\}$
- peerSig : $\text{Sign}_{WBB}\{\text{serialNo,district}\}$ (the returned peerSig forms the serialSig in future sequence charts)

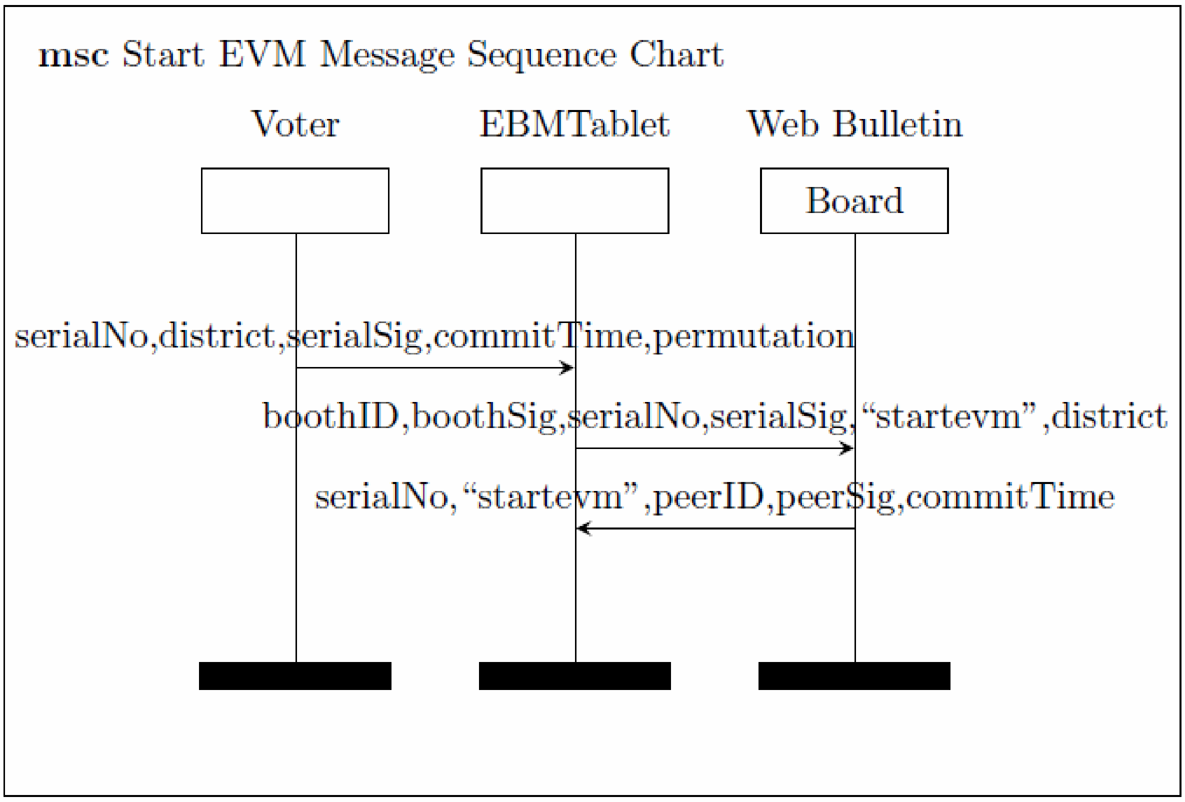
Figure 0-3 - Print on Demand Message Sequence Chart

Utilising the existing example from the ballot reduction process shown in 3.5.2.2.1 Ballot Reduction Overview the messages within Figure 0-3 can be further expanded upon.

Message	**Example**
district	Northcote
boothID, boothSig, serialNo, ”pod”, district, ballotReductions	PrinterA, $Sign_{PODTablet}\{Ballot\colon 01, Northcote\}$, Ballot:01, ”pod”, Northcote, [[{0,5,r_{F1}},{4,4,r_{E1}},{5,3,r_{D1}}] [{1,2r_{I1}},{3,3,r_{J1}}] [{5,6,r_{Q1}},{9,7,r_{R1}},{0,8,r_{S1}},{1,9,r_{T1}}]]
serialNo, ”pod”, peerID, peerSig, commitTime	Ballot:01, ”pod”, peer1, $Sign_{WBB}\{Ballot\colon 01, Northcote\}$, 101010101
peerSig,ballot	$Sign_{WBB}\{Ballot\colon 01, Northcote\}$, (Reduced ballot shown in Table 3-13)

Table 0-7 - Print on Demand messages expanded

Steps 6 to 12 in the protocol in 3.5.2.2.2 Print on Demand Protocol then describe the way in which a voter submits their ballot preferences and how they are then provided with an accompanying receipt which can be verified as in step 13.

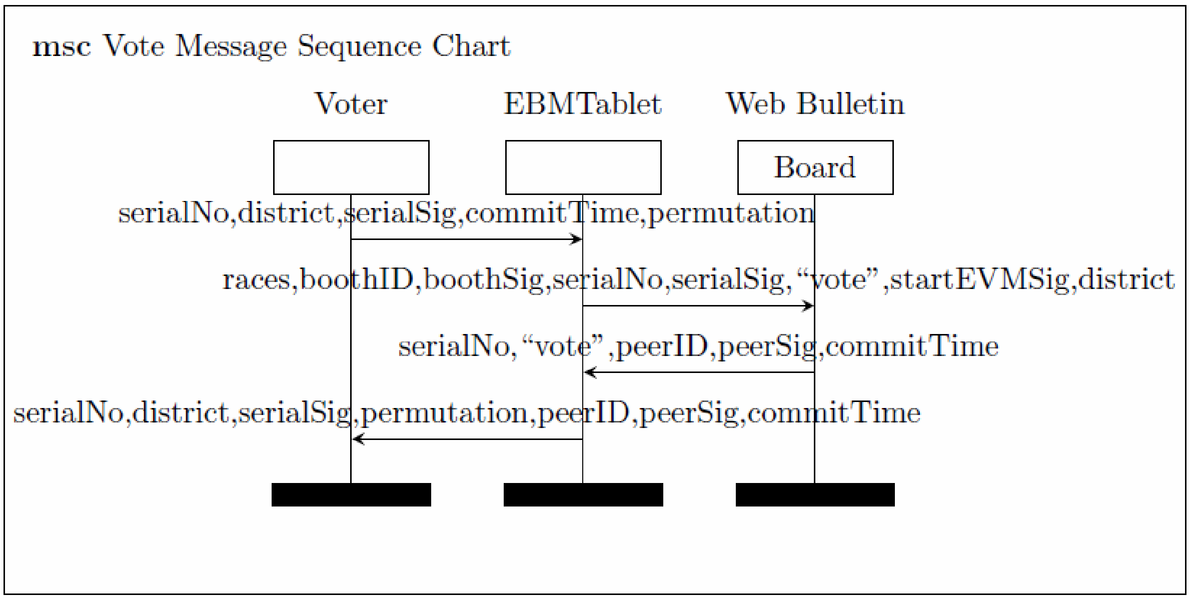

msc Start EVM Message Sequence Chart
Voter
EBMTablet
Web Bulletin
Board
serialNo,district,serialSig,commitTime,permutation
boothID,boothSig,serialNo,serialSig,“startevm”,district
serialNo,“startevm”,peerID,peerSig,commitTime

- boothSig : $\text{Sign}_{EBMTablet}\{$“startevm”,serialNo,district$\}$
- serialSig : $\text{Sign}_{WBB}\{$serialNo,district$\}$
- peerSig : $\text{Sign}_{WBB}\{$“startevm”,serialNo,district$\}$

Figure 0-4 - Start EVM Message Sequence Chart

Message	**Example**
serialNo, district, serialSig, commitTime, permutation	Ballot:01, Northcote, $Sign_{WBB}\{Ballot: 01, Northcote\}$, 101010101, π
boothID, boothSig, serialNo, serialSig, “startevm”, district	PrinterA, $Sign_{PODTablet}\{"startevm", Ballot: 01, Northcote\}$, Ballot:01, $Sign_{WBB}\{Ballot: 01, Northcote\}$, “startevm”, Northcote
serialNo, “startevm”, peerID, peerSig, commitTime	PrinterA, “startevm”, peer1, $Sign_{WBB}\{"startevm", Ballot: 01, Northcote\}$, 101010101

Table 0-8 - StartEVM messages expanded


msc Vote Message Sequence Chart
Voter
EBMTablet
Web Bulletin
Board
serialNo,district,serialSig,commitTime,permutation
races,boothID,boothSig,serialNo,serialSig,“vote”,startEVMSig,district
serialNo,“vote”,peerID,peerSig,commitTime
serialNo,district,serialSig,permutation,peerID,peerSig,commitTime

- boothSig : $\text{Sign}_{EBMTablet}\{\text{serialNo,district,preferences}\}$
- serialSig : $\text{Sign}_{WBB}\{\text{serialNo,district}\}$
- startEVMSig : $\text{Sign}_{WBB}\{\text{“startevm”,serialNo,district}\}$
- peerSig : $\text{Sign}_{WBB}\{\text{serialNo,district,preferences,commitTime}\}$

Figure 0-5 - Vote Message Sequence Chart

Message	Example
serialNo, district, serialSig, commitTime, permutation	Ballot:01, Northcote, $Sign_{WBB}\{Ballot:01, Northcote\}$, 101010101, π
races, boothID, boothSig, serialNo, serialSig, “vote”, startEVMSig, district	[{"id":"LA","preferences":["1"," "," "]}, {"id":"LC_ATL","preferences":["1"," "]}, {"id":"LC_BTL","preferences":[" "," "," "," "," "," "]}], PrinterA, $Sign_{PODTablet}\{"startevm", Ballot:01, Northcote\}$, Ballot:01, $Sign_{WBB}\{Ballot:01, Northcote\}$, “vote”, $Sign_{WBB}\{"startevm",\ Ballot:01, Northcote\}$, Northcote
serialNo, “vote”, peerID, peerSig, commitTime	Ballot:01, “vote”, peer1, $Sign_{WBB}\{Ballot:01, Northcote, 1,,:1,:,,,,,,101010101\}$, 101010101
serialNo, district, serialSig, permutation, peerID, peerSig, commitTime	Ballot:01, Northcote, $Sign_{WBB}\{Ballot:01, Northcote\}$, π, peer1, $Sign_{WBB}\{Ballot:01, Northcote, 1,,:1,:,,,,,,101010101\}$, 101010101

Table 0-9 - Vote messages expanded

Appendix I – Verifiable Component Pseudo code

Ballot Generation Pseudo code

```
for $i$ = 1 -> $p$ do
        Verify the Fiat-Shamir Signature for PoD Printer $i$ by recalculating it from public
        information – Step 1.
```

```
          Verify the serial numbers chosen for auditing for PoD Printer $i$ match using the newly
          calculated Fiat-Shamir Signature - Step 1.

          Get ballots for auditing and store in audit[$a$].

          for $j$ = 1 -> $a$ do
                  for $g$ = 1 -> $G$ do
                          for $h$ = 1 -> $n+1$ do
                                  Verify commitment from table $CRT_g$ using the randomness value $r_h$
                                  and witness value $R_h$ from table $RT_g$.

                  Create storage for $n+1$ combined randomness values in s[$n+1$].
                  for $g$ = 1 -> $G$ do
                          for $h$ = 1 -> $n+1$ do
                                  Update the current s[$h$] with the corresponding randomness values
                                  $r_h$ from $RT_g$ table.

                  s now contains $n+1$ combined randomness values produced through the combination
                  of $G$ random values from the different randomness generation servers.

                  Create storage for $n$ re-encrypted candidate identifiers reencryptions[$n$].
                  for $h$ = 1 -> $n$ do
                          Re-encrypt base candidate identifier at the index $h$ with the combined
                          randomness value s[$h$] and store in reencryptions[$h$].

                  Create storage for $n$ sorted, in a per race basis, re-encrypted candidate
                  identifiers sorted[$n$].

                  la_size = LA Race size
                  $k$ = la_size
                  Sort the re-encrypted candidate identifiers in reencryptions[1 -> $k$] and assign
                  to sorted[1 -> $k$].

                  lc_atl = LC ATL Race size
                  $k$ = lc_atl
                  Sort the re-encrypted candidate identifiers in reencryptions[la_size -> $k$] and
                  assign to sorted[la_size -> $k$].

                  lc_btl = LC BTL Race size
                  $k$ = lc_btl
                  Sort the re-encrypted candidate identifiers in reencryptions[la_size + lc_atl -
                  > $k$] and assign to sorted[la_size + lc_atl -> $k$].

                  Reconstruct the current ballot permutation using the original index of the re-
                  encrypted candidate identifiers as recalculated_permutation.

                  for $h$ = 1 -> $n$ do
                          Compare sorted[$h$] with audit[$j$][$h$]

                  Perform a commitment check on the permutation for audit[$j$] using the final
                  combined randomness value s[$n+1$] as the witness value. We verify that the
                  permutation for audit[$j$] is equal to recalculated_permutation and s[$n+1$]
                  concatenated and hashed together.
```
Code Extract 0-1 - Ballot Generation Verification pseudo code

Voting Packing Pseudo code

```
Store generic ballots corresponding to valid votes in generic_ballots[$s$]
Store voter preferences in votes[$s$].
Store ballot reductions for generic ballots used to submit preferences in reductions[$s$].
Create storage for reduced ballots, in a per race basis, for submitted ballots in
reduced_ballots_la[$s$], reduced_ballots_lcatl[$s$] and reduced_ballots_lcbtl[$s$].
Create storage for reordered reduced ballots in the LA race as reordered_la[$s$].
Create storage for reordered reduced ballots in the LC ATL race as reordered_lcatl[$s$].
Create storage for reordered reduced ballots in the LC BTL race as reordered_lcbtl[$s$].
Create storage for packed ballots in the LA race as packed_la[$s$].
Create storage for packed ballots in the LC ATL race as packed_lcatl[$s$].
Create storage for packed ballots in the LC BTL race as packed_lcbtl[$s$].

for $i$ = 1 -> $s$ do
        Get the preferences for current vote from votes[$i$]
        Lookup the corresponding generic ballot used which shares the same serial number as
        that of the vote message from generic_ballots[$i$].

        Apply and verify reductions[$i$] on generic_ballots[$i$] and store in
        reduced_ballots_la[$i$], reduced_ballots_lcatl[$i$] or reduced_ballots_lcbtl[$i$] as
        appropriate.

        Reorder reduced_ballots_la[$i$] using the preferences in votes[$i$] for the LA race not
        including ciphertexts with no preference provided and store in reordered_la[$i$].
        Reorder reduced_ballots_lcatl[$i$] using the preferences in votes[$i$] for the LC ATL race
        if an ATL vote not including ciphertexts with no preference provided and store in
        reordered_lcatl[$i$].
        Reorder reduced_ballots_lcbtl[$i$] using the preferences in votes[$i$] for the LC BTL race
        if a BTL vote not including ciphertexts with no preference provided and store in
        reordered_lcbtl[$i$].

        Pack the reordered reduced ballot in reordered_la[$i$] and store in packed_la[$i$].
        Pack the reordered reduced ballot in reordered_lcatl[$i$] and store in packed_lcatl[$i$].
        Pack the reordered reduced ballot in reordered_lcbtl[$i$] and store in packed_lcbtl[$i$].

for $i$ = 1 -> $s$ do
        Calculate the appropriate padding size for each packing value in each race.

for $i$ = 1 -> $s$ do
        Apply the appropriate padding to packed_la[$i$].
        Apply the appropriate padding to packed_lcatl[$i$].
        Apply the appropriate padding to packed_lcbtl[$i$].

for $i$ = 1 -> $s$ do
        Verify the padded packed ciphertexts in packed_la[$i$], packed_lcatl[$i$] and
        packed_lcbtl[$i$] match the values provided as input to the Mixnet in the corresponding
        btl file for the district and race.

number_of_csv equals the number of csv files.
Store csv files in csvs[number_of_csv]

for $j$ = 1 -> number_of_csv do
        for $k$ = 1 -> csvs[$j$] lines do
                Get the corresponding plaintext identifiers for csvs[$j$] which will vary between
                race type and district.
                Using the plaintext identifiers and the values in csvs[$j$][$k$] reorder the
                plaintext identifiers into preference values.

                Pack each of the sets of reordered plaintext identifiers using the preference
                value 1 -> race packing size.

                Verify that the packed value exists in the output from the Mixnet in the
                corresponding OUT file to verify the claimed preferences were correctly looked
                up from the corresponding look up tables.
```
Code Extract 0-2 - Vote Packing pseudo code

Public WBB Commits Pseudo code

```
Group together related commit objects so that each commitment $c$ contains a valid JSON Message
file, attachment ZIP file and a Signature JSON file.
for each commitment $c$ do
        message_file = commitment.getJSONMessageFile()
        SHA1 $h$
        for each message in message_file do
                Update $h$ with message.internalsignablecontent fields
                if message is file Message do
                        update $h$ with the attachment file relating to the current file message.

        SHA1 $sig$
        Update $sig$ with "Commit"
        Update $sig$ with commitment.getCommitTime()
        Update $sig$ with hash $h$

        Get the joint signature from the JSON signature file as $joint\_signature$.
        Verify that *sig* is equal to is equal to that data that was signed by *joint_signature* using
        the Public WBB public key.
```
Code Extract 0-3- Public WBB commit code extract