Software Design for VEC vVote System

Chris Culnane, James Heather, Steve Schneider and Zhe Xia

March 26, 2013
SOFTWARE DESIGN

for

VEC vVote System

Version 1.0

Sunday 13th January, 2013

Prepared by Chris Culnane
## Contents

1. **Introduction**  
   1.1. Purpose ................................................. 7  
   1.2. Scope .................................................. 7  
   1.3. Context ............................................... 8  
   1.4. Summary ............................................... 8  
   1.5. Project Contributors ................................. 9  

2. **References** ............................................. 10

3. **Glossary** ................................................ 12

4. **Overall Description** .................................... 15  
   4.1. Design Concerns ....................................... 15  
      4.1.1. Overview of Design ................................ 15  
      4.1.2. Pre-election ...................................... 16  
      4.1.3. Vote Casting ...................................... 17  
      4.1.4. Post Election ..................................... 18  
   4.2. Table Building ........................................ 19  
      4.2.1. Summary .......................................... 19  
      4.2.2. Candidate Identifiers ............................ 19  
      4.2.3. Base Value *baseValue* ........................... 20  
      4.2.4. Permutation Generation ........................... 20  
   4.3. Distributed Ballot Generation ......................... 21  
      4.3.1. Summary .......................................... 21  
      4.3.2. What is a ballot? ................................ 21  
      4.3.3. Interface to VEC Election Management System .... 22  
      4.3.4. Initial Ciphertext Generation .................... 24  
      4.3.5. Ballot Counter .................................... 25  
      4.3.6. Mixing ............................................ 26  
      4.3.7. Key Transformation ............................... 26  
      4.3.8. Batch Verification ................................ 28  
      4.3.9. POD Servers ....................................... 28  
   4.4. Print on Demand Service ............................... 29  
      4.4.1. Print on Demand Protocol ........................ 29  
      4.4.2. Entities .......................................... 29  
      4.4.3. Description of POD Protocol ....................... 31  
      4.4.4. Data Format ...................................... 32
4.4.5. Threshold Decryption and Key Transformation.
4.4.6. Auditing.

4.5. Mixnet Manager
4.5.1. Mixnet Setup
4.5.2. Download vote data from the WBB
4.5.3. Check vote data signature
4.5.4. Format vote data into Mixnet compatible format
4.5.5. Run Various Mixes and Decryptions
4.5.6. Vote Packing Look-up
4.5.7. Upload data to the WBB

4.6. Key Management
4.6.1. Certificate Signing Authority (CSA)
4.6.2. Certificate Signing Requests

4.7. Web Bulletin Board (WBB)
4.7.1. General Properties
4.7.2. Prerequisites
4.7.3. Vote Submission Protocol
4.7.4. Notation
4.7.5. Receiving Votes
4.7.6. End-of-Day Commit
4.7.7. Cancellations
4.7.8. Audit
4.7.9. Submission of Bulk Data

4.8. Public WBB
4.8.1. Nightly Updates
4.8.2. Mirroring

4.9. Testing, Error Management and Notification
4.9.1. Testing
4.9.2. Error Management and Notification

A. Mixnet Specification
A.1. Mixnet Specification
A.1.1. Functions
A.1.2. Interfaces
# Revision History

<table>
<thead>
<tr>
<th>Name</th>
<th>Status</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chris Culnane</td>
<td>Initial drafting of document structure</td>
<td>30th October 2012</td>
</tr>
<tr>
<td>Chris Culnane</td>
<td>Overview of system</td>
<td>2nd November 2012</td>
</tr>
<tr>
<td>Chris Culnane</td>
<td>Added Table Building Section</td>
<td>9th November 2012</td>
</tr>
<tr>
<td>Chris Culnane</td>
<td>Added Distributed Ballot Generation</td>
<td>23rd November 2012</td>
</tr>
<tr>
<td>Chris Culnane</td>
<td>Added Print on Demand Section</td>
<td>7th December 2012</td>
</tr>
<tr>
<td>Chris Culnane</td>
<td>Added Mixnet Manager Section</td>
<td>12th December 2012</td>
</tr>
<tr>
<td>Chris Culnane</td>
<td>Added Key Management Section</td>
<td>14th December 2012</td>
</tr>
<tr>
<td>Chris Culnane</td>
<td>Started WBB Description</td>
<td>17th December 2012</td>
</tr>
<tr>
<td>Steve Schneider</td>
<td>Edited Overall Document, minor additions</td>
<td>18th December 2012</td>
</tr>
<tr>
<td>Chris Culnane</td>
<td>Updated images and POD Protocol</td>
<td>18th December 2012</td>
</tr>
<tr>
<td>Chris Culnane</td>
<td>Added Mixnet Description</td>
<td>19th December 2012</td>
</tr>
<tr>
<td>Zhe Xia</td>
<td>Minor editing in Section 4.3</td>
<td>19th December 2012</td>
</tr>
<tr>
<td>Zhe Xia</td>
<td>Added Batch Proofs; Started Glossary</td>
<td>20th December 2012</td>
</tr>
<tr>
<td>Chris Culnane</td>
<td>Finished WBB Description</td>
<td>20th December 2012</td>
</tr>
<tr>
<td>James Heather</td>
<td>Added context and summary; minor edits throughout</td>
<td>21st December 2012</td>
</tr>
<tr>
<td>Steve Schneider</td>
<td>Edited overall document, further amendments and additions</td>
<td>21st December 2012</td>
</tr>
<tr>
<td>James Heather</td>
<td>Added \annot; additions to deal with two of CB’s comments; minor edits throughout</td>
<td>7th January 2013</td>
</tr>
<tr>
<td>Zhe Xia</td>
<td>Make some changes in Section 4.2 and 4.5</td>
<td>11th January 2013</td>
</tr>
<tr>
<td>James Heather</td>
<td>Edited overall document (mainly second half)</td>
<td>11th January 2013</td>
</tr>
<tr>
<td>Steve Schneider</td>
<td>Further amendments and additions; incorporated content from VT into 4.7.6 and addressed comments from PYAR</td>
<td>12th January 2013</td>
</tr>
</tbody>
</table>
1. Introduction

1.1. Purpose

The purpose of this Software Design Document (SDD) is to provide the design for the elements that need to be implemented to create the back end of the vVote election system for the Victorian Electoral Commission (VEC). The vVote election system is an end-to-end voter-verifiable election system based on the Prêt à Voter system. This document, in addition to providing the design for those elements requiring implementation, will also give some brief background into the properties that the vVote system aims to provide and how they relate to the end-to-end voter-verifiable properties defined in current academic literature.

Since electoral procedures differ considerably from one jurisdiction to another, this design document relates specifically to VEC elections, and would need appropriate adaptations for other environments. However, the overall structure of the system is entirely suitable for other elections, and particularly Australian elections, and the vVote election system has been designed in such a way that it could be enhanced to cover other jurisdictions without extensive replumbing. The enhancements would primarily deal with procedural variations (for instance, elections where voters can specify a ranking of parties) and modifications to provide better scalability for jurisdictions with different typical voting behaviour or larger numbers of candidates.

1.2. Scope

The SDD describes, in detail, those components that require implementation to complete the vVote system. There are additional external components that either have already been implemented or are currently being implemented. This document will not provide details of those designs but will refer to them and outline their properties and functionality. Where there is a direct interface between the components designed in this document and an external component the interface will be detailed within this SDD.

The overall scope of the components in this SDD concerns the back-end infrastructure for the vVote system. The front-end components are being designed and implemented separately; however, we will specify the format and nature of the interfaces between the front end and the back end in this SDD.
1.3. Context

The guiding principle behind the design is that the election system should be publicly verifiable: individual voters should be able to check that their votes have not been tampered with, and anyone should be able to verify the final election tally. Every step of the process should be verifiable, and should not require trust in any individual person or machine.

The system will provide mathematical proofs of all its sensitive operations. Verification of these proofs is something that members of the public should be able to perform using their own machines and, if desired, their own software. This means that voters will not need to trust electoral officials, software developers or hardware components to act correctly, because the integrity of the election will be verifiable at a mathematical level.

Part of this commitment to an open approach involves releasing the full source code. The code will be released under the GNU Public Licence (GPL) v3; this licence allows anyone to download, compile, reverse engineer or modify the system. It has important consequences for the development phase: no software libraries may be compiled into the system that are not also released under an open source licence that is compatible with the GPL v3. It also means that no patent-encumbered code may be included.

1.4. Summary

This document describes the overall design philosophy of the system, the components that need to be implemented to produce a complete system, and the interfaces between the various components.

Section 4.1 gives an overview, and discusses the principles behind the design structure, and the context in which the system will be deployed. A full system design then follows. In Section 4.2, we describe the tables that need to be precomputed for efficient operation of the system, and the algorithms that can be used to generate the tables. Section 4.3 then introduces the problem of distributed ballot generation, and the design detail for the programs that implement this part of the system. Next, in Section 4.4, we discuss the mechanism for on-demand printing of paper ballots (including ballots for those who are voting out of their home district). The Mixnet manager, which oversees the operation of the Mixnet, is then presented in Section 4.5. Issues around key management are dealt with in Section 4.6. Sections 4.7 and 4.8 then describe the distributed web bulletin board that provides a robust and fault-tolerant view of all public data generated during the whole election process. Finally, in Appendix A.1, the requirements on the external Mixnet component are described: a distributed component responsible for ensuring that all votes are provably correctly shuffled and that no individual machine (including the Mixnet manager) can discover anything about the permutation.
1.5. Project Contributors

The following people have participated in discussions and contributed to the design described in this document: Craig Burton, Chris Culnane, James Heather, Thea Peacock, Peter Y. A. Ryan, Steve Schneider, Sriram Srinivasan, Vanessa Teague, Roland Wen, Douglas Wikström and Zhe Xia. Earlier stages of the design are described in [BCH+12a, BCH+12b].
2. References


3. Glossary

**Ballot** In general, a ballot is used by the voter to cast her vote. But in this document, a ballot will pass through several stages in which it has different forms. A paper ballot or printed ballot is a physical ballot in which the candidate ordering is human readable. A digital ballot is stored digitally in an encrypted form, and only a party who possesses the correct secret key can read its information. We classify these two types of ballot when necessary. Otherwise, if we simply mention a ballot without classifying, the above difference does not matter and it can be either a printed ballot or a digital ballot.

**Ballot counter** A unique value assigned to each ballot. It records which ciphertexts are contained by a particular ballot. In the distributed ballot generation phase, this allows the ciphertexts of all ballots to be shuffled together rather than using a separate Mixnet for each ballot.

**Ballot manager** A single server architecture whose sole role is to apportion serial numbers.

**Candidate identifier** A publicly known numerical value assigned to each candidate. If we want to generate an encryption for a candidate, we encrypt this value rather than the candidate name.

**CSV: Comma-Separated Values** A simple text format in which the stored information is separated by commas.

**Distributed ballot generation** Ballots are generated by a number of independent parties in a distributed fashion, so that no single party learns the ballot information, and no single party controls the randomisation values used in the ballots.

**Early voting** The voting phase that takes place before Election Day. It lasts for around 2 weeks.

**EBM: Electronic Ballot Marker** A device to aid the voter in marking her vote choices.

**End-to-end voter-verifiable election system** A system in which all stages of the election can be publicly verified: individual voters are able to check that their votes have not been tampered with, and anyone is able to verify the final election tally.

**LA, LC, ATL, BTL** Legislative Assembly, Legislative Council, Above-the-Line and Below-the-Line respectively. These are aspects of Victorian voting: LA and LC are the two houses that are elected, and ATL and BTL are two ways of casting a vote for the LC.
Key management  To manage the signing and encryption keys generated by various
different components.

Multiparty computation  A computation performed by a number of mutually distrusting
parties in such a way that each party can check that the correct result is output.
Normally, some secret is involved in the computation, but no single party will learn
this secret unless all parties collude.

Prêt à Voter  An end-to-end voter verifiable election system that is suitable (with some
modifications) for VEC elections because of its user-friendly interface and ability
to handle ranked elections with large number of candidates.

Print-on-Demand service  A component that allows pre-prepared digital ballots to be
allocated and transferred to a print station in the polling booth, without any
central party learning the contents of these digital ballot.

Print-on-Demand client  The machine in the polling station that decrypts the digital
ballot received from the Print-on-Demand service and prints out the physical ballot
for the voter.

QR code  A 2D barcode containing stored information that can be easily read by com-
puter devices.

Race identifier  A numerical value to record the race information. It is pre-committed
and stored on the WBB.

SDD: Software Design Document  This document, which provides the design for the
back end of the vVote election system.

Table building and lookup  Algorithms to precompute and interpret a table built to
record the relationship between different ranking choices and their corresponding
numerical values. When a vote is decrypted, we get a numerical value that must
be decoded to retrieve the voter’s choices. This table enables us to avoid the need
to solve the discrete logarithm problem.

Verifiable Mixnet  The distributed component that ensures that all encrypted votes are
publicly and verifiably shuffled before decryption. A list of inputs will be processed
by a number of mix servers in sequence. Each mix server collects the list from the
previous mix server, re-encrypts and shuffles the list, and then outputs the result
to the next mix server; the relationship between the Mixnet input and output will
be kept secret unless all these mix servers collude. Each mix server also publishes
a proof of correct operation, so that the correctness of the Mixnet can be publicly
verified.

Vote packing  Each vote contains several ciphertexts. Vote packing is combining mul-
tiple ciphertexts into one in order to improve the performance of the Mixnet pro-
cessing the votes.
**WBB: Web Bulletin Board** It contains two services: a public facing website (called Public WBB) and an internal facing WBB (called Peered WBB or just WBB). The Peered WBB is run by a number of peers in a distributed fashion. It receives submissions in real time and provides appropriate administration of what to be published on the Public WBB. Once some information is written to the Public WBB, the information can not be altered or removed. Moreover, the information can be read by anyone, and everyone’s view of the Public WBB is the same.
4. Overall Description

4.1. Design Concerns

The overall design is based on the Prêt à Voter voting system \[\text{[CRS05, RS06, XSH}^+\text{07, RBH}^+\text{09]}\]. The vVote system \[\text{[BCH}^+\text{12a, BCH}^+\text{12b]}\] has taken the principles of Prêt à Voter and applied them to the setting of a VEC election. The overall election system must be end-to-end verifiable and without single points of trust, so that every step of the process can be verified by the voter or independent agents, but without revealing to any individual party or machine how an individual voter voted.

End-to-end schemes typically rely on advanced cryptography and multi-party computation protocols. This document will not describe the underlying details of such protocols or techniques, since they are covered in academic publications (Section 1.5 gives references for the results and techniques used in this design); however, it will refer to the properties and functions that are provided by such protocols and techniques.

4.1.1. Overview of Design

The key principle behind vVote is the ability to issue a printed record that allows the voters to verify the inclusion of their vote in the tally, without revealing how they voted. This is achieved using the randomised candidate ordering technique introduced in Prêt à Voter, and a printed ballot separable into two halves. In the original Prêt à Voter scheme, the voter is issued with a physical ballot that has the candidate names on one side in a randomised order, and the vote boxes (to be filled by the voter) on the other side, as illustrated in Figure 4.1. Additionally, underneath the vote boxes is the permutation of the candidate order, encrypted under a thresholded public key to which no individual person or machine holds the decryption key. Down the middle of the page, between the candidate names and the vote boxes, is a perforation line. The voter marks her choices in the vote boxes and tears down the perforation line. The side with the candidate names is then shredded, and the voter submits the vote boxes to the system, and keeps a copy of the completed boxes, signed by the system. The system is not able thereby to learn how the voter voted, because the ordering of candidate names is shredded before the vote boxes are scanned; the vote boxes on their own do not reveal the vote. The records of the choices are made available online, allowing voters to verify that their votes are included, unaltered, in the count.

The votes are counted by decrypting the permutation that is included (encrypted) with the choices. In order to prevent the printed record being associated with a decrypted vote, the submitted choices are securely and verifiably mixed multiple times by a Mixnet consisting of a collection of Mixnet peers. At the end of this process, provided that at
least one peer has behaved honestly, there is no traceable link between an encrypted vote and its decrypted output; however, the proofs supplied by the Mixnet will guarantee that the set of encrypted votes does correspond (as a whole) to the set of decrypted votes. The rest of this document will supply further details of these procedures.

An election has three fundamental phases:

- **Pre-election** preparation of election materials prior to polls opening
- **Vote Casting** the process of collecting and recording the votes cast
- **Post-election** the tallying of the votes to obtain the result

These three phases exist in the traditional paper voting scenario and also in the vVote system described in this document. The exact nature of the processes undertaken during each period is different, but fundamentally they have the same overall goal.

### 4.1.2. Pre-election

During this period the system must prepare digital ballots, in a fashion analogous to that in which ballots would need to be prepared in a paper-based election. However, owing to the randomised candidate ordering, each ballot is unique. It is important that no single party knows the ordering of any particular ballot; as part of the design, this document will specify a Distributed Ballot Generation protocol for creating these digital ballots. It forms the viewpoint specified in Section 4.3. The outcome of the Distributed Ballot Generation Protocol will be randomised ballots in an encrypted form, such that no single party knows anything about the permutation of the candidates.

Elections in Victoria, and in particular the Below-The-Line (BTL) ranked races, have a large number of candidates, often in the region of 35 candidates in a single race. Each ballot contains one ciphertext per candidate, and so if someone casts a BTL vote, all the ciphertexts for that ballot will need to be included in the mix. Multiplying the number of cryptographic operations that are required by a factor of 35 could easily make the computation infeasible within the time and resource constraints, however; and so, a Vote Packing technique is specified in the viewpoint in Section 4.5.4. This packs multiple ciphertexts into a single ciphertext to speed up the mixing and decrypting of the votes. It does, however, result in the need to construct a lookup table of all the possible voting
permutations in order to decode the decrypted value. This “Table Building” stage must be performed as part of the Pre-election, and possibly even several weeks ahead of the election. The “Table Building” is a computationally expensive task; however, the table lookup is very quick.

There are a number of different components involved in the running the election. Each component requires, at the very least, a signing key and corresponding public key certificate. This will include all EBM, Print, and Audit stations. The number of keys required is likely therefore to be of the order of several hundred. An internal Certificate Authority (CA) will be operated to sign the Certificate Signing Requests from the components. There is no need to use an external CA, since the network is closed, and the purpose of the signing and authentication is internal to the vVote system. This greatly reduces the cost since there is no per-certificate cost, and it provides greater flexibility. There are a number of open source CA packages: a suitable one will be selected. Prior to the start of the election the relevant Certificate Signing Requests will be created by the components and authorised centrally.

Additionally, the Mixnet must be set up and key generation must be performed. The setup of the Mixnet involves exchanging configuration files between each peer to ensure they are all aware of one another and able to communicate with one another. This service will be provided as part of the Mixnet, but will need to be jointly run by the various mix servers at the same time. This may happen as part of the Mixnet manager or as a separate one-off configuration step. The key generation procedure will be run through the Mixnet manager. Individual keys for other components will be generated locally by each component during its initialisation, with appropriate Certificate Signing Requests being produced.

All fixed values should also be committed to the WBB—for example, Candidate Identifiers, Race Identifiers and cryptographic group descriptions.

4.1.3. Vote Casting

The Vote Casting stage encompasses everything that will take place while the election is running. Because the election allows Early Voting, this phase will last around two weeks. There will be down time overnight for some components, but not all. Since votes could be cast from out of state, including overseas, the central system must be up and running whenever a poll station is open anywhere in the world.

One of the fundamental components covered in this design is the Print on Demand (POD) service. This allows pre-prepared digital ballots to be allocated and transferred to a print station in the polling booth, without any central party learning the contents of the ballot. The only machine that could ever see the contents of the ballot will be those in the polling station used either to print the ballot out or to cast the vote. The voter will interact with the POD service in order to get a physical ballot, before taking it to the EBM to cast the vote. Issues surrounding printing on demand are discussed in [Rya06].

The EBM (an external front-end component not detailed in this design document) will interact with the Web Bulletin Board to submit the vote and receive a digitally
signed receipt of the vote. This receipt will be printed out for the voter to take home. The receipt does not reveal any information about the candidates the voter has voted for, because it does not list the (randomised) candidate ordering.

The WBB is primarily used during the election, although it will be operational throughout to record and commit ballot data generated during the pre- and post-election stages. The WBB will be a key viewpoint discussed in this document.

There will be various administrative machines in poll stations to handle situations such as cancelling a vote if there is a problem during submission.

4.1.4. Post Election

The post election stage is focused on mixing and decrypting the votes. The votes need to be verifiably mixed in order to prevent anyone linking a decrypted plaintext vote with a submitted, encrypted, vote. The actual mixing will be performed by a verifiable Mixnet, which does not form part of this design document. However, the preparation of the files for the Mixnet, and the software to manage the running of the Mixnet, is part of this document and will form a viewpoint discussed here.

Figure 4.2 shows a high level overview of the design and how the various different

Figure 4.2.: Overview of Design
components will interact. In later sections we will provide individual viewpoints of each of the components. This design document contains the following viewpoints:

- WBB
- Table Building
- Print on Demand (POD) Service
- Distributed Ballot Generation
- Mixnet Manager
- Key Management

There are additional areas that will be covered; for example, network security and various administrative processes that will need to be implemented. However, these will not form separate viewpoints in this document. In general, polling station processes and procedures are not directly covered by this document. However, the communication between the components in the polling station and the back-end services is detailed here.

4.2. Table Building

4.2.1. Summary

The table-building process is an important step for efficiency. It is required to allow us to combine a permutation of multiple candidate ciphertexts into a single ciphertext for mixing, in such a way that we can unambiguously recover the permutation. Recovering the permutation is computationally intensive, so we take the approach of creating a look-up table to enable recovery to be efficient at run-time. The combining step greatly improves the efficiency of the mixing and thus reduces the time required before the tally can be produced. Although the construction of the table is a time-consuming and resource-intensive task, it can be performed in public (in that no secret information is involved, so it could be performed by a cloud service or on a cluster), and can be computed in advance of the election.

The table provides a means for efficient extraction of the permutation of candidates from its encoding as an exponent. It provides a mapping from the permutations of candidates \( \prod (CI_{Rank_j}^j \mod p) \mod p \) (see Algorithm 1 below) to the permutation \( Rank_j \), and needs to be pre-constructed since this extraction cannot be done efficiently in real-time.

4.2.2. Candidate Identifiers

Each candidate, or party, is allocated a candidate identifier \( CI \). (This will become clearer when looking at the Distributed Ballot Generation phase.) The candidate identifier is a value selected from the underlying group.
Candidate Identifiers must be unique at the point of decryption. The same underlying candidate identifier can be used in multiple districts/regions/races with the aid of a district identifier. This simplifies the table building process and allows the use of a single table for the entire election. It minimises space required and improves the efficiency of building the table.

If the underlying group is an Elliptic Curve group, the candidate identifiers will be points on that elliptic curve. Each one will be a co-ordinate \((x, y)\) as opposed to a single value. However, the \(x\) and \(y\) co-ordinates can be concatenated and treated as a single value provided the uniqueness requirement holds.

4.2.3. Base Value \textit{baseValue}

In order to encode several vote preferences (of a list of preferences) for candidates within a single data value, we first require a \textit{base value} \(b\). We use powers of \(b\) to represent the candidates, so candidate \(c_i\) is represented by \(b^i\). The preferences can then be encoded by multiplying each candidate value by its preference, and taking the sum of all these values, in the style of Baudron counting \([BFP+01]\). For example, if candidate 3 has preference 1, candidate 1 has preference 2, candidate 4 has preference 3, and candidate 2 has preference 4, the selection would be represented as

\[
1 \cdot b^3 + 2 \cdot b^1 + 3 \cdot b^4 + 4 \cdot b^2 = 3 \cdot b^4 + 1 \cdot b^3 + 4 \cdot b^2 + 2 \cdot b^1
\]

Henceforth we will use the name \textit{baseValue} as a meaningful variable name, rather than \(b\).

A concrete choice of \textit{baseValue} must be agreed upon prior to the election and start of table building. It must be larger than the maximum possible number of candidates, and should be committed to on the WBB and made public. We recommend that the value be derived from a hashed string to prevent any chances of manipulation. We recommend something along the lines of \(SHA256('VECVotePacking')\). The string used should be agreed by key stakeholders. Once a value has been selected it needs to be transformed using a pre-agreed method into a value that is in the subgroup, or on the curve in the case of Elliptic Curves. In fact the manipulations will be carried out on the exponents of the group generator \(g\), as shown in Algorithms 1 and 2 below. This is to enable the manipulations to be carried out homomorphically on candidate identifiers encrypted under ElGamal, as shown later in Algorithm 3 in Section 4.5.4.

4.2.4. Permutation Generation

The table holds every possible permutation of the votes for a given packing size \(PS\). We recommend a packing of size 6 initially\(^1\), although the construction phase should receive the packing size as a parameter and act accordingly. The actual table is the

---

\(^1\)The selection of such a value needs to take into account the balance between computational speed and storage space. By increasing this value, the mixnets and the decryption phase will be more efficient, but the size of the look up table will grow. The reverse is also true: by decreasing this value, the mixnets and the decryption will be less efficient, but the size of the look up table will decrease.
concatenation of all tables for 1 to $PS$. This allows us to handle partial rankings and races with fewer than $PS$ candidates.

### Table construction

Construct Candidate Identifiers (could be read in from a file)

```plaintext
foreach $i$ in $n$ do
    $CI_i = g^{(baseValue_i \mod q)} \mod p$;
end
```

**Algorithm 1**: Construct Candidate Identifiers

```plaintext
foreach subset $S$ of $\{1, \ldots, n\}$ of size $PS$ do
    foreach permutation $p$ of $S$ ($p_1$ to $p_{PS}$) do
        $Table_p = \prod_j (CI_{p_j} \mod p) \mod p$;
    end
end
```

**Algorithm 2**: Candidate Permutations

The result of the calculation would be stored into the table as $Table_p$ along with the respective candidate permutation $p$ in plaintext. Once the tables of all different sizes have been created, the table should be sorted completely and optimised for searching. The table should provide rapid searching of candidate permutations based on the lookup values calculated.

### 4.3. Distributed Ballot Generation

#### 4.3.1. Summary

The distributed ballot generation provides a verifiable way to produce ballots with a permuted list of candidates, without having to trust any single entity. No single device or entity learns the final permutation. The distributed ballot generation process will produce digital ballots that are encrypted under the POD Service’s public key. The approach is described in [Rya06, RT10].

The digital ballots will be committed to the WBB prior to their use. The aim is to produce all necessary digital ballots prior to the start of the election; however, if that is not possible, digital ballots may be committed during the election, as long as they are not used until they have been committed by the WBB onto the public WBB.

#### 4.3.2. What is a ballot?

A digital ballot consists of a collection of candidate ciphertexts, grouped into individual races. The ballot in the state of Victoria consists of two races, a Legislative Assembly (LA) race and a Legislative Council (LC) race. The Legislative Assembly consists of approximately 8 candidates that the voter will rank; there is one LA race for each of the 88 districts in Victoria. The Legislative Council race is larger, and run at a regional
level; there are eight regions across Victoria. A Legislative Council race consists of two voting options, Above The Line (LCATL) and Below The Line (LCBTL). ATL voting allows a voter to choose a party; the tallying rules say that such a ballot must be counted as if it were a BTL ballot filled in according to that party’s pre-published full preference ranking of all candidates. BTL voting requires the voter rank all the (roughly 20–40) candidates. Consequently, there is a ciphertext for each candidate in the Legislative Assembly, one for each ATL Party in the Legislative Council, and one for each BTL candidate in the Legislative Council. Each ballot therefore has in the region of 50 unique ciphertexts. These ciphertexts are permuted within their groups, LA, LCATL, LCBTL. The ciphertexts are not permuted across the groupings. The digital ballot will also contain a plaintext serial number that the voter can use as part of the verification process; in addition, the physical ballot contains a digitally signed copy of that serial number, embedded in a barcode.

What is in the ciphertexts?

Each ciphertext contains a pre-committed candidate identifier (or party identifier in the case of LCATL). This candidate identifier is a value from the underlying group that is used in the ElGamal encryption [Gam85]. Each identifier must be unique across all elections. Candidate Identifiers should be selected in a manner that is compatible with the table-building process described above. Where a race does not involve table building, the candidate identifier may be selected at random from the underlying group. Care should be taken to ensure that it is unique across the race/election.

Ballot Format

The physical ballot will be produced in a format specified by the VEC with the appropriate text and layout. The physical ballot will contain the serial number, printed in plaintext, as well as a QR Code. The serial number will be of the following form:

\[
\text{DistrictIdentifier: SerialNo}
\]

A list of district of identifiers will be provided to the ballot generation software. An example of a serial number is as follows:

\[
NTH: 245
\]

which refers to Northcote district, ballot 245. The contents of the barcode will be produced by the Print On Demand Client (POD Client) in the poll station, using the data provided by the POD Service. This will consist of a signed serial number and the appropriate candidate ciphertexts. The permutation information will be derived by the POD Client itself, since this information is not known to the POD Service.

4.3.3. Interface to VEC Election Management System

The list of candidates, parties, and races, will be provided in a CSV format from the VEC central system. This will consist of 3 separate files:
1. VEC-areas.csv
2. VEC-lower.csv
3. VEC-upper.csv

**VEC-areas**

This file contains 2 values per row in the following format:

\[ \text{region, district} \]

A region contains a number of districts; each district belongs to only one region. In consequence, each region appears in multiple rows, while each district appears only once. The recommended way of handling this file is to construct a folder hierarchy based on this structure. It will contain the following:

Regions
- RegionName1
  - DistrictName1
  - DistrictName2
- RegionName2
  - DistrictName3
  - DistrictName4

For example:

Regions
- Eastern Metropolitan
  - Bayswater
  - Box Hill
  - Donchaster

**VEC-lower**

This file defines the lower house race and consists of 4 values per row:

\[ \text{District, CandidateName, Index, PartyName} \]

The data contained within this file should be used to produce a list of candidate names, parties and identifiers for a particular district. The list should be in the order of the index values. The Party Name can be blank if a candidate is not affiliated to a party. It is recommended that this list is stored in the respective folder of the hierarchy created above.
This file contains the data regarding the upper house, regional, races. It consists of 8 columns per row:

- Region
- Grouping
- GroupingAlphaIndex
- GroupingIndex
- CandidateName
- Party
- Town
- Index

The region is self-explanatory; the Grouping refers to how the candidate is to be grouped and thus also defines an entry for the ATL portion of the ballot. This is important because some parties (coalitions in particular) submit joint ATL lists, and so the candidate party may not be the same as the grouping party. Each party listed in a region in the Grouping column should have an entry in the ATL list. The Grouping AlphaIndex and GroupingIndex refer to party ordering and are not needed during ballot generation. The candidate name and party are important for the BTL section of the ballot. The remaining two columns are not needed for ballot generation.

Once the various data files have been constructed, the next stage is to process this data. The processing step should generate an election specification file that will hold all the relevant parameters for each race. These will include the location of the public key, the location to store output files and whether the votes in that race are to be packed or not. This file will be used during the next stage to create the initial candidate identifiers and to prepare the initial ciphertexts for submission to the WBB and for ballot generation.

### 4.3.4. Initial Ciphertext Generation

The first step in the ballot generation stage is to produce a starting list of verifiable ciphertexts that can be passed to the Mixnet. In order to do this, a list of candidate identifiers must be generated. This step is dependent on whether the ciphertexts are due to be packed using the Vote Packing technique described in 4.5.4.

#### Candidate Identifier—No Packing

In situations where the ciphertexts will not be packed, candidate identifiers can be arbitrary elements of the underlying group. To avoid reuse of candidate identifiers across different elections these will be randomly generated. The first step is to generate a random number $r$, then create a random identifier that is in the underlying group, as follows:

$$g^r \mod p$$

This value should be recorded next to a candidate name and submitted to the WBB. This identifier will be the plaintext that is initially encrypted.

#### Candidate Identifier—With Packing

When the ciphertexts are to be packed, the candidate identifier is linked with the Vote Packing table generation. The candidate identifier is the following:

$$g^{\text{baseValue}^{\text{index}} \mod q} \mod p$$

24
where \textit{baseValue} is equal to the \textit{baseValue} used during the table construction in the vote packing, and \textit{index} is the index of the candidate in the list of candidate names.

Having constructed our candidate values, they should initially be encrypted with an ElGamal random value of zero, and the result published to the WBB. Others can use this to verify that the initial encryption was performed correctly, because the random value is fixed at zero and the candidate identifiers have been made public. After mixing, the proofs of mixing and re-encryption provide a guarantee that this initial list has been correctly mixed and re-encrypted, thus providing an overall proof that the ciphertexts have been correctly constructed.

4.3.5. Ballot Counter

In general, to achieve the ballot generation in a distributed fashion, the ciphertexts of each ballot need to go through a separate Mixnet. However, for large scale elections, e.g., millions of potential voters, this will cause the ballot generation phase to be very time consuming, since we need to run the Mixnet millions of times. Instead, we use a ballot counter here to record which ciphertexts are contained by a particular ballot. The benefit is that the ciphertexts of all ballots can be shuffled by a single run of the Mixnet, and in the Mixnet’s outputs, we can separate the ciphertexts of different ballots according to the ballot counter.

The process for creating the complete list and ballot counter is as follows:

1. Firstly, we assign a unique \textit{ballotCounter} to each ballot. This could take into account the race and district information. For example, ballots of different districts can be generated together provided that a record is kept of which race each \textit{ballotCounter} refers to.

2. Next, we create a pseudo ciphertext for each \textit{ballotCounter}, using value 0 as the special randomness. This pseudo cipher is denoted as \textit{Enc(ballotCounter)}.

3. Then for each ballot, we collect its ciphertexts which encrypt the candidate identifiers, and append the corresponding \textit{Enc(ballotCounter)} to each of its ciphertexts to form a two-column structure.

As a result, for all ballots, the result should be two column structure with the number of rows given by \textit{numberBallots} \times \textit{numberCandidates}. Each set of ciphertexts belonging to the same ballot should have the same \textit{ballotCounter}. The number of separate instantiations of the mixing should be minimised (with one important caveat, which will be explained shortly); consequently, it is highly likely that the most efficient approach will be to generate multiple districts or regions together. The mapping between district/region and \textit{ballotCounter} should be submitted to the WBB with the complete list.

This allows a verifier to check the complete initial list is correct.

Broadly speaking, it is desirable that the number of instantiations of the mix should be minimised: this reduces the overheads and improves the efficiency. However, a balance must be sought when instantiating very large mixes. The mixing requires that a threshold of the parties performing the mixing remain online throughout. If the mixing
is anticipated to take more than a few hours it is prudent to split the work up into separate instantiations, to mitigate the loss of time if the mix were to fail and need to be restarted as a result of network or system failure.

4.3.6. Mixing

The mixing will be performed by the Mixnet. The exact details are not covered here; however, an overview of steps is given.

The two column list needs to be exported from the WBB into the Mixnet required format. The mix servers will have been prepared in advanced in order that they have generated the required keys and have set up the necessary network connections. The Ballot Generation Mix can be started via the Mixnet manager. Once this has completed, the output list of ciphertexts should be submitted to the WBB along with the Proof directory. At this point, the first column of data should be decrypted. (Note that it is essential that only the first column is decrypted.) This will provide a list of ciphertexts that have been appropriately mixed. The decrypted first column provides an identifier for extracting the relevant ciphertexts for each race. For each \texttt{ballotCounter} used, extract the ciphertexts in the order they are given in the list. This set of ciphertexts is the permuted set of ciphertexts that will form part of a complete ballot. Having organised the ciphertexts into their \texttt{ballotCounter} groups the ciphertexts should now be grouped on a ballot-wide basis.

It will be necessary to generate more copies of the regional ciphertexts than of the district ciphertexts (since regions are subdivided into districts); the exact numbers will be provided as a parameter. Combine each group of district ciphertexts, both ATL and BTL, with the relevant regional ciphertexts. This gives an overarching grouping that forms a digital ballot. Each digital ballot should be issued with a plaintext serial number in the format specified above; the complete list should then be submitted to the WBB as the ballot ciphertexts.

4.3.7. Key Transformation

The final step of the ballot generation procedure is the key transformation protocol. This transforms the ciphertexts from encryption under the public key of the election into encryption under the public key of the Print on Demand service.

We transform a message under one key into a message under another key without revealing the underlying message \cite{Jak99}. This needs to be performed in a way that is verifiable both to the other parties taking part and to the general public. We assume that the two different key pairs were generated from the same group description. We have the following prerequisites:

- There is a joint public key \( y \), whose corresponding secret key \( x \) is shared amongst \( t \) ballot generation servers
- There is a joint public key \( z \), whose corresponding secret key \( w \) is shared amongst \( s \) POD servers
• Both keypairs share the same \( g \) value

• The message from the ballot generation mix is of the form \((u, v) = (g^r, my^r)\)

Most of the calculations for the Key Transformation can be performed in public and would commonly be performed by the WBB. However, the presence of a distributed WBB makes this process harder, and so we utilise a pre-commitment stage to simplify the process during runtime. This is of particular important during the Print On Demand protocol that makes use of the same key transformation technique, except it must perform it in real-time. There are therefore two stages, a pre-commitment that must be performed prior to the start of the election and any key transformations, and a live portion that is performed during the run of the actual protocols.

**Pre-Commitment Stage**

Each server (whether it is the Ballot Generation Servers during ballot generation or Print On Demand Servers during Print On Demand) \( i \) performs the following \( k \) times, where \( k \) is from 1 to \( \text{numberBallots} \times \text{numberCandidates} \)

- Select a random value \( e_i \) and \( f_i \)
- Calculate \( g^{e_i}, y^{e_i} \) and \( g^{f_i}, z^{f_i} \) along with a proof that the respective \( e_i \) and \( f_i \) are the same
- Submits \( g^{e_i}, y^{e_i} \) and \( g^{f_i}, z^{f_i} \) along with the respective proofs and the value of \( k \), all signed by \( i \), to the WBB
- The WBB will commit to these values at the end of the day. Ideally a separate commitment would be made to the Public WBB to ease checking of signatures later, but it is not essential.
- Following the commitment to the Public WBB each \( i \) downloads the set of all committed to values and verifies the WBB signatures and server signatures.

**Live Stage**

Each server \( i \) performs the following:

- The Server calculates the following using the values obtained from the Public WBB during the Pre-Commitment Stage:

\[
\begin{align*}
    u'' &= \prod_{i=1}^{t} g^{f_i} \\
    u' &= u \times \prod_{i=1}^{t} g^{e_i} \\
    v' &= v \times \prod_{i=1}^{t} (z^{f_i} \times y^{e_i})
\end{align*}
\]  

\( ^2 \)this is a standard \( \Sigma \)-proof of re-encryption
• The above values are submitted by $i$ to the WBB

• Each server $i$ performs a partial decryption, along with proof generation, for each $(u', v')$

• The partial decryptions and proofs are sent to the WBB. In the case of the Print On Demand protocol they are also returned to the POD Client

• The values can be combined by the POD Client or any public service to form the output $v''$

$$v'' = zf \times m$$

where $f = \sum_{i=1}^{t} f_i$.

At the end of the process, we have $(u'', v'') = (g^f, mz^f)$, which is the same message $m$ as we started with, but now under a different key. These ciphertexts, along with all proofs of the transformation and decryption, should be published on the WBB. The only values that should not be published, and which must be destroyed, are the random values $e_i$ and $f_i$.

4.3.8. Batch Verification

At an abstract level, a $\Sigma$-proof works as follows: firstly the prover makes some commitment and sends the commitment to the verifier; then the verifier randomly selects a challenge and sends it back to the prover (or the prover can generate the challenge herself using the Fiat-Shamir heuristic [FS86] in order to make the proof non-interactive); finally the prover uses her private knowledge to compute some values, denoted as $L$ and $R$, and then makes them public. To verify the $\Sigma$-proof, the verifier uses some public functions $f(x)$ and $g(x)$ to verify whether $f(L) = g(R)$. If the verifier needs to verify a number of independent $\Sigma$-proofs, instead of verifying them separately, she can use batch verification [BGR98] to verify all these proofs together. To achieve this, the verifier randomly selects a value $e_i$ for each of the $\Sigma$-proofs, then the equation $\prod_i f(L_i)^{ei} = \prod_i g(R_i)^{ei}$ will imply that for every $i$, we have $f(L_i) = g(R_i)$. In our case, a digital ballot will contain a number of ciphers, and for each cipher, we need to prove that the above key transformation has been correctly performed. Thanks to batch verification, one combined proof will be enough to prove that all ciphers have been transformed correctly.

4.3.9. POD Servers

Each POD Server receives a copy of the transformed ciphertext, which it will use during the POD protocol detailed in Section 4.4.1. The POD server can independently verify that the final ciphertext it gets is correct using the information on the WBB.
4.4. Print on Demand Service

The goal of the Print on Demand Service is to provide distribution of digital ballots for any district at any poll station in real-time. The components involved are shown in Figure 4.3.

The ciphertexts for ballots need to be pre-constructed and committed to the WBB. We therefore cannot create digital ballots in the polling station on demand: we need to distribute previously constructed ballots. The ballot generation procedure given above describes how such digital ballots are constructed and how they are transformed into encryptions under the public key of the POD Service. The same key transformation procedure described above will again be used during the Print on Demand protocol, to transform the digital ballot into the designated printer’s public key.

4.4.1. Print on Demand Protocol

Figure 4.4 provides a message sequence chart of the Print on Demand protocol.

4.4.2. Entities

Voter

This is an abstraction of a voter. The voter is assumed to have some additional functionality, for example, the ability to check signatures.
Poll Worker
This is a combined abstraction of the poll worker and the system the poll worker is using. Later in the document we will provide a more explicit split of the tasks, but for the sake of this protocol the two roles can be combined.

POD Service
This will be a thresholded service that uses some form of joint/threshold signatures. For simplicity in the message sequence chart it is treated as a single entity. When a signature is issued in a message or a decryption/encryption, it should be considered as if a threshold of the POD Servers have co-operated to perform that action.

POD Client
This will be a tablet with an attached printer or similar. It will be located in a booth-type structure to provide privacy, but will be online.

Ballot Manager
The ballot manager will be a single server architecture whose only role is to apportion serial numbers. This is utilised to simplify the process of assigning a serial number.
without having to run a complex agreement protocol. It does not need to be trusted, since its assignments must be backed by a signed confirmation from the WBB.

**Web Bulletin Board**

This is the bulletin board as described in other design documents. It will also be a threshold-based service, but for the sake of simplicity we abstract it to a single entity. Like the POD Service, when it issues a signature, it is a joint/threshold-based signature.

4.4.3. Description of POD Protocol

The following describes a typical use case for the POD Protocol.

A voter walks into the polling station and presents appropriate ID to the poll worker. The poll worker checks the voter off on the register and requests the administrator machine to generate a random *sessionID*. This *sessionID* is just a random number, containing no identifying information. If the POD Client is separate to the administrator device the *sessionID* could be in the form of a 2D barcode, magnetic strip card or even a smart card; it could also be selected from a box by the poll worker or by the voter. However, it is currently assumed to be generated by the administrator machine controlled by the poll worker. Having generated, or scanned, the *sessionID*, the poll worker submits it, along with the district of the voter, to the POD Service. The POD Service signs the *sessionID* and stores this information in its database. When the poll worker has received confirmation of reception, he either hands the *sessionID* to the voter or submits it, via his administrator machine, to the POD Client.

It could be that the POD Client is attached to the administrator machine, if the ballots are to be printed at the registration desk. If they are to be printed in a separate, private, booth the signed *sessionID* will be produced in barcode form and handed to the voter. We maintain the abstraction of a POD Client device, even in the case of its being attached to the administrator machine, to distinguish the roles being performed.

If the POD Client is separate, the voter moves to the POD Client and scans the barcode (*sessionID*). (Alternatively, the POD Client can be pre-loaded with the *sessionID* by the administrator machine.) The POD Client signs the *sessionID* and submits it to the POD Service. The POD Service looks up the district it recorded previously. It signs both the district and the *sessionID* and submits the result to the Ballot Manager. The Ballot Manager looks up the next available serial number for that district and assigns it to the submitted *sessionID*. It then notifies the WBB of this assignment and waits for signed confirmation of the assignment from the WBB. This signed response is returned to the POD Service. The POD Service will only release an actual ballot once it receives a signed serialNo and *sessionID* from the WBB. The POD Service transforms the relevant ciphertexts into the relevant printer’s public key. It then signs the serialNo and returns both the signed serial number and the transformed ciphertexts to the POD Client. The POD Client decrypts the ciphertexts and prints the resulting plaintexts, along with the signed serial number and any other relevant information (for example, the permutation needed by the EBM).
The content of a ballot is a collection of ciphertexts, one for each of the \( n \) candidates on the ballot. The key transformation will therefore be performed \( n \) times, and \( n \) candidate ciphertexts will be returned to the POD Client. The proofs of decryption and key transformation will be submitted to the WBB as evidence that the voter can later check. The details of the key transformation are identical to those used during the distributed ballot generation, except for the keys used. Here we are transforming from the POD Service key to an individual POD Client key; since this is performed on a per ballot basis, it must be done individually rather than using the batch approach adopted during the distributed ballot generation. This will have an impact on the number of proofs produced, since it will not be possible to generate batch proofs across multiple ballots.

4.4.4. Data Format

Communication between the client and servers will be performed using JSON. The basic structure is as follows:

Listing 4.1: Poll Worker Request

```json
{
  "district" : "Northcote",
  "nonce" : "RandomValue",
  "signature" : "Sign\_PWK\{district, nonce\}"
}
```

Listing 4.2: Poll Worker Response

```json
{
  "signature" : "Sign\_POD\{Nonce\}"
}
```

Listing 4.3: POD Client Request

```json
{
  "nonce" : "RandomValue",
  "signature" : "Sign\_POD\_C\{Nonce\}"
}
```

Listing 4.4: POD Client Response

```json
{
  "serialno" : "NTR:123",
  "signature" : "Sign\_POD\_S\{SerialNo\}"
  "candidates" : [ 
  {"alpha":"g^r", "beta":"Cand_1 \times g^r" },
  {"alpha":"g^{r_1}", "beta":"Cand_2 \times g^{r_1}" },
  {"alpha":"g^{r_2}", "beta":"Cand_3 \times g^{r_2}" },
  {"alpha":"g^{r_3}", "beta":"Cand_n \times g^{r_3}" }
  ]
}
```
4.4.5. Threshold Decryption and Key Transformation

The key transformation protocol requires a decryption of each ciphertext during the key transformation. In the ballot generation phase we can use the built in decryption routine included in the Mixnet, since we will be operating on large batches and relatively few runs of the protocol. A threshold cryptography approach \cite{DF89,Ped91} must be implemented for the POD Service. The construction of the joint public key and secret keys should be performed in a distributed manner to ensure that no single party has knowledge of the complete secret key.

4.4.6. Auditing

The auditing procedure provides a method of auditing the actions of the Print on Demand Client. In standard Prêt à Voter, the process is used to audit the construction of the physical ballot. However, with the distributed ballot generation detailed in Section 4.3, the construction of the underlying ballot data, including the ciphertexts submitted to the WBB and the ciphertexts held by the POD Service, can be audited by checking the proofs created by the ballot generation mix and key transformation protocol. These proofs can be checked for all digital ballots, and not just those that are audited; assurance can thereby be obtained that the ciphertexts on the WBB and the ciphertexts held by the POD Service are well formed. However, this check does not provide assurance that the POD Client has performed honestly and accurately. For this, we need an audit process to check that the final step is performed correctly.

As mentioned above, when a physical ballot is printed out, the ciphertexts are transformed into the key for the POD Client device and then sent to the device. Those ciphertexts cannot then be decrypted by anyone other than the POD Client itself. When a voter receives a printed ballot, she is free to choose to audit the printing of the ballot. To do so, she must go to the Ballot Auditing Station. She will scan the barcode on the physical ballot, which includes the SerialNo. The SerialNo will be sent to the WBB in the form of an Audit request, whose structure is detailed in Section 4.7. On receipt of authorisation from the WBB, the Ballot Audit Station will submit the signed authorisation, along with the serial number, to the POD Service.

When the POD Service receives an audit request, it will first check that the authorisation from the WBB is valid. Each POD Server will then decrypt the ciphertexts directly and construct a signature, under the \textit{PODS} joint signing key, of the serial number and plaintexts. Each server will return this signature share to the Ballot Audit Station. Once the Ballot Audit Station has received a threshold set of these signatures, it combines them and checks the signature is valid and correct for the permutation contained in the barcode on the ballot paper. It then constructs a receipt to print out with the signature along with the plaintexts. The Ballot Audit Station is able to construct the plaintexts from the permutation value that the POD Client placed into the barcode on the ballot; the POD Service therefore needs communicate only the signature, and not the plaintexts. If the POD Client has been honest the signature will be valid; otherwise, any attempts to cheat will be detected.
The audit signature along with the plaintexts should be printed onto an audit receipt. The barcode on the audit receipt should also contain the same data as that within the ballot barcode. This allows the voter to use a smartphone to scan the physical ballot and check the signature and reconstruct how the ballot should look for that signature, thus enabling an independent audit of the ballot without having to trust the Ballot Audit Station.

Details of the protocol are shown in Figure 4.5.

4.5. Mixnet Manager

The Mixnet manager should provide a simple-to-use command-line and graphical interface to run the processes associated with the mixing of the votes, as well as the mixing itself. It should provide adequate feedback to assist the administrator to monitor and react to any problems. The Mixnet manager will perform the following tasks, each of which will be expanded on in the following sections:

- Mixnet setup
- Ballot Generation
- Download vote data from the WBB
- Check vote data signature
• Format vote data into Mixnet-compatible format (including vote packing and district identifiers)
• Run various mixes and decryptions
• Vote Packing Look-up
• Upload proofs to WBB
• Upload plaintext vote data to the WBB

4.5.1. Mixnet Setup
The Mixnet setup phase will be run prior to the start of the election and before ballot generation.

4.5.2. Download vote data from the WBB
Each mix server should be prepared to download the vote data from the WBB independently. A suitable web-based interface will be provided to allow this to take place. In certain circumstances, the mix servers may all be located in a single room, and the vote data will be provided from a single source. In such circumstances, the Mixnet manager should be capable of accepting such offline data, while still independently checking the veracity of the data by checking the digital signatures posted to the WBB.

4.5.3. Check vote data signature
The data will have been committed to the WBB each night, and so the list of nightly signatures will need to be checked. This will involve splitting the downloaded data into the appropriate daily submissions and checking that the digital signatures covering those commitments are valid. This should be an automated process that provides feedback to the user on the successful checking of signatures as well as keeping a log of the signatures checked. Should a signature fail, a user must be alerted. It should not be possible to proceed with the mixing until the signatures have been successfully verified.

4.5.4. Format vote data into Mixnet compatible format
The data returned by the WBB will be the raw vote data submitted from the EBM. This data must be combined with the ciphertexts that were submitted to the WBB during the ballot generation stage. The first step is to retrieve the appropriate ciphertexts for each ballot, via the matching serial number. Having retrieved the ciphertexts they should be ordered by preference. Note, the ciphertexts generated as part of the ballot generation are in the appropriately permuted order for the ballot and thus the re-ordering is purely by preference. For example, if there were six ciphertexts \((\theta_a, \theta_b, \theta_c, \theta_d, \theta_e, \theta_f)\) and the submitted preferences are \([3,1,4,6,5,2]\), the ciphertexts would be re-ordered as follows: \((\theta_b, \theta_f, \theta_a, \theta_c, \theta_e, \theta_d)\). This is the data that would be submitted to the mix servers for the mixing.
Vote Packing and Race Identifier

In order to optimise the mixing process and reduce the number of individual mixes that will need to be run, all the votes for a particular type of race will be mixed together, even when they come from different districts. For races which do not use Vote Packing, the candidate identifier will be unique across the entire election, and no race identifier needs to be applied. In situations where Vote Packing is being used, a Race Identifier must be used during the Vote Packing process.

The Race Identifier will be a pre-committed value stored on the WBB. It must be applied to the Packed Vote only, not the individual ciphertexts. Votes are packed into groups of size $PS$, the packing size described in Section 4.2.4. In order to do this, the set of ordered ciphertexts constructed above are separated into groups of size $PS$; the last group can be of size $PS$ or less. For example, if we have $n$ candidates and a set of ciphertexts $Ciph_{i}$ from 1 to $n$, in preference order, the following would be performed to provide a sequence of $PackedCiphs$:

$$j = 1; k = 1;$$
$$PackedCiphs_k = 1;$$

for $i = 1 \rightarrow n$
do

$$PackedCiphs_k = (PackedCiphs_k \times (Ciphs^j_i \mod p)) \mod p;$$

if $j == PS$ then

$$j = 1; k = k + 1;$$
$$PackedCiphs_k = 1;$$
end

$$j = j + 1;$$
end

Algorithm 3: Vote Packing Algorithm

Each $PackedCiphs_k$ will therefore be an encryption of one of the permutations generated by Algorithm 2 on Page 21. For example, consider the sequence of ciphertexts $(\theta_b, \theta_f, \theta_a, \theta_c, \theta_e, \theta_d)$, which is ordered by preference. If the packing size $PS$ is 3, then Algorithm 3 will pack a triple of votes into each packed data value, as follows:

$$PackedCiphs_1 = (\theta_b^1 \times \theta_f^2 \times \theta_a^3) \mod p$$
$$PackedCiphs_2 = (\theta_c^1 \times \theta_e^2 \times \theta_d^3) \mod p$$

Recall (from Section 4.3.4) that each $\theta_i$ is an encrypted candidate identifier, and so of the form $E(g^{baseValue \mod q \mod p})$. So if candidates $a, b, c, \ldots$ are numbered 1, 2, 3, \ldots then the product of powers used to compute the $PackedCiphs_i$ combine homomorphically to provide encryptions of the entries of the table computed in Algorithm 2.

$$PackedCiphs_1 = E(Table_{(2,6,1)})$$
$$PackedCiphs_2 = E(Table_{(3,5,4)})$$

Once the votes have been packed accordingly, each $PackedCiphs$ must have the race identifier applied. The race identifier $ri$ is inferred from the serial number; the appropriate value should be looked up on the WBB. The requirement is that the value $ri$ must
be in the group $G_q$, because $r_i$ will be multiplied with the $PackedCiphs$ rather than the plaintext encrypted in the $PackedCiphs$. There may be multiple race identifiers to apply—for example, one for the district race and one for the regional race. For each of the races, the following is performed on the packed ciphertexts:

```
foreach element pc of the line PackedCiphs do
    pc = (pc \times r_i) \mod p;
end
```

**Algorithm 4: Race Identifier Algorithm**

Once the ciphertexts have been either packed or placed into the appropriate order, the data must be converted into a suitable file format for the mix servers.

### 4.5.5. Run Various Mixes and Decryptions

The Mixnet manager should be configurable to run various mix and decryption operations on the underlying Mixnet. The Mixnet manager will provide a graphical user interface for selecting the appropriate Mixnet to be run. It will provide protection against accidental running of stages; for example, key generation after a key has already been generated. The Mixnet manager will also provide feedback on the currently running Mixnet, to enable the end user detect any errors or connectivity problems.

### 4.5.6. Vote Packing Look-up

Following the decryption step, any votes that have undergone the vote packing step will need to be decoded. Those votes that were not packed will decrypt to the correct candidate identifier; however, the packed votes will decrypt to the combined value of the packed votes. In this case, to get the plaintext vote, the combined value must be looked up in the Vote Packing table, which will provide the appropriate decoding. Additionally, if a race identifier has been applied, a number of lookups will need to be performed to identify both the race the vote is for and the decoded vote. It is worth remembering that the output from the Mixnet is just the decrypted vote: there is no serial number, so we need to identify (by searching) the appropriate race the vote should be applied to.

**Vote Packing—No Race Identifier**

Where no race identifier has been applied, the decrypted value should be looked up in the previously constructed vote packing table. Race identifiers will be applied across all inputs to a particular mix, but they may not be applied to all mixes. The vote packing table will be a sorted table, so the lookup is a simple binary search. This will need to be performed for each decrypted ciphertext, and so should be optimised for maximum efficiency. There are a number of techniques for optimising the lookup—for example, fixed length values and cached mapped lookups. An optimisation could be obtained by taking further consideration of the structure of the overall table. The vote packing table should exhibit a uniform distribution of values, since the output of any individual value will be pseudorandom. Hence, it should be possible to predict a smaller section of
the table where the lookup will start. At an abstract level, the overall table will have approximately equal numbers of values starting with a ‘1’, as it does starting with a ‘2’, and a ‘3’, etc. Thus, if the value to be looked up starts with a ‘1’, it is possible to narrow the search to a little over 10% of the entire table. By also looking at the next digit it will be possible to narrow it further. There is a trade off between time taken to predict the location and the time saved doing a bigger binary search. However, if the starting size of the table being searched can initially be reduced, it will help the performance of the underlying disk cache.

**Vote Packing—With Race Identifier**

Where a race identifier has been applied during the vote packing stage, the only way to recover the race identifier is to try multiplying the decrypted value by the inverse of each pre-committed race identifier \( ri^{-1} \mod p \), and then performing a search on the table. If no value is found within the table, the next race identifier is tried, and so on until a match is found. On finding a match, the relevant race identifier is known as well as the contents of the first ciphertext. From that point on, the race identifier can be used to process any other ciphertexts in the same row, since all ciphertexts in that row will be from the same race.

**4.5.7. Upload data to the WBB**

Once the Mixnet has successfully finished both mixing and decrypting, the relevant proofs and plaintext votes must be committed to the WBB. The WBB will provide a web-based interface for the Mixnet manager to communicate with for uploading the data. The exact contents and files to be uploaded will be specified in the Mixnet configuration document. The proofs are likely to be constructed as a folder containing a number of files, while the plaintext votes will consist of a single file for each mix that has been performed.

**4.6. Key Management**

A number of different signing and encryption keys will need to be generated by various different components. The keys for the central services—WBB, Print on Demand and Mix Servers—will be generated as part of those components. However, the signing keys for each EBM and Print on Demand Client will need to be generated on the devices themselves. Each Print on Demand Client will also need to generate an encryption key pair for use in the Print on Demand protocol. The standard way for verifying the validity of public keys is to produce an X.509 certificate to accompany the public key. There are commercial organisations that provide such a service; however, almost all of the keys we produce will only be used internally, so there is no need to engage with such a service. The one exception to this is the joint public key used by the WBB that produces publicly verifiable signatures. This public key will need to be signed by a recognised certificate signing authority.
4.6.1. Certificate Signing Authority (CSA)
An internal signing authority will be created. Existing open source packages already exist (for example, http://www.ejbca.org/) that provide all the necessary functionality. The Key Management package should provide a simple user interface for an administrator to authorise certificate signing requests, without requiring in-depth knowledge of the underlying system. All actions should be appropriately logged to provide accountability. Near real-time revocation lists should also be created to allow rapid revocation of a certificate, should a client device (EBM/Print on Demand client) become compromised. The underlying system may use an existing open source package or implement its own simplified authority; however, whichever approach is selected, the implementation should comply with appropriate PKI standards.

4.6.2. Certificate Signing Requests
The client devices and servers will produce standards-compliant Certificate Signing Requests. The CSA should provide a web-based interface for the submission of these requests. The requests should be displayed to alert an administrator that they need authorising. There is no automatic authorisation of a request. Once a request has been completed it should be included in the directory of certificates provided by the CSA. The client device will periodically poll the CSA, while waiting for a response to the signing request, until it has received either a positive or negative response. It is not anticipated that certificate signing requests will be submitted during the run of the election. This would only occur in the situation where a new device or re-initialised device has been added to the network, and in such a case, particular caution would need to be taken. The CSA signing key should be held offline during the election.

4.7. Web Bulletin Board (WBB)
The WBB will consist of two services; a public facing website (Public WBB) and an internal facing WBB (referred to as Peered WBB or just WBB). At an abstract level, this consists of two components. The internal component receives submissions in real time and provides appropriate signatures and authorisations. At the end of each day the internal-facing component commits to its contents and sends the contents and commitment to the public facing component. A voter must therefore wait till the end of the day before checking the WBB for their submission. The advantage of such an approach is that it makes serving the content easier and allows the voter to be certain that the contents will not change over time; in addition, it means that there is only a single view of the WBB at any one time. This is because the commitment is publicly broadcast.

The design aims to remove any single points of failure, and therefore avoids having a single machine responsible for the real-time, internal-facing component. The public-facing component can easily be mirrored since it does no processing and is just a conduit. In making the internal component robust, we have to manage the added complexity of having multiple peers, and be able to handle the possibility that some of the peers may
be acting dishonestly. The design is intended to work in a distributed manner, such that the individual peers could be run by different stakeholders. However, initially, all peers will be managed by the VEC.

The following functionality must be provided by the WBB:

- Receive votes and provide a signature guaranteeing that the data has been correctly stored
- Receive protocol data (proofs, ciphertexts, serial numbers, etc.) and provide a guarantee that they have been correctly stored
- Authorise audit requests for unused ballots and refuse requests for used ballots
- Allocate serial numbers for ballots
- Store all election data (proofs, ciphertexts, submissions)
- Commit all new data to the Public WBB at the end of each day

4.7.1. General Properties

These are the generalised properties:

1. The signature scheme threshold is \( \frac{2}{3} \)
2. The protocol can handle any number fewer than \( \frac{1}{3} \) parties being dishonest (including the EBM as a party)

4.7.2. Prerequisites

The following are the prerequisites on the threshold keys:

1. Each peer \( i \) has two key pairs \((SK_{1,i} \text{ and } SK_{2,i})\). The \( SK_1 \) and \( SK_2 \) (secret keys) are threshold signature private keys. Each has a corresponding publicly known public key \( PK_1 \) and \( PK_2 \).
2. The voter should be able to access \( PK_2 \) on a smartphone, and can check data signed with \( SK_2 \).

4.7.3. Vote Submission Protocol

1. Voter presses submit button on EBM
2. EBM signs vote data and simultaneously sends to all peers
3. On receipt of a submission the peers (concurrently) perform the following:
   a) Peer logs receipt of a communication
   b) Peer checks the serial number has not been used for auditing or previously submitted as a vote
• If serial number has already been used, peer creates a signature share of relevant error message and returns it to the EBM. Protocol ends without anything being written to the database.

c) Peer stores data in its database
d) Peer signs vote with its share $SK_1$, and sends it to all other peers
e) When a peer receives a signature share, it does the following:
   i. If it does not currently hold a share from that peer it stores it, otherwise it ignores it
   ii. Check if it has a threshold set of valid shares—a share is valid if it is a signature of the same data as the peer has signed
   iii. Once a valid signature has been combined it is stored in the peer’s database
   iv. Peer signs vote with $SK_2$ and returns it directly to the EBM

4. EBM waits until it receives a threshold set of signature shares or times out
5. Threshold set of shares could be a signature of the vote or an error signature
6. Signature is printed on receipt
7. Voter verifies signature and leaves polling station if valid, and cancels vote if invalid or no signature was returned

4.7.4. Notation

Some notation that will be used throughout the diagrams to aid brevity:

• The signed serial number: $\text{Sign}_{PODS}\{\text{SerialNo}\} : \text{SignedSerial}$

• The preferences $pPrefs$ are the voters preferences or choice in the races on the ballot

• A vote $v$ consists of $pPrefs, \text{SerialNo, SignedSerial}$

• EBM Signature $\text{EBMSig}$ is of this data $\text{Sign}_{EBM}\{pPrefs, \text{SerialNo, SignedSerial}\}$

• The submission from the EBM is: $\text{SerialNo, SignedSerial, v, EBMSig, EBMID}$, where $\text{RBMID}$ is the unique identification of the respective EBM

4.7.5. Receiving Votes

Figure 4.6 shows the communication between the EBM and three peers. We show only three peers for brevity, but the system must be capable of working with an arbitrary number of peers that will be configured prior to the start of the election.

Figure 4.7 provides a flow chart of the operation of the EBM and the actions it performs. Where timeouts are referred to, these should be configurable parameters, and
must not be hard-coded into the system. The timeouts will be adjusted according to the real-world setup. For example, if network communication is over a mobile connection, the timeout will be extended. Where a timeout exists on the Server and on the EBM, the timeout on the server should be shorter than that on the EBM. The flow chart in Figure 4.7 is linked to the flow chart in Figure 4.10 and illustrates what is occurring on the back-end.

Figure 4.8 illustrates the communication between the peers within the WBB service. This is further detailed in the flow chart in Figure 4.10. As above, this is illustrated with just three peers, but should be able to handle an arbitrary number of peers. Ideally, peers should be run in a multi-threaded environment to ensure the greatest throughput possible—the aim is to get as close to real-time as we can achieve. However, for security reasons, it is vital that there are not two concurrent instances of the protocol running on the same serial number, and so peers should carefully manage the threading of operations. When a request is received, it should be checked, in a synchronised manner, that an existing run of the protocol is not currently in operation for that serial number. If such an instance is already running, the request must be queued until the existing run finishes. If the serial number is different, the peer is free to run the protocol concurrently in a new thread.

The same process is followed if the contents of the message is different—for example, if a cancellation request or audit request is being sent. The exact contents and underlying actions will be different, as will be explained further on. However, the underlying communication structure is the same.
Start

Prepare and sign vote data

Send vote to all WBB Peers

Sleep for 1 second

Have I received a valid signature

Have I exceeded the timeout?

Print signature on receipt

Print error on receipt

End

End

Figure 4.7.: EBM to WBB
Figure 4.8: WBB Round 1

Figure 4.9: WBB to EBM
Figure 4.10: WBB Round 1
4.7.6. End-of-Day Commit

The end-of-day commit procedure transfers the joint collection of valid data to the public-facing WBB. The specific data to be provided to the Public WBB is yet to be decided: for transparency it should include the complete collection of transactions that have been processed by the peered WBB (votes, cancellations, and audits), and should enable voters to conveniently check that their votes have been included. The Database Commitment below supplies the complete collection of transactions to the Public WBB. The Hash Tree Commitment below provides a data structure that enables efficient checking of votes. The appropriate approach is still to be decided.

The end-of-day commit should be performed nightly when the service is under minimum load. Note, there is a chance that the service will never be under zero load, owing to the requirement to provide a service at foreign embassies in different time zones across the world. The process is fully detailed in the flow chart in Figure 4.11.

The following is a summary of the protocol to transfer the database of transactions to the Public WBB:

**Database Commitment**

1. Each peer creates a hash of its database, signs it (using an individual, not threshold, signing key) and distributes the hash to all other peers

2. On receipt of a signed hash of another peer’s database the peer stores it

3. Once a peer has received a signed hash from all other peers, or a timeout has occurred and it has received at least the threshold number of hashes, it compares all of the hashes

4. If all the hashes match, it means all databases are the same
   a) Each peer will now create a signature of its hash using the joint signing key $SK_2$ and distribute it to all other peers
   b) Once a threshold set of signatures is obtained the peer will contact the public WBB and offer to provide the database and associated signature

5. If some of the hashes do not match the following takes place
   a) Each peer sends its database to all other peers
   b) On receipt of a database from another peer the receiving peer will check for any valid votes/cancellations/audits that are not present in its own database and add them to it
   c) Once a peer has received a database from all other peers, or a timeout has been reached, it will hash the new database and sign it under the joint $SK_2$ and send that signature to all other peers
   d) Once a threshold set of signatures is obtained the peer will contact the public WBB and offer to provide the database and associated signature
A hash tree provides a possible data structure to allow voters to check their votes. A hash tree can be constructed from the database of votes, and transferred to the Public WBB with the following protocol:

**Hash Tree Commitment**

**Hash Tree Construction** Each Peer $i$ computes from its database $D_i$ the appropriate state for all serial numbers for which it has an entry—whether it is associated with a vote, a cancellation, or an audit—and constructs a hash tree $HT_i$ of this information. Call the root hash $H_i$.

**Optimistic Commit Step** This consist of two rounds:

- **Round 1**: Each Peer $i$ signs $H_i$ with $s_i^1$ and sends it to all other peers.

  $[\text{Peer } i \text{ to all peers:}] \text{commitmsg}_1 = H_i, \text{sig}_{s_i^1}(H_i)$

- **Round 2**: If Peer $i$ received correctly signed hashes from all other peers, or a timeout has occurred and it has received at least a threshold number of correctly signed hashes, and all of the received hashes are equal to $H_i$, then it signs $H_i$ with $SK_i^2$ and sends it together with its entire hash tree $HT_i$ to the public WBB.

  $[\text{Peer } i \text{ to public WBB:}] \text{commitmsg}_2 = HT_i, \text{sig}_{SK_i^2}(H_i)$

Protocol ends (success).

- Otherwise, go to the Fallback Commit Protocol. (This might be due to time-out or to getting different hashes.)

**Fallback Commit Protocol (One extra round)** This has another two rounds. If we know there has been an error, we can skip the optimistic commit step and do this first.

- **Round 1**: Each Peer $i$ sends $D_i$ to all other peers.

  $[\text{Peer } i \text{ to all peers:}] \text{commitmsg}_3 = D_i$

- **Round 2**: Having received $D_j$, Peer $i$ updates $D_i$ as follows: For all valid SerialNos $s$, if $D_j$ contains an update request $v$ on $s$ and Peer $j$ has also sent $t_1$ valid distinct $SK_j^1$ signatures on $v$, add $v$ to $D_i$.

Then recompute $HT_i$ and redo the optimistic commit step.
Figure 4.11.: WBB Commitment Flow Chart
4.7.7. Cancellations

Cancellations are a key component of the protocol and are essential for its security. A cancellation will be issued by an administrator device. This could be centrally managed, or run in the poll station (to be decided). Fundamentally, the process is similar to vote casting, although the exact contents of the cancellation request are slightly different. It should be noted that there will be manual procedures and a paper audit trail completed in order to effect a cancellation; however, the operation of such a process is outside the scope of this document. The cancellation procedure is as follows:

1. Administrator sends a signed cancellation request consisting of:
   \[ \text{Sign}_\text{Admin}\{\text{Cancel}, \text{SerialNo}, \text{Sign}_{\text{PODS}}(\text{SerialNo})\} \]
   to all peers

2. On receipt of a cancellation request the peer logs the request and stores it in its database. It signs the request using \( SK_1 \) and distributes the signature share to all other peers

3. On receipt of a signature from another peer the peer will store it in its database

4. Once a threshold of valid signatures is received the peer will sign the cancellation request under \( SK_2 \) and return it to the administrator machine

5. Once the administrator machine receives a threshold of valid shares it provides a cancellation receipt, including the signature, for both the administrator and the voter

6. If the process times out or a signature is not constructed the protocol is re-run until it is successful

The re-running of the protocol needs to be carefully managed. The same restrictions on concurrent runs of the protocol on the same serial number should still be enforced. Additionally, the fundamental requirement of the WBB is that nothing should ever be deleted or overwritten. In consequence, there could be multiple runs of the cancellation procedure. All the runs should be stored in relation to that serial number, until a successful cancellation has taken place. A successful cancellation is reached when a peer has a threshold set of signatures under \( SK_1 \). After this, the peer can reject a further cancellation request on the grounds that the cancellation has already taken place, but it should also return a copy of the cancellation signature to the administrator.

4.7.8. Audit

The audit process will be performed by the Print on Demand Service, once the Print on Demand Service has first obtained an authorisation from the WBB to undertake the audit procedure. The audit is very similar to a vote, except it does not contain any vote data, and is prefixed with the word “Audit”. The WBB will authorise an audit only if the ballot has not already been used for voting and has not been cancelled. The same process as a voting request is undertaken by the Audit Machine in the poll station.
It will contact the WBB and request an audit. The audit request can be performed multiple times if it fails; however, once a peer has received any form of audit request, whether successful or not, it will not permit that ballot to be used for voting. Each audit request should be recorded in the same way as a cancellation. Once a successful audit request has been processed and signed under $SK_1$, the peer can reject the request, but should also return the valid audit request signature.

4.7.9. Submission of Bulk Data

In addition to voting data, there are points within the system where data should be submitted for commitment to the WBB. For example, each mix server will need to publish the proofs of shuffling and key transformation following the ballot generation protocol. This is a slightly more complicated process, since there is no implicit identification for the request, unlike the serial number used in other such requests. However, the data submission is similar in nature to the submission of vote data, and should proceed in a similar manner. An unsuccessful request should be cancelled and a new request submitted. It is the role of the submitter to append a randomly generated requestID to identify a submission. The requestID should be signed by the submitter and the WBB should check that the signature is valid before accepting a request. The WBB must log the requestID along with the submitted data and should reject any attempt to resubmit a previously used requestID. The contents of the data being submitted is somewhat arbitrary. The message will therefore be of the following form: submitter, requestID, Sign$_{submitter}$\{requestID\}, data. Additionally, a signature of the complete request under the key of the submitter will also be appended. The data could be a file or raw data; how that is stored is left up to the WBB. The same procedure as the vote submission is then followed. If the submitter receives a valid signature under $SK_2$, they can be assured that a threshold set of parties have successfully stored their data. If the process fails, the submitter must request a cancellation of that data in a similar way to the vote cancellation procedure. However, in this scenario, only the submitter can request a cancellation of its own submission. Once the submission has been cancelled a new request with a new requestID is constructed and the protocol is run again. It should be noted that we do not expect these requests to fail; however, we must ensure suitable procedures are in place in case they do.

4.8. Public WBB

The Public WBB is a simple service that stores data and accepts updates from the Peered WBB. It should be implemented in such a way that it can be easily mirrored to allow both widespread distribution of the data and provision of robust access. It is likely that the Public WBB will be a cloud-based service that can be easily scaled to user/voter demand. The key operation is to serve the data it has from previous commitments. Essentially, this task involves simply serving the raw data and signatures; however, it is important to serve it in a user-friendly manner. For example, if a vote has been cancelled following a successful submission, the original submission will still be included
in the commitment, but it will now have a corresponding cancellation record; when a request for a serial number is received, all the relevant data should be displayed. The system should also provide easy access to download the complete set of data for both the entire election and for the daily commitment containing a particular ballot. This is to enable easy verification of the data by other organisations and voters.

4.8.1. Nightly Updates

The process for nightly commitments is detailed in Section 4.7.6. It was mentioned that once a peer has a successful signature on a submission, it will contact the Public WBB to notify it of the available update. On receipt of such a notification, the Public WBB will request the update from the relevant peer, which will send the complete database along with the necessary signatures. The Public WBB will receive multiple notifications of an available update, because each peer that has successfully taken part in the protocol will send such a notification. It is up to the Public WBB to select which peer to download the data from. Once it has downloaded the data, it should check that the contents of the database and the signature are valid. If the signature does not validate, the Public WBB should discard the data and request it from a different peer.

4.8.2. Mirroring

The Public WBB, as discussed above, should be set up to be easily scalable and mirrored. All updates from the Peered WBB will be sent through the Public WBB, so anyone wishing to set up a mirror of the Public WBB should be able to request notification of any updates received by the Public WBB from the Peered WBB; the Public WBB should provide such notifications so that the mirror service can download the relevant update from the Public WBB. There should be no external access to the Peered (internal-facing) WBB.

4.9. Testing, Error Management and Notification

There is an overarching requirement that almost everything should be logged, both in terms of outgoing and incoming communication and in operations being undertaken by processes. Clearly, there are certain pieces of data that should not be appearing in logs, and should never be stored to the hard disk, primarily the randomness values used in both signatures and encryptions/re-encryptions.

A significant portion of the protocols are in place to handle situations when something goes wrong or communication is lost. The process must always have a way of moving forward, and should never get into a state that prevents the election continuing or the result being obtained (subject to certain assumptions on the number of failures handled). Although failures are unlikely, it is essential they are handled and logged, and a suitable notification made.
4.9.1. Testing

At the heart of preparedness for failures is testing. The protocols have been designed to handle a certain number of failures. Such scenarios should be tested against, and detailed results provided. It is also necessary to test what happens when we break the assumptions on the number of failures. Procedures need to be prepared to handle a variety of situations, and such testing will feed into the preparation of those procedures. Unit testing is an important component of software development, but on its own is not sufficient to provide the level of assurance required of the system detailed in this design. The testing procedure will be a joint process between the front-end development and the back-end development. However, automated stress testing and failure testing of the back end should be prepared and run whenever changes have been made.

4.9.2. Error Management and Notification

The design has to be prepared for errors and failures. Failure scenarios should be logged and a suitable notification system should be put in place to provide election authorities with a global view of the system. The severity of the failure should be clear, and different viewpoints provided. For example, if a WBB peer stops responding, it is a serious issue and must be flagged accordingly. A failed connection between and EBM and a peer is not as serious; however, it should be logged. A view of the aggregated errors should also be provided, to give administrators an idea, and early warning, of any problems arising. For example, if there are many failed connections coming from a particular polling station, or several cancellation requests related to a particular EBM, this should be flagged to administrators for further investigation. This will require a threshold associated with different errors to determine when the issue is escalated and highlighted to the administrator. These thresholds should be configurable both before and during runtime to handle the dynamic nature of the system.
A. Mixnet Specification

A.1. Mixnet Specification

The background literature on Mixnets relevant for the purposes of this document, covering mixes, re-encryption mixes, and proofs of shuffles, is [Cha81, SK95, FS01, Nef01, Wik09], [TW10]. An example Mixnet is described in [Wik12].

The exact Mixnet that will be used by the vVote system has not been finalised at the time of writing, and so there is a level of flexibility required for interfacing to the Mixnet. The following section will provide a high level overview of the functionality that the Mixnet component will offer.

A.1.1. Functions

The Mixnet is at the core of the system and will perform a number of functions:

- Distributed key generation of a threshold cryptography key pair
- Mixing of rows of ciphertexts; able to handle both single and multi-column structures
- Threshold decryption in a column-wise manner
- Production of proofs of mixing and decryption in a format and style that will allow independent verifiers to be written

The Mixnet will provide interfaces for the submission of ciphertexts for mixing, as well as the production of files of the output plaintexts and the relevant proofs. The structure of the proofs directories and files will be fully documented to allow the implementation of third-party, independent, verifiers (see, for example, [Wik11]). The individual mix servers will be operated by different organisations (Electoral Commissions) and so the implementation must work over both an internal and an external network. The setup phases and safe shutdown phases will also be implemented and provided. The various operations will be called from the command line; however, the Mixnet Manager specified within this document should provide a simple front-end interface to the command line operations.

A.1.2. Interfaces

Because the Mixnet has not been finalised, it is not possible to state the exact file formats or interfaces that will be provided. However, such interfaces and formats will be
specified prior to the commencement of implementation of the system specified within this document. The file formats will be generic formats that are commonplace when handling data. For example, communication over the web will almost certainly use a JSON structure, while file formats for local operations are likely to use a Comma Separated structure or similar.