An STV Voter Verifiable Scheme

Wayne Thomson

Supervisors: Kris Bubendorfer, Ian Welch

Submitted in partial fulfilment of the requirements for
Bachleor of Information Technology.

Abstract

A Voter-Verifiable Election scheme (VVS) empowers voters to verify that their vote was correctly included in an election. An ideal VVS would allow the system to be audited, but not reveal any connections between voters and their votes. There is such a scheme, but it is targeted at First Past the Post which although widely used, is not the only election system. In this project, an analysis is carried out on the requirements of Single Transferable Voting (STV) and modifications are made to an existing electronic VVS system for use with STV. A prototype is designed and built as a proof of concept.

Contents

Chapter 1

Introduction

Electronic voting is becoming increasingly used throughout the world. However problems with current implementations have often lowered public confidence. There is a push for verification of elections, and in some of the states in America the requirement for a paper trail has been legislated [18]. A paper trail allows for checks on the system, but may conflict with the obvious concern for freedom from persecution, safety and privacy of voters.

People often think of electronic voting as exclusively Internet voting but Internet voting is just one type. Electronic voting is using electronic technology in some part of the voting process, whether the voting is for an election, referendum or a poll.

There are three main advantages that electronic voting has;

- Increased Speed.

  Ballots are collected and transported at a faster rate, as electronic transmission can move relatively instantly to where they need to be counted. Computers are much faster than humans at processing numbers and running calculations on digital data, so tallying as well as recounts and audits would be much faster.

- Increased Accuracy.

  Computers have a high accuracy when processing digital data. Accuracy is also increased by the ability to audit votes where it may have been a too expensive or lengthy process for manual counting. The vote data may be able to be shared easily amongst third parties who before, would not have been able to deal with the great number of ballots.

- Reduced Cost

  The huge cost of temporary election works can be reduced if no staff are required to tally votes or the number of employees at each polling booth are reduced. If ballots are not printed, then printing and transportation costs are saved.

Although these are important, it is also critical to consider the maintenance of voter anonymity.

A Voter-Verifiable election scheme (VVS) is where each voter is empowered to audit components of the election scheme. One VVS approach to electronic voting is [6] which provides a paper trail, preserves the privacy of voters and can be audited by third parties. It is well designed and should be flexible enough to be compatible with the major voting systems. The scheme specifies how:

- Authorities create paper ballots.

- Voters make their vote.
- Authorities count the votes.
- Authorities, voters and special groups can audit the system.
- The number of votes made for each candidate can be totaled and audited without the possibility of any links made between voters and particular votes.

Currently only First Past the Post (FPP) implementations of electronic VVS exist, so designing and implementing an STV VVS is worthwhile. The scheme [6], is targeted at FPP and so extending it to STV system would be useful.

Single Transferable Voting is used in elections throughout the world [15] including the Australian senate and New Zealand local body elections. It is and is often promoted as the system which offers more choice for the voter. In addition to electoral process, an STV VVS Receipt system could prove advantageous to other fields of Computer Science.

1.1 Project Objectives

There are two objectives set for this project. These are:

1. To modify the VVS scheme [6], to allow ranked voting as per Single Transferable Voting.
2. A proof of concept implementation addressing open issues in [6].

1.2 Contributions

This project makes the following contributions to the Voter-Verifiable Election Scheme:

- A clear description of the VVS in [6].
- An analysis of the changes required to extend VVS [6] to handle Single Transferable Voting.
- A method of permuting and unpermuting a candidate list for ranked voting under a VVS, that is if $\pi(original\ candidate\ list)$ = a permutation, then $\pi^{-1}(that\ permutation) = original\ candidate\ list$.
- A method to detect illegitimate ballots.
- An analysis of the use of digital as opposed to paper ballots.
- An analysis of different representations of the identifier a voter uses to check a receipt.
- An implementation as a proof of concept of the STV VVS.

1.3 Project Guide

Chapter 2 discusses related work to the field of electronic voting, voter-verifiable schemes and election systems. It also explains a method of identifying and informaly analysing security issues within a system. Chapter 3 gives a clear description of the VVS system this project is based on.[6]. Chapter 4 reports on the required abilities for an STV VVS, and develops modifications to allow VVS to be used for STV. Chapter 5 analyses three issues which are required for an implementation of VVS; detecting illegitimate ballots, deciding on digital or paper ballots and how the receipt identifier will be represented to the voter. Chapter 6 develops the functional and nonfunctional requirements of a VVS implementation, and explains the design of a prototype called STV-VVS.

Chapter 2

Related Work

This chapter outlines other material in the fields of voting systems, electronic voting and voting systems. Section 2.1 gives brief definitions of different election systems. Section 2.2 runs through identified problems with electronic voting. Section 2.3 discusses the history of voter-verified electronic election schemes. Section 2.4 explains the use of attack trees for ensuring that a system fulfills security requirements.

2.1 Voting Systems

A voting system is a system of procedures to poll a group for the answer to a particular question with predetermined answers. The system should outline how to poll the group and how to collate the results to form an answer. A General election is an example which uses a poll to answer the question of who should run the country.

Different voting systems are used for many different purposes and in many different countries. The major systems being; First Past the Post(FPP), Single Transferable Voting (STV) and Supplementary Member (SM). In 1992, New Zealand had a referendum to decide whether the voting system should be changed to either of these systems. A comparison of the different methods in a New Zealand context is given by [4, 17].

2.2 Problems with Electronic Voting

There are many papers dedicated to shortcomings of electronic voting, two opinions on implementation problems are [14, 9]. The common problems seem to be:

- Technical Oversights - Several cases had basic technical problems, both security and reliability issues. Examples include hardcoded passwords, bad programming and badly chosen hardware. Security vulnerabilities often stem from these technical oversights.
- Useability Issues - Many voters and election officials do not possess enough computer experience to deal with complicated computer systems.
- General Computer System Risks - Problems such as power outages and hardware failure can be inherent in all computer systems.

Any good electronic voting system must take into account these potential problems. The majority of electronic voting systems on the market are proprietary and little is known about the quality of the product and so the intention of [5] is to have an open solution which can be evaluated and built on by anyone.

2.3 Voter Verified Electronic Voting

This earliest electronic VVS is [5], which uses visual cryptography to print two receipts which separately do not reveal the vote. The voter can take one of the receipts out of the booth for crosschecking later, while the other is destroyed.

The main advantages promoted are :

- Voters can verify their own votes inside the booth and on the internet.
- The election's integrity can be verified mathematically.
- The need for blackbox voting is eliminated.
- System transparency improves robustness.

Requirements of an electronic voting scheme are given by [2], matched against [5]. The requirements set out in this paper are:

- Accuracy. Detect any errors and recover from them.
- Secrecy. Allow only the voter to know their vote and detect any failures of this. This provides voters with their privacy and reduces the chances of coercion and vote buying.
- Usability. Anyone on the election role should be able to participate in the process.
- Efficiency.
- Unbiasness. The voter's choice should not be affected.
- Scaleability. The Scheme should be practical in any election.

The same paper briefly describes three other electronic election schemes (Diebold, Schnier and VoteHere) and conclude that they believe that [5] is the closest match to their requirements. A comparison between their requirements and how or if their requirements are met by the other systems is not given.

After assessing the requirements of an electronic voting scheme a simplified version of [5] is given as [11]. Some of the original's dependence on technology is sidesteped by removing the use of visual cryptography. Two strings are created which when put together, the difference between them displays the vote. The strings are separated into two receipts in a similar manner to the original. Cyphertext is introduced into the chosen receipt which an authority can use to create the other string.

In a variant [12], the fixed list of candidates(or options) is replaced by one which varied between vote forms. Cyclic shifts are used to reorder the base candidate list between ballots, ensuring that once the candidate list is destroyed, the vote is hidden. This concept is a simplified equivalent to divvying the two pieces of visual cryptography in [5].

2.4 Attack Trees

Attack trees are a way to brainstorm, develop and display security attacks for a system. A comprehensive attack tree is a powerful tool as it highlights areas where mitigation strategies are necessary. An attack tree is kept as working documentation and in a perfect world, different trees would be reusable, crossing over many applications. It is working documentation because it is unlikely that someone or even a group of people would be able to produce every security requirement in one or a few sittings, let alone all attacks which need consideration (if this was possible, there would probably be no need for tools such as this).

High-level attack goals are used as a starting points(root nodes) for attack trees. Methods of achieving a goal are inserted into the tree as children of the goal to be achieved, and the tree becoming deeper as the high-level gain complexity. Nodes may have children which are all required for achieving that node's goal denoted as AND nodes or nodes may have alternative methods of accomplishment denoted as OR nodes.

Starting at the deepest level, nodes have zero, one or more associated weightings. An example of a weighting would be how much skill, money or time is required to accomplish a goal. By choosing different groups of attackers, weightings are used to determine whether the attackers are able to succeed in breaking system security as well as determining which attacks are probable. By moving up the tree from the lowest nodes, the weightings are added at each level. For example by choosing a scheme where nodes can either be determined as high cost or low cost, a node would have a high cost if:

- it is the bottom node in a path, and is determined as a high cost node.
- it's children are all required (AND nodes) and one child is high cost.
- only one child is required (OR nodes) for accomplishment, and all are high cost.

The steps that Schneier suggests for creating the tree are as follows:

1. Identify possible attack goals, which become separate trees. Note that trees can join where they share common methods.
2. Possible attacks which make a goal or attack possible will become a child of that node.
3. Apply weightings to the nodes.
4. Return and reevaluate the trees when ever new possibilities are thought of or modifications are made to the system.

As security goals have already been discussed, it is easier to introduce attack goals in the context of those security goals. This is done in a similar manner to [16]. A level will be added to the top of the tree which contains the security goals, the second level becoming the attack goals for those subjects.

Chapter 3

Understanding The Voter-Verifiable Scheme

It is difficult to understand the complex scheme given in [6] without a background in voting systems, electronic voting and network security. This chapter gives simpler definitions, with the same or similar notation, providing background information on some of the concepts which are assumed by the authors. Section 3.1 explains the reasoning behind the scheme; what goals are accomplished and the approach taken to accomplish those goals. Section 3.2 contains the definitions, breaking the system into four components; ballot construction, vote collection, vote processing and audits. These processes move through the different entities in the system depicted by figure 3.1.

3.1 Hiding Secrets

Secrets are hard to distribute securely; they can need to be communicated in a way that no one can evesdrop, by either using a closed channel or communicating in a way which no evesdroppers can understand. Cryptography can be used as a mathematical equivalent to an language which an evesdropper cannot understand, where some mathematical attribute or formula needed to comprehend communications is only known by the communicating parties. Cryptography is used within Chaum's scheme to hide the connection between voters and their votes.

Sometimes it is important to hide the overall picture from individuals, by breaking up a secret it can be distributed amongst a group so that something can be accomplished or is known jointly but not individually. Chaum's scheme encrypts parts of the secret (vote) independently to split up the vote counting so that no individual knows who voted for who and yet the entire process can be verified.

3.1.1 Important Definitions

- Candidate List: Every voter uses a ballot which contains a vote column for marking their vote and a candidate column which contains a candidate list. Each of the items in a candidate list lines up with a row of the vote column. Voting for a particular candidate, involves placing a mark in the row on the vote ballot corresponding to the candidate to be voted for. Each voter has a random candidate list.

- Vote Column Destruction: Once the vote is made, the candidate list is destroyed leaving the vote column which contains the position of the candidate voted for, but the information about the candidate voted for is effectively lost to any individual entity.

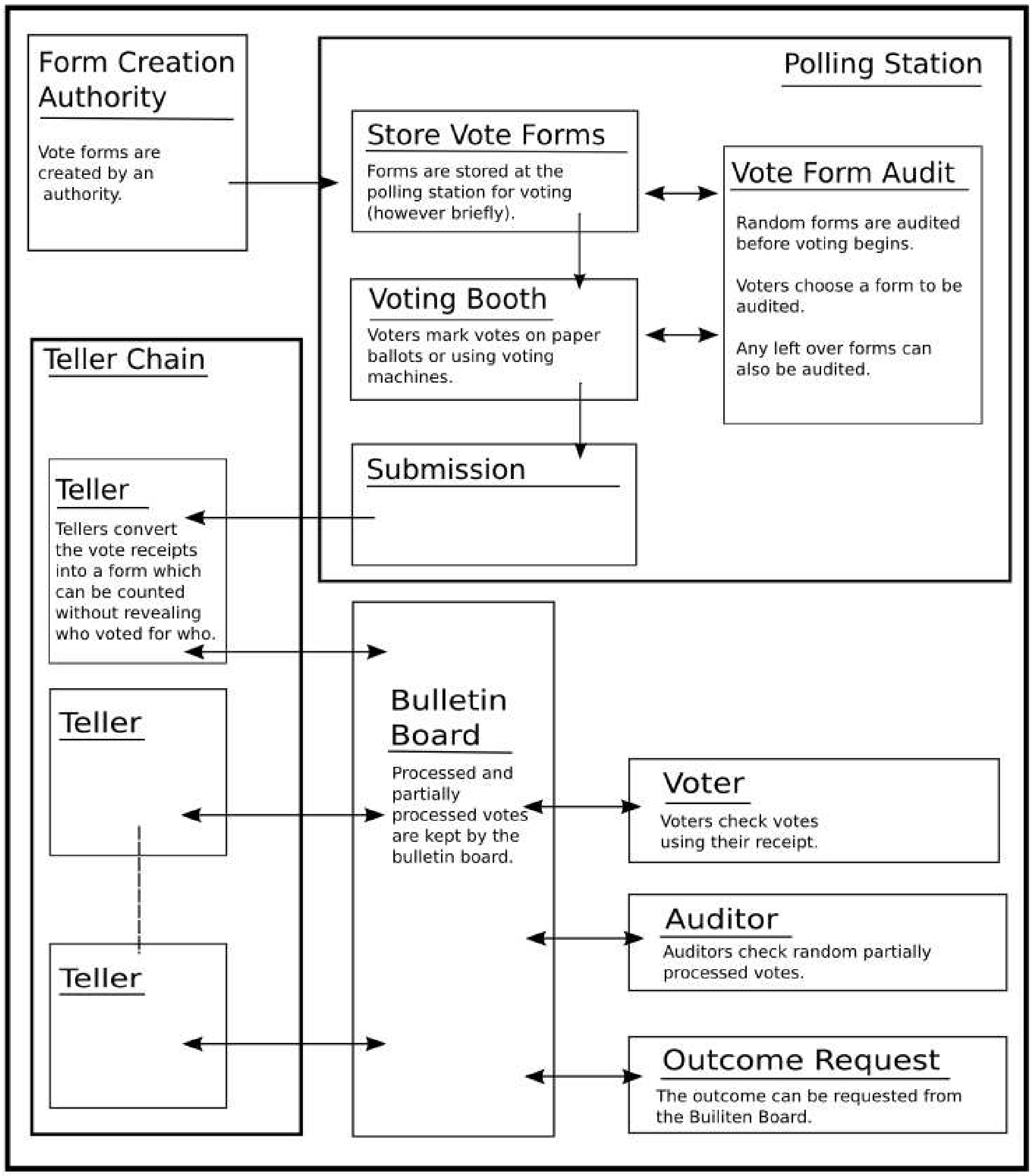

Form Creation Authority
Vote forms are created by an authority.
Polling Station
Store Vote Forms
Forms are stored at the polling station for voting (however briefly).
Vote Form Audit
Random forms are audited before voting begins.
Voters choose a form to be audited.
Any left over forms can also be audited.
Voting Booth
Voters mark votes on paper ballots or using voting machines.
Submission
Teller Chain
Teller
Tellers convert the vote receipts into a form which can be counted without revealing who voted for who.
Teller
Teller
Bulletin Board
Processed and partially processed votes are kept by the bulletin board.
Voter
Voters check votes using their receipt.
Auditor
Auditors check random partially processed votes.
Outcome Request
The outcome can be requested from the Builiten Board.

Figure 3.1: System Overview.

Once the candidate list is destroyed the remaining part of the ballot is called the receipt.

- Tellers: These are the entities that process and count the votes. There are multiple tellers to divide the overall secret (who voted for who). Each teller is the only entity that can unlock (decrypt) their portion of the secret.
- Teller Chain: A teller chain is a group of tellers, all it does in the system is set up a group of tellers.
- Bulletin Board: This controls intermediate transformations performed by tellers.
- Germs: A function using the sum of random values called germs is used to permute each candidate list. It is these germs that are the secret to be distributed. By encrypting each germ in a way which only one teller can access it by, enables the overall division of the secret. All of the encrypted germs together are called an onion, because they are encrypted in layers, one inside another.

3.2 How the scheme works

Ballots are preprinted by an authority containing two columns. The left column contains the list of options or candidates and the right is for the voter's choice or choices.

This scheme provides secrecy of the voter's choice/s by creating a different(reordered original) candidate list for each voter and only allowing that voter to see the candidate list they used. Secrecy can be assured by destroying the candidate list used once the vote has been made. The attributes used to reorder the candidate list are hidden under layers of encryption and can only be revealed through the group of tellers. Each teller knows only their part of the vote counting process, allowing votes to be tabulated and verified but not matched against any voter.

For example a vote is represented by fig.1. From this image it is impossible to tell who the vote was made by and even who the vote is for.

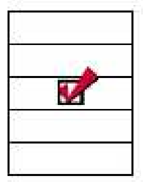


Figure 3.2: A Hidden Vote

3.2.1 Creating the Ballot

The first thing to be done in this scheme is the creation of a new candidate list based on the original. A random distribution is required so a set of random values (germs) is needed for some manipulation, creating a new ordering. It is these germs that are the attributes which will allow the tellers to recreate the list at a later point. The manipulation must allow the germs to be used to work backwards from the reordered candidate list to the original, and be able to manipulate each germ independently as a germ will have a particular teller associated with it.

A reordering of the candidate list is done using a single unique number called an offset. This is created by manipulating $2k$ germs, where k is the number of tellers. The steps to create the offset with publicly known cryptographic hashes are:

1. Create the set of germs:
   $G = \{g_0, g_1, ..., g_{2k-2}, g_{2k-1}\}$
2. Create intermediate hashed values: $hash(g_i)(mod\ v)i = 0, 1, 2, ..., 2k-2, 2k-1$ where v is the number of candidates.
3. The offset $\theta = \sum_{i=0}^{2k-1} d_i (mod\ v)$

The offset is used to permute the original list with cyclic shifts. The candidates are moved down by this value. When bottom candidates need to be moved, they are moved to the top.

The example figure 3.2.1 shows B shifted from A by an offset of three. The list is moved down by three. When an item needs to be moved down but is at the bottom of the list it is moved to the top. Cyclic shifts are simple to implement and understand.

A.		**B.**
Durian		Starfruit
Rambutan		Papaya
Starfruit		Mangosteen
Papaya		Durian
Mangosteen		Rambutan

Figure 3.3: An cyclic shift example

3.2.2 Introduction to the Onion

The onion is the tool used to hide the germs in an encrypted form for the tellers to match the vote against the original candidate list. It is passed from teller to teller and each teller can only decrypt their part of the secret.

Each teller has two public and private key pairs, of which the public keys are used to wrap their associated germs under onion layers. Each teller wraps two layers so that half of the manipulations can be revealed for checking without revealing the entire process. The core layer is created by taking a nonce and the first germ g_0 wrapping it with the 2^{nd} public key of the final teller (i.e. the teller which will be passed the inside core). The second layer is created by wrapping the core layer and g_1 under the final tellers 1^{st} key. The next layers are made by taking the next germ and the previous layer and wrapping it with one of the tellers corresponding public key.

An onion can be depicted as:
$\{g_{2k-1}, \{g_{2k-2}, \{...., \{g_1, \{g_0, D_0\}PK_{T0}\}PK_{T1}\}....\}PK_{T2k-3}\}PK_{T2k-2}\}PK_{T2k-1}$
or : $D_{i+1} := \{g_i, D_i\}PK_{Ti}\ Onion := D_{2k}$

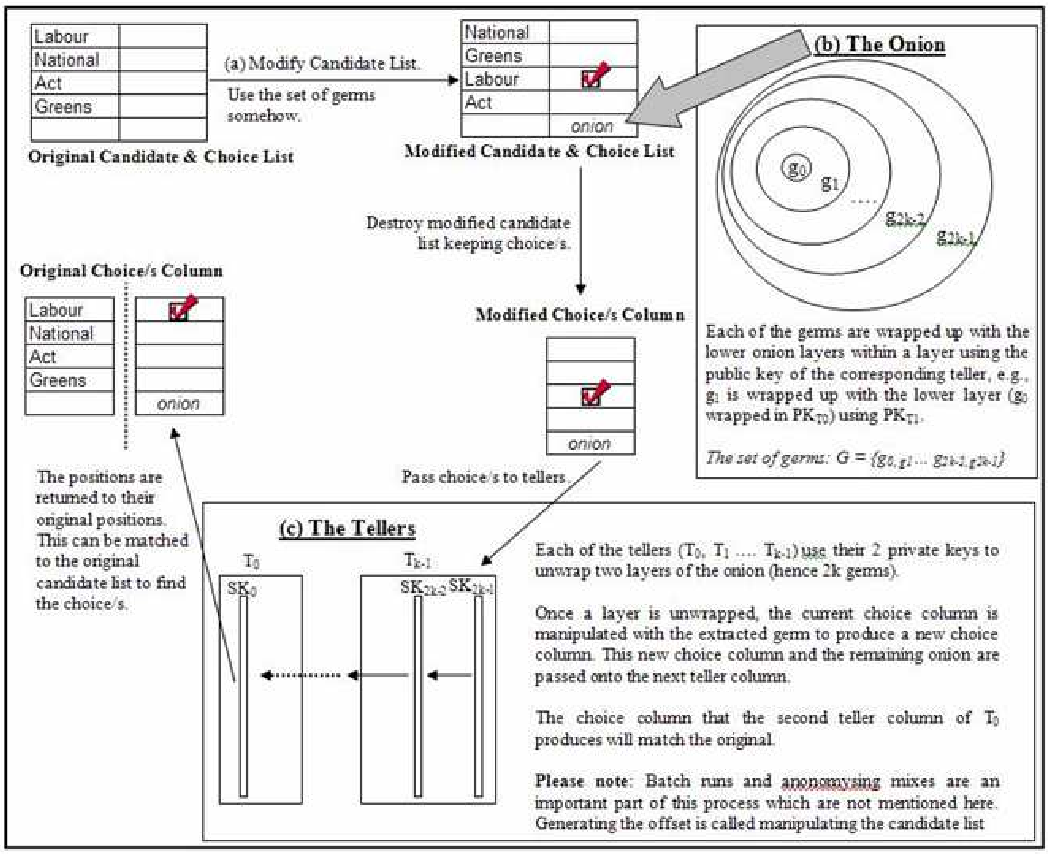
3.2.3 Processing the Votes

Once the vote is made, the choice represented by the vote column is input along with the onion into the teller process. Each teller successively strips two layers from the onion, revealing the associated germs. The germs are used to manipulate the vote column in the opposite way to the way which the candidate list was manipulated. After going through all of the tellers the vote column should be in the same order as the original candidate list, hence able to be matched against the original list.

The connection between voter and vote is hidden by processing batches of vote columns. The order of the vote columns that are output from a teller is different from that which was input, so unless all tellers are corrupted secrecy will hold. The column is taken by a teller, modified and then passed to the next teller until it passes through the last teller, ready to be matched against the original list. The column is represented by r which is the row number of the selection from the vote column.

The process for teller $i-1$:

1. Use the first private key PK_{2i-1} to strip the outer layer from the current onion D_{2i}, revealing the germ g_{2i-1} and the next layer D_{2i-1}.
2. Find the intermediary amount to shift the vote back; $d_{2i-1} = hash(g_{2i-1})(mod\ v)$.
3. Create the new r. $r_{2i-1} = r_{2i} - d - 2i - 1(mod\ v)$.
4. Using the second private key PK^{2i-2} to unwrap D_{2i-1} revealing g_{2i-2} and the next layer D_{2i-1}. The new r value is created and passed onto teller i with D_{2i-1}.


Labour
National
Act
Greens
Original Candidate & Choice List
(a) Modify Candidate List.
Use the set of germs somehow.
National
Greens
Labour
Act
onion
Modified Candidate & Choice List
(b) The Onion
g0
g1
....
g2k-2
g2k-1
Destroy modified candidate list keeping choice/s.
Original Choice/s Column
Labour
National
Act
Greens
onion
Modified Choice/s Column
onion
The positions are returned to their original positions. This can be matched to the original candidate list to find the choice/s.
Pass choice/s to tellers.
(c) The Tellers
T0
Tk-1
SK0
SK2k-2 SK2k-1

Each of the germs are wrapped up with the lower onion layers within a layer using the public key of the corresponding teller, e.g., g_1 is wrapped up with the lower layer (g_0 wrapped in PK_{T0}) using PK_{T1}.

The set of germs: $G = \{g_0, g_1 \ldots g_{2k-2}, g_{2k-1}\}$

Each of the tellers (T_0, T_1 .... T_{k-1}) use their 2 private keys to unwrap two layers of the onion (hence 2k germs).

Once a layer is unwrapped, the current choice column is manipulated with the extracted germ to produce a new choice column. This new choice column and the remaining onion are passed onto the next teller column.

The choice column that the second teller column of T_0 produces will match the original.

Please note: Batch runs and anonomysing mixes are an important part of this process which are not mentioned here. Generating the offset is called manipulating the candidate list

Figure 3.4: Technical Overview.

3.2.4 Finding the end result

Once each vote column has been completely reordered, it will match the original candidate list and the vote can be counted.

Chapter 4

Adaption for Ranked Voting

If an election requires voting for multiple candidates or ranked voting such as in Single Transferable Voting (STV), the original scheme [6] needs to be extended as conventional cyclic shifts will not suffice to hide patterns. As the candidates are always in the same order, a good guess can be made as to who was voted for. Any modifications must retain the principles of secrecy, accuracy and transparency of the original scheme. Leaving the well reasoned existing scheme as untouched as possible will reduce the chances of damaging these principles.

Section 4.2 identifies the required extensions to run an STV system under the Voter-Verifiable Scheme [6]. An analysis is given in section 4.3 of the current scheme's cyclic shifts, while section 4.4 outlines and evaluates several ideas to produce completely randomised full permutations before giving a definition Knuth permutations. Section 4.5 shows how an index of a permutation can be created so the tellers can undo the Knuth permutations.

4.1 Introduction to STV

4.2 Required Abilities of a Candidate Permutation

It has been suggested that full permutations should be used to vary the candidate list in practice regardless of the voting system [12, 6]. This is because the larger the number of permutations the less likely it is that any sort of guessing or pattern matching can be used. With cyclic shifts there is only v possible permutations (where v is the number of candidates) while there is $v!$ different permutations with full permutations.

1. Manipulations on candidate lists need to be reversible (using the germs) so that if $\pi(Originalcandidatelist)$ = a permutation,
   then $\pi^{-1}(thatPermutation) = OriginalCandidateList$ [6]

2. Nothing, not even an individual teller should be able to connect a particular vote column to it's vote. This is required to preserve voter secrecy. To achieve this the process of reordering a vote column to match the original should be able to be broken up, distributing parts of the secret to each teller.

3. The scheme should be scaleable.

Required Attributes		
Attribute	Achieves Attribute	Description
Reversible	YES	Provided by the simple addition and subtraction of intermediaries.
Connection between tellers	NO	Shifting the vote column back teller by teller removes any connection from vote column to actual vote. When using multiple or ranked votes, a reasonable guess can be made as to who the votes are for. The order of candidates will always be known which means that knowing one candidate's vote will reveal all.
Scaleable	YES	

Table 4.1: Fulfilled Requirements using Cyclic Shifts

4.3 Analysing Chaum's Cyclic Shifts

The original system works by modifying the original candidate list using cyclic shifts. The germs are hashed and added together and taken mod v, producing the offset. When bottom candidates need to be moved, they are moved to the top.

When returning the choice column to the original order, each of the tellers peels off two of the layers, extracting two of the germs. The columns can be shifted back teller by teller without revealing any more than two germs to a single teller. This is important to ensure that without compromising exactly all tellers, the votes remain secret.

The amount that cyclic shifts fulfills requirements of a candidate permutation is given by table 4.3.

4.4 Creating a Permutation for Ranking

4.4.1 Multiple Forms

The easiest way to allow rankings is to use multiple voting forms. This would not change the basic form of Chaum but would require a separate form for each of the rankings for each question. For example; if there were two questions and we wanted five rankings for each there would be 10 forms (A lot of calculations, including the calculation of many random numbers). American elections often have more than 20 questions and so this approach would not be viable. The amount that multiple forms fulfills requirements of a candidate permutation is given by table 4.4.1.

4.4.2 Full Permutation by List Creation

This scheme works in a similar way to hash tables. The idea behind this is to take the original list of candidates and add each candidate to a new list. Each candidate is assigned a position by taking a hashed germ mod $(v - x)$ where x is the number of candidates already placed. Only v germs would be used, which modifies the scheme in a negative way as we need to distribute the secret as much as possible. e.g. if there were only three candidates we would only have one and a half tellers which is not good enough. The amount that list creation fulfills requirements of a candidate permutation is given by table 4.4.2.

Required Attributes		
Attribute	Achieves Attribute	Description
Reversible	YES	Provided by the simple addition and subtraction of intermediaries.
Connection between tellers	YES	Shifting the vote column back teller by teller removes any connection from vote column to actual vote.
Scaleable	NO	Multiple forms may become very expensive in terms of computation and paper.

Table 4.2: Fulfilled Requirements using Multiple Forms

Required Attributes		
Attribute	Achieves Attribute	Description
Reversible	NO	This is not reversible unless all the tellers have access to all the germs.
Connection between tellers	NO	There is no way to reverse the permutation unless we give all the tellers access to all germs. Every teller (in which case we may as well remove all the tellers) will be able to show the connection. The amount of tellers will be directly related to the amount of candidates. A reduced amount of tellers may increase the possibility of corruption.
Scaleable	YES	This scheme would have amount the same computational cost as cyclic shifts.

Table 4.3: Fulfilled Requirements using List Creation

4.4.3 Full Permutation by Swapping

In this scheme we take the original candidate list and swap pairs until all germs are used. Four different ways of manipulating germs to find candidate positions to swap are outlined below:

1. **Dual hash** - This method requires two significantly different hash functions. We take the set of germs created as in cyclic shifts and apply each of the two hash functions to it, creating two new sets. The new sets will contain the candidate positions to swap. Figure **??** shows a candidate list being permuting using two hash different functions.

   $A = \{hasha(g_0)mod|C|, hasha(g_1)mod|C|, ..., hasha(g_{2k-2})mod|C|, hasha(g_{2k-1})mod|C|\}$
   $B = \{hashb(g_0)mod|C|, hashb(g_1)mod|C|, ..., hashb(g_{2k-2})mod|C|, hashb(g_{2k-1})mod|C|\}$

2. **Halves** - This scheme takes the original germ list and halves it. Two new sets are created by hashing those halves and taking them mod $|C|$

   $A = \{hash(g_0)mod|C|, hash(g_1)mod|C|, ..., hash(g_{k-2})mod|C|, hash(g_{k-1})mod|C|\}$ $B = \{hash(g_k)mod|C|, hash(g_{k+1})mod|C|, ..., hash(g_{2k-2})mod|C|, hash(g_{2k-1})mod|C|\}$

   The original candidate list has pairs swapped using the values from A and B. This scheme will not have the same amount of swaps as probably needed, i.e. k instead of $2k$.

3. **Dual Germ Sets** - The problems with halves is that it tries to create effectively two sets from one, so either the new sets and going to be smaller or replicate pairs. By introducing a new set of germs, there are $2k$ pairs that are unrelated without replication. The two sets of germs:

   $G1 = \{g1_0, g1_1, ..., g1_{2k-2}, g1_{2k-1}\}$
   $G2 = \{g2_0, g2_1, ..., g2_{2k-2}, g2_{2k-1}\}$

   Each hashed germ can be taken $mod|C|$ to create a new set of pairs:

   $A = \{hash(g1_0)mod|C|, hash(g2_1)mod|C|, hash(g1_2)mod|C|, hash(g2_2)mod|C|, ...,$
   $hash(g1_{2k-2})mod|C|, hash(g2_{2k-2})mod|C|, hash(g1_{2k-1})mod|C|, hash(g2_{2k-1})mod|C|\}$

   These two sets can be used for swapping in the same manor as halves.

4.4.4 Full Permutations per Germ

All the permutation methods stated above except cyclic shifts permute the list one candidate at a time. This means that depending on the number of candidates, the final candidate list permutation may not be unbiased. If the list was permuted completely and unbiasly once per germ then the final permutation should be unbiased.

- Sorted List of Permutations

  If all the permutations for a possible candidate list were computed it would be possible to use the germs to pick a permutation from that list. This would not work unless an algorithm allowed for the list to build all permutations in the same order, or if the list was sorted everytime it was created. It would be too expensive to even create the list of permutations with any number of candidates above seven.

- Knuth Shuffle

  A Knuth shuffle [8] is an algorithm for creating a permutation using random numbers. The candidate list is ran through from top to bottom, a random number being generated for each. The random number is taken $mod\ (v - x)$, where x is the number of candidates already shuffled including the current one. The candidate from the unshuffled candidates which matches the result is swapped with the current candidate.

When computers create random numbers another value called a seed is used as the base. The same seed given to a random number generator will always result in the same sequence of numbers. When the random numbers for a shuffle are created the germ is used as the seed, which means that the shuffle can be duplicated.

4.5 Reordering with a Permutation Index

The germs are known by their tellers when trying to reverse the shuffle, and can be used as the seeds for recreating the original shuffle. Instead of shuffling candidates, a list of integers is shuffled which can be used as an index:

1. The teller unwraps the germ as per normal.
2. A new index list is created, containing the numbers from 0 to $v - 1$.
3. The index list is shuffled with the germ as the seed for generating random numbers.
4. A new vote column is created.
5. Moving from top to bottom of the index list, the indexes are read. The item from the vote column in the position matching the index read is moved into the new vote column at the position matching the current position in the index list.

Chapter 5

Implementation Issues

In the series of papers [6, 3, 5, 12, 11, 10], there are several concerns for a voter-verifiable election scheme implementation that have not been disscussed, investigated or were left open for discussion. These are aspects which would not necessarily make sense to complicate the various schemes with ideas that would change according to best practices and different implementations.

These concerns are described and placed in the context of the current implementation. A choice of one approach to a concern is given after each problem is defined. Section 5.1 discusses the problem of identifying ballots which should not be inside the system, i.e. fake ballots created by an attacker or genuine ballots corrupted by an attacker. A comparison of using digital ballots with vote machines as opposed to paper ballots which builds on previous discussions [6, 10] is given in section 5.2. Different ways of representing the onion and the benefits of each are given in 5.3.

5.1 Ballot Authentication

5.1.1 Ballot Stuffing

Some precautions are needed to prevent invalid ballots entering the system [6]. Invalid in this case includes receipts that are cast by illegitimate voters or receipts that are cast by legitimate voters but are malformed.

Entering the same legitimate completed receipt into the tellers multiple times (a replay attack) is a form of ballot stuffing with the purpose of changing the outcome. Simply matching the number of votes cast with the number of receipts will identify when ballot stuffing has occurred but will not identify which votes have been inserted or where they came from [6]. The nonce encrypted in the first layer will identify the reoccuring votes, but this will not be found until the last layer is unwrapped at the final teller. This goes against the idea of distributing risk; i.e. by only corrupting the final teller, it would be possible to change the nonces, allowing the illegitimate receipts to be counted. Also this will not be able to identify where the vote was made.

To solve this receipts need to be authenticated. The most obvious answer is to authenticate using some identifier attached to a voter which would be ideal. Unfortunately this is unacceptable as it would reduce voter anonymity. Instead of authenticating by voter, authentication could be done by vote machine, polling booth or polling station. The danger of this would be if someone has access to records of where voters made their vote or watched where voters made their vote. Some sort of statistical analysis may be used to find which candidate someone voted for. In the same way that it might be useful to leave spare candidate lists for voters in each booth [6], having multiple voting machines within booths would stop observers from matching voters to particular voting machines.

Authenticating votes involves generating a public and private key pair for each machine or site. A signature digest of the vote column (including the rankings and onion) marked by the voter and a nonce would be encrypted under the machine or sites private key. The digest is taken of the entire vote column so that no vote can be submitted which contains a valid signature from another ballot. The nonce is generated by the machine or booth or site and is unique to that respective entity. Unlike the nonce encrypted under the onion, access will be possible from all tellers. A plaintext identifier will be written on the receipt so that the tellers have a reference to the public key required to decrypt the signature. Figure 5.1 shows a First Past the Post receipt where X is the mark indicating the row of the chosen candidate, Mx is the identifier and K^{-}_{Mx} is the private key.

X
$onion, Mx, K^{-}_{Mx}(votecolumn, nonce)$

Figure 5.1: An authenticated receipt.

Tellers can now flag and throw away any receipts not signed by legitimate vote machines. If ballot stuffing occurs or is suspected and all receipts have legitimate signature, checks can be made on the number of receipts signed by each vote machine, booth or polling station. The resulting number of signed receipts would be compared with the expected number of voters for each signing device to determine where errors have occurred. Another possible security benefit of this is that if one signing device is determined to be a problem due to corruption, all votes from that device can be located and discarded. This could potentially be misused by corrupt electoral or governmental officials, receipts from certain polling stations could be deleted or strategically removed with the excuse of erroneous or corrupt data. However paper ballots have the same vulnerability.

Using this method of authentication would need to be considered carefully, particularly the choice of signing device needs to be one which cannot be tracked down to particular voters. Ensuring accuracy must be balanced against the need for voter privacy.

5.1.2 Vote Form Substitution

The main motivation for substituting legitimate vote forms for illegitimate ballots is to change a voters vote. This could involve manipulating the vote form in some fashion along with possibly using social engineering techniques such as always making a particular candidate appear at the top of a candidate list. Someone may also try to find additional information about a voter by putting some sort of tracing information on the ballot, for example to find who they voted for.

If manipulated malformed voteforms are present, audits should identify that problems exist but in this section the concern is catching illegitimate forms which are wellformed. It would be possible to create a batch of well formed ballots, recording the onion and candidate list for each. By substituting these created ballots for legitimate ones, voters and their votes could be identified by their receipts. This needs to be stopped to avoid votebuying or forced voting by coercion.

To authenticate the ballots a public and private key pair for each ballot authority could be generated. The authority takes a signature of the encrypted onion and signs it with its private key. Note that a signature could somehow be created using the entire ballot, but the information contained within the candidate list will be destroyed during voting making the signature invalid after that point.

When the receipt gets to the teller or a vote machine, the authority's public key is used to obtain the plaintext signature. A signature is created using the same method as the authority and is compared with the decrypted version. If the two signatures match the receipt is legitimate. While this solution identifies illegitimate vote forms it does not identify ballots created by misbehaving authorities. If an authority is identified as a problem, it would be easy to identify all the ballots and remove them from the tally.

5.1.3 STV Prototype Implementation

In this implementation, the two suggestions from 5.1.1 and 5.1.2 will be included in the requirements. A public and private key pair will be created for the authority to sign each voteform. A public and private key pair will also be created for each vote machine, used to sign each receipt.

5.2 Digital Vote Forms

Througout the scheme, consideration needs to be given to the mechanism for transmission and use of data. Data can be used and transferred digitally for example, over the internet or using physical means such as paper. Digital data must be processed at some point as this scheme uses digital cryptography to create, verify and tally the votes. However there is no technical reason for storing forms in either a digital or physical medium. If ballots are not to be transmitted to polling stations digitally then they will have to be created at the polling or pre-generated paper ballots will need to be created by authorities in a batch process.

Three reasons for, and two reasons against using physical transfer methods to bring the voteform to the voter are explored below.

- Voter trust in vote forms

  In recent elections electronic voting machines such as those used in the 2000 American elections, have created a huge amount of stigma. It is important for voters to trust the the voting scheme otherwise voter turnout may suffer or the elected government may not be trusted by its own people or the international community. It is not useful to gain voter trust by giving voters receipts if they do not trust the voting scheme itself. The software required to show a voteform, take a vote and print the receipt is complex enough that a software engineer who walks into a polling booth would know nothing of the inner workings of the system, nevermind the average person who knows less about computers. Physical paper ballots which seem to hide nothing may be the most likely to be trusted by the voter, however it may be possible to increase voter trust in electronic forms by introducing and explaining the auditing groups through advertising. An interesting paper on increasing voter trust in a VVS scheme suggests using ballots familiar to voters such as scratchy cards [10].

- Ease of use

  In western societies today every living person who has voted in a few elections is probably familiar with physical vote forms. However not every voter is famlier with computers. It is easier for the voter to make their vote using a paper ballot than a voting machine, and in the rare case that a voter has never used a paper ballot it is very natural to pick up as paper forms are used in eveyday life. Under an electronic system physical vote forms could be created for those who are not familiar with computers. If paper ballots are not available for those who dislike voting machines, it could reduce voter turnout.

- Digital Complexity

  Designing computer systems that account for all possible scenarios of security attacks is difficult, if not impossible. Desiging a non computerised system which accounts for every scenario is much more likely to be possible, as non-computerised systems are more natural to comprehend and there are fewer types of attacks possible. Without the use of micro or nano technology almost everyting in a voting booth should be visable to the average voter. It may be possible to hide something, for example a small camera inside a booth but the chances of detection by average voters are much greater than detecting computer attacks.

  Attacks upon digital systems are much more feasible in an election situation. To target multiple polling stations or even multiple booths in a manual system would be difficult without time, manpower and money. If a digital system was to be used, distributed attacks may be possible on multiple voting machines by a single attacker with no or little cost from the safety of their home.

  The problem with digital transmission is the hidden complexity. The levels of possibilities goes deeper and deeper with the attackers imagination and knowledge. The state at which attacks can be made in electronic scheme is blurred compared with a physical scheme. For example even when the candidate column is destroyed in a digital system, it is likely that memory containing a copy of that candidate list may remain somewhere. Depending on the implementation, the attacks which can be made on physical systems may also be able to be made on a digital system.

  Over the last twenty or so years of computing, humans have accepted risks for certain advances to daily life. Examples of these are electronic banking and email which have both been compromised to hurt individuals or groups. As opposed to the damage of an individual's monetary loss, miscommunication or the consequences of stolen or forged communication, election problems could affect large groups, society or the entire world. Knowing this, the risk taken in an election should be minimised as much as possible.

  It is important to consider the level of trust required and balance the benfits of technology with the risks involved. An example of a system where the benefits would outweigh the risks is one which uses automated voting for non-critical matters. The risks may not be of as much of a concern in some systems such as electing a class representative. With advanced micro or nano technologies it may be the case that the problems involved with non computerised systems become as complex as those with computer systems. Advanced development in security techniques may make it possible to identify all potential problems, but until then it would be best to isolate the risks as much as possible when dealing with important systems such as elections. Until risks are reduced significantly, less important systems could be used as pilots to discover potential problems.

5.2.1 STV Prototype Implementation

Although stated that it is more important for elections to reduce risk, this implementation will use digital transactions. There are two reasons for using digital transmissions here; it is easier to implement a system in this case which ignores printing forms and scanning receipts and the risk of attack is nil.

5.3 Onion Representation

Each onion, once created needs to be printed on the vote form with the candidate list it represents. There are two reasons for a representation of the candidate ordering to be placed on the form. Firstly, the tellers know nothing about each form and once the ordering is destroyed they need the onion to recreate the candidate list for matching votes. The second reason is that it is used to identify the vote, the pairing of the encrypted onion with the vote position uniquely identifies the vote.

Using the onion as an identifier requires that every onion be different. There is a very limited chance of getting duplicate onions as the random germs should be drawn from a modest size and they will be encrypted differently though different layers. By increasing the size of germs or increasing the number of tellers, the chance of producing the same vote form is reduced, as the number of possible permutations becomes very large. However, even with a small number of tellers and small germ size, the nonce should prevent two ballots from having the same onion. It is also possible to ensure that two equivalent onions are never created, this could be done by ballot creation authorities remembering some aspect of the onion, i.e. the signature or adding an identification number.

An alternative solution to ensure that onions are different is to change the scheme to use the tellers in a different order for different vote forms as this could also reduce the chance of the same onion being created, as the layers would be encrypted differently. This would change the scheme considerably and will not be considered in this project.

The representation of vote forms needs to be useful, that is:

1. The onion can be read into a format that the tellers can use.
2. The voter can check their receipt easily enough that they actually want to.
3. The voter trusts the representation.
4. Anyone should be able to verify their own vote, no matter their computer skill level or access to a computer.

5.3.1 Physical Representations

Physical representations of the onion are those which a voter can see and touch, they will generally be representations that are transparent to the voter; that is the voter will be able to see the entire representation even if they cannot understand the value contained within it.

In the following discussions, the first required quality of reading the onion for the tellers is ignored, as it would be relatively easy in all cases.

- Textual Onions

  Textual onions are onions represented by written characters. It does not matter which character set is used, except that it all voters should be able to recognize all characters.

  The concern is the ability for the voter to check their receipt. The tellers will be able to deal with long and complicated numbers, but voters will be challanged and annoyed if they have to read long strings of numbers into a computer. Ensuring that strings are as short as possible will make it easier for a voter to remember large chunks. The larger the percentage of characters that a voter can read into a checker at once without re-reading the receipt, the faster or easier it is to check that receipt.

  Chaum suggests a germ range of $0 - 2^{32}$ (32bits), which represented in binary is thirty two characters. Using Chaum's example; if there are three tellers this would result in a total of 192 binary characters (two germs for each teller) without encryption. Encrypting each of the germs with an RSA public key would increase the size even further, as there is a minimum amount of data which can be encrypted. If for example a layer is four bytes, the exact size of plaintext to encrypt must be in 54 byte chunks (with a 512 bit key) and so the remaining 50 bytes is made up in padding. This means that each extra layer will double in size and it will be a multiple of 54 bytes of plaintext and a multiple of 64 bytes of ciphertext.

By representing an onion using hexadecimal, the string can be cut down considerably, a 192 character binary string can be represented with forty eight hexadecimal characters. Using all the letters in the alphabet and the numbers zero to ten, it can be reduced even further; with a base thirty six counting system 9 bytes can be represented with two characters, reducing the 192 binary onion to three characters. With ten tellers, 640 bits would need to be represented, using the base thirty six scheme, it could be represented with 9 characters.

It would be possible to extend this idea by using a base sixty four representation (the alphabet in both upper and lower case). This would make it possible to represent thirteen bytes with two characters.

Although it seems beneficial to ensure that the onion string is as short as possible, it should also be considered whether an onion could be too short. Up until now is has been taken it for granted that it is better to have an onion which is easier to remember as it will be easier to enter. Being able to easily remember onions may in some way or in some implementations have a negative effect on voter privacy, for example someone may be able to memorise an onion as it is passed to someone.

- Barcode Onions

  It is possible to encode the onion as a barcode. There are two concerns with onions represented by barcodes; Will the barcode fit effectively on the receipt yet still contain enough data and how can a voter check their receipt.

  The amount of data which can be stored on a standard barcode is small as each vertical bar can encode one number from zero to nine. However, two dimensional barcodes can store a large amount of data in comparison, some in the thousands of bytes. A list of different two dimensional bardcode schemes are given by [7].

  The problem of enabling voters to check receipts containing onions represented by barcodes. It is very unlikely that a reasonable amount of people would have access to barcode scanners to check their receipts. Even if voters had access to barcode scanners, the hardware and software would need to use the same format which was used to create the barcode onion originaly; probably requiring a download from somewhere. The next problem is what to do with the data once it is read into the computer. If a program was going to be downloaded to scan the barcode, then it could connect to the internet and check for the lodged vote. It would also be possible for the program to turn the onion into text which could then be copied and pasted into a browser for searching, but this is quite complicated.

  The main aspects to consider are scaleability, portability, usability and trust. For example, voters may trust a scheme more where they enter a plaintext onion into a web browser on their own computer (if they trust their computer) than using a program which ”magically” scans a barcode and shows a screen with their receipt on it. At present it would be too expensive to give each voter a barcode scanner, and it would be difficult to create software which worked for everyone, taking into consideration everyones needs. A much simpler and cost effective scheme would to create checking booths. However, this would reduce the usefulness of receipts as voters would be unlikely to check receipts more than once, and possibly not at all if their were queues. The receipts could also only be checked while the booths were in place.

5.3.2 STV Prototype Implementation

The method of onion representation for the prototype be be a base sixty four counting system. This is because it will be the smallest possible representation which would be usable by voters.

Chapter 6

Design and Implementation of STV-VVS

This chapter describes a prototype implementation of a Voter Verifiable Scheme (VVS) for Single Transferable Voting (STV) called STV-VVS. This proof of concept shows how the scheme works. A formal definition of requirements is given in section 6.1 which is taken through to an implementation design in section 6.2. Section 6.3 gives the test strategy used, with recent test results. Section **??** discusses the major issues with implementing this scheme and section **??** outlines possible improvements and future development.

In each section, STV-VVS is discussed as three separate subsystems:

1. Ballot creation is a batch process of creating a specified amount of ballots. This area also includes the problem of transporting ballots to polling stations.
2. Voting is the process of allowing voters to make their votes using a vote machine.
3. Processing votes, involves converting receipts into a form which can be tallied.

Although these three phases are distinctly separate, other implementations may require them to be intertwined, for example if ballots were created in an online manner by the vote machines. Such modifications could be accomplished with minor modifications to STV-VVS.

6.1 Requirements

6.1.1 Functional Requirements

Functional requirements are concerns which are directly related to the core aspects of the system.

Ballot Creation

The following requirements must be met by an implementation of the ballot creation subsystem.

1. Ballot Creation. A form creation authority produces ballots. Each ballot contains a candidate list, an onion, a signature of the candidate list and onion, and an identifier for the authority that can be used to lookup the authority's public key.
2. Ballot Numbering. The form creation authority creates the specified amount of forms in a batch process.

3. Ballot Transportation. Ballots are send to Polling Stations, which can then be used for voting.

4. Ballot Authentication. To authenticate a ballot, authorities use their private key to sign a hash taken of the onion and candidate list.

5. Onion Creation. An onion is created by producing and wrapping random numbers(germs) with tellers' public keys. The onion can be said to contain layers of germs, each layer containing one germ, and the layer encrypted before it or a unique nonce in the case of the core (initial) layer.

6. Encryption. Each layer is encrypted with the public key of the teller who decrypts it.

7. Creation of multiple germs. The number of germs is equal to twice the number of tellers, as each teller must do two transformations on each receipt during vote processing so that half the process can be revealed for auditing without revealing complete paths of permutations through the tellers (which would enable the identification of voters).

8. Permutations. The candidate list is created by iteratively permuting the base candidate list with Knuth shuffles. The input into the shuffle is a sequence of germs which are used in the opposite order to which they were encrypted in.

9. Polling Station Ballot Storage. A polling station keeps unfilled ballots for voters to vote on. When the polling stations receives a ballot, it checks the signature of the creation authority and drops the ballot if invalid and signals an error.

10. Polling Station Ballot Audit. Polling station auditors randomly choose a specified percentage(sample size) of vote forms to audit as they arrive at the polling station. The onion is sent to the tellers, which return the germs. The auditors permute the candidate list using a Knuth shuffle. An error is flagged if the list does not match the vote form.

Voting

The following concerns must be met by an implementation of the voting subsystem.

1. Ballot Choice. The polling station offers two ballots to each voter whom selects one at random to vote on. The other is returned for checking and is audited by supplying the tellers with the onion under oracle mode and then discarded.

2. Vote. The voter uses a vote machine to reorder the candidate list of the chosen form to match their preference for the candidates. The voter informs the vote machine once the choices have been made, commiting their selection.

3. Candidate Column Destruction. Once the vote is committed, each candidate name in the list is replaced with a number corresponding to the position that the candidate held in the list when the voter first received the form.

4. Vote Receipt Authentication. The vote machine signs a signature of the lodged receipt with its private key. The signature is of the onion and a textual representation of the candidate positions. This becomes the voters receipt which is printed for the voter to keep, and an electronic copy is sent to the bulletin board by the polling booth.

Vote Processing

The following requirements must be met by an implementation of the vote processing subsystem.

1. Teller Data Retrieval. Each teller requests a set of onion and candidate-position list pairs from the bulletin board. The first teller is given pairs from polling booth submissions, the remaining tellers use the output from the previous teller.

2. Teller Transformations. Each teller performs two transformations, each transformation peels one layer from the onion revealing a germ and the next onion layer (or the nonce, if the layer is the core layer. The results of the transformations are sent to the bulletin board. The first transformation creates an intermediate list of pairs and the second forms the input for the next teller (or in the case of the final teller produces a list of candidate positions to be matched against the base list.

3. Onion and Candidate-Position List Transformations. Each transformation consists of three steps; firstly the teller decrypts the top layer of the current onion (extracting the next). Next the candidate list is unpermuted with the germ found by recreating the Knuth shuffle and working backwards. Finally, a shuffle is performed on the newly created list of pairs and is posted to the bulletin board along with the current germ.

4. Random Partial Checking. After two transformations are made a teller reveals one path between permutations for each of the pairs posted to the bulletin board. By moving down the intermediate list, a fifty percent chance exists to reveal either the connection between the initial list input and the intermediate or the intermediate and the final output.

5. Voter Lodgement Audit. A voter can request to see a copy of their receipt stored by the bulletin board. The voter is identified by sending their onion with the request.

6. Result Request. Upon request, the bulletin board returns each permutation of candidates which has been voted for. The permutations can then used to work out the percentage of votes as per STV.

7. Drop Repeats. Receipts containing the same nonce (within the onion) is dropped, and a warning issued.

8. Transformation Audit. Auditors check that the teller transformations are performed correctly by checking the revealed connections at the bulliten board. For each connection an empty connection is created and the candidate is placed at the position it is in at the left hand side of the connection. The list is shuffled using the germ as input and the position is matched against the position that it holds in the list on the right side of the connection.

The following parameters are given specific values that are hard coded in this implementation, but in future should be customizable.

- The base candidate list is the original candidate list to be permuted, it consists of five fruit: Durian, Mangosteen, Rambutan, Starfruit and Papaya.
- Ten tellers are used. This is an arbitrary choice, without investigation it seems to be a nice number for distributing secrets.
- The size of the germs is 2^{32}, as suggested in the original design [6].
- The sample size for audits on ballots arriving at polling booths is twenty percent. .
- The representation of a candidate list is the concatenation of candidate names with commas between them. This represents the elements as well as the order.
- Each teller works on eight receipts at one tile. This is the number used in examples from [6].
- Onion Representation. An Onion is displayed to the voter in Base64.
- Signature Representation. Authorities signatures are represented in Base64.

6.1.2 Non-Functional Requirements

Non functional requirements describe system concerns which are not part of the main functionality of the system. They are requirements which constrain how the system works, not what it achieves. This implementation is a proof of concept, so the greater concern is that the system does what is needed. This means that some of the nonfunctional requirements outlined below are not direct goals of the system, but are areas to be measured or analysed. Four types of nonfunctional requirements from [1] are relevant: Usability, Performance, Supportability and Implementation Requirements.

- Usability

  Usability is not considered in this implementation. While very important, possibly affecting voter trust and attitude towards the system, it is a topic which could be the basis for another full paper. Voter trust is the subject of [10].

- Performance

  Only a few forms are generated in STV-VVS for testing, and so generation performance is not critical, however if forms were to be batch generated for an entire country , performance would be important. A performance analysis of ballot creation and processing time is given in section **??**. It is likely that these two processes will be the most intensive, and are also important to test as they were subject to modification for STV-VVS.

- Supportability

  The prototype needs to be easily extensible and should be well documented. As many different voting schemes could be imagined (and have been) it is important that the prototype is quite general. The use of class interfaces gives flexibility, allowing core responsibilities to be declared, yet leave room for extension.

- Implementation Requirements

- STV-VVS is written in Java as the cryptographic tools are accessible and easy to use. Memory management allows concentration on developing the prototye, and simple serialization allows for simulation of data storage.
- RSA Public Key encryption is used for all encryption, but this could be easily interchanged with any other asymmetric encryption type. The encryption mode is electronic codebook mode (ECB), which should be replaced by Cipher-block chaining (CBC) or similar. The importance is that ECB encrypts blocks of data the same way every time, which makes it vulnerable to dictionary attack. A dictionary attack is where the attacker somehow works out what a patten means and then every time that pattern occurs, they know what is being communicated. CBC creates dependence between blocks.
- The Bulletin Board, Tellers, Polling Booths, Vote Machines, Ballots and receipts are stored as serialized flat files. In future, implementations of STV-VVS would use a Database and Key Distribution Center for storage. Serializing private keys is not safe.
- Polling stations manage ballots and Bulletin Boards manage receipts though directories and flat files (the serialized objects).
- There is only one ballot creation authority, one polling station and one voting machine. For the purpose of testing STV-VVS there is no need for any more. In a real poll, multiple creation authorities would allow for different sources of forms if one was corrupted or forms needed to be created quickly. Multiple polling stations and voting machines would be required for a network application.

6.2 Implementation Design

An authority creates ballots, which are used by another or the same authority to collect and tally votes. This section describes the technical details for different components, the relationship between them and the flow of messages through the system.

6.2.1 STV-VVS Process

The system runs through different stages: ballot creation, voting and vote processing. The responsibility of the system management belongs to the poll, which could also be called an election authority. It is the election authorities responsibility to arrange all the aspects of the poll before giving control to the various components. The poll class in STV-VVS first creates or loads the ballot creation authority supplying it with the base candidate list, then the bulletin board and teller chain (controls the group of tellers) with tellers. The polling station is loaded, supplied with voting machines and ballots created. Once the polling station has collected all votes and sent them to the bulletin board, the poll uses the bulletin board to process the votes which can then be tallied. The following list contains system components which are serialized and saved for future use:

- If a ballot authority is not found on disk, the poll creates one, the only input being the identifier to use for that authority. The authority creates a public and private key pair for itself. Once created, the poll serializes the authority so that it can be loaded next time the program runs.

- A teller chain class allows system components to be able to communicate with the tellers. If a teller chain is not found on disk, a new one is created. The teller chain is responsible for setting up tellers; the poll tells the chain how many tellers are required, then the chain creates that many and returns the public keys for those tellers (in order of creation). In STV-VVS the tellers are contained within the teller chain, so the poll serializes the tellers along with the teller chain. In other implementations it may be useful to create a teller link class to control the communication between the teller and other system components. This would make the tellers independant of the communication channels, simplifying the design.

- If a vote machine is not found on disk, the poll creates one, the only input being the identifier to use for that machine. The vote machine creates a public and private key pair for itself.
- When the bulletin board is created it is supplied with an array of teller identifiers (which the poll gets from the teller chain) so that it can set up directories for each. Each teller has a directory assigned to it, containing two subdirectories for each of the transformations . It also creates a directory to store receipts which have been processed.
- A polling station is created by the poll if not found. It requires an identifying number, a vote machine and the bulletin board. Once created, the polling station is serialized. A serialized polling station contains both the vote machine and bulletin board it was created with.

6.2.2 Ballot Creation

When a poll needs ballots, it requests them from the ballot creation authority by calling the method "createBallot". It supplies the authority with the tellers' public keys for onion encryption, the base candidate list for permuting and the required byte size of the germs. The class diagram for this subsystem is given in figure 6.1, a sequence diagram is given in figure 6.2. The ballots purpose is to create a vote form containing a permuted candidate list for voters to vote against and an onion for use by the tellers to unpermute that list during vote processing. A signature is also attached to identify fraudulent ballots.

Ballots each create the same number of germs as there are keys (two for each tellers because of the random partial mixing during vote processing). The base candidate list is copied and permuted iteratively using each germ as the seed, creating a permutation of the candidate list that the voter will vote against. The ballot will then create a nonce; a random number which is unique between onions. A layer is created for every germ, and allocated two layer items before being encrypted. A layer item is a class containing a byte array and a method to access it. Nonce, germ and layer are all sub-classes of layer item. Although these could be represented simply by byte arrays it is logical to think of these entities as individual constructs, and other implementations are not required to use byte arrays. One of the layer items given to a layer will be the next germ which has not been used (but in the opposite order to which they were used for permuting the candidate list), the second is either the onion's nonce if it is the core layer, or the previously encrypted layer.

Once a layer has been created by an onion, the two layer items are joined for encryption. As the germ and previous layer or nonce are byte arrays this simply involves concatenating the layer or nonce and germ. Although the layer can be a variable length, the germ will always be a fixed length and so tellers are able to split them. The concatanated byte array is encrypted using the cryptographic libraries cipher class and given the 512 bit key public key of the teller who will eventually decrypt the layer.

A signature is created from a hash of the final layer (the onions byte array) concatenated with the nonce encrypted at the center of the onion and a byte array representation of the concatenation of candidate names. The hash is signed with the private key of the authority and added to the ballot with the nonce. Once the required number of ballots are created, the poll sends the ballots to a polling station along with the public key of the authority who created them.

If the polling station has received a ballot containing the nonce it discards the ballot, otherwise it uses the public key of the authority to decrypt each of the ballots signatures, performs the same hash that the form authority performed and compares the two. If the two resulting hashes do not match, an exception is thrown and the program exits. The nonce is recorded as having been seen from that teller.

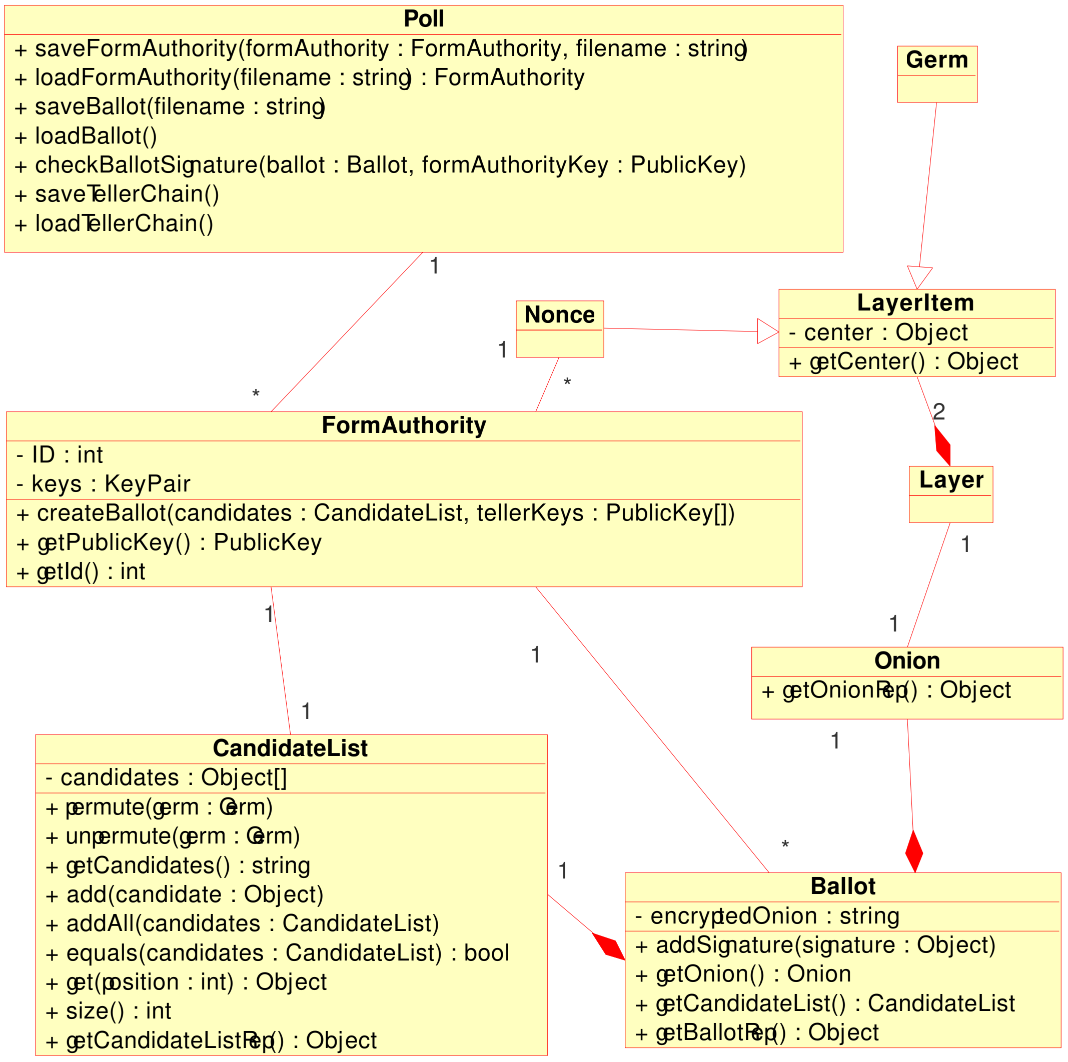

Poll
+ saveFormAuthority(formAuthority : FormAuthority, filename : string)
+ loadFormAuthority(filename : string) : FormAuthority
+ saveBallot(filename : string)
+ loadBallot()
+ checkBallotSignature(ballot : Ballot, formAuthorityKey : PublicKey)
+ saveTellerChain()
+ loadTellerChain()
Germ
Nonce
LayerItem
- center : Object
+ getCenter() : Object
FormAuthority
- ID : int
- keys : KeyPair
+ createBallot(candidates : CandidateList, tellerKeys : PublicKey[])
+ getPublicKey() : PublicKey
+ getId() : int
Layer
Onion
+ getOnionRep() : Object
CandidateList
- candidates : Object[]
+ permute(germ : Germ)
+ unpermute(germ : Germ)
+ getCandidates() : string
+ add(candidate : Object)
+ addAll(candidates : CandidateList)
+ equals(candidates : CandidateList) : bool
+ get(position : int) : Object
+ size() : int
+ getCandidateListRep() : Object
Ballot
- encryptedOnion : string
+ addSignature(signature : Object)
+ getOnion() : Onion
+ getCandidateList() : CandidateList
+ getBallotRep() : Object
1
*
1
*
2
1
1
1
1
1
1
*
1

Figure 6.1: Ballot Construction Class Diagram.

6.2.3 Voting

The polling station controls the voting process, it asks whether a vote is to be made, allows a voter to make their vote, sending it to the bulletin board when completed and then asks whether another vote is to be made. It will continue doing this until it runs out of ballots or told there are no more votes. Figure 6.3 shows the classes involved with voting, and figure 6.4 is the sequence diagram for a vote.

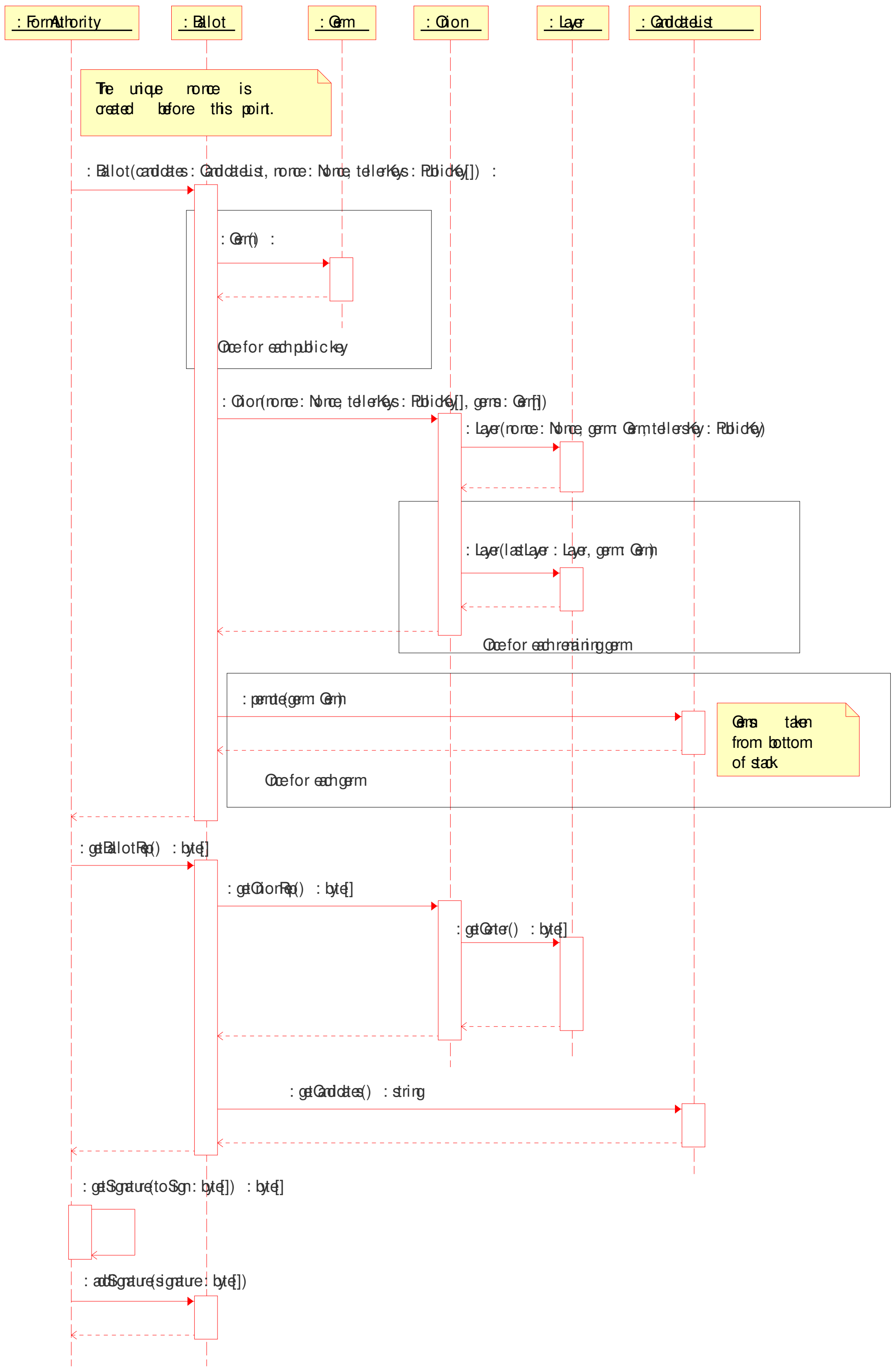

: FormAuthority
: Ballot
: Germ
: Onion
: Layer
: CandidateList
The unique nonce is created before this point.
: Ballot(candidates : CandidateList, nonce : Nonce, tellerKeys : PublicKey[]) :
: Germ() :
Once for each public key
: Onion(nonce : Nonce, tellerKeys : PublicKey[], germs : Germ[])
: Layer(nonce : Nonce, germ : Germ, tellerKey : PublicKey)
: Layer(lastLayer : Layer, germ : Germ)
Once for each remaining germ
: permute(germ : Germ)
Germs taken from bottom of stack
Once for each germ
: getBallotRep() : byte[]
: getOnionRep() : byte[]
: getCenter() : byte[]
: getCandidates() : string
: getSignature(toSign : byte[]) : byte[]
: addSignature(signature : byte[])

Figure 6.2: Ballot Construction Sequence Diagram.

When a polling station is told that a vote is required, it picks two forms and allows the voter to select one. The form not selected is used for auditing by sending the onion to the teller chain which returns a permuted candidate list back. If the candidate list does not match the one on the form, the program throws an exception and quits. Otherwise the vote machine is sent the chosen form for voting.

The STV-VVS vote machine creates a swing JFrame called visual vote form which contains two JPanels; visual vote column and visual candidate column. The way that the vote machine displays the ballot is different from examples previously given [6], it is equivaliant, but slightly easier to use. A screenshot of the vote machine is given in figure 6.5. The candidate list is loaded from the ballot, and the items are copied and placed in the same order inside a ranked list. The list can be manipulated to promote and demote items but retains the original values (the positions are moved up and down the list). The ranked list is placed within the candidate column. The onion is loaded, the base64 string representation of it is found by calling toString on the layer contained within the onion. This string is placed within the visual vote column. A voter can select the rankings for candidates, promoting and demoting as they wish by using the corresponding promote and demote buttons until their desired ranking order is achieved. The voter then casts the vote using the self selected form.

Once confirmation is given, the vote machine goes through the list, finding the item which was originaly at each position. The number corresponding to the position that original item holds in the current list is recorded as the vote for that position. Figure 6.1 gives an example of a receipt being generated from a ranked list. The vote machine then creates a receipt, giving it the candidate list containing the positions generated and the onion and nonce from the original ballot. The vote machine then creates its own nonce, and creates a hash of a byte array representation of the concatenation of the candidate list, onion and two nones. The hash is then signed with the vote machines private key. Each complete receipt is posted to the bulletin board. Upon receiving receipts, the bulletin board places serialized receipts in the directory corresponding to the first tellers first transformation (the first teller which it was informed about when it was set up).

Row	Initial candidate list on ballot	Voter ranking	Receipt Ranking	Movement
1	Mangosteen	Rambutan	5	(1→5)
2	Rambutan	Starfruit	1	(2→1)
3	Durian	Papaya	4	(3→4)
4	Starfruit	Durian	2	(4→2)
5	Papaya	Mangosteen	3	(5→3)

Table 6.1: Visual Receipt Generation.

6.2.4 Vote Processing

Once the vote machine is told that no more votes are to be made, it returns control to the poll, which informs the teller chain to start processing votes. The teller chain iteratively calls tellers to process votes associated to them. Figure 6.6 shows the sequence diagram for a teller making transformations on batches or receipts. A teller will run through two batches of transformations: the first using it's first private key and the second using it's second. In each transformation, the teller will continue to request an array of eight receipts until the number of receipts received is less than eight (because of the random partial mixing). When the bulletin board receives a request for receipts it will include the teller identifier and

Figure 6.3: Vote Class Diagram.

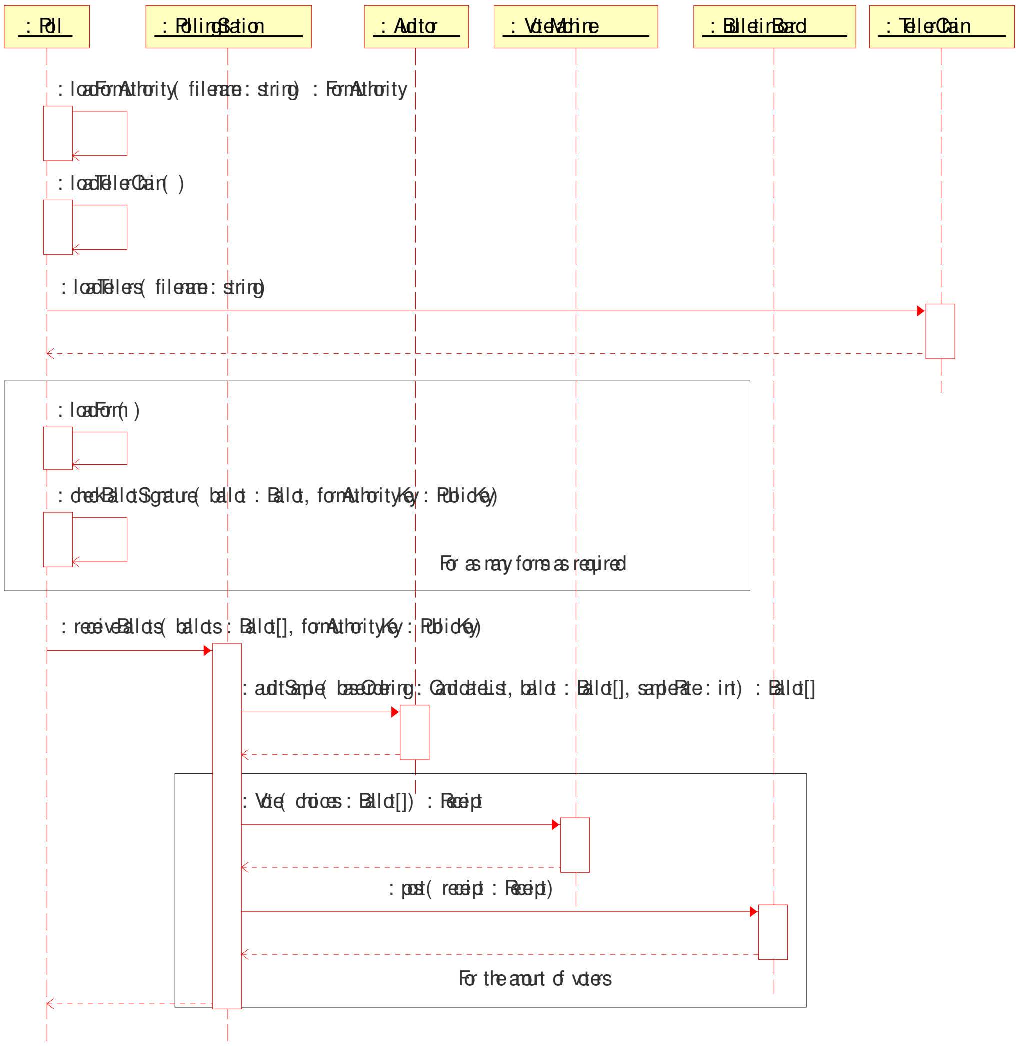
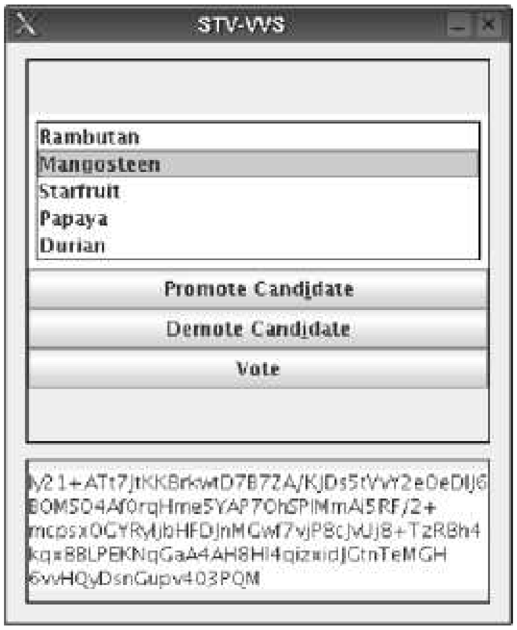

: Poll
: PollingStation
: Auditor
: VoteMachine
: BulletinBoard
: TellerChain
: loadFormAuthority(filename : string) : FormAuthority
: loadTellerChain( )
: loadTellers(filename : string)
: loadForm( )
: checkBallotSignature(ballot : Ballot, formAuthorityKey : PublicKey)
For as many forms as required
: receiveBallots(ballots : Ballot[], formAuthorityKey : PublicKey)
: auditSample(baseOrdering : CandidateList, ballot : Ballot[], sampleRate : int) : Ballot[]
: Vote(choices : Ballot[]) : Receipt
: post(receipt : Receipt)
For the amount of voters

Figure 6.4: Run Vote Machine.



Rambutan
Mangosteen
Starfruit
Papaya
Durian

```
y21+ATt7JtKKBrkwtD7B7ZA/KJDs5tVvY2eDeDIJ6
BOMSO4Af0rqHme5YAP7OhSPIMmAI5RF/2+
mcpsx0CYRyljbHFDJnMCwf7vjP8cJvUJ8+TzRBh4
kq=BBLPEKNqGaA4AH8HI4qiz=idJGtnTeMGH
6vvHQyDsnGupv403PQM
```

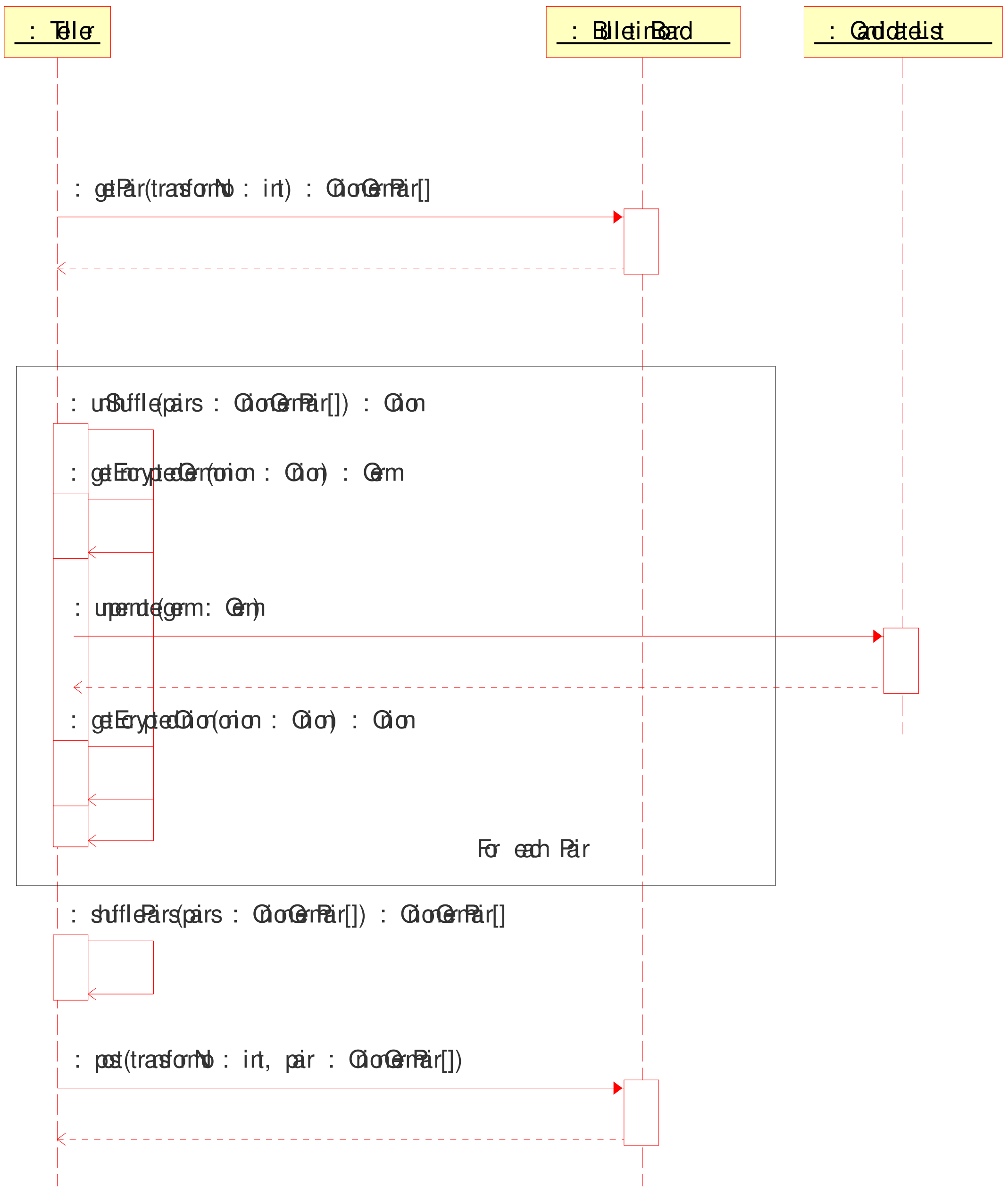
Figure 6.5: A Sample Ballot.

transformation number. The bulletin board then reads up to eight serialized receipts from the directory corresponding to the teller and its transformation e.g. *./receipts/teller0/trans1* for the receipts for teller 0's first transformations, deserializes them to send the teller.

Before the transformation on a set of receipts, a teller will authenticate them. This involves checking the signature of the receipt using the public key of the machine who signed it. If the decrypted hash does not match a hash generated or the nonce has already been seen, the program throws an error and exits. The first step in the transformation is to decrypt each of the receipt layers, retrieved from the onion, using the tellers private key matching the transformation. The first x bytes where x is the encryption strength is the germ and the remaining byte array is the next encrypted layer. The unpermute method of the candidate list is called with the germ as input. Details of the permutation function is given in figure 6.7 (as outlined in 4.5). The teller forms a new receipt containing the new candidate list, the next encrypted layer (or none if there are none left) and posts it to the bulletin board, with the tellers identifier and the current transformation number. When the bulletin board receives a post from a teller it writes serialized receipts to the directory corresponding to the next teller in the list. If the teller is the final teller in the list of tellers, and the transformation is that tellers second, then it will place the serialized receipts in a completed directory.

Receiving less than eight receipts signals to the teller that the bulletin board has run out of receipts for that transformation, and either moves on to the next transformation if the current one is the first or returns control to the teller chain.

Once all the tellers have processed their receipts, the poll requests and then prints the results from the bulletin board. To find the results the bulletin board reads all the receipts in the completed directory and deserializes them. It goes through each of the candidate lists and using a hash map, creates a mapping from candidate list permutations to occurrences. The candidate list permutation is the string representation of the candidate list. Once the different permutations have been tallied, they are printed out with their corresponding number of occurrences. Because the permutations contain rankings not candidates, the results need to be matched with the base candidate list.



: Teller
: BulletinBoard
: CandidateList
: getPair(transformNo : int) : OnionCipherTermPair[]
: unShuffle(pairs : OnionCipherTermPair[]) : Onion
: getEncryptedCipherTerm(onion : Onion) : CipherTerm
: updateTerm(term: CipherTerm)
: getEncryptedOnion(onion : Onion) : Onion
For each Pair
: shufflePairs(pairs : OnionCipherTermPair[]) : OnionCipherTermPair[]
: post(transformNo : int, pair : OnionCipherTermPair[])

Figure 6.6: Process Receipts.

```
Create the index; a list with the numbers 0 to
the number of candidates -1

Shuffle the list using the germ as the random seed

Create a new empty candidate list

For each position in the index

        Copy the item from the receipt's candidate list at
        the current position tothe position in the new array
        at the current positions value.
```

Figure 6.7: Permutation Reversal Pseudo Code

6.2.5 Auditing

The three types of auditing are; voter audits performed during voting (see 6.2.3), Sample audits on ballots arriving at the polling station and checks of teller transformations. The last two have not been implemented and so are explained below:

- When a polling station receives ballots, a certain percentage need to be audited. The polling station needs to be assured that the forms are welformed. Figure 6.8 shows this process. The polling station chooses random ballots from the batch and sends the onions to the teller chain. The chain iteratively passes each layer as it is decrypted to each teller. Each teller decrypts a layer with its first private key, revealing a germ and the next encrypted layer, it then decrypts the next layer with its second private key. The teller returns both the next encrypted layer (or not if it was the final layer) and two germs. The tellerchain is then able to return the set of germs to the polling station in the same order that they were decrypted.

  The polling station creates a new candidate list by permuting the original candidate with each of the germs, in the opposite order to which the teller chain returned them in. If the resulting candidate list does not match the list found of the ballot, an exception is thrown, and the program exits.

- Once vote processing is complete, auditors check teller transformations to ensure that tellers have not performed any corrupt permutations. This is done by requesting that each teller reveal half of the transactions between the first transformation directory's receipts and their second transformation directory and the other half of the second transformation directory's receipts which have not been revealed and the next tellers first directory. For each link between receipts, the teller will reveal the germ used so that the auditor can create the permutation to ensure that the teller has done it correctly. Note that this definition is slightly different from [6] as no information is stored about links.

6.3 Testing

There are two tests detailed below, these are the most important tests that STV-VVS should pass. The first ensures that ballot creation and encryption is being correct managed and the

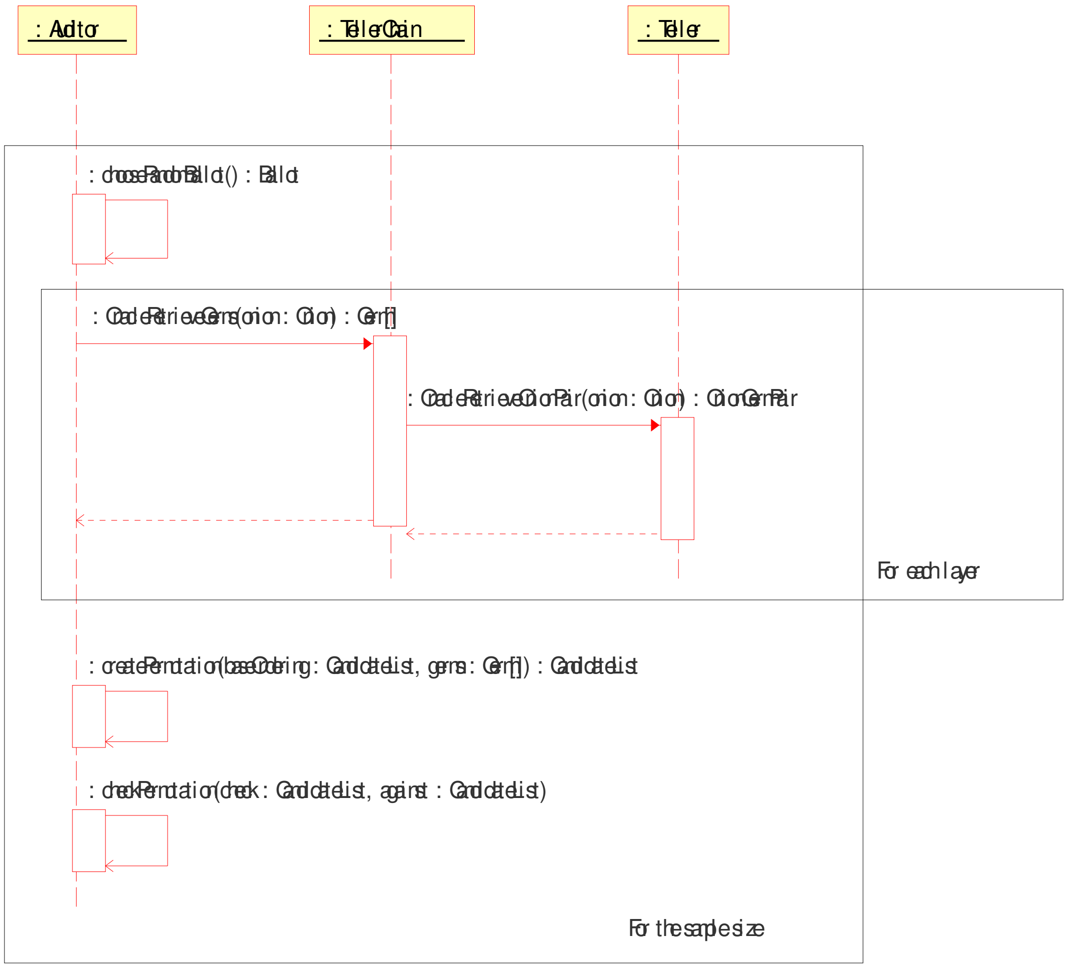

: Auditor
: TellerChain
: Teller
: chooseRandomBallot() : Ballot
: OracleRetrieveGerms(onion : Onion) : Germ[]
: OracleRetrieveOnionPair(onion : Onion) : OnionGermPair
For each layer
: createPermutation(baseOrdering : CandidateList, germs : Germ[]) : CandidateList
: checkPermutation(check : CandidateList, against : CandidateList)
For the sample size

Figure 6.8: Poll Booth Audit.

second proves that permutations are randomly distributed.

- To ensure that ballots are created correctly, a test class creates ten public and private key pairs, and creates a ballot. The ballot is given a candidate list of five items, the ten public keys, a new nonce and an encryption strength of four. Once the ballot has been created, the test class uses the ten private keys in the opposite order to which it gave the ballot and unencrypts the onion in the same way as a teller would.
- Ballot permutations should be randomly distributed. This is measured by checking whether each different permutation of three items (six different permutations) occurred fairly evenly. 100, 1,000, 10,000 and 100,000 ballots are created for this test. Details for the test of 100,000 ballots are given. FormAuthority createBallot 16,735 16,730 16,744 16,536 16,673 16,580

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The objectives in this project were to design modifications to a Voter-Verifiable scheme (VVS) for use with Single Transferable Voting (STV) and to demonstrate that the modifications work using a prototype.

In order to find what modifications were required, a description of [6] has been provided which lays out clearly the different parts of the VVS system. The form authority creates forms which are given to a polling station for candidates to vote on. Each receipt is lodged to the bulletin board to be processed. Each teller takes receipts from the bulletin board and decrypts the receipt using the germ that is hidden from other components.

By analysing the existing VVS [6], it was discussed how an STV VVS would require full permutations to each candidate list instead of simple cyclic shifts. The permutation method needed to be reversable using each germ in the opposite way to which it was created, scaleable keep knowledge of each transformation independant to hide connections between voters and their vote. The new candidate permutation is created by iteratively permuting it with a Knuth shuffle with germs as input. This new permutation type is safe for use with ranked voting for STV.

Before the modifications to [6] for VVS could be implemented in a prototype, some implementation considerations needed to be analysed. A way to identify illegitimate ballots and receipts were identified, that is assigning public and private keys to ballot creation authorities, and vote machines. Hashes can be taken of the components on the ballot or the receipt and signed with the private key of the ballot creation authority or vote machine respectively forming a signature. When a component in the system receives a ballot or receipt it decrypts the signature with the corresponding entities public key and compares it to a hash it makes itself. If they match then the ballot or receipt is legitimate.

The differences between using digital and paper ballots was discussed to determine whether a complete digital scheme was acceptable for a prototype. The disadvantages of using digital ballots were that the hidden complexity in digital systems is greater than standard voting practices, voters may be sceptical of electronic vote machines and some voters would not be familiar with computers. The advantages of digital ballots were found to be: reducing transfer of data between digital and paper was complex, and is not as flexible.

Different onion representations were discussed with the two different types being textual and barcodes. There was a range of different textual representations, the most efficient one was a base sixty four counting system which can represent thirteen bytes with two characters. Standard barcodes were found to be inefficient as they can only represent very small nubers, but two dimensional barcodes can represent thousands of bytes. Unfortunately two dimensional barcodes pose an even greater challange than standard barcodes, as voters would not be able to check their vote using any computer without special devices. By using a base sixty four onion, voters can type it into their computers with a standard keyboard.

Requirements for an implementation called STV-VVS have been developed and explained, which have been turned into a system design. A detailed description is given along with UML diagrams to aid in understanding. It is this implementation that proves an STV VVS system based on [6] is possible.

7.2 Future Work

There are many different fields which the STV VVS scheme touches on requiring investigation. One of the biggest obstacles in replacing paper ballots is voter trust. One problem is explaining the system to others, it is important, as people often do not trust things they do not undestand and trust may affect voter turnout. A series of useability studies would show what onion representations are most popular to voters. The theory that there may be a difference in trust of digital and papper ballots should be investigated, including both the computer literate and illiterate.

STV-VVS needs additional work, to allow for customisation of the paramaters in section 6.1.1.

Futher time needs to be invested into security analysis of the scheme.

7.2.1 Security Analysis

Given below is the beginning of a security analysis based on the work from [13, 16] explained in section 2.4. It is currently at step two of [13] as the attack goals have been identified. The next step is to create the trees of attacks which make each possible. It is important to note that, a perfect tree will never be found in the first instance and they will be working documentation.

A sense of security from voters should be provided by a Voter Verifiable Scheme (VVS) but there is a need to ensure that any implementations of it must withstand stringent security requirements. The main goal will be to first ensure that the election results are correct, that is candidates get enough votes to place them in the positions where they should be (an election can afford to accept some error level, but not enough to pervert the outcome). Secondly, to keep voters choices secret and thirdly, voters checking their vote should see what is correctly recorded. Some formal method is needed to check and prove these three goals.

For this security analysis, attack trees [13] will be used to explore and discuss potential security problems. Section 7.2.1 contains a security analysis on the VVS. Each discussion explains the problem, mitigation strategies including ways to isolate problems, so that they can be detected or recoverable. Section 7.2.1 contains a security analysis on the implementation including considerations of the java cryptographic libraries and virtual machine.

Analysing the Scheme

The approach take is to assume that Chaum's original scheme is perfect, analysing potential attacks on the modifications will help identify and allow discussion of any weaknesses have been introduced by the modifications made in chapters four and five. For ease of analysis the system is broken into three areas; cryptographic components, processing/distribution components and human factors. Note that some of the same high-level goals will be present amongst multiple parts, which can be added together to form bigger trees.

Security Goal	Attack Goal
1	Lodge an illegitimate voteform.
1	Lodge a vote twice.
2	Remove an already lodged vote.
2	Change an already lodged vote
3	Add a vote to the counting process.
3	Remove a vote from the counting process.
3	Force a vote to be counted incorrectly.
4	Discover what went on inside the vote booth.
4	View a votecolumn with its candidate column which is used for voting and be able to find who owns the receipt.
5	View a votecolumn with its candidate column which is used for voting.

Table 7.1: Attack Goals

Identifying Security and Attack Goals

The security goals are derived from the requirements already stated; correctness, privacy and verifiability. If the following sub requirements hold, then this should be possible:

1. Votes should be lodged correctly.
2. Votes should be stored correctly.
3. Votes should be counted correctly.
4. Votes should not be able to be matched with their vote, even given their receipt.
5. Voters should be able to match their receipt to the corresponding vote in the system.
6. Requesting the outcome of the poll should produce the correct results.
7. Audits should produce correct results to all parties.

These become the top level goals of the security tree and the attack goals for the second level are given in table **??** with the corresponding security goal each was derived from.

Bibliography

[1] BRUEGGE, B., AND DUTOIT, A. H. *Object-Oriented Software Engineering*. Pearson Education Inc., 2004.

[2] BRYANS, J., AND RYAN, P. A dependability analysis of the chaum digital voting scheme. Tech. Rep. CS-TR-809, Newcastle University of Computing Science, 2003.

[3] BRYANS, J., AND RYAN, P. A dependability analysis of the chaum digital voting scheme. Tech. Rep. CS-TR-809, Newcastle University of Computing Science, 2003.

[4] CATT, H., HARRIS, P., HARRIS, N., AND ROBERTS, N. S. *Voters Choice*, first ed. The Dunmore Press Limited, Palmeston North New Zealand, 1992.

[5] CHAUM, D. Secret-ballot receipts and transparent integrity. Tech. rep., 2003.

[6] CHAUM, D., RYAN, P., AND SCHNEIDER, S. A. A practical, voter-verifiable election scheme. Tech. Rep. CS-TR-880, Newcastle University of Computing Science, 2004.

[7] COMMUNICATIONS, A. 2-dimensional bar code page. `http://www.adams1.com/pub/russadam/stack.html`, viewed 20/09/2005.

[8] KNUTH, D. E. *The Art of Computer Programming:Seminumerical Algorithms*, vol. 2. Addison-wesley Publishing Company, 1981.

[9] LARSEN, L. R. Voting technology implementation.

[10] RANDELL, B., AND RYAN, P. Voting technologies and trust. Tech. Rep. CS-TR-911, Newcastle University of Computing Science, 2005.

[11] RYAN, P. A variant of the chaum voter-verifiable scheme. Tech. Rep. CS-TR-864, Newcastle University of Computing Science, 2004.

[12] RYAN, P., AND BRYANS, J. A simplified version of the chaum voting scheme. Tech. Rep. CS-TR-843, Newcastle University of Computing Science, 2004.

[13] SCHNEIER, B. Attack trees. Tech. rep., December 1999. `http://www.schneier.com/paper-attacktrees-ddj-ft.html`, viewed 03/10/05.

[14] SIMONS, B. Electronic voting systems: the good, the bad, and the stupid.

[15] SOCIETY, E. R. Voting systems: Multi-member systems. http://www.electoral-reform.org.uk, 2005. `http://www.electoral-reform.org.uk/votingsystems/systems3.htm`, last viewed 18/10/05.

[16] STROUD, R., WELCH, I., WARNE, J., AND RYAN, P. A qualitative analysis of the intrusion-tolerance capabilities of the maftia architecture. In *International Conference on Dependable Systems and Networks (DSN)* (June 2004), pp. 453–465.

[17] VARIOUS. *Taking it to the People - The New Zealand Electoral Referendum Debate*. Hazard Press Limited, 1993.

[18] VERIFIEDVOTING.ORG. Verified voting sweeps the states! http://www.verifiedvoting.org/, 2005. `http://www.verifiedvoting.org/`, last viewed 21/10/05.