THESIS

IMPLEMENTATION OF A SECURE AND ANONYMOUS ELECTRONIC VOTING PROTOCOL

Submitted by

Burke Webster

Computer Science Department

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2008

COLORADO STATE UNIVERSITY

July 2, 2008

WE HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER OUR SUPERVISION BY BURKE WEBSTER ENTITLED IMPLEMENTATION OF A SECURE AND ANONYMOUS ELECTRONIC VOTING PROTOCOL BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE.

Committee on Graduate work

Dr. Indrajit Ray, Thesis Adviser

Dr. Indrakshi Ray, Inside Committee Member

Dr. Leo Vijayasarathy, Outside Committee Member

Dr. L. Darrell Whitley, Department Chair

ABSTRACT OF THESIS

IMPLEMENTATION OF A SECURE AND ANONYMOUS ELECTRONIC VOTING PROTOCOL

Our current world is becoming more and more dependent on widespread, electronic communication and functionality. We expect to perform banking activities from home, renew our car insurance through an easy to use website, and submit important business proposals via e-mail. Yet when it comes to voting, an important aspect of our democratic society, we follow outdated and cumbersome procedures. In this age of technology, we strive to streamline this process but are met with many obstacles. Some are technical in nature, while others are socially complex and politically charged. The main issues we face when looking to transition this process into its digital equivalent are many. How do we identify ourselves in a digital world? How do we cast out vote without divulging our identity? How do we ensure the secrecy and accuracy of our vote once cast? Finally, how do we trust the digital presence of the voting authority with which we never really see?

In this paper, we will target these issues through the implementation and analysis of the eVoting system. This system, based on the work by Ray et al. [39], provides an elegant solution whereby voters can cast secure votes without the need to trust the voting authorities of the system. The voting protocol provides for voter privacy and anonymity through a mixture of cryptography and anonymous networks. The system enforces democratic constraints required in any election, ensuring that only registered voters may vote. The system protects voters that wish to not vote, by providing a level of accuracy beyond that which current systems offer. A voter can easily verify his vote was properly cast and recorded, and furthermore the voter, or any other entity, can easily verify the final results of the election.

We feel that a working prototype of such a strong and secure system is the first step in advancing towards an end goal of full electronic voting. Thus, in this paper we will present the eVoting system, an implementation of a secure and anonymous electronic voting protocol.

Burke Webster

Computer Science Department

Colorado State University

Fort Collins, CO 80523

Summer 2008

Acknowledgements

I would like to thank my family and closest friends for their never-ending support and encouragement throughout the years.

Table of Contents

Table of Contents v

List of Tables vii

List of Figures viii

1. Introduction 1

2. Related Work 3

 2.1 Anonymous Channels 3

  2.1.1 Trusted Relays 3

  2.1.2 Mix Systems 5

 2.2 Electronic Voting 6

  2.2.1 Complex Cryptographic Protocols 6

  2.2.2 Anonymous Channel Protocols 7

  2.2.3 Electronic Voting Systems 9

3. The Voting Protocol 11

 3.1 Necessary Conditions for Electronic Voting 11

 3.2 Participating Entities 11

  3.2.1 Certificate Authority 12

  3.2.2 Election Manager Authority 13

  3.2.3 Ballot Distributor Authority 13

  3.2.4 Voter Attesting Authority 14

  3.2.5 Vote Counter Authority 14

  3.2.6 Voting Client 15

 3.3 Communication Channels 15

4. Implementation 17

 4.1 Messages 17

  4.1.1 General Messages 18

  4.1.2 Certificate Authority Messages 19

  4.1.3 Election Manager Messages 22

  4.1.4 Ballot Distributor Messages 23

  4.1.5 Voter Attesting Authority Messages 25

  4.1.6 Vote Counter Messages 26

4.2 Receiving and Sending Messages ... 27
4.2.1 Communication Protocol ... 28
4.2.2 Implementing the Communication Protocol ... 28
4.3 Ballot Description Language ... 30
4.4 Authoritative Entity Long-term Storage ... 31
4.4.1 Schemas ... 31
4.5 Anonymous Channel ... 34
4.6 Blind Signatures ... 36
4.6.1 Implementing the Blind Signatures ... 36
4.6.2 Key and Certificate Management ... 38
5. Analysis ... 41
5.1 In the Absence of Fraud ... 41
5.2 In the Presence of Fraud ... 43
5.2.1 Collusion involving BD and VAA ... 44
5.2.2 Collusion involving BD and VC ... 45
5.2.3 Collusion involving VAA and VC ... 45
5.2.4 Collusion involving BD, VAA, and VC ... 45
5.2.5 Collusion with the involvement of EM ... 46
5.2.6 Collusion with the involvement of voters ... 46
6. Future Work ... 47
6.1 Graphical User Interfaces ... 47
6.2 Anonymous Channel Integration ... 47
6.3 Distributed Architecture ... 48
6.4 Certificate Revocation Lists ... 48
6.5 Extending BDL ... 48
7. Conclusion ... 49
Bibliography ... 51

List of Tables

4.1 External libraries and dependencies used in the eVoting system. 17

4.2 Valid XML messages. 18

4.3 BDClaimedSerial table definition. 32

4.4 VAACertifiedVotes table definition. 32

4.5 VCCastVotes table definition. 33

4.6 Ballot001 table definition. 34

List of Figures

3.1 The voting protocol. 12

4.1 VotingProtocolError DTD. 18

4.2 VotingProtocolError example. 19

4.3 GetCertificate DTD. 19

4.4 GetCertificateResponse DTD. 20

4.5 ApproveCertificate DTD. 21

4.6 ApproveCertificateResponse DTD. 21

4.7 ApproveCertificateAlreadyReceivedResponse DTD. 22

4.8 GetOpenBallots DTD. 22

4.9 GetOpenBallotsResponse DTD. 23

4.10 GetBallot DTD. 23

4.11 GetOpenBallotsResponse DTD. 24

4.12 RegisterVoter DTD. 24

4.13 RegisterVoterResponse DTD. 25

4.14 CertifyVoter DTD. 26

4.15 CertifyVoterResponse DTD. 26

4.16 CastBallot DTD. 27

4.17 CastBallotResponse DTD. 27

4.18 The various servlet classes, used to implement the authoritative entities. 29

4.19 The various client classes, used to communicate with the authoritative entities. 29

4.20 Ballot Description Language DTD. 30

4.21 A sample ballot, described in the Ballot Description Language. 31

4.22 DDL statement to add unique constraint. 32

4.23 Example overlay mix netword used by TOR. 34

4.24 Connection setup through the network from source to destination. 35

4.25 Applying the blind signature to a ballot. 36

4.26 BlindSignature.java source code sample. 37

4.27 Creating the voters keystore. 38

4.28 Creating voters certificate signing request (CSR). 39

4.29 Generating CA's self-signed certificate. 39

4.30 Importing CA's RSA keys. 40

4.31 Signing a CSR. 40

1. Introduction

Our current world is becoming more and more dependent on widespread, electronic communication and functionality. We expect to perform banking activities from home, renew our car insurance through an easy to use website, and submit important business proposals via e-mail. Yet when it comes to voting, an important aspect of our democratic society, we follow outdated and cumbersome procedures. We must first register to vote, typically in person, by standing in queues at local government facilities. We must then drive or walk to our predetermined voting precinct and stand in yet another line. After providing proof of our identity, we then wait in another line before finally entering a curtain-clad booth. Once there, in the privacy of a confined space, we read through line after line of ballots, casting out vote and hoping the machine accurately counts our vote. Out of social courtesy, we try to expedite the process as to not hold up the growing lines behind us. We then leave the precinct and wait to see vote tallies, all the while wondering if our vote was handled correctly and counted in the final results.

The above situation seems a bit tedious, yet we perform it regularly to comply with our civic duties. In this age of technology, we strive to streamline this process but are met with many obstacles. Some are technical in nature, while others are socially complex and politically charged. The main issues we face when looking to transition this process into its digital equivalent are many. How do we identify ourselves in a digital world? How do we cast out vote without divulging our identity? How do we ensure the secrecy and accuracy of our vote once cast? Finally, how do we trust the digital presence of the voting authority with which we never really see?

In this paper, we will target these issues through the implementation and analysis of the eVoting system. This system, based on the work by Ray et al. [39], provides an elegant solution whereby voters can cast secure votes without the need to trust the voting authorities of the system. The voting protocol provides for voter privacy and anonymity through a mixture of cryptography and anonymous networks. The system enforces democratic constraints required in any election, ensuring that only registered voters may vote. The system protects voters that wish to not vote, by providing a level of accuracy beyond that which current systems offer. A voter can easily verify his vote was properly cast and recorded, and furthermore the voter, or any other entity, can easily verify the final results of the election.

We feel that a working prototype of such a strong and secure system is the first step in advancing

towards an end goal of full electronic voting. Thus, in this paper we will present the eVoting system, an implementation of a secure and anonymous electronic voting protocol. Section 2 will provide an overview of other systems and protocols that address the issues in electronic voting. Section 3 will present the voting protocol and its design goals and requirements. Section 4 will discuss the transformation of the design goals and requirements into a working system. We will then present an analysis of both the voting protocol as well as the eVoting system in Section 5. Finally, Section 6 will present the topics for further exploration and Section 7 will provide a summary of the work presented here.

2. Related Work

In this section we will present related work. Section 2.1 will begin by discussing the development of various anonymous channels. Section 2.2 will then discuss the various electronic voting protocols and systems that have been proposed and implemented over the years.

2.1 Anonymous Channels

Anonymous channels have been around for quite some time, and have been leveraged in various situations to help reduce the chance that a users communications or actions can be linked back to them. There are various key ideas that all factor into anonymous communications. First, we can define anonymity as the state of being not identifiable within a set of subjects [16]. When dealing with anonymity, it is also important to consider unlinkability, unobservability, and psuedonymity. Unlinkability states that a user should be able to make multiple requests via the anonymous channel without observers linking these uses together. Unobservability states that messages generated by or for the user within the system should not be discernible from random noise within the system. This is especially important in mixed networks. Finally, we have pseudonymity, which is the use of pseudonyms within a system as a means of identification.

Anonymous channels are typically implemented using trusted and semi-trusted relays or mix systems. We will present examples of trusted relays in Section 2.1.1, and examples of mix systems in 2.1.2.

2.1.1 Trusted Relays

Trusted relay systems are anonymous communication systems that rely on some trusted or semi-trusted central node through which messages are annonymized. These systems typically provide low levels or security and are susceptible to black box attacks against anonymity.

Johan Helsingius provided an anonymous mailing system at anon.penet.fi in 1993 [16] that used a roster of pseudonyms for sending and receiving messages. The system would accept mail from registered users, strip the messages of any identifying marks, apply a pseudonym and send the message out. Responses to the message would be sent back to the relay using the pseudonym, whereby the system would map the pseudonym back to a real address and forward the message

along. Due to legal issues, the service was shut down in 1996.

Hughes and Finney introduced Type I remailers in 1996 [38]. In this scheme, the user would send mail to the remailer after encrypting it using Pretty Good Privacy (PGP). The remailer would decrypt the message with its private key, strip all identifying information, and forward the message along, possibly through a string of participating remailers. The advantage to this scheme is that the remailer didn't need to keep a mapping of pseudonyms to real addresses for routing of return electronic mail. Instead, the remailer would construct a reply block and include it with the outgoing mail. The reply block consisted of a set of encrypted attributes used by the remailer to forward responses to the originator. While the reply block solved one issue, it introduced other insecurities that could undermine the anonymity of the user. The reply block provided a means of easily identifying a message through traffic analysis since the reply block was identical for all messages originating from a single user. Later systems, such as the Type III remailer attempted to address this issue by using one-time use reply blocks.

Various anonymous web proxies have been implemented that offer differing degrees of anonymity. Anonymizer and SafeWeb are two examples of web proxies that attempt to provide anonymous web browsing to its users. Both provide a trusted proxy that all web browsing traffic is directed through. The proxy strips the traffic of any identifying characteristics and hides the originating IP address. The proxy also examines included and embedded scripting resources, such as JavaScript files, in an attempt to strip any code that might be able to determine a users identity and transmit it back to the originating site. SafeWeb provides an additional benefit in that it offers an SSL-enabled proxy to encrypt traffic from the user to the proxy. These services suffer from various issues, the largest being that simple traffic analysis can be used to link incoming requests with outgoing messages.

To address the single point of observation issue with basic anonymizing proxies, Reiter and Rubin proposed an approach that used multiple proxies, called Crowds [41]. A user wishing to browse websites anonymously first contacts a central server to retrieve a list of participating nodes - the crowd. The user then directs her request at a random member of the crowd. When the selected crowd member receives the request, it randomly choses to either forward the request to another member of the crowd, or send the request to the destination web site. The strength in this scheme is that traffic analysis is much more difficult. An adversary must observe a significant portion of the crowd to successfully infer any information.

Katti et al. [31] later proposed a similarly distributed scheme whereby the user splits their message into a number of pieces, and sends each piece to an intermediary. The intermediary then

reassembles the message and forwards it on. The strength of this system is that an adversary must observe or control all of the channels used to transmit each piece of the message, otherwise they will be unable to reassemble the full message.

2.1.2 Mix Systems

The Type I remailer presented in 2.1.1 forms the basis of an entire group of anonymous channels, called mix systems or mix networks. While the Type I remailer is an insecure version, it lays the foundation whereby a set of nodes are used to "bounce" messages around an overlay network, with a random exit node finally forwarding the message on to its destination. The strength of this approach lies in the concept of initiator anonymity, whereby a node receiving a message does not know if the message originated, or was simply forwarded, from the previous node.

The original pioneering work for mix systems was provided by Chaum in [7], where he introduced the concept of a mix node that lowers the linkability of incoming and outgoing messages. In this scheme, messages where padded with random bits, and a header was attached that contained the address of the next mix node to forward the message to. The message was then block encoded using RSA public key encryption, and sent to the first mix node. The goal of the system was to achieve unlinkability, whereby an adversary could not identify a message as it traveled through the mix network. To increase the unlinkability and unobservability, dummy messages are used to increase the amount of traffic within the mix network. Thus, an observer cannot determine if a given message is an actual message originating from a user, or a dummy message created by the mix nodes.

Gülcü and Tsudik [28] and Möller et al. [35] introduced Babel and Mixmaster respectively as a mix network approach to remailing. Babel offers both sender and receiver anonymity, whereas Mixmaster only provides sender anonymity. These systems use a layered encryption approach, wrapping an outgoing message in layers of encryption, each of which is removed by the next mix node in the path. These systems make use of various encryption techniques, and apply padding schemes to help increase unlinkability. Danezis et al. [15] later introduced Mixminion, a very similar remailer that offered both sender and receiver anonymity, and bi-directional anonymity. The major contribution of the Mixminion scheme is the reduction of tagging-based attacks. Messages are split into a two-part header and a body. The header contains the addresses of the mix nodes used to transmit the message, and applies special encryption techniques to destroy messages that have been tagged by an adversary or dishonest mix node.

A circuit-based approach to mix networks was introduced through the idea of onion routing,

proposed by Goldschlag et al. [27], Reed et al. [40], Goldshlag et al. [26], and Syverson et al. [43]. In this scheme, a circuit is first established, and messages are then forwarded to their destination via the established route. Goldberg [25] later introduced the Freedom network, a follow-on to the onion routing network, and Dingledine et al. [18] proposed TOR, a second generation onion routing anonymous network. In TOR, a list of participating servers is used to select a subset of nodes to participate in a circuit. An iterative approach is used at establishing bi-directional channels between participating nodes. Additionally, TOR offers hidden services through the mix network. These services must be advertised or known a-priori, and can be accessed through the establishment of an anonymous channel within the TOR network. We will chose to implement the TOR network as part of the eVoting system to provide a means of anonymously casting our filled voting.

2.2 Electronic Voting

A number of electronic voting protocols have been proposed and implemented over the last seventeen years. These protocols typically are aimed at solving the necessary conditions for electronic voting, given in Section 3. Each utilizes various techniques to facilitate electronic voting, and typically base the security of the system on either complex cryptographic protocols or anonymous channels. We will present a discussion of protocols of both form, as well as other electronic voting systems in the following sections.

2.2.1 Complex Cryptographic Protocols

DeMillot and Merritt [17] propose a scheme where a voter's vote is encrypted multiple times using a pair of public keys from all other voters. The voting protocol requires the interaction of all voters in order to succeed. Thus, if a voter abstains from the protocol, or decides to not co-operate, the protocol must be restarted.

Cohen and Fischer [9], Benaloh and Young [1], and Cramer et al. [12], [13] all proposed very similar protocols that made use of zero-knowledge techniques to provided anonymity to the voter. The protocols in [9] and [1] are based on higher degree residue encryption, while [12] uses discrete logarithm encryption. The protocol in [9] provides anonymity based on the assumption that the election administrator is trusted. The protocol in [1] improves on the protocol in [9] by spreading the cast ballot over a number of different tallying authorities, thus offering more privacy to the voter. The approach given in [12] is slightly different, in that the voter sends a verifiable piece of

their ballot to a number of vote compiling authorities. These authorities then feed the information to a central authority, which assembles the ballot and tallies the results. The strength in this scheme is the anonymity provided to the voter by the splitting and reassembly of their vote. However, all of these systems suffer from high computational overhead. The scheme given in [12] also suffers from high communication overhead, and restricts the voter to a yes/no answer to a fixed question. The work given in [13] improves upon some of these limitations by optimizing the computational overhead associated with the protocol.

A scheme based on probabilistic privacy homomorphisms [29] is given by Iversen in [30]. This scheme relies on communication between the voters and the candidates being voted for. The scheme cannot be disrupted by a voter, or subset of voters, and preserves the privacy of voters against corrupt or dishonest voting authorities and candidates. However, much like the other protocols presented in this section, these schemes suffer from high computation and communication overhead. Additionally, we find that this scheme is not suitable for electronic voting due to the required participation of all the candidates.

Our protocol, however, does not suffer from the issues presented above. No voter, or subset of voters, can disrupt the voting process. Additionally, our protocol does not require any computationally intensive or non-widely available cryptographic techniques. Finally, our protocol has very little communication overhead.

2.2.2 Anonymous Channel Protocols

Chaum first proposed an electronic voting protocol in 1981 that made use of public key cryptography, rosters of digital pseudonyms, and anonymous channels [7]. In the proposed scheme, a mix-net was used to conceal the voter's identity. However, the full anonymity of the voter could not be guaranteed due to the necessity of trusting at least one of the mix agents in the mix-net. Chaum later proposed a scheme that provided full anonymity to the voter in [5]. This approach used a dc-net approach, but suffered from that fact that a single voter could disrupt the voting process.

Boyd proposed a protocol in [2] that used a multiple key encipherment scheme, and an anonymous channel to cast the final vote. While this scheme provides privacy and some levels of accuracy, a dubious administrator could affect the results of the election by submitting forged votes.

Two protocols proposed by Nurmi et al. in [37] - the Two Agency Protocol and the One Agency Protocol - split the voting process up between a validator and a tallier. The validator ensures the voters are eligible registered voters, while the tallier is responsible for collecting the cast votes and

computing the results. This approach is proposed to help provide anonymity to the voter. In the Two Agency Protocol, tags are issued to voters by the validator, and used by the voter when casting their final vote. However this approach suffers from two major issues. First, if the tallier and validator collude, they can link a cast vote back to the individual voter. Secondly, the tallier and validator can collude and cast spurious votes in the place of voters that have not participated in the voting process. The collusion issues are addressed in the One Agency Protocol by eliminating the validator. The tallier is thus responsible for all aspects of the voting protocol, and uses an all-or-nothing disclosure of secrets protocol when disbursing tags. While this solves the anonymity concerns, it does nothing to address the fact that the tallier can cast spurious votes in the place of non-participating voters.

Chaum first introduced the idea of blind signatures in [4] as a method for implementing an anonymous electronic payment system, with proposed applications in conducting secret ballot elections. Nearly ten years later, Fujioka et al. introduced a voting protocol that used blind signatures [24]. Anonymous channels were used, combined with blind signatures, to solve the collusion issues seen with other similar protocols, such as the Two Agency Protocol. In this protocol, the voter encrypts their ballot and applies a blind signature before signing it and sending it to the validator. The validator verifies the voter's signature and ensures the voter has not already submitted a prior vote. The validator then signs the encrypted, blinded ballot and returns it to the voter. The voter then removes the blinding factor, revealing an encrypted filled vote, which has been signed by the verifier. The voter then submits this to the tallier via an anonymous channel. The tallier verifies the vote, and then places it on a list that is published at the end of the election. Voters can then verify their vote is on the list, and send in their keys so that the tallier may decrypt and tally the votes. While this protocol provides many of the necessary qualities for a secure electronic voting system, there are a few major issues. One issue is that the voter must be involved in the voting protocol throughout the entire process, submitting their decryption keys after the end of the voting phase. Additionally, voters cannot abstain from voting as this allows spurious votes to be cast through collusion of the tallier and validator.

Cranor and Cytron later proposed the SENSUS protocol in [14] that overcame the two major issues in [24]. The voter can submit his decryption key immediately after notification from the tallier that his cast vote was received. This alleviates the need for the voter to participate in the scheme until the end of the voting period. The voter could also specifically abstain from voting by indicating his decision in the cast ballot. While this solved part of the issue, a voter that chooses to abstain

but doesn't cast a vote indicating so leaves the door open for a colluding tallier and validator to cast spurious votes.

Our protocol is very similar to the protocol proposed by Fujioka et al. [24] and the SENSUS protocol. The advantage of our protocol is that no authoritative entity within the voting system can take advantage of voters that have chosen to abstain from voting and not specified so through a cast ballot.

2.2.3 Electronic Voting Systems

There are a number of electronic voting systems that have been proposed and are currently in use. These are typically machines that run on specific hardware and implement voting-specific software. Roughly 39% or voters participating in national election in the United States of America in 2006 cast their votes using electronic systems [42]. Direct-recording electronic (DRE) voting systems are the most popular. These systems display a ballot to the voter and receive instructions from the voter via button or touch-screen feedback. The results are processed by the software, and stored, typically in encrypted format, in memory components. At the end of the election, a hard copy printout is generated showing the election results and voting summaries.

Diebold Election Systems Inc. produces the DRE-based Diebold AccuVote-TS and AccuVote-TSx systems, used by nearly 10% of registered voters in the 2006 elections [19]. Despite the widespread use of these systems, Kohno et al. [32] revealed several vulnerabilities whereby voters could vote more than once and perform administrative actions. They also indicate a lack or cryptographic techniques used in the software, as well as improper uses in the cases where it was applied. Finally, they note a strong lack of artifacts indicating a well-developed and mature software engineering process. These results were confirmed and expanded upon by the State of Maryland and SIAC [11] and RABA [3], and the State of Ohio and Compuware [10].

More recently, Feldman et al. [19] have also shown that the Diebold systems are vulnerable to both vote-stealing attacks and denial-of-service attacks. In the vote-stealing attacks, an adversary can inject code into the voting machine that will alter the cast votes without leaving any trace of the fraudulent activity. The denial-of-service attacks are detectable, and in most cases are aimed at disrupting or preventing the voting process. For example, a malicious voting machine could be set to turn off at a preset time during the election, or erase all votes and audit logs at the end of the election.

The electronic Voting and Counting System (eVACS), developed by Software Improvements, is

an electronic voting system that has successfully run multiple elections in Australia. The eVACS system allows for both electronic and paper ballots, and is unique from many of the DRE systems in that its source code is open source and publicaly available.

David Chaum announced the Punchscan system in 2006, which uses a mixture of computer systems as well as more traditional paper-based ballots [8]. In this system, a paper ballot is given to voters to mark using an ink-blotting pen. The voter must fold the paper ballot in half, thereby revealing the full ballot and allowing the voter to cast his vote. The foldable ballot is a direct implementation of visual cryptography, proposed by Naor and Shamir in [36] and later applied by Chaum in [6]. The ballot is then torn in half, resulting in two ballot pieces, neither of which can be used on its own to determine how the voter voted. The voter selects either half of the ballot to be scanned and cast, and shreds the other half. The voter is also given a receipt which can be used to verify the vote was successfully recorded, but which does not contain the actual vote. The system has proven successful in various small-scale elections, but has yet to be used in large-scale elections.

3. The Voting Protocol

3.1 Necessary Conditions for Electronic Voting

A set of necessary conditions has been identified by various researchers [39],[14] that need to be satisfied by any electronic voting system.

1. **Accuracy** - A system is accurate if (1) a vote cannot be modified without detection, (2) valid votes cannot be removed or miscounted in the final tally, and (3) invalid votes are not counted in the final tally.

2. **Democracy** - A system is democratic if (1) only eligible voters may cast votes, and (2) each eligible voter can only cast one vote.

3. **Privacy** - A system is private if it is not possible for any entity involved in the voting process to link a voter to his or her cast vote.

4. **Verifiability** - A system is verifiable if a voter can independently verify that his or her vote has been counted correctly.

5. **Un-authorized Proxy** - A system is secure against un-authorized proxies if no entity involved in the protocol can involuntarily cast a forged vote for a voter wishing to not cast his or her ballot.

Ideally, these conditions should also be satisfied by the pre-existing paper-based voting systems. However, we find that in many cases these conditions are not met, and thus we find that electronic voting systems (that do satisfy these conditions) provide a higher degree of confidence to the voter that his vote has been correctly counted and anonymity maintained.

3.2 Participating Entities

The voting protocol implemented by the eVoting system is an extension of the protocol proposed in [39]. The original protocol made use of three authoritative entities to facilitate the voting process. These entities were responsible for distributing ballots, certifying filled and blinded ballots, and receiving and tallying cast ballots. We will maintain the functionality and scope of these proposed entities, and add two more authoritative entities into the protocol. These entities will be responsible

for managing and distributing certificates, and managing and distributing ballot information. Thus, the eVoting system, as shown in Figure 3.1, will leverage the functionalities provided by the Certificate Authority, Election Manager, Ballot Distributor, Voter Attesting Authority, and Vote Counter authoritative entities, as well as the implicit functionality of the voting client (used by the voter).

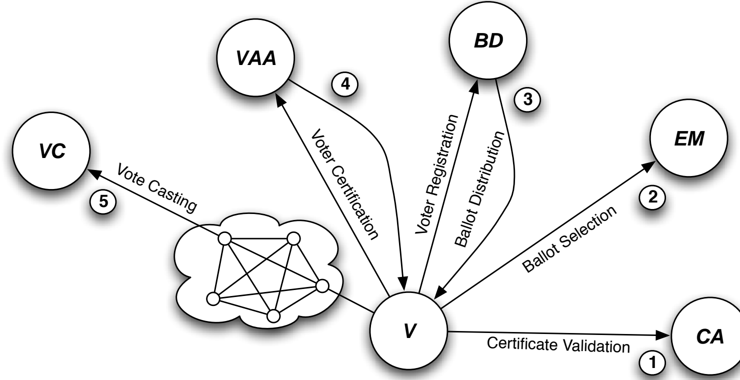

Figure 3.1: The voting protocol.

3.2.1 Certificate Authority

The Certificate Authority (CA) is responsible for managing the public certificates of the voters within the system, as well as the other four authoritative entities. The CA must fulfill the following list of responsibilities within the system: (1) receiving of non-validated certificates for processing, (2) allowing administrators to approve non-validate certificates, and (3) distributing validated certificates to voters as well as the other authoritative entities.

When a new voter wishes to interact with the eVoting system, they will create a public certificate that uniquely identifies them in the system. The voter will then send a request, through the voting client, to CA that will contain their newly created and non-verified public certificate. CA must receive this request and place it on its work queue for processing. An administrator should process this queue of non-verified public certificates by validating the details of the public certificate, and ensuring the identity is correct and accurate for the given voter. Once the administrator has verified the voter, and his public certificate, the certificate is then added to CA's list of validated certificates. The voter may then begin the voting process.

CA may be contacted by any of the other four authoritative entities and asked to produce a validated public certificate for either a voter, or one of the authoritative entities. For instance,

the Ballot Distributor might want to verify a public certificate presented by a voter, or the Vote Counter might want the Voter Attesting Authorities public certificate to verify a signature. The CA is responsible for servicing all of these requests.

3.2.2 Election Manager Authority

The Election Manager (EM) is responsible for managing and distributing ballots to voters and other authoritative entities within the system. The EM must fulfill the following list of responsibilities within the system: (1) allow administrators to add, modify, or delete elections, (2) ascertain which configured elections are currently in the open, votable state, and (3) distribute election information to voters as well as other authoritative entities within the system.

The EM can manage any number of ongoing elections, where each election has its own ballot. Ballots are described using our XML-based Ballot Description Language (BDL). A ballot will have various details associated with it, including a start and stop time, as well as ballot line items. A ballot may only be accessible, and thus open for voting, in the time period described by the start and stop times. The ballot will have a number of ballot line items, where each ballot line item is a direct question with a finite set of defined answers. Additionally, a voter has the option to specifically abstain from voting for any given ballot line item.

The opening and closing of ballots should be automated, and require no interaction from an administrator. However, an administrator should be able to easy compose a new ballot through the use of BDL, and add the ballot to the system. Once the current time has reached the specified start time of the ballot, the ballot will become accessible to the voters and other authoritative entities within the system.

3.2.3 Ballot Distributor Authority

The Ballot Distributor (BD) is responsible for distributing blank ballots to registered voters in the system. The BD must fulfill the following list of responsibilities within the system: (1) distribute blank ballots for elections in the open phase to registered voters, and (2) generate and track serial numbers guaranteed to be unique within a given election.

A voter will request a blank ballot from the BD for a given election they wish to participate in. BD must first verify the election is still in the open phase, and thus a blank ballot can be generated. Each blank ballot should have a unique serial number, which is used throughout the voting protocol. This serial number must be unique within a given election. Thus, it is possible for two different

ballots, each from a different election, to contain the exact same serial number. BD should thus track each allocated serial number. BD must also be able to produce a list of allocated serial numbers for a given election once it has closed, as well as the total number of voters registered for an election.

3.2.4 Voter Attesting Authority

The Voter Attesting Authority (VAA) is responsible for certifying filled ballots that have been obtained from BD. VAA must fulfill the following list of responsibilities within the system: (1) accept incoming ballot certification requests from voters, (2) verify a filled ballot did in-fact originate from BD, (3) sign and transmit verified filled ballots back to the voter, and (4) keep track of all certified ballots for each ongoing election.

A voter, after filling in their blank ballot, will then blind the ballot and send it to VAA for certification. The blinding process, accomplished through blind signatures, ensures that VAA cannot decipher the contents of the filled ballot while still allowing for a blind signature, similar to traditional public key cryptography-based signatures. Before VAA signs a blinded ballot, if must first ensure the ballot has originated from BD and that the voter has not already submitted a ballot certification request for the given election.

Once VAA has verified the ballot as being authentic, it will then sign the blinded ballot and send it back to the voter. VAA should also maintain a list of all certified ballots for each ongoing election. This will be used to produce a list of certified votes at the close of an election, as well as ensure voters only submit one ballot for certification for a specific election.

3.2.5 Vote Counter Authority

The Vote Counter (VC) is responsible for receiving cast ballots from the voter, and tallying and displaying the results of elections. Thus, the VC must fulfill the following list of responsibilities within the system: (1) accept and store filled certified ballots from voters, (2) tally the results of cast ballots, and (3) display the results of elections.

The VC must be available to receive cast votes from voters, either through a regular communication channel or an anonymous channel. While not strictly required in all situations, the anonymous submission channel will provide a greater degree of privacy during the voting process. As VC receives cast ballots, it should inspect them to ensure they have already been certified by VAA. If this condition is met, and the election is still open for voting, VC should store the cast ballot and recalculate the election tallies.

Once an election has closed, VC should be able to produce a raw list of all cast ballots for a given election. Additionally, VC should display the tallied results of the election, declaring the most popular votes for each ballot line item in a given ballot. Since voters may either select a predefined choice for each ballot line item, or choose to abstain from voting, the VC must take this in to account when tallying the results.

3.2.6 Voting Client

The Voting Client (V) is implicitly used to communicate and facilitate the voting protocol with the five previously mentioned authoritative entities. To sum up the responsibilities of the voting client, it must provide the following: (1) generate and use locally held public key cryptography keys and certificates, (2) display lists of ongoing, open elections and allow the voter to select an election to participate in, (3) request a blank ballot from BD, (4) provide a mechanism for the voter to vote for each ballot line item with the ballot for the selected election, (5) blind and submit a filled ballot for certification, (6) un-blind a certified ballot and verify it's contents, and (7) cast a certified ballot.

The voting client should be easy to use, and guide the voter through the voting protocol. As the voter progresses through the voting process, the voting client should save its current state frequently so the voter can pause the voting process and return at a later time to continue. The voting client should also be conscience of locally stored information, and make all attempts to securely store and clean up any sensitive information.

3.3 Communication Channels

The five authoritative entities as well as the client implicitly use two different communication channels to facilitate the voting process. The first, and main communication channel, is a plain HTTP channel. The various authorities and entities communicate by crafting XML-based messages and sending them via the HTTP 1.1 POST method. This protocol was chosen due to its widespread use and availability. This is especially important for voters sitting behind proxies and firewalls.

The second channel leveraged during the voting protocol, specifically at the ballot casting stage, is an anonymous channel. The anonymous channel is provided by the TOR network, and implicitly uses HTTP 1.1 POST methods. The TOR network is an overlay network that implements onion-routing protocols to securely and anonymously provide a TCP-based HTTP protocol through a mixture of participating nodes. The use of the TOR network provides anonymity to the voter when

they cast their vote, as it obscures the originating location of the voter from the Vote Counter.

4. Implementation

The eVoting system as described below was implemented using the Java Platform, Standard Edition 5. The implementation was performed on a Apple-based system, and integration and testing was performed on a combination of Apple and Linux-based systems. The source code totals over 15,000 lines of code, and uses the external libraries and capabilities shown in Table 4.1.

Table 4.1: External libraries and dependencies used in the eVoting system.

Library	File
Apache Commons Codec	commons-codec-1.3.jar
Apache Commons Collections	commons-collections-3.2.jar
Apache Commons Configuration	commons-configuration-1.5.jar
Apache Commons Lang	commons-lang-2.4.jar
Apache Commons Logging	commons-logging-1.0.4.jar
JDOM XML Parsing Library	jdom.jar
MySQL Java Connector	mysql-connector-java-5.0.8-bin.jar
Apache Struts Framework	struts2-core-2.0.11.1.jar
Object-Graph Navigation Language (for Apache Struts)	ognl-2.6.11.jar
XWork Command-pattern Framework (for Apache Struts)	xwork-2.0.4.jar

The remainder of this chapter will describe specific implementation choices and decisions. In Section 4.1, we will describe the various messages used within the system, and Section 4.2 will describe the specific technologies chosen to facilitate communication of the messages. Section 4.3 will describe the developed XML schema for describing ballots, Section 4.4 will describe the database schemas, Section 4.5 will describe the selected anonymous channel implementation and integration, and Section 4.6 will describe how we implemented blind signatures.

4.1 Messages

The eVoting system uses various XML formatted messages to facilitate communications. All requests to, and responses from, authoritative entities are always formatted as an XML message. The table below lists the XML messages the voting client will send during the protocol, and the responses it might receive:

Table 4.2: Valid XML messages.

Request	**Response**
GetCertificate	GetCertificateResponse VotingProtocolError
ApproveCertificate	ApproveCertificateResponse ApproveCertificateAlreadyReceivedResponse VotingProtocolError
GetOpenBallots	GetOpenBallotsResponse VotingProtocolError
GetBallot	GetBallotResponse VotingProtocolError
RegisterVoter	RegisterVoterResponse VotingProtocolError
CertifyVoter	CertifyVoterResponse VotingProtocolError
CastVote	CastVoteResponse VotingProtocolError

4.1.1 General Messages

There are various general-purpose messages that may be encountered at different stages of the voting process. These general-purpose messages will be described here, and referenced in later sections where appropriate.

VotingProtocolError

The `VotingProtocolError` message is a general-purpose error message that may be used during any phase of the voting protocol. This message is typically used to relay information related to errors in the communication or processing of other messages. For instance, a communication error between the Voting Client and the Certificate Authority would be reported through a `VotingProtocolError` message created by the Voting Client.

The `VotingProtocolError` takes the form given in Figure 4.1, described in XML Document Type Definition (DTD) format.

```
<!ELEMENT VotingProtocolError ( ErrorString, InResponseTo? ) >
  <!ELEMENT ErrorString ( #PCDATA ) >
  <!ELEMENT InResponseTo ( #PCDATA ) >
```

Figure 4.1: VotingProtocolError DTD.

The `<ErrorString />` tag will contain a meaningful message describing the error encountered.

The optional `<InResponseTo />` tag will contain the name of the message that generated this error. For example, say a communication problem occurred during the transmission of the `CertifyVoter` message. The Voting Client would receive (or generate) a `VotingProtocolError` message of the form shown in Figure 4.2.

```
<VotingProtocolError>
  <ErrorString>
    There was a communication error while attempting
    to certify your ballot
  </ErrorString>
  <InResponseTo>CertifyVoter</InResponseTo>
</VotingProtocolError>
```

Figure 4.2: VotingProtocolError example.

4.1.2 Certificate Authority Messages

The following messages may be received or generated by the Certificate Authority. Note that each of the five authoritative entities, as well as by the Voting Client, use the `GetCertificate` message very frequently in the voting protocol. As such, this message could also be categorized as a generate purpose message. However, for the sake of clarity, it is presented as a CA message.

GetCertificate

The `GetCertificate` message is used by the Voting Client as well as all the authoritative entities to retrieve a specified certificate. The message will specify some attribute by which the CA will locate the requested certificate. If the certificate is found, it will be returned via a `GetCertificateResponse` message. If the certificate is not found, a `VotingProtocolError` message will be generated.

The `GetCertificate` message takes the form shown in Figure 4.3.

```
<!ELEMENT GetCertificate ( Alias | Serial ) >
  <!ELEMENT Alias ( #PCDATA ) >
  <!ELEMENT Serial ( #PCDATA ) >
```

Figure 4.3: GetCertificate DTD.

Note that the `<Alias />` and `<Serial />` elements are mutually exclusive. We only need to specify one or the other when requesting a certificate. In practice, the `<Alias />` field is typically

used when requesting the certificate for one of the authoritative entities, as they are well known to all parties working within the system. In the case of an authoritative entity requesting a voters certificate, they will already hold the certificate presented by the voter. Thus the authoritative entity will extract the serial number or alias from the X.509 Certificate and include this in the `<GetCertificate />` request. The authoritative entity can then verify that the certificate returned by the CA matches the certificate presented by the voter.

GetCertificateResponse

When CA receives a valid `<GetCertificate />` request, it will search it's list of validated certificates, trying to match on either the `<Alias />` or `<Serial />` elements value. If a matching certificate is found, CA will construct a `<GetCertificateResponse />` message, and embed the valid certificate. If no matching certificate is found, CA will return a `<VotingProtocolError />` response message. The GetCertificateResponse message form is given in Figure 4.4.

```
<!ELEMENT GetCertificateResponse ( Certificate, Alias | Serial ) >
  <!ELEMENT Certificate ( #PCDATA ) >
  <!ELEMENT Alias ( #PCDATA ) >
  <!ELEMENT Serial ( #PCDATA ) >
```

Figure 4.4: GetCertificateResponse DTD.

The `<Certificate />` element above will contain a base-64 PEM encoded version of the X.509 certificate so that it can easily be sent as valid payload data in an HTTP 1.1 POST response.

AproveCertificate

The ApproveCertificate message is used for bootstrapping a new voter into the system. When a voter starts up the Voting Client for the first time, they can either load in pre-existing key and certificate data, or generate a new public/private key pair and corresponding X.509 certificate. In the case where the user has generated a new certificate, they must send this certificate to CA for validation. Thus, the Voting Client will initiate a ApproveCertificate message and enclose the newly generated certificate. The ApproveCertificate message format is given in Figure 4.5.

The `<Alias />` element in the above message is a translated form of the principal field in the X.509 certificate. The principal field is extracted, and spaces are translated to underscores (_), and equals signs (=) are translated to dashes (-).

```
<!ELEMENT ApproveCertificate ( CSR, Alias ) >
  <!ELEMENT CSR ( #PCDATA ) >
  <!ELEMENT Alias ( #PCDATA ) >
```

Figure 4.5: ApproveCertificate DTD.

Upon receiving one of these messages, CA will generate a `ApproveCertificateRequest` message, and will add the certificate to a queue of non-validated certificates. An administrator must interact with the system in order to verify and validate the queued certificates. Once a certificate has been validated, it is loaded into CA's list of validated certificates.

ApproveCertificateResponse

As mentioned previously, the `ApproveCertificateResponse` message is used by CA to notify the voter that their `ApproveCertificate` request was successfully received and the certificate has been queued for approval. The `ApproveCertificateResponse` message format is given in Figure 4.6.

```
<!ELEMENT ApproveCertificateResponse ( Received ) >
  <!ELEMENT Received ( #PCDATA ) >
```

Figure 4.6: ApproveCertificateResponse DTD.

The `<Received />` element is simply a boolean value that indicates successful reception of the certificate. In practice, this field should always contain a true value. Otherwise, a `VotingProtocolError` message would have been generated and returned detailing the error.

ApproveCertificateAlreadyReceivedResponse

Due to the asynchronous nature of the approval of newly generated and submitted certificates, the voter could easily send multiple `ApproveCertificate` messages. The first message will be processed, and the certificate will be added to CA's queue of certificates needing approval. All subsequent `ApproveCertificate` requests by the same voter will elicit a `ApproveCertificateAlreadyReceivedResponse` message. This is used to simply inform the voter that their request for approval has already been received. The `ApproveCertificateAlreadyReceivedResponse` message format is given in Figure 4.7.

The `<Alias />` element takes the same form as described above for the `ApproveCertificate`

```
<!ELEMENT ApproveCertificateAlreadyReceivedResponse ( Alias ) >
  <!ELEMENT Alias ( #PCDATA ) >
```

Figure 4.7: ApproveCertificateAlreadyReceivedResponse DTD.

request.

4.1.3 Election Manager Messages

The following messages may be received or generated by the Election Manager.

GetOpenBallots

When a voter starts up the Voting Client, the first task they encounter (after proper certificate validation and approval) is the selection of the ballot they wish to participate in. To accomplish this, the Voting Client must enumerate the ballots that are currently open for voting. This is done by submitting a `GetOpenBallots` request to the Election Manager. The `GetOpenBallots` message format is given in Figure 4.8.

```
<!ELEMENT GetOpenBallots  >
```

Figure 4.8: GetOpenBallots DTD.

When the Election Manager receives one of these requests, it will scan through its list of ballots and determine which ones are currently available for voting. This is determined by examining the start and end date for each ballot. The current date and time must be in between these two values for the ballot to be considered available for voting. The Election Manager will thus compile this list, and return the list of open ballots via the `GetOpenBallotResponse` message.

GetOpenBallotResponse

The `GetOpenBallotResponse` message is used by the Election Manager to return a list of all open ballots to the Voting Client. The definition of an open ballot is described above in the definition for the `GetOpenBallotResponse`. The `GetOpenBallotsResponse` message format is given in Figure 4.9.

As shown by the DTD in Figure 4.9, this message may contain multiple ballots. Each ballot is fully described, and the `id` attribute will be unique across all ballots in the message.

```
<!ELEMENT GetOpenBallotsResponse ( BallotList )  >
  <!ELEMENT BallotList ( Ballot* ) >
    <!ELEMENT Ballot ( Name, Description, Start, End, LineItems ) >
    <!ATTLIST Ballot id ( #PCDATA ) >
      <!ELEMENT Name ( #PCDATA ) >
      <!ELEMENT Description ( #PCDATA ) >
      <!ELEMENT Start ( #PCDATA ) >
      <!ELEMENT End ( #PCDATA ) >
      <!ELEMENT LineItems ( LineItem+ ) >
        <!ELEMENT LineItem ( Narative, Choices ) >
        <!ATTLIST LineItem line_item_id ( #PCDATA ) >
          <!ELEMENT Narative ( #PCDATA ) >
          <!ELEMENT Choices ( Choice+ ) >
            <!ELEMENT Choice ( #PCDATA ) >
            <!ATTLIST Choice choice_id ( #PCDATA ) >
```

Figure 4.9: GetOpenBallotsResponse DTD.

GetBallot

The `GetBallot` message can be used by any entity within the eVoting system to request a particular ballot. The ballot can be in either the open state (as previously described) or in a closed state. When crafting this message, the `<BallotId />` element should be used to specify the particular ballot one is interested in obtaining. If no matching ballot is found, a `<VotingProtocolError />` message is generated and returned. The `GetBallot` message format is shown in Figure 4.10.

```
<!ELEMENT GetBallot ( BallotId )  >
  <!ELEMENT BallotId ( #PCDATA ) >
```

Figure 4.10: GetBallot DTD.

GetBallotResponse

This message is generated in response to a `GetBallot` request. The `GetBallotResponse` message contains a complete description of the specified ballot including all ballot line items. The `GetBallotResponse` message format is given in Figure 4.11.

4.1.4 Ballot Distributor Messages

The following messages may be received or generated by the Ballot Distributor.

```
<!ELEMENT GetBallotResponse ( BallotId, Ballot )  >
  <!ELEMENT BallotId ( #PCDATA ) >
  <!ELEMENT Ballot ( Name, Description, Start, End, LineItems ) >
  <!ATTLIST Ballot id ( #PCDATA ) >
    <!ELEMENT Name ( #PCDATA ) >
    <!ELEMENT Description ( #PCDATA ) >
    <!ELEMENT Start ( #PCDATA ) >
    <!ELEMENT End ( #PCDATA ) >
    <!ELEMENT LineItems ( LineItem+ ) >
      <!ELEMENT LineItem ( Narative, Choices ) >
      <!ATTLIST LineItem line_item_id ( #PCDATA ) >
        <!ELEMENT Narative ( #PCDATA ) >
        <!ELEMENT Choices ( Choice+ ) >
          <!ELEMENT Choice ( #PCDATA ) >
          <!ATTLIST Choice choice_id ( #PCDATA ) >
```

Figure 4.11: GetOpenBallotsResponse DTD.

RegisterVoter

The `RegisterVoter` message is used by the voter, after selecting a ballot, to begin to voting process. This message is sent from the Voting Client and received by the Ballot Distributor. The `RegisterVoter` message format is given in Figure 4.12.

```
<!ELEMENT RegisterVoter ( BallotId, VoterCertificate )  >
  <!ELEMENT BallotId ( #PCDATA ) >
  <!ELEMENT VoterCertificate ( #PCDATA ) >
```

Figure 4.12: RegisterVoter DTD.

The `<BallotId />` element will specify the ballot which the voter wishes to begin the voting process on. This ballot must be valid and in the open state for this request to be successfully processed. If the ballot doesn't exist, or is in a close state, the Ballot Distributor will return a `VotingProtocolError` message. Additionally, the Ballot Distributor will ensure the voter has not already sent a `RegisterVoter` request for the specified ballot. As in Section 4.1.2, the `<VoterCertificate />` element will contain a base-64 PEM encoded string of the voters X.509 certificate.

RegisterVoterResponse

The `RegisterVoterResponse` message is generated in response to the successful reception and processing of a `RegisterVoter` request by the Ballot Distributor. This response message will con-

tain a blank ballot for the specified ballot, which the voter may use to participate in the voting process. The Ballot Distributor will sign the voters certificate to help ensure integrity through the voting process. The Ballot Distributor will also include the generated, unique serial number. The `RegisterVoterResponse` message format is given in Figure 4.13.

```
<!ELEMENT RegisterVoterResponse ( BlankBallot )  >
  <!ELEMENT BlankBallot ( BallotId, SignedCertificate, Vote, SignedSerial ) >
    <!ELEMENT BallotId ( #PCDATA ) >
    <!ELEMENT SignedCertificate ( Certificate, SignedDigest ) >
      <!ELEMENT Certificate ( #PCDATA ) >
      <!ELEMENT SignedDigest ( #PCDATA ) >
    <!ELEMENT Vote ( VoteValue, Comments ) >
      <!ELEMENT VoteValue ( #PCDATA ) >
      <!ELEMENT Comments ( #PCDATA ) >
    <!ELEMENT SignedSerial ( SerialNumber, Signature ) >
      <!ELEMENT SerialNumber ( #PCDATA ) >
      <!ELEMENT SIgnature ( #PCDATA ) >
```

Figure 4.13: RegisterVoterResponse DTD.

4.1.5 Voter Attesting Authority Messages

The following messages may be received or generated by the Voter Attesting Authority.

CertifyVoter

The `CertifyVoter` message is used by the voter to submit a filled ballot, blinded through a blind signature, to the Voter Attesting Authority for certification. The Voter Attesting Authority will verify the Ballot Distributors signature on the voters certificate, and that the voter has not already submitted a certification request for the specified ballot. The Voter Attesting Authority will then sign the filled ballot using a blind signature, and returned the signed ballot via the `CertifyVoterResponse` message. The `CertifyVoter` message format is given in Figure 4.14.

The filled and blinded ballot is contained in the `<SaltedVote />` element. This field is sufficiently blinded such that if this message were to be intercepted or captured, the voters ballot selections would be unrecoverable.

CertifyVoterResponse

The `CertifyVoterResponse` message is generated after the Voter Attesting Authority has fully processed a `CertifyVoter` message, and ensured that the ballot originated from the Ballot Distrib-

```
<!ELEMENT CertifyVoter ( SaltedBallot )  >
  <!ELEMENT SaltedBallot ( SignedCertificate, SaltedVote,
                            SignedSerial, BallotId, SignedDigest ) >
    <!ELEMENT SignedCertificate ( Certificate, SignedDigest ) >
      <!ELEMENT Certificate ( #PCDATA ) >
      <!ELEMENT SignedDigest ( #PCDATA ) >
    <!ELEMENT SaltedVote ( SaltedVoteValue ) >
      <!ELEMENT SaltedVoteValue ( #PCDATA ) >
    <!ELEMENT SignedSerial ( SerialNumber, Signature ) >
      <!ELEMENT SerialNumber ( #PCDATA ) >
      <!ELEMENT SIgnature ( #PCDATA ) >
    <!ELEMENT BallotId ( #PCDATA ) >
    <!ELEMENT SIgnedDigest ( #PCDATA ) >
```

Figure 4.14: CertifyVoter DTD.

utor and that the voter has not already submitted a certification request for the specified ballot. The `CertifyVoterResponse` message format is given in Figure 4.15.

Much like the `<SaltedVote />` element in the `CertifyVoter` message, the `<SaltedVote />` element in the above message will contain the sufficiently blinded filled ballot. Thus, an observer can garner no useful information by inspecting or storing this information. Likewise, the `<BlindSignature />` element will contain the results of VAA blindly signing the `<SaltedVote />` elements value. No useful information can be garnered from this element, except by the originating voter.

```
<!ELEMENT CertifyVoterResponse ( SignedSaltedBallot )  >
  <!ELEMENT SignedSaltedBallot ( SaltedVote, BlindSignature ) >
    <!ELEMENT SaltedVote ( SaltedVoteValue ) >
      <!ELEMENT SaltedVoteValue ( #PCDATA ) >
    <!ELEMENT BlindSignature ( #PCDATA ) >
```

Figure 4.15: CertifyVoterResponse DTD.

4.1.6 Vote Counter Messages

The following messages may be received or generated by the Vote Counter.

CastVote

The `CastVote` message is the final step in the voting process, whereby the voter submits their filled, certified ballot to the Vote Counter. The ballot and message are stripped of any identifying marks that would tie it back to a particular voter, and the submission is made through an anonymous

channel, thus ensuring privacy. Upon successful reception and storage of the `CastVote` message, the Vote Counter will respond with a `CastVoteResponse` message. If any error is encountered during the processing or storing, a `VotingProtocolError` message is generated. The `CastVote` message format is given in Figure 4.16.

```
<!ELEMENT CastVote ( CastBallot )  >
  <!ELEMENT CastBallot ( BallotId, SignedMarkedVote, SignedDigest ) >
    <!ELEMENT BallotId ( #PCDATA ) >
    <!ELEMENT SignedMarkedVote ( MarkedVote, MarkedVoteSignature ) >
      <!ELEMENT MarkedVote ( Vote, VoterMark) >
        <!ELEMENT Vote ( VoteValue, Comments ) >
          <!ELEMENT VoteValue ( #PCDATA ) >
          <!ELEMENT Comments ( #PCDATA ) >
        <!ELEMENT VoterMark ( #PCDATA ) >
      <!ELEMENT MarkedVoteSignature ( #PCDATA ) >
    <!ELEMENT SignedDigest ( #PCDATA ) >
```

Figure 4.16: CastBallot DTD.

CastVoteResponse

The `CastVoteResponse` message is generated by the Vote Counter after receiving and processing a `CastVote` request. This message is delivered back to the voter via the anonymous channel it was submitted from. The `CastVoteResponse` message format is given in Figure 4.17.

```
<!ELEMENT CastVoteResponse ( SignedDigest )  >
  <!ELEMENT SignedDigest ( #PCDATA ) >
```

Figure 4.17: CastBallotResponse DTD.

The `<SignedDigest />` element above contains a Vote Counter signed digest of the cast ballot. This value is used by the Voter to validate the successful reception of their cast vote by the Vote Counter.

4.2 Receiving and Sending Messages

In this section, we will describe the selected communication protocol that will be used to facilitate the transfer of information. We will also describe the technology selections used to implement the communication protocol.

4.2.1 Communication Protocol

Sending and receiving of messages is important in the eVoting system in facilitating all communications between the Voting Client and the authoritative entities. The communication channels used for the sending and receiving of messages should interoperate easily with a variety of networks and host systems. The communication channels should also be flexible and robust enough to support the sending and receiving of XML data. The voting protocol, as described in this paper, is secure in its own right. Thus, we will not require the communication channel to implement or provide any cryptographic, or other, privacy protection for protecting the contents of messages in transit within the channel.

We have chosen the widespread Hypertext Transfer Protocol (HTTP), version 1.1 [20] as the medium by which we will send all messages. Specifically, we will use the HTTP 1.1 GET and HTTP 1.1 POST methods. The HTTP 1.1 GET and POST methods are described in sections 9.3 and 9.5 respectively of RFC 2616.

The HTTP 1.1 GET method is used by the voters to access the public interfaces of the various entities, such as the vote tally provided by the Vote Counter. This access will be done via standard web browsers, or other HTML interfaces, and not through the voting client.

The HTTP 1.1 POST method will be the primary method used to exchange information, and will facilitate all communications between the voting client and the authoritative entities. The POST method allows for data to be included in the request body. Thus, we will use this ability to attach our XML-based messages into the request to facilitate the transfer of information. In cases where the XML messages contain might contain non-text characters, the content will be base-64 encoded to ensure it is transferred and decoded properly.

4.2.2 Implementing the Communication Protocol

In Section 3.3 and 4.2.1, we laid the basic foundation for the communication channels. We have selected the HTTP 1.1 POST and GET protocols, specifically over a basic HTTP connection, and an anonymized connection via the TOR network. In this section, we will talk more specifically about the implementation choices and issues faced.

Given that HTTP 1.1 was selected as the main communication protocol, we wanted to leverage existing systems and tools to avoid reimplementation. Each authoritative entity would need to accept and process incoming HTTP requests, and respond with HTTP responses. The Voting Client

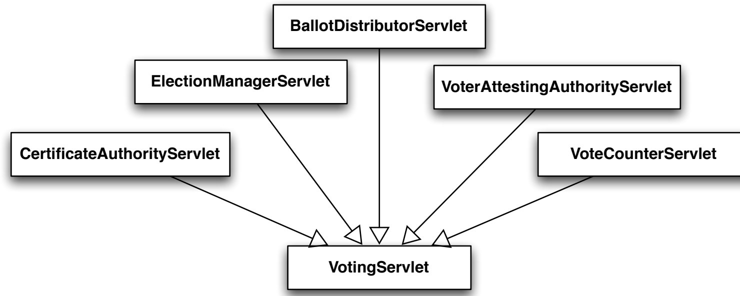

Figure 4.18: The various servlet classes, used to implement the authoritative entities.

needed to create and send HTTP requests, and read the responses sent back by the authoritative entities. Thus, we sought a basic client-server architecture. Given the widespread use of HTTP, the large number of pre-existing tools, we had many choices. We decided to leverage the capabilities provided by the Apache Tomcat server [23], a Java-based application server. The Tomcat server provides basic implementations of the Java Servlet [33] and JavaServer Pages [34] technologies. In our implementation, each authoritative entity would implement a Servlet class. The final class layout is given in Figure 4.18. A base class defines general functionality used by each authoritative entity, while the specific subclasses implement behavior specific to each.

To communicate with the authoritative entities via the chosen server technology, we implemented a very basic socket-based client. The client makes strong use of `URLConnection` class, and leverages the `TorLib.TorSocket` method for anonymous socket connections. The general structure of the various clients is shown in Figure 4.19. Here we find a superclass, `ServletClient`, which provides the basic I/O operations to communicate with the various authoritative entities. This superclass is then subclassed for each authoritative entity we wish to communicate with. Each subclass implements functionalities specific to a given authoritative entities, and provides methods to send the entity-specific messages given in Table 4.2.

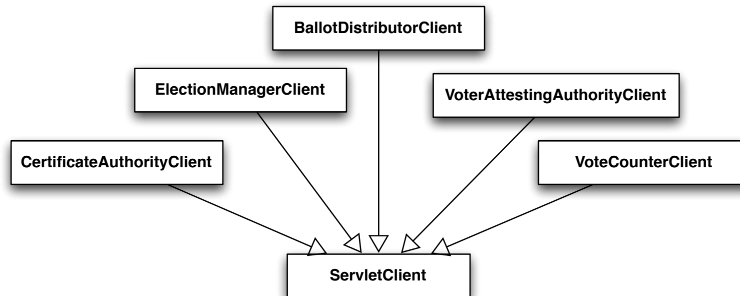

Figure 4.19: The various client classes, used to communicate with the authoritative entities.

4.3 Ballot Description Language

The Ballot Description Language (BDL) was developed to provide an intuitive and expressive language for defining ballots. In the eVoting system, an administrator would define a new ballot using the BDL and add it to the Ballot Distributor authority to enable voting on the new ballot. The typical form of a ballot described in BDL must follow the DTD described in Figure 4.20.

```
<!ELEMENT Ballot ( Name, Description, Start, End, LineItems )>
<!ATTLIST Ballot id ( #PCDATA )>
  <!ELEMENT Name ( #PCDATA )>
  <!ELEMENT Description ( #PCDATA )>
  <!ELEMENT Start ( #PCDATA )>
  <!ELEMENT End ( #PCDATA )>
  <!ELEMENT LineItems ( LineItem+ )>
    <!ELEMENT LineItem ( Narative, Choices+ )>
    <!ATTLIST LineItem line_item_id ( #PCDATA )>
      <!ELEMENT Narative ( #PCDATA )>
      <!ELEMENT Choices ( Choice+ )>
        <!ELEMENT Choice ( #PCDATA )>
        <!ATTLIST Choice choice_id ( #PCDATA )>
```

Figure 4.20: Ballot Description Language DTD.

The `<Name />` element is used to give the ballot a human-understandable name for the ballot. There are no strict requirements when defining a name, but it should be descriptive and allow the voter to easily identify the ballot given a list of all available ballots. The `<Description />` element allows a more verbose description of the ballot. The `<Start />` and `<End />` elements describe the time window in which voters may cast votes for the ballot. This time range is defined as `[Start, End)`, where voters can begin voting once the current time is equal to or after the `Start` time, and prior to the `End` time.

The ballot also will contain a `<LineItems />` element, which will contain a collection of `<LineItem />` elements. Each `<LineItem />` element will contain a `<Narative />` element that gives a description of the line item. A set of `<Choice />` elements will be contained in the `<Choices />` element defining the valid responses a voter can select for the line item. While not represented by the BDL DTD, each line item contains an implicit choice to abstain from voting. Thus, we can represent either a specific selection by the voter for a defined choice, or the selection of the voter to abstain from voting.

A sample ballot, described in BDL, is shown in Figure 4.21.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Ballot id="001">
  <Name>Ballot 001</Name>
  <Description>Vote for the president</Description>
  <Start>2008-03-20 00:00:00</Start>
  <End>2008-03-21 00:00:00</End>

  <LineItems>
    <LineItem line_item_id="001">
      <Narative>Select a candidate for president</Narative>
      <Choices>
        <Choice choice_id="001">Alice</Choice>
        <Choice choice_id="002">Bob</Choice>
      </Choices>
    </LineItem>
    <LineItem line_item_id="002">
      <Narative>Select a candidate for vice president</Narative>
      <Choices>
        <Choice choice_id="001">Charlie</Choice>
        <Choice choice_id="002">Dave</Choice>
      </Choices>
    </LineItem>
  </LineItems>
</Ballot>
```

Figure 4.21: A sample ballot, described in the Ballot Description Language.

4.4 Authoritative Entity Long-term Storage

Many of the authoritative entities have a need for long-term storage. This storage must be persistent, and support multiple concurrent connections. In a large-scale election, with millions of voters, it is imperative that we maintain read and write consistency within the system.

We will achieve these needs through the use of a relational database system (RDBMS), installed and accessible for each authoritative entity that requires long-term storage. We have chosen to use the MySQL Community Edition RDBMS system. The Ballot Distributor, Voter Attesting Authority, and Vote Counter will all require their own, unique instance of the MySQL RDBMS.

The schema and design considerations for each database will be given below.

4.4.1 Schemas

The schemas for the Ballot Distributor, Voter Attesting Authority, and Vote Counter will be described in this section. We will provide an overview of the table definitions, and provide commentary on our choices.

First, we will define the Ballot Distributor schema. If you recall from Section 3.2.3, we need to

track all distributed blank ballots, including the uniquely generated serial number, and ensure that a given voter receives only one blank ballot for each election they register for. We will accomplish these requirements using the schema given in Table 4.3.

Table 4.3: BDClaimedSerial table definition.

Column	**Type**	**Null Allowed**	**Constraints**
id	bigint unsigned	N	PK
ballot_id	varchar(255)	N	UNIQ(ballot_id, voter_identity)
issued_on	datetime	N	
ballot_serial	varchar(255)	N	
voter_identity	varchar(255)	N	UNIQ(ballot_id, voter_identity)

As shown in Table 4.3, we implement one of our requirements - a voter can only register and receive one blank ballot for each election - by defining a unique constraint across the `ballot_id` and `voter_identity` columns. A sample SQL DDL statement is given in Figure 4.22 that will implement this constraint.

```
ALTER TABLE BDClaimedSerial
ADD UNIQUE KEY BDClaimedSerial_voter_identity_serial (ballot_id, voter_identity);
```

Figure 4.22: DDL statement to add unique constraint.

The other columns listed in Table 4.3 will allow us to store the generated serial number, the voters identity, and the date and time the blank ballot was distributed.

Next, we will define the schema used by the Voter Attesting Authority to store blinded ballots received for certification. Table 4.4 gives the column definitions.

Table 4.4: VAACertifiedVotes table definition.

Column	**Type**	**Null Allowed**	**Constraints**
id	bigint unsigned	N	PK
ballot_id	varchar(255)	N	
ballot_serial	varchar(255)	N	
certified_on	datetime	N	
salted_vote	text	N	
voter_signature	varbinary(255)	N	
vaa_signature	text	N	

The `ballot_serial` will be the unique serial number generated by the Ballot Distributor. The

salted_vote field will contain the actual blinded ballot. Recall that the ballot has been sufficiently obscured through a blind signature before it is sent to the Voter Attesting Authority. Thus, the rows contained in this table will provide no usefully information to an adversary regarding the choices selected by a voter. The voter_signature and vaa_signature are also stored so that the Voter Attesting Authority can later reproduce all ballots sent for certification.

Table 4.5: VCCastVotes table definition.

Column	**Type**	**Null Allowed**	**Constraints**
id	bigint unsigned	N	PK
cast_date	datetime	N	
ballot_id	varchar(255)	N	
vote	text	N	
comments	varchar(255)	Y	
voter_mark	varbinary(255)	N	
vc_digest	varbinary(255)	N	

Next, we will define the schema used by the Vote Counter to store and process all cast ballots. Table 4.5 gives the table definition used by the Vote Counter. One important concept to notice is that we store the vote in a single field, the vote field. Given that a ballot might have multiple line items, we will use a serialization technique to condense the voters answers for multiple line items into a single value. This value can easily be unserialized and mapped to the specific line items to determine the actual choices selected by the voter. We will also store the voters mark, a unique mark created by each voter and used to audit their vote, and the Vote Counters digest created over the fields of the cast ballot. These two fields, voter_mark and vc_digest, can be used after an election is closed for voting to audit and verify the results of the election.

As previously mentioned, we serialize the voters line items choices into a single value and store that value in the VCCastVotes table. Thus, we must have a way to deserialize these votes into the particular line items so that we can tally the results. To accomplish this, we will define a table for each Ballot added to the system. The table will follow a pre-defined form, which will be used when mapping a ballot described in BDL to the table representation in the MySQL database. Table 4.6 gives the table definition for Ballot 001, described in BDL in Figure 4.21.

The table for each ballot must have an id column, which will match the id attribute of the <Ballot /> element in the BDL description. The table should also include a datestamp column to record the time the vote was received. The table will then have a number of columns, named by

Table 4.6: Ballot001 table definition.

Column	**Type**	**Null Allowed**	**Constraints**
id	bigint unsigned	N	PK
datestamp	datetime	N	
li001	varchar(16)	N	
li002	varchar(16)	N	
comments	text	N	

prefixing the line items `id` attribute with `li`, where each column maps to a line item given in the ballots BDL specification. For example, Table 4.6 has two columns, `li001` and `li002`, representing line items `001` and `002` respectively. Finally, a `comments` field is given to record any comments specified by the voter.

4.5 Anonymous Channel

The original voting protocol proposed in [39] stated that it did not rely on an anonymous channel. The vote casting protocol was assumed to take place via a pseudo-anonymous channel, such as an anonymous ftp session. While operationally this approach succeeds, it posses major threats to the overall privacy of the voter. Given the proposed vote casting process, a dubious Vote Counter could easily track a vote back to an IP address. Given a single IP address, it is a simple task to trace it back to a geographic location, and in some cases an individual person. We find this lack or privacy to be a compelling restriction with the original protocol, and thus propose a more sufficient anonymous channel.

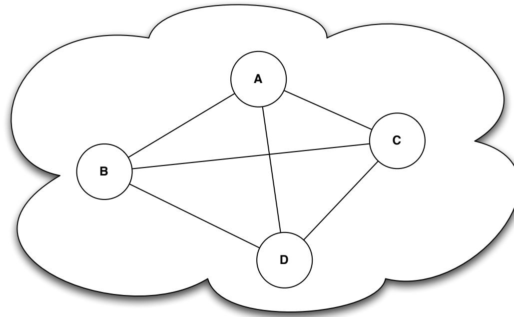

Figure 4.23: Example overlay mix netword used by TOR.

We have chosen to leverage the capabilities of the TOR network to cast votes. In this scheme, the Voting Client will proxy HTTP requests into the TOR network, finally emerging through a TOR

exit node and reaching the Vote Counter. The TOR network is widely used for anonymous web browsing, and other anonymous service implementations. TOR is based on a mix network, as shown in Figure 4.23. In this mix network scheme, a fully connected overlay is setup, whereby each Onion Router (OR) maintains a TLS socket connection to all other ORs in the network.

Suppose `X` in Figure 4.24 wants to establish a TCP connection with `Y` through the TOR network. In order to do this, `X` chooses an OR to begin the establishment of the circuit. Thus, `X` establishes a connection with `A`, the first hop in the circuit. Next, `X` asks `A` to extend the circuit to `D`. When this is done, `X` asks `D` to create the TCP connection to `Y`. Once this has been done, `X` is notified that the circuit has been setup and is ready to accept data. `X` then sends stream data through the circuit via a SOCKS proxy, which transparently provides a TCP connection to `Y` via the TOR circuit.

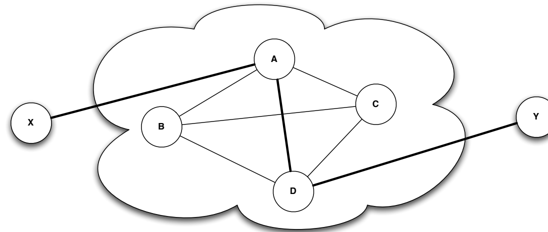

Figure 4.24: Connection setup through the network from source to destination.

In order to tie our implementation into the TOR network, we made use of a Java library provided by Joe Foley [21] called TorLib. TorLib is a library part of a larger project, the TinFoil project. TorLib provides a set of methods that can be used to proxy requests into the TOR network, as well as provide anonymous DNS resolutions. The main methods used from the TorLib are `TorLib.TorResolve()` and `TorLib.TorSocket()`. The `TorLib.TorSocket()` method is used to establish a socket connection with a remote host. This method relies on the existence of a local TOR client to implement its functionality. The `TorLib.TorResolve()` method is used to anonymously resolve a hostname to an IP address, and implicitly creates a socket connection through the local TOR client.

In order to provide the highest level of flexibility to the users of the eVoting system, the Voting Client is configurable as to whether an anonymous connection should be used when casting votes. If the Voting Client is configured to use an anonymous connection, the local machine must have a TOR client installed and running before casting the vote. While the end goal is to integrate the TOR client code into the Voting Client thus eliminating the need to run a separate TOR client, the

state of the TOR project does not readily facilitate this.

4.6 Blind Signatures

The eVoting system uses blind signatures in the voting protocol to achieve privacy for the voter while submitting their filled ballot for certification with the Voter Attesting Authority. The blind signature scheme was initially proposed by Chaum in 1981 [7] and conceptually works as depicted in Figure 4.25.

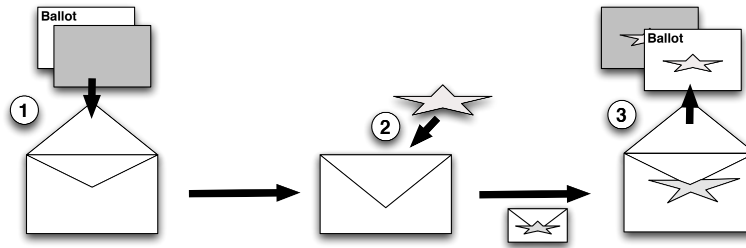

Figure 4.25: Applying the blind signature to a ballot.

We enclose the ballot, and a piece of carbon paper, in an envelope. We then seal the envelope and send it to the recipient. The recipient applies their signature to the outside of the envelope. The signed envelope is then sent back to the sender. The sender removes the contents from the envelope. The existence of the carbon paper inside effectively transfers the signature to the ballot underneath, without allowing the signer to see the contents.

More formally, let's suppose Alice has a message m that she wishes to have Bob sign. Let (n, e) be Bob's public key, and (n, d) be Bob's private key. Alice first generates a random value r, which she will use to blind m, where r and n are coprime ($gcd(r, n) = 1$). Alice then creates the blinded message x (step 1) by computing $x = (r^e m) \ mod \ n$, and sends x to Bob. Bob receives the blinded message x, which he can derive no useful information from, but wishes to sign. Bob creates the signed blinded message t (step 2) by computing $t = x^d \ mod \ n$, and sends t back to Alice. Alice receives the signed blinded message t, and wishes to extract the signature s (step 3). Since $x^d \equiv (r^e m)^d \equiv rm^d \ mod \ n$, we can obtain the signature by computing $s = r^{-1} t \ mod \ n$.

4.6.1 Implementing the Blind Signatures

Implementing blind signatures in Java proved to be a fairly straightforward task. The `BlindSignature` class was created to handle the task of computing and verifying blind signatures.

The `blind()`, `blindSignature()`, `unblind()`, and `verify()` methods are shown in Figure 4.26.

```
private static BigInteger parseMsg(byte[] msg, BigInteger n)
    throws BadPaddingException {
    BigInteger m = new BigInteger(1, msg);
    if (m.compareTo(n) >= 0) {
        throw new BadPaddingException("Message is larger than modulus");
    }
    return m;
}

public BigInteger blind(String message) throws BadPaddingException {
    BigInteger e = this.pubKey.getPublicExponent();
    BigInteger n = this.pubKey.getModulus();
    byte[] messageDigest = this.digest(message);
    BigInteger m = parseMsg(messageDigest, n);
    BigInteger x = this.r.modPow(e, n).multiply(m).mod(n);
    return x;
}

public BigInteger blindSignature(BigInteger message) {
    BigInteger d = this.privKey.getPrivateExponent();
    BigInteger n = this.privKey.getModulus();
    BigInteger t = message.modPow(d, n);
    return t;
}

public BigInteger unblind(BigInteger message) {
    BigInteger n = this.pubKey.getModulus();
    BigInteger s = this.r.modInverse(n).multiply(message).mod(n);
    return s;
}

public boolean verify(BigInteger signature, String message)
    throws BadPaddingException {
    BigInteger e = this.pubKey.getPublicExponent();
    BigInteger n = this.pubKey.getModulus();
    BigInteger t = signature.modPow(e, n);
    byte[] messageDigest = this.digest(message);
    BigInteger m = parseMsg(messageDigest, n);
    return t.equals(m);
}
```

Figure 4.26: BlindSignature.java source code sample.

The `blind()`, `unblind()`, and `verify()` methods would be used by Alice, the sender, in the blind signature process. The `blindSignature()` method would be used by Bob, the receiver and signer, during the process. There are also public constructors methods (not shown in source code sampe) that are used to create the `BlindSignature` object by passing in a `RSAPublicKey` or `RSAPrivateKey`.

4.6.2 Key and Certificate Management

An important part of the eVoting system is the RSA key pair and corresponding X.509 certificate. Each authoritative entity, as well as all eligible voters, must maintain an RSA public and private key pair. Additionally, an X.509 certificate containing the associated RSA public key must also be maintained.

The eVoting system provides key management through the use of the Java keystore database, utilizing the Sun Microsystems proprietary JKS format. Each authoritative entity will utilize an instance of a keystore, which resides in the `Keys/` directory. The keystore will contain the authorities RSA public and private keys, as well as the associated X.509 certificate. Additionally, each eligible voter will maintain a similar keystore. The keystore for CA will also contain every verified X.509 certificate known within the system.

In the remainder of this section will we provide an overview of how the users keystore is generated in Section 4.6.2, as well as certificate management tasks provided by CA in Section 4.6.2.

Creating the Users Keystore

```
> keytool -genkey -keyalg RSA -keysize 1024 -alias Voter \
   -keystore KeyFile.jks -storetype JKS
Enter keystore password:  KeyFile
What is your first and last name?
 [Unknown]:  John Doe
What is the name of your organizational unit?
 [Unknown]:
What is the name of your organization?
 [Unknown]:
What is the name of your City or Locality?
 [Unknown]:  Fort Collins
What is the name of your State or Province?
 [Unknown]:  CO
What is the two-letter country code for this unit?
 [Unknown]:  US
Is CN=John Doe, OU=Unknown, O=Unknown, L=Fort Collins,
    ST=CO, C=US correct?
 [no]:  yes
```

Figure 4.27: Creating the voters keystore.

Prior to the voter participating in any voting process, he must establish a keystore, which in-turn generates a RSA public and private key and associated X.509 certificate. A script is provided with the eVoting client software to automate this task. For the sake of clarity, we will present the high

level tasks that the setup script provides.

Figure 4.27 shows the process of generating a 1024-bit RSA key pair and public certificate for user `John Doe`. The information must be entered accurately by the voter, as this will be verified at a later stage by CA. Any inaccuracies will result in an invalid certificate, which will prevent the user from proceeding through the voting process. Also note that we are using the `keytool` binary, an application that is provided with all standard distributions of the Java 5 and later run-time environment (JRE).

One the voter's keystore has been initialized, the setup script will generate a certificate signing request (CSR), which can be sent to CA to initiate the certificate verification process. Figure 4.28 shows the command used to generate the CSR.

```
> keytool -certreq -alias BurkeWebster -keyalg RSA \
    -file Voter.csr -keystore KeyFile.jks  -storetype JKS
```

Figure 4.28: Creating voters certificate signing request (CSR).

Once CA verifies the certificate in the CSR, CA will sign the voters certificate and import the certificate into its eligible voters list. From that point forward, the voter will be know and uniquely identifiable within the system based on properties of his or her X.509 Certificate. The task of importing a CA-verified certificate is handled by the eVoting client.

Certificate Management Tasks

The Certificate Authority (CA) will perform various tasks to verify and manage X.509 certificates for the eVoting system. CA will leverage both the `keytool` binary mentioned in the previous section, as well as OpenSSL libraries. The OpenSSL libraries are needed to perform the certificate signing tasks.

Figure 4.29 shows the task of generating CA's certificate and self-signing the certificate. Note that we store the certificates in the default PEM encoded format.

```
> openssl genrsa -out ROOTCA.pem -des 2048
> openssl req -new -key ROOTCA.pem -x509 -days 3650 -out ROOTCA.pem
```

Figure 4.29: Generating CA's self-signed certificate.

Importing the OpenSSL generated RSA keys into the Java keystore is shown in Figure 4.30. We

first must export the PEM encoded keys in DER format, which is the format required for importing into the Java keystore (lines 1 and 2). We then use a helper program, `ImportKey` to accomplish the import task.

```
> openssl pkcs8 -topk8 -nocrypt -in ROOTCA.pem -inform PEM \
   -out ROOTCA.der -outform DER
> openssl x509 -in ROOTCA.pem -inform PEM -out ROOTCA.der -outform DER
> java ImportKey ROOTCA.der ROOTCA.der ROOTCA ROOTCA
```

Figure 4.30: Importing CA's RSA keys.

Finally, the task of signing a CSR is shown in Figure 4.31. After signing a CSR, CA will import the signed certificate into its keystore, thus making it accessible for download by other entities within the eVoting system.

```
> openssl x509 -req -in Voter.csr -CA ROOTCA.pem -CAkey ROOTCA.pem \
     -out Voter.pem -days 365 -CAcreateserial -CAserial ROOTCA.seq
```

Figure 4.31: Signing a CSR.

5. Analysis

First, we will give an analysis showing that the voting protocol satisfies the five necessary conditions set forth in Section 3.1 in the absence of any fraud and collusion. We will then show that the voting protocol still satisfies the necessary conditions in the presence of fraud, and collusion between the authoritative entities.

5.1 In the Absence of Fraud

1. **Accuracy** can be proven by showing that votes cannot be modified without detection, valid votes are correctly counted, and invalid votes are excluded from the final tally.

   When a voter casts their vote, VC will generate and send a signed digest of the cast ballot via a return path in the anonymous network. Thus, when the voter receives this signed digest, they can verify the signature and digest, thus ensuring that the vote was not altered or changed during the transmission process. If the voter is unable to verify the signed digest returned by VC, he can claim fraud. This guarantees that votes cannot be modified without detection.

   Additionally, the signed digest generated by VC can be used to ensure valid votes are correctly counted. The signed digest will be transmitted to the voter during the vote casting process, and also published in a public place. This ensures that VC cannot later claim that it did not receive a valid vote. The public listing of cast votes, which includes the voter mark for each vote, can also be used by the voter to ensure that their cast vote appears on the list, and that it represents the manner in which the voter voted. These two facts guarantee that valid votes are counted correctly.

   Finally, we show that invalid votes are excluded from the final tally. There are two reasons, excluding fraudulent activity, that lead to invalid votes: an ineligible voter cast the vote, or an eligible voter cast more than one vote. In the first case, BD ensures that only eligible voters receive a blank ballot. VAA guarantees that the votes it certifies originated from BD by verifying the vote contains BD's signature. These facts together guarantee ineligible voters cannot cast a vote. In the case of an eligible voter attempting to cast more than one vote, VAA will detect and deter the vote from being cast. In the process of certifying a vote, VAA ensures that the voter has not already submitted a vote for certification. This is done by inspecting

the voter's signature on the vote. If VAA determines that the voter has already presented a vote for certification, it simply does not sign any additional vote. If the voter casts a vote that has not been signed by VAA, VC will reject the vote. Further more, each cast vote contains a voter mark, which is guaranteed to be unique. This ensures that two cast votes cannot be attributed to a single voter. Together, these provide the guarantee that an eligible vote cannot cast more than one vote.

2. **Democracy** can be proved by showing that only eligible voters may cast votes, and each eligible voter may cast one and only one vote. Due to the accuracy properties of the system already presented, we can guarantee the necessary properties of democracy.

3. **Privacy** can be proven by showing that only the voter will know how he or she voted, and that a cast vote cannot be linked back to the voter who cast it.

   Two authoritative entities will receive a copy of the filled vote - VAA and VC. However, the copy that VAA receives has been sufficiently randomized through blind signatures so that VAA cannot determine the contents of the vote. The copy that VC receives must not be blinded or obscured to ensure that VC can properly count the vote in the tallies. However, the voter does encrypt the vote via symetric key encryption in the TOR circuit, ensuring that the vote is known only to the voter until the point at which the vote has been received by VC.

   The cast vote does contain the voters mark, which is generated by the voter using the unique serial number assigned by BD. However, the methods used to generate the voter mark are sufficiently strong such that it is computationally infeasible to reduce the voter mark back to the original serial number. Thus, no information within the vote can be used to link a vote back to the voter who cast it.

   Finally, the anonymous submission channel used to cast the vote ensures that VC can only track a cast vote back to an IP address. Due to the topology of the mix networks provided by TOR, VC cannot know if the voter cast the vote from that IP address, or if this was the exit node used by the voter when casting their vote through the anonymous network. This ensures that a cast vote cannot be linked back to the individual voter that cast the vote.

4. **Verifiability** can be proven by showing that a voter can independently verify that his or her vote was correctly counted.

   At the end of the voting process for a given election, the list of cast votes is published by VC.

A voter can locate their vote on this list by using the voter mark, and ensuring the cast vote information is accurate and truthfully represents the selections of the voter.

5. **Un-authorized Proxy** can be proven by showing that a forged vote cannot be cast in the place of a voter choosing to not vote.

   At the end of the voting process, VC will have a set of cast ballots and VAA will have a set of salted ballots. There will be a one-to-one correspondence between a cast vote held by VC and a salted ballot held by VAA. Furthermore, each salted ballot held by VAA must be signed by an eligible voter. If required, only the eligible voter will be able to produce both the salted ballot and cast ballot matching those held by VC and VAA. Given this, nobody can cast a spurious vote if the voter decides to not cast his vote.

5.2 In the Presence of Fraud

We present two different scenarios involving fraud - one where the voter does not participate and one where the voter does participate - and show that the eVoting system can detect and deter all such activity. As given in [39], we define the following items, which will exist at the end of voting for a given ballot.

1. β - A set of blank ballots, generated by BD, where each ballot contains a unique serial number generated and signed by BD. Only an eligible voter and BD can each legitimately hold a copy of a blank ballot.

2. S - A set of filled, blinded ballots, each one being signed and sent by an eligible voter to VAA for certification (signing). A $S_i \in S$ contains an eligible voters signature, as well as BD's signature (on the serial number). A voter has a copy of one $S_i \in S$, and VAA will also have the same copy. VAA cannot forge a S_i, it must have originated from an eligible voter.

3. S' - A set of filled, blinded ballots that have been certified by VAA by applying its signature. Each eligible voter will hold a $S'_i \in S'$, and VAA will also hold a copy. Again, a S'_i cannot be forged by VAA, as it must have been signed and sent by an eligible voter.

4. V - A set of filled, unblinded, and certified ballots. Each $V_i \in V$ must contain the signature of an authorized voter, the signature of VAA, and the signature of BD.

5. F - A set of cast ballots, sent to VC by eligible voters. F is equivalent to the set of ballots contained in V with the voters' signature the serial number signed by BD removed. Each

eligible voter holds an $F_i \in F$, and VC also holds a copy. Note that a valid F_i will contain VAA's signature, thus unless VC and VAA are colluding, F will not contain any fraudulent ballots. Furthermore, there is a one-to-one correspondence between a $F_i \in F$, a $S_i \in S$, and a $S'_i \in S'$, via the voter mark. Thus, if fraud is suspected, only an eligible voter can produce all three together with the random number that generates S'_i and F_i from S_i.

Given the items defined above, which will exist at the end of the voting process, we now examine the different scenarios where the authoritative entities and voters are colluding, or participating in fraudulent activities. We will first present the scenarios where the voter is not participating in collusion, followed by the scenario where the voter is colluding. Of these situations, we will consider scenarios where EM does not collude, followed by the scenario where EM does collude. Similar to any other system that makes use of public key cryptography, we will assume that CA is a trusted entity within the system, and thus there is no opportunity for collusion.

5.2.1 Collusion involving BD and VAA

If BD and VAA collude the can have the following:

1. $\hat{\beta}$ - A set of valid blank ballots.
2. $\hat{S}$ - A set of filled and blinded ballots.
3. $\hat{S}'$ - A set of filled, blinded ballots signed by VAA, where there is a one-to-one correspondence between a $\hat{S}'_i \in \hat{S}'$ and a $\hat{S}_i \in \hat{S}$.
4. $\hat{F}$ - A set of cast ballots.

By casting votes in the proper manner, each ballot in $\hat{F}$ will contain VC's signature. Thus, these votes will appear to have been cast by eligible voters. However, $\hat{S}'_i$ and $\hat{S}_i$ will not contain the signature of an eligible voter. Although no ballot in $\hat{F}$ can be proven directly to be fraudulent, if BD and VAA cast some $\hat{S}'_i$ producing $\hat{F}_i$, then it is possible the number of cast votes will exceed the number of eligible voters. Furthermore, BD and VAA cannot produce a $\hat{S}_i \in \hat{S}$ and a $\hat{S}'_i \in \hat{S}'$ corresponding to a $\hat{F}_i \in \hat{F}$, such that the $\hat{S}_i$ and $\hat{S}'_i$ contain the signature of an eligible voter, without the voter claiming that his or her vote has not been counted. Similarly, no voter will be able to produce the random number that produces $\hat{S}'_i$ from $\hat{S}_i$, since this random value would have been calculated by the colluding parties.

5.2.2 Collusion involving BD and VC

If BD and VC collude they can have the following:

1. $\hat{B}$ - A set of blank ballots.

2. $\hat{F}$ - A set of cast ballots.

Each $\hat{B}_i \in \hat{B}$ can be proven as a valid blank ballot since it will contain a unique serial number, which has been signed by BD. Each $\hat{F}_i \in \hat{F}$ will contain VC's signature, but will not contain VAA's signature. These ballots can be proven invalid, thus preventing BD and VC by themselves from committing fraud.

5.2.3 Collusion involving VAA and VC

If VAA and VC collude they can have the following:

1. $\hat{S}$ - A set of filled and blinded ballots.

2. $\hat{S'}$ - A set of filled, blinded ballots certified by VAA and containing VAA's signature.

3. $\hat{F}$ - A set of cast ballots.

Since a $\hat{S}_i \in \hat{S}$ and $\hat{S}'_i \in \hat{S'}$ will not contain an eligible voters signature, nor BD's signature, they can be proven as fraudulent. Thus, VAA and CA cannot, by themselves, commit fraud.

5.2.4 Collusion involving BD, VAA, and VC

If BD, VAA, and VC all collude together they can produce the following:

1. $\hat{\beta}$ - A set of valid blank ballots.

2. $\hat{S}$ - A set of filled and blinded ballots.

3. $\hat{S'}$ - A set of filled, blinded ballots signed by VAA, where there is a one-to-one correspondence between a $\hat{S}'_i \in \hat{S'}$ and a $\hat{S}_i \in \hat{S}$.

4. $\hat{F}$ - A set of cast ballots, where each $\hat{F}_i \in \hat{F}$ will contain BD, VAA, and BD's signature.

This is the strongest scenario presented thus far, enabling the three authoritative entities to commit fraud. This is possible because VC can substitute some valid vote $\hat{F}_i \in \hat{F}$ for a fraudulent

vote, thus altering the final tallies. This scenario does not suffer from the issues that arise when only BD and VAA collude whereby the number of tallied votes could exceed the number of eligible voters.

However, if a valid vote it removed from the final tally, the voter will be unable to locate and verify their cast vote in the list published by VC. Thus the voter can claim fraud. Similarly, if VC does not alter the published list of cast votes, but instead only alters the tallies, a voter, or other entity that is independently verifying the results of the election could detect this, and fraud could be claimed. The trio must then produce a $\hat{F}_i \in \hat{F}$, a $\hat{S}_i \in \hat{S}$ and a $\hat{S}'_i \in \hat{S}'$, such that $\hat{S}_i$ and $\hat{S}'_i$ contain the signature of an eligible voter and $\hat{F}_i$ contains the voter mark also found in $\hat{S}_i$ and $\hat{S}'_i$.

5.2.5 Collusion with the involvement of EM

If EM colludes, the best it can do is change some property of a ballot, such as the end date. However, due to the significance of the voting process, their will be many interested parties both monitoring and verifying the results. Any inconsistencies introduced by EM would be flagged, and could be reconciled by auditing the list of cast ballots.

5.2.6 Collusion with the involvement of voters

The only way a voter can commit fraud within the system is by providing one copy of a filled and blinded ballot containing their signature. This, however, only changes the actual vote submitted by the voter and with the full knowledge of the voter. We consider this situation to be voting by proxy, and thus do not consider this as fraud. Furthermore, voters cast their votes independently of each other. Thus, unless all voters are colluding together, a very unlikely situation in any large-scale election, they cannot alter or affect the overall outcome of the voting process.

A final situation is one in which a voter decides to not participate in the voting process, or only partially completes the voting process. Both of these scenarios are covered by the un-authorized proxy property, and have already been shown to not allow fraud in the previous sections.

6. Future Work

The voting protocol and eVoting system presented here are full featured, and could easily facilitate an election. However, there are few improvements that should be addressed in the future. We will list these here, and describe the future direction of the system.

6.1 Graphical User Interfaces

The implementation provided here consists of simple text-based command line interfaces for both the voter as well as administrators of the authoritative entities. While the command line interface is easy to understand, and can provide the full functionality required by the eVoting system, we suggest the implementation of a graphical user interface, especially for the voting client.

The graphical user interface should be very simple, presenting the minimum amount of information. It should provide for all current features of the system, and could extend the current capabilities by providing more descriptive actions for moving through the voting process. Additionally, we believe that the graphical user interface will be much easier for less savvy voters to use, and will provide them the confidence needed to feel safe and secure voting via the system.

6.2 Anonymous Channel Integration

The eVoting system, as presented here, requires the voter to run a separate piece of software, namely the TOR client. This limitation was due to the current state of the TOR project. Libraries and documentation were not readily available to build the TOR client functionality into the voting client.

Future iterations of the eVoting system should provide transparent anonymous channel mechanisms by integrating the anonymous channel capabilities into the voting client. The voter should not even need to be aware of the existence of the anonymous network client. This will help reduce any usability issues and run-time errors that could be generated from the improper use of the stand-alone TOR client.

6.3 Distributed Architecture

The eVoting implementation and voting protocol given here uses one single instance of each of the five different voting authorities. While this is sufficient for small-scale elections, such as department or company-type scenarios, we feel that a more robust architecture would be needed for larger-scale elections. Due to the nature of the voting protocol, we do not feel that it would be infeasible to implement multiple instances of the authoritative entities. These could be geographically located to service a widespread population of voters.

6.4 Certificate Revocation Lists

The current system does not provide for Certificate Revocation Lists (CRL). A CRL is typically used in public key cryptography certificate-based systems as a means of blacklisting certificates that are either invalid, have expired, or need to be revoked for some reason. The CRL is maintained by the CA, and distributed to any entity that uses the CA for certificate revocation tasks. Thus, any entity that receives a certificate that is on the CRL would know that the certificate should not be trusted.

6.5 Extending BDL

The Ballot Description Language (BDL) that was developed provides a means to express simple ballots in XML. Simple ballots consist of a number of questions, where each question has a predefined set of answers. The BDL does not provide the capability for single party votes, where a voter votes for her chosen party, thereby automatically selecting the parties candidates in each question. BDL also lacks support for varying ballot schemas based on geographic location and precincts. Thus, BDL must either be extended to support the full list of requirements for national election, or a new representation must be chosen. A formal OASIS standard is currently developing, the OASIS Election Markup Language [22], which provides full support for the aforementioned requirements, and is in current use in Europe.

7. Conclusion

In this paper, we have presented an implementation of a secure and anonymous voting protocol - the eVoting system. This system, based on previous work by Ray et al. [39], provides for all the necessary conditions required by any electronic voting system. First, accuracy is given by the fact that votes cannot be modified without detection, only valid votes are counted in the final tally, and invalid votes are rejected. Secondly, the eVoting system enforces democratic constraints by allowing only eligible voter to vote, and ensuring a voter may only cast one vote. Thirdly, the system protects the privacy of the voter by ensuring that no entity can link a voter to his cast vote through the use of blind signatures and anonymous submission channels. Fourthly, the system provides a level of verifiability beyond that which is provided by our current national voting systems in that a voter can verify their vote was received and counted correctly, and any entity may audit the final tallies by computing and verifying them via the final published list of cast ballots. Finally, the system ensures that no entity involved in the voting process may cast a forged vote for a voter not participating in the voting process.

We have presented an implementation based on open-source and freely available software. The protocols developed do not rely on any proprietary technology or communication channels. Basic HTTP transport protocols are used for a majority of the communication during the voting process, thus ensuring our system will work in a wide variety of operating environments. Furthermore, the TOR network is used when casting a vote to ensure the voters privacy. TOR is a well-established mix network with nodes operating worldwide.

While the protocol has been shown to be secure and complete, there are two current issues that must be addressed before this system can be deployed and used in actual wide-scale elections. First, the eVoting system and underlying protocol do not prevent vote buying. In order to prevent vote buying, a voter must not be able to prove the manner in which they voted. In the eVoting system, the voter generates a unique voter mark and includes this voter mark in their cast ballot. Thus, a one-to-one correspondence can be established from a voter mark to their cast ballot; this allows the voter to prove that he or she voted a particular way.

Secondly, our system is vulnerable to fraud if VAA and VC collude and the voter obtains certification on his ballot but fails to cast it. In this scenario, the set of cast ballots F will contain fewer ballots than the set of certified ballots S', $|F| < |S'|$. Since the protocol relies on a one-to-one

correspondence between the members of S' and F, CA and VAA could collude and generate a set of $|S'|-|F|$ ballots and cast them. These ballots would contain the signature of VAA, and a valid voter mark. In this situation, the fraud cannot be detected. However, following the reasoning in Section 5.2.4, we can prove fraud. A $F_i \in F$, a $S'_i \in S$, and a $S_i \in S$ must be produced such that S'_i and S_i bear the signature of an eligible voter and F_i has been generated from the voter mark in S'_i and S_i. While these items can be produced, the voter must be involved to present the random number used to generate the voter mark from the serial number given by BD. Since the voter did not cast his vote after certification, he will then be made aware that a spurious vote has been cast using his identity and fraud can be proven. We would suggest the investigation of anonymous commit protocols that can ensure that the certification and casting either both occur successfully, or neither occurs, thus preventing the situation presented here.

Bibliography

[1] Josh C Benaloh and Moti Yung. Distributing the power of a government to enhance the privacy of voters. In *PODC '86: Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 52–62, New York, NY, USA, 1986. ACM.

[2] Colin Boyd. A new multiple key cipher and an improved voting scheme. In *EUROCRYPT '89: Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology*, pages 617–625, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[3] RABA Innovative Solution Cell. Trusted agent report: Diebold accuvote-ts voting system. http://www.raba.com/press/TA_Report_AccuVote.pdf, January 2004.

[4] D. Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology - CRYPTO '82*, pages 199–203. Springer-Verlag, Berlin, 1983.

[5] D. Chaum. Elections with unconditionally-secret ballots and disruption equivalent to breaking rsa. In *Lecture Notes in Computer Science on Advances in Cryptology-EUROCRYPT'88*, pages 177–182, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[6] D. Chaum. Secret-Ballot Receipts: True Voter-Verifiable Elections. *IEEE SECURITY & PRIVACY*, pages 38–47, 2004.

[7] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, 1981.

[8] Steven Cherry. Making every e-vote count. *Spectrum, IEEE*, 44(1):13–14, Jan. 2007.

[9] J.D. Cohen and M.J. Fischer. A robust and verifiable cryptographically secure election scheme. In *26th IEEE Symposium on Foundations of Computer Science*, pages 372–382. IEEE Computer Society, October 1985.

[10] Compuware Corporation. Direct recording electronic (dre) technical security assessment report. http://www.sos.state.oh.us/sos/hava/files/compuware.pdf, November 2003.

[11] Science Applications International Corporation. Risk assessment report: Diebold accuvote-ts voting system and processes. http://www.dbm.maryland.gov/SBE, September 2003.

[12] Ronald Cramer, Matthew Franklin, Berry Schoenmakers, and Moti Yung. Multi-authority secret-ballot elections with linear work. *Lecture Notes in Computer Science*, 1070:72–??, 1996.

[13] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. *Lecture Notes in Computer Science*, 1233:103+, 1997.

[14] L. Cranor and R. Cytron. Design and implementation of a practical security-conscious electronic polling system. Technical Report WUCS-96-02, Washington University, 1996.

[15] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: design of a type III anonymous remailer protocol. *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 2–15, 2003.

[16] George Danezis and Claudia Diaz. A survey of anonymous communication channels. Technical Report MSR-TR-2008-35, Microsoft Research, January 2008.

[17] R. DeMillo and M. Merritt. Protocols for data security. *Computer*, 16(2):39–51, 1983.

[18] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13 table of contents*, pages 21–21, 2004.

[19] Ariel J. Feldman, J. Alex Halderman, and Edward W. Felten. Security analysis of the diebold accuvote-ts voting machine. In *EVT'07: Proceedings of the USENIX/Accurate Electronic Voting Technology on USENIX/Accurate Electronic Voting Technology Workshop*, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[20] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[21] Joe Foley. Torlib. http://web.mit.edu/foley/www/TinFoil/.

[22] Organization for the Advancement of Structured Information Standards (OASIS). Election markup language. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=election.

[23] Apache Software Foundation. Apache tomcat. http://tomcat.apache.org/.

[24] Atsushi Fujioka, Tatsuaki Okamoto, and Kazuo Ohta. A practical secret voting scheme for large scale elections. In *ASIACRYPT '92: Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques*, pages 244–251, London, UK, 1993. Springer-Verlag.

[25] I.A. Goldberg. *A Pseudonymous Communications Infrastructure for the Internet*. PhD thesis, UNIVERSITY of CALIFORNIA, 2000.

[26] D. Goldschlag, M. Reed, and P. Syverson. Onion Routing for anonymous and private internet connections. *Communications of the ACM*, 42(2):39–41, 1999.

[27] David M. Goldschlag, Michael G. Reed, and Paul F. Syverson. Hiding routing information. In *Information Hiding*, pages 137–150, 1996.

[28] C. Gülcü and G. Tsudik. Mixing E-mail with Babel. *Network and Distributed System Security, 1996., Proceedings of the Symposium on*, pages 2–16, 1996.

[29] Kenneth R. Iversen. A novel probabilistic additive privacy homomorphism. In *International Conference on Finite Fields, Coding Theory and Advances in Communications and Computing*, 1991.

[30] Kenneth R. Iversen. A cryptographic scheme for computerized elections. In *CRYPTO '91: Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, pages 405–419, London, UK, 1992. Springer-Verlag.

[31] S. Katti, D. Katabi, and K. Puchala. Slicing the Onion: Anonymous Routing Without PKI. *ACM HotNets*, 2005.

[32] T. Kohno, A. Stubblefield, AD Rubin, and DS Wallach. Analysis of an electronic voting system. *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 27–40, 2004.

[33] Sun Microsystems. Java servlet technology. http://java.sun.com/products/servlet/index.jsp.

[34] Sun Microsystems. Javaserver pages technolog. http://java.sun.com/products/jsp/.

[35] U. Möller, L. Cottrell, P. Palfrader, and L. Sassaman. Mixmaster ProtocolVersion 2. *Draft, July*, 2003.

[36] M. Naor and A. Shamir. Visual cryptography. *Lecture Notes in Computer Science*, 950(1):1–12, 1995.

[37] Hannu Nurmi, Arto Salomaa, and Lila Santean. Secret ballot elections in computer networks. *Comput. Secur.*, 10(6):553–560, 1991.

[38] S. Parekh. Prospects for remailers-where is anonymity heading on the internet. *First Monday*, 1(2), 1996.

[39] I. Ray, I. Ray, and N. Narasimhamurthi. An anonymous electronic voting protocol for voting over the internet. In *Third International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems*, 2001.

[40] M.G. Reed, P.F. Syverson, and D.M. Goldschlag. Anonymous connections and onion routing. *Selected Areas in Communications, IEEE Journal on*, 16(4):482–494, May 1998.

[41] Michael K. Reiter and Aviel D. Rubin. Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[42] Election Data Services. 69 million voters will use optical scan ballots in 2006, 66 million voters will use electronic equipment, 2006.

[43] P. Syverson, G. Tsudik, M. Reed, and C. Landwehr. Towards an Analysis of Onion Routing Security. *Designing Privacy Enhancing Technologies: Workshop on Design Issue in Anonymity and Unobservability*, pages 96–114, 2000.