Acknowledgments

The code for this project was developed by Mark Herschberg, Ben Adida, and Randy Milbert, under the supervision of Prof. Ron Rivest. I would like to thank them all for their contributions. Ben's knowledge of Java proved invaluable to the project. He created the server's basic functionality and is credited with the wonderful interface it has. Ben also developed the lower layers of the communications protocol. Randy implemented such crucial components as the Blowfish algorithm, a terrific random number generator, and the ballot. Ron's supervision was helpful throughout the development process.

We would also like to thank DARPA, which sponsored our research under grant number DABT63-96-C-0018, and Dr. Karger, for the use of his machine during the coding.

Chapter 1

Introduction

1.1 Background

In 1869 Thomas Edison received US patent 90,646 for an “electronic voting device.” He tried to sell his invention to the Massachusetts legislative bodies, unsuccessfully. A century later, we are once again attempting to apply electronic wizardry to expedite the democratic process.

It seems as though everything is being automated by computers today. With the recent explosion of growth on the world wide web, the ability to communicate more information faster and cheaper is at our fingertips. We have email, electronic newspapers, and video conferencing all leading the trend towards a paperless society.

Elections themselves have not remained completely static. Absentee ballots have long been common. This idea was extended in April,1997, when Monterey County, California experimented with the first voting by mail (VBM) system. Additionally, Direct Recording Electronic (DRE) systems have been used in polling stations since the 1970s. In DRE booths, unlike their mechanical counterparts, the tallies are stored electromagnetically. [Kir95]

Thanks to the recent advances in the field of cryptography we can bring all these trends together and create a secure electronic voting system. Our system, E-Vox, combines the flexibility of a VBM with the speed and power of modern day computers. Although some philosophers may disagree, we take the value of these properties to be self evident.

1.2 History

The cases listed above are, unfortunately, exceptions, rather than the rule. The fact is, progress in the field of electronic voting has moved very slowly over the last hundred years. It was not until cryptography become a field of public interest, as opposed to one used exclusively for military purposes, did the first protocols begin to surface.

1.2.1 Properties of a Secure Secret Voting Scheme

Fujioka et al. defines seven requirements of a secure, secret election.

1. Completeness: All valid votes are counted correctly.

2. Soundness: The dishonest voter cannot disrupt the voting.

3. Privacy: All votes must be secret.

4. Unreusability: No voter can vote twice.

5. Eligibility: No one who isn't allowed to vote can vote.

6. Fairness: Nothing must affect the voting.

7. Verifiability: No one can falsify the result of the voting.

Curiously, the above properties are often taken for granted in the US electoral system. After a voter drops the ballot in the box, he goes home and awaits the return. Certainly the need for a photo ID and a signature provides some authentication, but that's the extent of it. Perhaps his vote was secretly recorded; maybe the tallies from his booth were not reported correctly; it is possible that the election officials stuffed the ballot box when he wasn't looking. Most systems rely heavily on our trust in the government appointed officials overseeing the election.

There is significantly less trust when computers are involved. This is not without some justification. Anyone who has spent any time working on a computer has seen it crash.

Bugs are an accepted part of most software. Recently it seemed as though every other week college students were discovering security flaws in supposedly secure web browsers. These problems, coupled with the general public's lack of understanding about computers, makes for a very distrustful populous. (In defense of computers, though, US elections do not exactly have a spotless record either.)

Before any computerized balloting system can be accepted it must stringently meet the above requirements. There are also four additional properties a system may, or may not, possess.

8. Receipt-Freeness: The voter does not need to keep any record of his vote.

9. Non-Duplication: No one can duplicate anyone else's vote.

10. Public Participation: Everyone knows who did, and did not, vote.

11. Private Error Correction: A voter can prove his vote was miscounted without revealing how he voted.

(Properties 9, 10, and 11 are taken from Schneier [Sch96].)

The value of these four properties really depend on the values of the specific society. Most elections require the user to record some or all of his ballot, to later verify that it was counted correctly. This receipt also allows voters to easily sell their votes. The buyers can use the receipts to verify the way in which bought voters voted. [Gen96] (ckme)

Duplication does not seem to be such a significant problem in most real elections. Schneier gives the example of a three person election between Alice, Bob, and Carol. If Alice does not care about the outcome of the election, she can simply duplicate Bob's vote. The winner is then the person for whom Bob voted (because Alice will have voted with him). Alice knowing the result of the election, effectively sees Bob's ballot. Given an election with any reasonable number of people, who are assumed to genuinely care about the outcome, this will not be an issue.

The property of public participation is one which may or may not be of value to a society. Private error correction is one which most societies would probably consider useful.

1.2.2 Secure Election Strategies

There are three main approaches to secure electronic elections. Within each of these categories there are a number of variants, each trading complexity for functionality.

Self-Adjudicating Protocols

The most basic protocols require no external parties. Security is created by multiple layers of encryption and/or signing. Anonymity is generated by repeated reordering of the votes during various steps of the algorithm. Michael Merritt designed one of the earliest schemes. [DeM82] All voters need to have public keys which are assumed to have been distributed before the election beings.

In this scheme, each voter performs the following steps, as summarized by Schneier.

1. He attaches a random string, R, to his vote, V.
2. Then he encrypts his vote with public keys of Voters 1 through N, in that order.
3. Again, he repeats step two, but this time includes a random string within each layer of encryption.

At this point the votes look like:

$$E_N(R_N,E_{N-1}(...(R_2, E_1(R_1, E_N(E_{N-1}(...(E_1(V,R))...))))...))$$

where R_i is random string of voter i, and E_j is the encryption of the parenthesized expression using Voter j’s public key.

4. All votes are passed from voter to voter, starting with voter N and ending with Voter 1. Each voter decrypts the message and strips off the random string, making certain it is the one he had used. The voter then scrambles the votes and sends them onto the next voter (with Voter 1 sending the votes on to Voter N).

Now only the inner encryptions remain.

$$E_N(E_{N-1}...(E_1(V,R))...)$$

5. Again each Voter from N down to 1 decrypts his layer, but then signs the message and sends it on. Voter i checks the validity of the signature of voter i+1 and if it is valid decrypts, signs, and passes the message onward.

Partway through this step the votes looks like

$$S_{i+1}(E_i...(E_1(V,R))...)$$

where S_i is the signature of Voter i.

6. All voters confirm the signature of Voter 1 and check the list of votes for their initial random string to insure their vote was counted.

The number of votes is constant throughout the process and so ballot stuffing or dropping is easily detected. Votes cannot be replaced by a malicious party. An attempt to do so in the second round of decryptions (step 5) will be discovered as the signed object will not be correct. The signatures at this stage make it easy to trace back and find the malicious party.

An attempt to substitute votes during the first round of decryptions (step 4) will either be detected later in the round because the random number is incorrect; or, if Voter i substitutes for Voter j's vote, for $j > i$, then Voter j will detect it during the start of the second round of decryptions. As opposed to second round decryption substitutions, the malicious party cannot be uniquely identified in this case.

The scrambling of votes provide anonymity; and the inner random string R allows participants to insure their Vote is in the final tally.

There are quite a few problems with this scheme. If for no other reason, it is simply too computationally intensive to be useful.

Central Vote Repository

Excessive computation can be avoided by creating a Central Vote Repository (CVR). This system requires far less computational work. Again the voters are presumed to have a public/private key pair {k,d}.

1. The CVR asks each voter whether or not he will participate in the upcoming election.
2. A list of all participants is made public.
3. Each voter receives an ID number using an All-Or-Nothing-Disclosure-of-Secrets (ANDOS) protocol.
4. Each voter anonymously sends the CVR his ID number, I, along with the encryption of his vote, V, paired with his ID number.
5. The CVR publishes all encrypted votes $E_k(I,V)$.
6. After step 5 is complete, each voter anonymously sends {I,d} to the CVR.
7. All votes are decrypted and their values published alongside them.

This system prevents unauthorized voters from voting, as well as registered voters from repeatedly doing so. While the ANDOS system [Nur91] is too complex to be described here, suffice it to say that votes cannot be traced to the voter, because the

ANDOS system prevents the ID distribution center from knowing which voter got which ID.

There are some significant limitations to this system. The main drawback is that the central facility is a single point of failure or corruption. It can forge votes in the name of people who abstain (although step 1 is supposed to reduce the number of abstainers). Conversely, it can drop valid votes and claim they were never sent. The voter has no way of proving that he did submit a vote. If, instead, the CVR simply chooses to miscount a vote, the voter's only recourse is to show the triplet {I, E_k(I,v), d} at which point his vote is revealed. Finally, the ANDOS protocol is rather complex.

Simpler variations on this theme do not have all the necessary properties. More complex ones overcome some of the difficulties. [Sak94]

Multiple Voting Organizations

In the spirit of checks and balances, the next improvement to election schemes comes from using two centers, instead of one. Now, instead of a single CVR, there are two entities: a Validation Agency (VA) and a Tabulation Facility (TF). A valid vote must pass through both bodies to be counted. The first recognizes the voter's right to vote, without seeing the actual ballot, and gives the voter some token confirming this authorization. The second party is then anonymously passed the validation token and the vote. This type of scheme, of course, assumes that the two groups are set up so as not to collude with one another.

1. Each voter, after providing his identity, asks the VA for an authorization number.

2. The VA randomly generates authorization numbers and distributes them.

3. The list of all such authorization numbers is given to the TF.

4. Each voter picks a random ID number and sends it, along with his vote and authorization number to the TF.

5. The TF checks the authorization number and, if it is on the list, crosses it off and publishes the vote along with the ID number.

While this protocol does not require that the voters all have public keys, all communication needs to be encrypted in some manner. Additionally the voter needs to prove his identity to the VA somehow. Finally, an anonymous channel is again needed by the voter to communicate with the TF.

This type of system has some notable improvements over the previous two examples. Neither body, by itself, has enough information to link a specific ballot with a voter. The VA's validation numbers prevent both unauthorized voters from participating and valid voters from voting repeatedly.

Still, this system is not perfect. The VA can create false voters and vote in their name. Collusion between the VA and TF can break the system. Despite these flaws, however, this system is a good starting point, and the Fujioka et al. system we implemented is an improvement on this theme.

1.3 Motivation

1.3.1 Impediments to Development

There have only been a handful of papers in the area of secure electronic voting. The result is that there have been only a handful of implementations. This is likely due to societal needs. Electronic data transfers and communications are commonplace in today's world. Business, government, and military applications all require strong, efficient, encryption and authentication schemes. Elections, on the other hand, are occasional events; and not very popular ones in places like the United States with only a 55% turnout rate for presidential elections. It is no surprise that most cryptographic research has been devoted to mainstream topics such as data encryption and public key systems, whereas electronic voting is considered by experts like Bruce Schneier to be an “esoteric protocol.”

However, as the use of the internet increases, we expect work in this field to grow. Elections are really just a special case of secure multiparty computation, [Kil90] in which a group of people wish to perform a calculation together, perhaps each using personal data, without revealing their individual data. Contract bidding, negotiations, and aggregate demographic data, just to name a few examples, are related computations which we can expect to see calculated on the internet in the future. Our work will be relevant to these areas, as well as general security developments. At the very least, it is, to our knowledge, the first secure electoral system built under Java.

This lack of need for secure elections has resulted in a rather large gap between theory and practice. Although the algorithms are sound, certain assumptions they make are far from trivial to implement, as we’ll see in the case of E-Vox, based on a paper by Fujioka et al. (2.2.1)

There are also a number of logistical problems. Two parties desiring a secure communication channel, can, together, develop their own program and each take a copy. An election usually has a much larger base of participants, each of whom will need to receive the code. Whereas a physical meeting could be used to verify the authenticity of the code in the former case, that is unfeasible in the latter. The code must also be trusted by the voters not to perform any malicious activities, such as erasing the hard drive, or recording the keystrokes and forwarding them to someone wishing to defeat the anonymity of the election. Again, this is easier among a small group of people such as in the former case.

Finally, there are a number of policy issues that come with elections. By policy we mean problems which cannot be solved by cryptography alone. The fundamental problem of key distribution becomes a registration problem in this case. Throughout this implementation there will be a number of places in which the problem is left open to the users of the system.

1.3.2 Previous Work

Despite the obstacles listed above, a number of universities have begun supporting electronic voting, in the past few years, often for student elections. These systems, however, all have some drawbacks.

Pericles (MIT)

At MIT, student government elections are held both electronically and on paper during disjoint periods. The computer-based voting system, Pericles was developed by Paul Kirby. [Kir95] It is a C based system, which runs over Mosaic.

There are two drawbacks to this system. First, it relies on the Kerberos system for authentication and message encryption, making its application to the general public limited. Second, it is a single server system. Although it is set up to protect the privacy of the students and insure a fair election, anyone with access to the server can defeat the system. Despite this cryptographic dependency, Pericles does have a very good user interface, and works extremely well for the type of election for which it was designed.

Princeton

The scheme at Princeton, developed by Ben Davenport, Alan Newberger, and Jason Woodward is also an implementation of the Fujioka, et al. [Dav96] The details of this system will be discussed fully in section 2.2.

We believe our implementation has certain advantages over theirs. Their implementation is written in Perl and uses a web based interface. The downside to this is that the web browser must connect to a server to perform the actual computations required of the voter. This system is vulnerable to a spoofing attack from which their suggestion of using SSL will not protect them (see section 6.3.4). This server also must be trusted to reliably perform a step critical to the voting process, hours after the voter has interacted with the system (see section 2.2.2).

The servers themselves are limited in that they can only handle one connection at a time. The messages passed between them, although all encrypted with PGP, are subject to all sorts of timing analysis, which could defeat the anonymity of the system.

Sensus

Lorrie Cranor and Ron Cytron have implemented Fujioka et al., as well. [Cra96] Again, we believe our system has certain benefits over theirs. First, they employ a three stage protocol. That is, the voter must perform three separate actions, two of which are absolutely necessary for their vote to be tallied. Our implementation requires only two actions on the part of the voter, only one of which is necessary (see section 2.2.2).

Second, their system is written in C and Perl, and makes use of CGI scripts. This has the code logistical problems mentioned earlier (see section 1.3.1). In the environment of a college campus (or even business), where the computers are maintained by a vigilant and knowledgeable group of system administrators, this is not a problem. However porting it across intranets, so otherwise disjoint groups can participate in the same election still is a problem.

Finally, it presumes the use of a public key system for all voters. Again, this may not be a problem on a given intranet, but will be in the general case.

1.3.3 Goals

With the above systems in mind, our goal was to develop a secure, user-friendly, stand-alone system for a small scale election.

Secure

We define secure to mean meeting the seven requirements as listed by Fujioka et al.

User-Friendly

The system was designed to be completely user-friendly. By this we mean that the voter himself needs to perform the bare minimum number of actions required of any electoral process and no more. The two steps absolutely required of any election are registration and voting. Note that our requirement is procedural rather than computational. There is no constraint on the computation, save the implied limit of it being finished within a reasonable amount of time. Much of the literature has been devoted to improving the efficiency of elections, but that is not one of our goals.

From the voter's standpoint, both activities are performed quickly and easily. To register, the voter must go to the appropriate registration office, prove his identity (the process by which this is done is left as a matter of policy), and then simply enter his name, or any unique ID, and password into a program (both of which he needs to remember).

When it comes time to vote, he can simply download the applet using a web browser, enter his name and password, and then click a few buttons. At that point, the voter can walk away from the process knowing he has completed the act of voting, just as if he had stepped out of the election booth. (As explained in section 2.2.2, because the user interface was such an important goal of this system, we did significantly alter the security model of the protocol to achieve it, but we still meet our definition of secure nonetheless.)

Both registration and voting programs take on the order of seconds to complete, and the user interface is self-explanatory.

Stand-Alone

Our system was designed to be a stand-alone system. Some current systems (e.g. Pericles) assume certain structures are in place, such as key-distribution systems. We make no assumptions of this sort. All we require are some servers and a web browser which supports the JDK1.1 (Java Development Kit). Specifically, the user (voter) does not need to download any code ahead of time; and because Java applets are run inside a sandbox, there is no need to worry about system security.

Size

E-Vox can easily support elections on the order of a hundred people. With sufficiently fast servers, a few thousand can be quite reasonable. Moreover, the system can be scaled further without significant effort. The limit is really the speed and size of the servers, and the bandwidth of their connections. There is also a slight bottleneck in terms of file system latency. Currently, each registered voter is stored in a separate registration file. With the release of the JDBC (Java Database Connectivity), a database backend can be set up to more efficiently manage a larger number of people. We claim that this system can potentially handle tens of thousands of voters. Note that for any sufficiently large system, separate elections still need to be held, much like the way in which current US federal elections are broken into a number of small districts.

Chapter 2

Theory

2.1 Cryptographic Elements

Election protocols are built from a number a low level cryptographic structures. These structures, alone or in combination, create the various properties we desire in election systems. The reader is presumed to have a basic understanding of public key cryptosystems. [Dif77]

2.1.1 Digital Signatures

Digital signatures are intended to be the electronic analog of written signatures. That is, some object “attached” to another (say a document or file), undeniably associating it with the signer. The signature must have three properties. First, it must be unique; the signatures of different parties must be different. Second, the signature must not be forgeable, Alice cannot create Bob’s signature. Third, the digital signature needs to be verifiable, so anyone can confirm the authenticity.

E-Vox implements digital signatures using the RSA public key cryptosystems. [Riv78] They work as follows. Alice, with public key e, private key d, and modulus of n, signs an object, M, by encrypting it with her private key.

$$S = M^d \bmod n$$

Because she is the only person who knows her private key, she is the only person who can create the signature, S, for this object, M.

Anyone can verify that S is indeed her signature of M by encrypting S with her public key.

$$M = S^e \bmod n = (M^d)^e \bmod n = M^{de} \bmod n = M \bmod n$$

2.1.2 Blind Signatures

Our voting system's need for signatures has an additional constraint in that it must be a blind signature. To illustrate what a blind signature is, suppose Bob doesn't trust Alice. He does, however, trust Trent, who trusts Alice. Bob is willing to accept a message from Alice only if Trent signs it. Unfortunately, the message is of a rather personal nature and Alice doesn't want Trent to see it. She can use a blind signature.[Cha82]

Schneier provides a good analogy of the solution. In the real world, Alice can seal her message in an envelope filled with carbon paper. Trent can then sign the outside of the envelope and his signature will get transferred to Alice's message, without Trent ever actually seeing it. Alice can then remove the message from the envelope, and give the signed message to Bob, who can verify Trent's signature.

The cryptographic version is as follows. Trent has a public key e, private key d, and modulus n. First Alice blinds her message, M, using a random value, k, between 1 and n.

$$B = Mk^e \bmod n$$

Then Trent signs it by

$$S' = B^d \bmod n = (Mk^e)^d \bmod n = M^d k \bmod n$$

Alice can unblind this to yield Trent's signature of M as

$$S = (S'/k) \bmod n = M^d \bmod n$$

This can be verified as it is now a normal digital signature.

2.1.3 (One-Way) Hashing

A one-way hash is a mathematical function. We say h is the hash of M for hashing function H

$$h = H(M).$$

A one-way hash has the following properties. For any size M (or any size within a given range of sizes), the size of the output, h, is constant. The inverse, H^{-1}, is hard to compute, such that given h and H, it is hard to find any M such that H(M) = h. Given a message M, it is hard to find another message M’, such that H(M) = H(M’). Finally, it is hard to find two messages M, M’ such that H(M) = H(M’).

We employ a version of SHA, the Secure Hash Algorithm, designed by the NIST and the NSA. Specifically, we use SHA-1. SHA takes any input up to 2^{64} bits in length and produces 160 bit output.

Hashing is often used in conjunction with digital signatures. The signing of large documents can be computationally expensive. Because the hash, sometimes referred to as the fingerprint, of an object is unique, and often of a shorter length than the original object, it is the hash of an object which is often signed, rather then the object itself. For a given object and signed hash, the object itself can be hashed by the recipient, and then compared to the signed hash after the “un-signing” function has been applied to it.

2.1.4 (Blind) Commitment

A commitment is a way in which one party can commit to an object (e.g. string of bits, message, contract) without the anyone else seeing what that object really is. However any attempt by the first party to change the object can be detected.

A good analogy would be for Alice to lock away a message in a safe requiring two keys, which she would then split between her and Bob. Alice cannot open the safe to change the message, and Bob cannot open the safe to see the message, without the help of the other.

A few ways to commit to a bit pattern are available. In our protocol we use one-way hashing to insure bit commitment. Alice generates two random bit strings, R_1 and R_2. She then hashes those, along with her message, M, and sends it to Bob with one of the keys, say R_1.

$$C = H(R_1, R_2, M)$$

Bob cannot compute M from C because of the properties of the hashing function, H. Similarly, Alice cannot not find another message and bit string pair (M', R') such that

$$C = H(R_1, R', M')$$

and so she cannot change her message to M' without detection by Bob. By keeping R_2 secret, Alice prevents Bob from hashing every possible string, along with R_1, to try and find the message to which Alice committed. (This type of attack is known as a dictionary attack.)

Specifically, we use

$$h= SHA(k1, SHA(message, k2))$$

In the above equation, the message is actually hashed twice. During each hash, a key is first appended, and then prepended, to the object being hashed. In our application, the keys are used to add random bits to the hashed message.

2.1.5 Anonymous Channels

The final piece needed by our system is an anonymous channel. This is not necessarily a purely cryptographic beast. An anonymous channel is simply one in which Alice can send a message to Bob without revealing her identity (some channels allow Bob to reply to Alice, while maintaining Alice's anonymity). [Anon]

Anonymous channels will be discussed in more detail later in section 3.2.

2.2 "A Practical Secret Voting Scheme for Large Scale Elections"

2.2.1 Core Protocol

E-Vox is based on "A Practical Secret Voting Scheme for Large Scale Elections" by Fujioka, Okamoto, and Ohta [Fuj93]. The paper itself is quite concise. It gives a mathematical framework for a secure election. However, many details needed for a full implementation were left out (see section 3.2).

The protocol is similar to the third scheme listed in section 1.2.2. It involves an administrator, a counter, and the voter. Like most election protocols, it requires an anonymous channel.

The administrator is responsible for rubber stamping ballots. That is, after a voter proves his identity to administrator, the administrator will sign the committed, blinded ballot it is given. Then the voter will be removed from the list of those eligible to vote. At the end of the protocol, the administrator will publish a list with the committed, blinded ballots, those ballots when signed, and the voters to whom they were given.

The voter will use this signature as proof of eligibility with the counter. The vote is sent to the counter through an anonymous channel. The counter has no way of matching the ballots it receives to any voter. The counter does, however, know to count the votes, because they have the administrator's signature.

The vote is actually sent in two parts. First the committed ballot signed by the administrator is anonymously passed to the counter. While the counter knows the vote is valid, it cannot break the commitment scheme to actually see it. Rather it must wait for the keys to uncommit the vote to be sent through a second anonymous channel. At the end of the protocol, a list of the committed ballot, the administrator's signature of it, the keys used to uncommit it, and the actual ballot are publicly posted.

The steps listed in Fujioka, et al. are as follows.

1. The voter selects his candidates and commits to this ballot.
2. This committed ballot is then blinded and signed by the voter. It is then sent to the administrator.
3. The administrator verifies the right of the voter to vote, and the signature of the blinded vote. If the signature is valid, the administrator signs the committed, blinded ballot, returning this signed ballot to the user, and publishes its log.
4. The user unblinds the ballot, and verifies the administrator's signature, which, because of the blinding properties, should still be valid for the committed (but no longer blinded) ballot.
5. The committed ballots, now signed by the administrator, are then sent, through an anonymous channel, to the counter which publishes it along with an index number.
6. After all the committed votes have been sent in, the voters can confirm that their vote is listed, and that all votes have valid signatures.
7. After everyone has had a chance to confirm the entries in the counter's published list, each voter sends in the keys needed to uncommit his vote, along with the index of the committed vote. Again the communication is through an anonymous channel.

8. The counter then adds to the published list the keys and the uncommitted votes (which can be confirmed to match the committed votes).

A list of all voters who have had their vote signed is published by the administrator. This list includes their name, blinded ballot, and its signature. The counter's intermediate published list (from step 5) has the committed (unblinded) ballot and its signature. The final published list of the counter contains the values form the intermediate list as well as the keys used to uncommit, and the uncommitted (plaintext) vote, itself.

2.2.2 Modifications to the Protocol

The core protocol above does not quite satisfy our design requirements. Specifically, it fails to be user-friendly. Fujioka, et al. requires the voter to perform three steps.

1. Get the administrator to sign the vote and send it to the counter
2. Check that the vote is listed by the counter, confirm any of signatures listed, and, if everything appears on the level, send in the keys to uncommit.
3. Confirm that all votes were uncommitted and counted correctly.

While a program can easily do any of the steps, it still involves a button click on the part of the user. People want to drop the ballot in the box and go home. An election of this nature might require 3 days to run, one for each step, an unappealing prospect to many potential users of the system.

In theory, during step one, the executables for the later steps could be set up to sleep in the background of a computer, and run at a later time without further effort on the part of the voter. However, this requires foreign code to be downloaded onto a workstation.

There are two problems with this approach. First, the code must be trusted not to hurt the system (or be confined to a sandbox which must then be maintained until the code is finished and removed). Second, the code must be safely stored until it executes; on a public workstation, someone could come along and delete the sleeping programs before they execute the remaining steps.

There may be a social solution to this problem. Prof. Rivest cleverly suggests that as electronic voting becomes commonplace, people will entrust other entities to perform these later steps for them. Political parties, for example, have a vested interest in making sure keys do get sent, and signatures are confirmed, more so than the average voter. They would be likely to provide such services to their membership.

In order to meet our goal of user-friendliness, requiring a bare minimum of work on the part of the voter, our revised protocol requires only two steps. Specifically, steps one and two, listed above, get combined, so that if the voter wishes to leave at this point, he may do so knowing that the election will not be disrupted and that his vote will be counted. The voter simply yields his right to confirm that his vote was counted, and also that any other votes are correct. Others may still do so.

The justification for this stems from current election systems. As noted in section 1.2.1, after a citizen walks out of the polling station, he doesn’t think about his vote again, but rather assumes everything is on the level. The same can be done in our implementation. However, for those who desire to check, the individual votes and the tally can be confirmed.

Chapter 3

The E-Vox System

3.1 Assumptions

As noted in section 1.3.3, our system was designed as an improvement over others in that it makes very few assumptions about the environment in which it is run. That is, we do not presuppose any public key system or other basic cryptosystem is in place. We do, however, make a few basic underlying assumptions.

3.1.1 Assumptions Made by the Protocol.

1. The cryptographic systems used are hard to break.
2. Each of the following parties: voter, administrator, anonymizer, and counter do not collude with each other. They may work with other parties outside the system in an attempt to defeat it.

Cryptography

Almost all of the properties we desire in our system are achieved by cryptographic methods (e.g. blind signatures, encrypted messages). Although certain systems like RSA are unproven to be secure, conventional wisdom thinks they are. Their continued use in other applications suggests our assumption is quite reasonable.

Independence

The security of this system is based on an adversarial model. To assume collusion defeats the premise around which the system was designed.

3.1.2 Assumptions about the Physical System

The following assumptions are all that are required of a network to run E-Vox.

1. The communication channels provide a low level of data confirmation such as TCP/IP.
2. The server machines have the JDK1.1 (or better) installed.
3. The host machines run a web browser that supports the Java 1.1 (or better) and allows the (potentially signed) applet to open connections to multiple hosts.
4. The host machines are secure in that they will not explicitly maintain a record of operations performed on them.

3.2 Creating a Well-Defined System

Fujioka et al. only described a core theoretical system for voting. Many of the details needed to actually build the system were left out. There were four general issues not sufficiently addressed in their paper.

1. Authentication. Although specified, it called for authenticaion of the voter by the administrator using a digital signature scheme. We preferred a different solution.
2. Communication. No communication issues were considered. This included message interception, prevention of data tampering during transmission, and the anonymous channel itself.
3. Keys. The distribution of keys between servers is not addressed.
4. Errors. Descriptions of how to use the receipts and server logs are mentioned but no formal complaint process is specified.

In defining this system, we leave open policy issues. Such issues include registration, time, and the handling of errors. See section 6.4 for a more detailed description.

3.2.1 Authentication

The two options considered for voter identification were a public key system, suggested by the use of digital signatures in Fujioka, et al. and a password system. The former was discarded for two reasons. First, one of our highest priorities is user-friendliness. Passwords are a familiar and accepted concept, even to the non-technical user. Public keys are less familiar, and far more confusing. Additionally, the public keys needed for signing would be on the order of several hundred bits. Second, either a public key system must already be in place, or the keys must be distributed in a secure manner. The most likely form of distribution would be for voters get their keys during registration, which requires that they either remember the unwieldy number, or have some sort of secure electronic transfer available. From a non-technical user's perspective, passwords are clearly easier to manage. However, our system is a very modular one, such that you could replace the authentication sub-system with one using public keys transparently, with respect to the rest of the system.

3.2.2 Communication

In the protocol as given by Fujioka et al, simple communications (i.e. those which did not require an anonymous channel) had satisfactory cryptographic protection from a passive attacker. That is, an attacker who can view, but not alter the message.

In reality, communications are more complex, and so are the malicious parties. (See section 6.2). Any number of things could happen during a communication. In addition to eavesdropping, there could be noise on the channel, partial or total loss of a message, and partial or total message substitution. To protect against this, all communications between any two parties (e.g. the voter applet and the administrator server) are handled by the following protocol for “secure connections.” Note that we do not assume the use of SSL.

Secure Channels

Alice, the initiator, needs a random number generator, and Bob, the recipient, should have a public key. Alice sets up a connection with Bob (e.g. opens a TCP socket).

$$\begin{tikzcd}\text{Alice} \arrow[rrr, "E_s\{\text{message}, k_1, k_2, \text{MAC}\}, E_{\text{pk-B}}\{S\}"] & & & \text{Bob}\end{tikzcd}$$

Figure 3.1: Secure Message passing

We use Blowfish, a block cipher designed by Bruce Schneier, to encrypt all communications. Alice generates a session key, S, which is a random byte array used as a Blowfish key. Alice also generates two random byte arrays for padding, k_1 and k_2. These byte arrays, along with the message, are then hashed to generate a MAC (Message Authentication Code). The MAC is the HMAC-SHA hash of these values, used to ensure the integrity of the bits during transmission. [Kra97] The message, padding keys, and the MAC are then encrypted with Blowfish using the random session key generated earlier. The Blowfish key is then encrypted with the recipient’s public key (pk-B). Together these are sent to Bob, who can decrypt the second part of the transmission to get the Blowfish key, and use that to get the message and padding keys. Bob can rehash the message and keys to confirm the integrity of the communication. Note that a regular hash with a single random key would suffice.

Protection from eavesdropping is provided by the Blowfish encryption. The TCP layer guards against noise and occasional packet loss. The MAC prevents partial substitution or loss from going unnoticed. The code for sending and receiving the messages through our connections (secure or otherwise) makes use of timeout functions. Finally, the protocol itself will catch any total message substitutions.

Replay attacks, being somewhat equivalent to attempts of stuffing the ballot box, are also ineffective. The administrator will accept exactly one request per voter for a signature. If the message from the voter to the administrator is replayed, the administrator will not honor it, but rather will send a complaint tot he commissioner. All replayed messages to the anonymizer will be passed on by the server. However, the counter will remove any duplicate votes, so this attempt to vote twice will fail.

While these attacks will not succeed in the cryptographic sense, it is not good enough for our purposes. Replayed messages waste time, bandwidth, and computational power. To further guard against reply attacks (as well as more general attacks) we implement a method of blacklisting to limit messages in the system (see section 4.7.1).

Anonymous Channels

The anonymous channel was largest unspecified component. There are a number of anonymous remailers currently in use throughout the internet. [Anon] Many of these could have been the basis for our channel. In the end, the approach we took is different from most, because we can make certain assumptions about the messages being transferred and optimize our channel for it. Namely:

1. There is one message per voter.
2. The flow of information is unidirectional, meaning no information about the sender’s address needs to be saved for a reply message.
3. Messages are all approximately the same size.
4. All messages are sent during a relatively short, fixed time span.

5. Messages need only be received by the deadline, there are no chronological or other ordering requirements.

With this in mind, our anonymous channel uses a single server which works by employing secure connections. Specifically, we layer a secure connection from the voter to the anonymizer on top of a secure connection from the voter to the counter. The voter applet will take the signed object, and encrypt it, along with the plaintext vote, hash keys, and a MAC, using a Blowfish session key. The applet will then encrypt the session key with the counter's public key. These two encrypted items become the message transmitted in the secure connection to the anonymizer, layered on top of this secure connection to the counter.

The anonymizer will, up until the voting deadline, receive votes encrypted in this manner. It will store these votes without any information about their origin. Shortly after the deadline, the server will scramble these votes and send them on, en masse, to the counter. Consequently, no timing analysis can be performed to correlate messages from the voter to the anonymizer, with those from the anonymizer to the counter.

Serialization

All objects are passed through communication channels in serialized form, that is, as a string of bytes. Any object can be serialized. The structure of the object, along with its data is recorded using special demarcation bytes. The object can then easily be transmitted and deserialized on the other end of the channel. [Cor97]

3.2.3 Key Distribution

Key distribution is a traditional problem in cryptography. How can parties send keys to one another and be certain the key is valid? How can we prevent a man in the middle from switching keys during transmission?

The solution is to use a secure channel employed specifically for this purpose. The design and implementation of this secondary channel is left to the user of the E-Vox system.

Included in the E-Vox package is a RSA key generator. Before an election is run, each server must generate its own keyset. The public keys are then exchanged between servers in whatever manner is deemed appropriate. At start-up, all servers, as well as the voting applet, will have all the keys, including its own, “hardcoded” (i.e. written explicitly in the code). By exchanged, we mean passed through a secure channel set up by the servers (or their human overseers) beforehand. This could involve trusted couriers, or a previous meeting at which IDs were confirmed and cryptosystems to allow this exchange were composed and distributed. While a corrupt server could distribute incorrect keys at the outset, it would be immediately detected once the voting commences.

3.2.4 Error Detection and Response

The process outlined in Fujioka et al. does allow certain types of voting fraud to be found somewhat early in the protocol. For example, if, after the second stage, the counter had received more committed votes than the administrator had signed, it would be known before all the votes are opened and counted.

Unfortunately, with our modification, such errors will not be discovered until after the election is over. It also suggests that the voter, who we presume may vote from a private workstation, must make the effort to diagnose the error and report it to the appropriate authority. The code for the applet and servers recognizes different types of errors and automatically sends complaints to the commissioner server, which logs them. Servers also keep their own error logs in case communications are problematic.

The commissioner server itself is ultimately overseen by humans. During and after the election, they must make decisions about appropriate responses to complaints. For example, if half the voters claim an invalid signature from an administrator, they may wish to shut down and investigate the administrator server immediately. On the other hand, if one of a thousand voters claims the administrator server refused his connection for no good reason, a noisy connection might have been the culprit and the commissioners may choose to ignore the isolated incident.

3.3 The Revised Voting Protocol

Our specific steps are as follows (displayed in Figure 3.2)

1. The voter selects his candidates and commits to this ballot using HMAC-SHA (requiring two commitment keys).
2. This committed ballot is then blinded by the voter and sent to the administrator, along with the voter's name and password, using a secure connection.
3. The administrator verifies the right of the voter to vote, and the validity of his password. The administrator then signs the committed, blinded ballot, returning this signed ballot to the voter. (After the deadline, the administrator publishes a list of voter names, blinded ballots, and their signatures.)
4. The voter verifies the administrator's signature and then unblinds the ballot.
5. The signed, committed ballots, along with the (unsigned) committed ballot, the plaintext and commitment keys, are then sent to the anonymous server, using the two layered secure connection (with the counter server).

6. All votes received by the anonymous server before the deadline are then randomly reordered and forwarded, en masse, to the counter just after the deadline. (The anonymous server publishes a list, in the scrambled order, of messages it sent to the counter).
7. The counter confirms the administrators' signatures, and tallies the votes. The counter publishes a list containing the plaintext ballot, commitment keys, committed vote and signed vote.

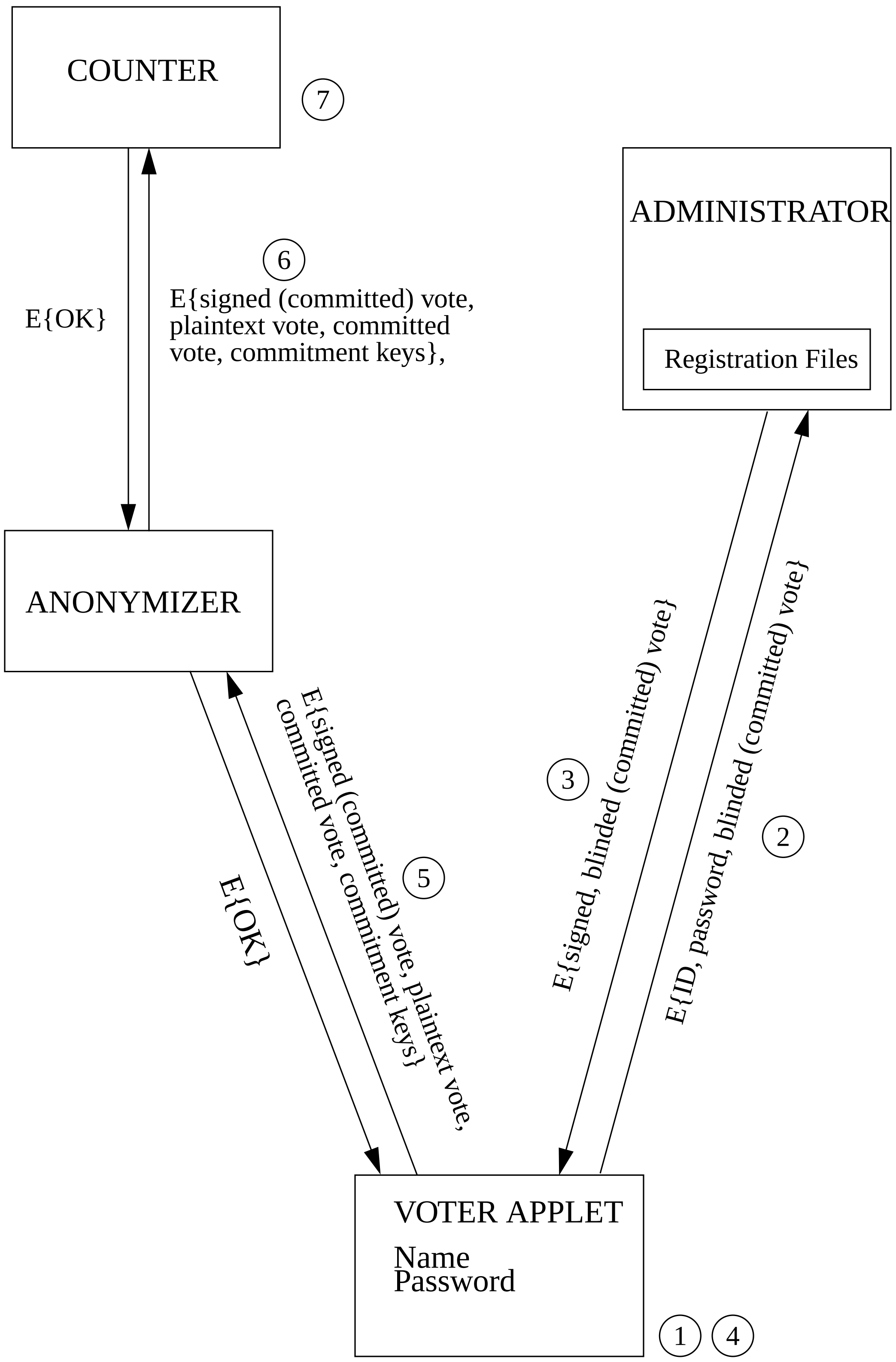

COUNTER
7
6
E{OK}
E{signed (committed) vote, plaintext vote, committed vote, commitment keys},
ADMINISTRATOR
Registration Files
ANONYMIZER
E{signed (committed) vote, plaintext vote, commited vote, commitment keys}
5
E{OK}
3
E{signed, blinded (commited) vote}
2
E{ID, password, blinded (commited) vote}
VOTER APPLET
Name
Password
1
4

Figure 3.2: E-Vox Process Chart

The voter selects candidates in all the races and creates a vote object, which contains the ballot with his choices. This ballot is then committed using a hash function. Specifically, HMAC-SHA, requiring two keys, is used. This hash, in addition to acting as a commitment, allows for smaller, and in our case, constant size, messages requiring signatures. This hash is then blinded and sent to the administrator to be signed.

The administrator verifies the right of the voter to vote, the validity of his password, and checks to see if he has voted before. If the voter is confirmed as eligible to vote, the administrator and signs the committed, blinded ballot, returning this signed ballot to the applet. After the deadline, the administrator publishes the voter's name, his (committed) blinded vote, and the administrator's signature of it.

Upon receipt of the signed vote, the applet verifies the administrator's signature, and then unblinds the signed object. The signature, because of the blinding properties, should still be valid for the committed (but no longer blinded) vote. A secure connection is created by the applet to the counter. Layered on top of this is a secure connection from the applet to the anonymous server. The applet then sends the plaintext vote, the keys used to commit, the committed vote, and the signed committed vote to the anonymizer. This is a bit redundant, because the plaintext and keys can be used to reconstruct the committed vote which can be compared with the signed committed vote. However this redundancy is useful as an extra check, especially by suspicious humans, against errors.

Note that the anonymizer and counter both send replies. Under normal circumstances they will each send an OK message. If something is awry, either server can send a complaint notice back to the sender. This reply is not required by the protocol, and so is not explicitly listed in it. However, the replies are a useful safety check. The reply is encrypted using the session key chosen by the sender. Recall from section 3.2.2 that the session key is encrypted with the recipients public key during the transmission from the sender to the receiver. Only the intended receiver should be able to decrypt the message to get the session key to create the correctly encrypted reply.

The anonymizer saves each vote in a separate file, without any information about its origin. After the deadline passes, the anonymizer sends the votes using a regular channel, in a random order, to the counter. Note that the votes are still encrypted with the counter's public key because the lower layer of the secure connection is still in place. Upon completion of the transmission to the counter, the anonymous server publishes a list of what it sent. The list is published using the same random ordering that was used to create anonymity when forwarding the votes to the counter.

The counter first removes duplicate votes. By duplicate, we mean every bit is the same, including the session keys and message encrypted with them. Having done this, the counter confirms all administrator signatures, and lists: the committed signed votes, the committed votes, the keys used to commit, and plaintext votes (as well as the final tallies).

All lists are published after the election deadline. Anyone can confirm that the signatures are valid, that no extra votes were added by a server, and that the tally is correct.

3.4 Proof of Correctness of the Revised Protocol

Because this system is only a slight modification of the protocol listed in Fujioka. et al., our proofs are based on those from the paper.

Theorem 1 (Security):

No voter or external party can prevent the election or maliciously alter the results.

Sketch of Proof:

First we examine the problem cryptographically. A simple eavesdropper cannot gain any information because everything is encrypted by Blowfish or RSA. Note that because they are randomly generated and used for a single communication between two parties, no Blowfish session key is intentionally used more than once by party involved (two parties could independently generate the same key, but this is unlikely given sufficiently large keys). A man-in-the-middle attack is prevented because the public keys of the recipients, which are known ahead of time, are used to effectively encrypt all messages.

If the message is intercepted during transmission, it cannot be altered because then the MAC will not be valid. If instead the interceptor simply blocks the receiver from getting the message, the communication channel will timeout and the sender will be alerted to a problem. (Note that this is the best we can do in the real world as well. If someone welded shut the doors to the polling station, voters could not vote then, either.)

From the standpoint of the physical system, a voter can do one of two things. A registered voter can send a bad or partial vote (e.g. no keys to uncommit). This only hurts the voter and prevents him from exercising his franchise. The other option is for a voter to send repeated votes (or external party to send garbage). In theory, the counter detects repeated ballots and discards them (as well as any invalid ballots), so they will not be counted more than once. In practice, to maintain efficiency, the number of connections per host are limited (see section 4.7.1), so the voter would have to move the vote from host to host to send a large number of message, greatly slowing down the attack.

(Repeated voting by a registered voter with a signed vote is also addressed in Theorem 3.)

Theorem 2 (Privacy):

Only if the counter and the anonymizer conspire can privacy be broken. [*Note this violates our assumption of non-collusion.*]

Sketch of Proof:

The blinded ballot given to the administrator server cannot be correlated to a specific plaintext vote, committed vote, or (administrator) signed vote sent to the counter. The administrator and counter together, along with all their published information, cannot break the privacy. Any external party would have exactly the same information because the lists are public.

When claiming an error occurred, privacy may be maintained. Given a problem during execution of the protocol, the voter has the following recourse. If the administrator's signature is invalid, the voter can show the blinded vote and its signature. (We address the problem of an administrator switching votes on the voter in section 6.2.4.) The blinded vote cannot be matched to the same committed vote, blinded differently when the voter re-initiates the protocol. If the vote is "lost" before or during the counting stage, the voter can show those values in this case, too. Again, the blinded vote (as well as its signature) cannot be traced to any other value shown.

If the anonymous server is corrupt, by itself, it can do nothing. The votes that it sees are encrypted with the counter's public key in the lower layer of the secure connection. Only if the anonymous channel is broken, that is, it is no longer anonymous because it reveals information about the origin of votes to the counter, can privacy be compromised. However this violates our assumption.

Theorem 3 (Unreusability):

Assume that no voter can break the commitment or blind signature scheme. Then the voter cannot reuse the right to vote.

Sketch of Proof:

The first vote signed by the administrator for that voter is valid. The administrator will not sign another vote given to it by the voter. The voter must change the committed ballot, while still keeping the signature valid, which it cannot do by our cryptographic assumptions.

Theorem 4 (Eligibility):

Assume no one can break the blinded signature scheme. Then a nonregistered person cannot vote.

Sketch of Proof:

For a nonregistered person to vote, he must be able to forge the administrator's signature, since the administrator will never accept one of his votes for signature. This violates our cryptographic assumption.

Theorem 5 (Recoverability):

Assuming that a voter will never lose his ballot in the process, and given that no two parties collude, a vote dropped by any party can be recovered.

Sketch of Proof:

If the counter drops a vote, the anonymous server can give its list to the commissioner who will then have the counter decrypt them and find the missing vote. If the anonymizer drops a vote, the voter can show his signed, blinded vote. If the administrator drops it (which is equivalent to refusing to sign it), the voter complains to the commissioner. If the administrator did sign it, it can show the signed vote to the commissioner.

Note that although we can recover a lost vote, we cannot determine which of two parties lost the vote. This issue is addressed in section 6.3. The assumption that the voter will not purposely lose his vote is discussed in section 6.2.1. A malicious administrator and forged voting is analyzed in section 6.2.4.

Chapter 4

Components

4.1 Cryptographic Library

When this project began, the JDK1.0 was available. During its progression, the JDK1.1 was released in various beta forms, and then in its final version. In terms of security, JDK1.1 provides much more support than its predecessor. It includes basic hashing functions, as well as key generation, support for arbitrarily large numbers, pseudo-random number generation, and object (including applet) signing. *Note: At this time, most web browsers do not yet support JDK1.1, but are expected to in the near future.*

The cryptographic library packaged with E-Vox is relatively straightforward. It includes a class/interface hierarchy for keys, encryption, decryption, blinding, and committing. The library can easily be further extended both vertically and horizontally.

While Fujioka et al. does not call for any particular cryptographic class to be used, we have implemented the necessary functions in RSA, SHA-1 (referred to as SHA), and Blowfish. Our motivation for RSA was due to both RSA's simplicity, and out of respect for the author's thesis supervisor. SHA was chosen because it is the NIST standard and provided by the JDK1.1. Blowfish was chosen because it is faster than most other algorithms in its family (although to our knowledge, no comparative speed tests have been performed in Java). However, this speed is partially offset by the large number of subkeys which must be precomputed before encryption/decryption. Also Blowfish allows for a variable length key up to 448 bits.

The base algorithms have been implemented as follows. An object to perform a specific type of function is constructed with the appropriate input. This object's methods then perform the actual computations. For instance, we might create an R*saEncryption* object called *myRsa* using an *RsaKey* object, some subset of keys (say just the public keys), or a seed from which to generate keys. To encrypt a message, we would then perform

```
RsaEncryption myRsa = new RsaEncryption(seed);
cyphertext = myRsa.encrypt(message);
```

At this level, all objects passed into, and returned from the objects' methods, are done so as byte arrays. The decision to use byte arrays instead of *BigIntegers* was made because the former are more general (easier to concatenate and use in functions such as Blowfish). However, all of the mathematical computations are done after the input has been converted into a *BigInteger.*

The hashing is specifically HMAC-SHA. The HMAC requirements do not add any extra load on the system, as the HMAC keys simply replace the commitment keys. Because we are signing the SHA of a vote, rather than the vote object itself, the size is pre-determined and therefore not too unwieldy to use in our algorithms.

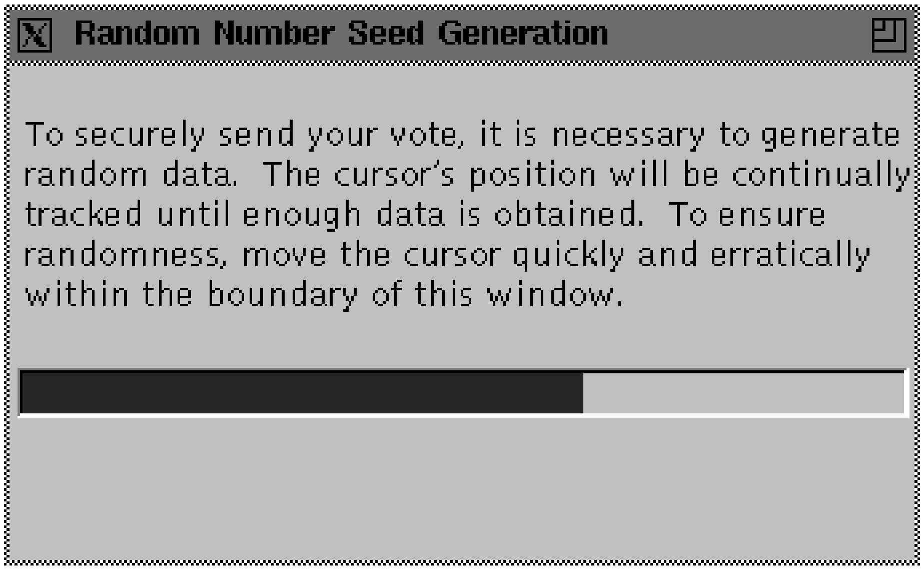
4.2 Vote Object

At the heart of the protocol is the *Vote* itself, a Java class. This object contains within it fields for the various stages the ballot goes through. This includes a field to hold the actual candidate selections (wrapped in an *Choice* object built for this purpose), as well as fields to hold the value of the *Choice* object when committed, blinded, signed, and unblinded. The object also has methods to perform those functions, given the appropriate tools (e.g. a signing method which takes an *RsaKey* object as input).

Code using the object will set the appropriate fields, serialize the Vote, and send it on to another party. We may, however, still need some of the original data later. For example, the voter will set both the committed and blinded fields, in addition to the *Choice* object. The administrator, on the other hand, only needs, and, in fact, should only be allowed to see, the values in the committed-and-blinded field. The solution is to make a copy of the object, clear the fields the administrator should not see, and send that on to the administrator server. The administrator sends back this object with an additional field, blinded-and-signed, filled in. This new field from the administrator's *Vote* object, is then copied back into the original *Vote* object created by the applet. In this manner the *Vote* object is passed from one party to the next throughout the protocol.

4.3 GenRand, the Random Number Generator

For key generation throughout the protocol, we need a source of random bits. This is achieved by the GenRand object. The various components (servers and applet) all contain an object of this type.



To securely send your vote, it is necessary to generate random data. The cursor's position will be continually tracked until enough data is obtained. To ensure randomness, move the cursor quickly and erratically within the boundary of this window.

Figure 4.1: GenRand dialog box

Just after construction, a GenRand object will be seeded. When this occurs, a dialog box will pop up (Figure 4.1) and the user will be asked to randomly move the mouse around in the window. A bar across the middle of the object indicates how many bytes have been recorded.

The mouse positions are tracked as the mouse is moved. Each position consists of a 32-bit x, and 32-bit y, coordinate. Those are cast to 8 bit bytes (that is, the higher bits are removed). These bytes are then hashed using SHA to create a seed for Java's *SecureRandom* object, a pseudo random number generator provided by the JDK. Subsequent (pseudo) random bits are generated by this Java object.

4.4 Network Connections

Java provides basic networking functionality (i.e. the java.net API). This includes the ability to create sockets and send object streams through them. Objects are also serializible, that is, transformable to and from byte arrays, which are more readily transferable. On top of this we have added our own protocols to allow for secure connections, ones which do not require the use of a SSL connection.

We offer two types of connections, simple connections and secure connections. Both are used in the protocol, though mostly the latter. A simple connection is provided directly by the JDK. A secure connection then adds a cryptographic layer on top of it (see section 3.2.2).

Specifically, a secure connection is created using the server address and port number, the RSA keys of the receiver (again, a different family of encryption functions can be used), and a *GenRand* object.

Using GenRand, we create a Blowfish session key of a specified length. This key is used to encrypt the serialized object we are sending. The session key is then encrypted using the receiver’s public key and also sent to the receiver, who can decrypt the session-key, and then decrypt the message.

The Blowfish encrypted message also carries with it a MAC (Message Authentication Code). This is used to confirm the integrity of the bit stream. The MAC itself is simply the HMAC-SHA of the plaintext message, and two random keys, which are also sent in the communication. Normally, a simple single hash is sufficient, i.e. SHA. Assuming most people will vote for candidates on the ticket, there are a rather small number possible ballots, and consequently, a small number of possible messages. The keys add some randomness to the messages to reduce their similarities.

Secure connections can be “layered” an arbitrary number of times. That is we can create a secure connection, which consists of cyphertext and an encrypted key, as listed above, and then take that pair as a new message, which is sent using another secure connection.

Note that there is no handshaking in this protocol. The sender encrypts everything, and sends it without requiring any work on the part of the recipient. (The recipient’s public key is known by the sender ahead of time.) The result is that we allow for middle men in layered, multi-party secure connections. That is, Alice can send a message to Carol via Bob in such a way that 1) Bob cannot see the messages being sent because the they are encrypted with Carol’s public key during transmission, and 2) Carol cannot match Bob’s incoming and outgoing messages with each other, because the former are encrypted with Bob’s public key, whereas the latter are not. This type of layered connection is necessary for creating the anonymous channel. If desired, handshaking can be added on top of a layer used by any two parties in “direct” connection.

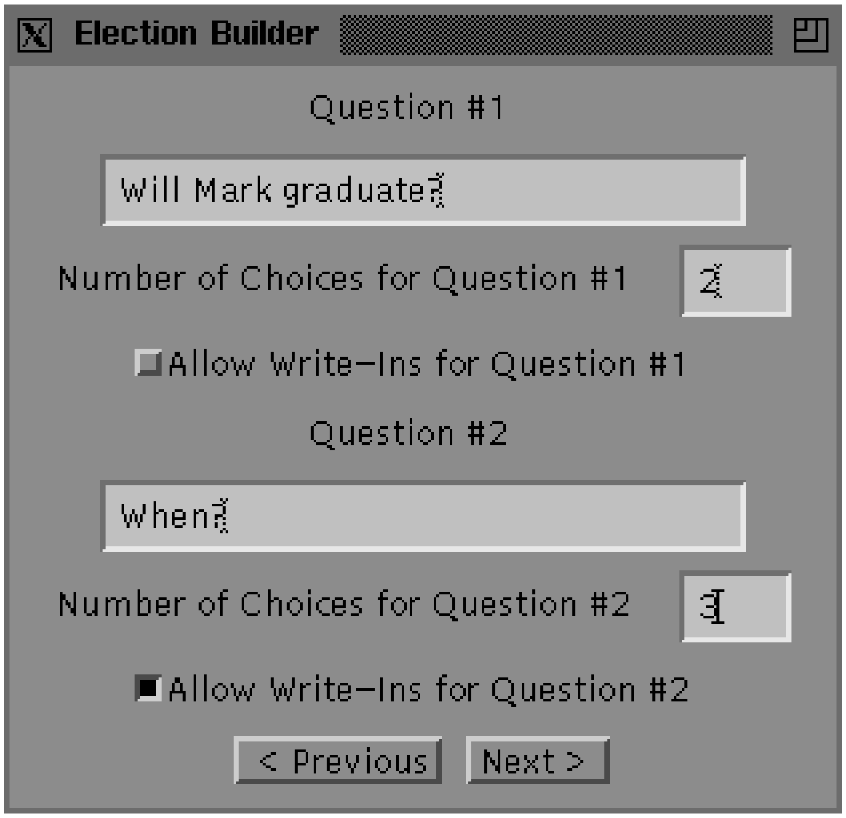
4.5 Registrar

There needs to be some person or persons who ultimately confirms people as being eligible to vote. Those that are eligible can then register using the registration program. This simply records each voter's information - unique ID (e.g. name), password, status of vote ("Has the person voted yet?" initially false) - in a file named as the unique ID. These files are given to the administrator server. The status field is marked true by the administrator during the signing process, to prevent repeated voting by a single person.

4.6 Election Builder

There is an application which allows users to create an Election Object. This object contains the "list" of questions and possible answers (including write-ins). The object is used by both the applet and counter in performing their duties.

The election builder creates an Election Object within a file specified by the user. The user can select the number of questions, and the number of choices for each question. For any question write-ins may or may not be allowed. The program pops up a dialog box which runs through a series of "cards," prompting the user for input. Two cards are shown in Figures 4.2 and 4.3.


Election Builder

Question #1

Will Mark graduate?

Number of Choices for Question #1 2

☐ Allow Write-Ins for Question #1

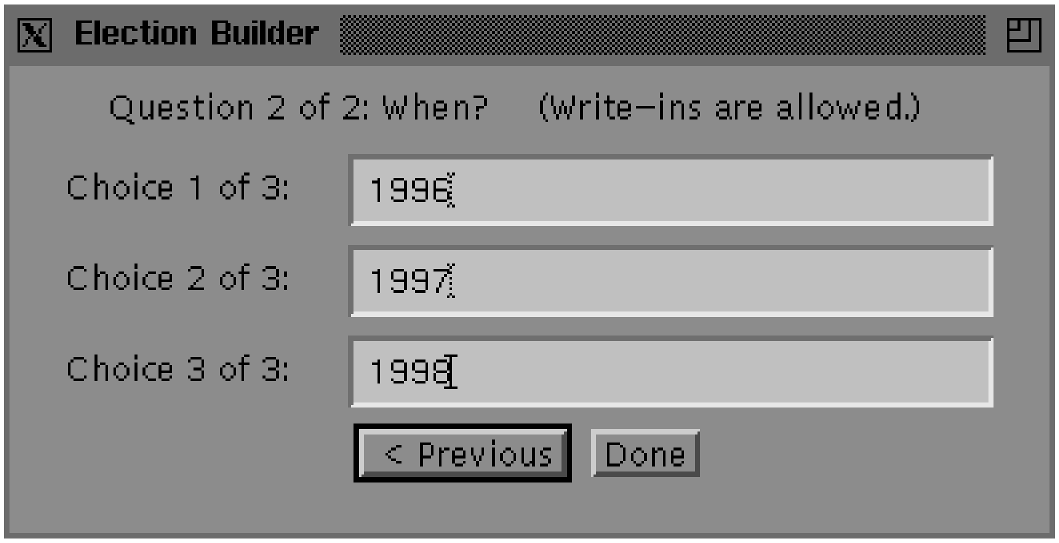
Question #2

When?

Number of Choices for Question #2 3

■ Allow Write-Ins for Question #2

Figure 4.2: Election Builder Questions Card


Election Builder

Question 2 of 2: When? (Write-ins are allowed.)

Choice 1 of 3: 1996

Choice 2 of 3: 1997

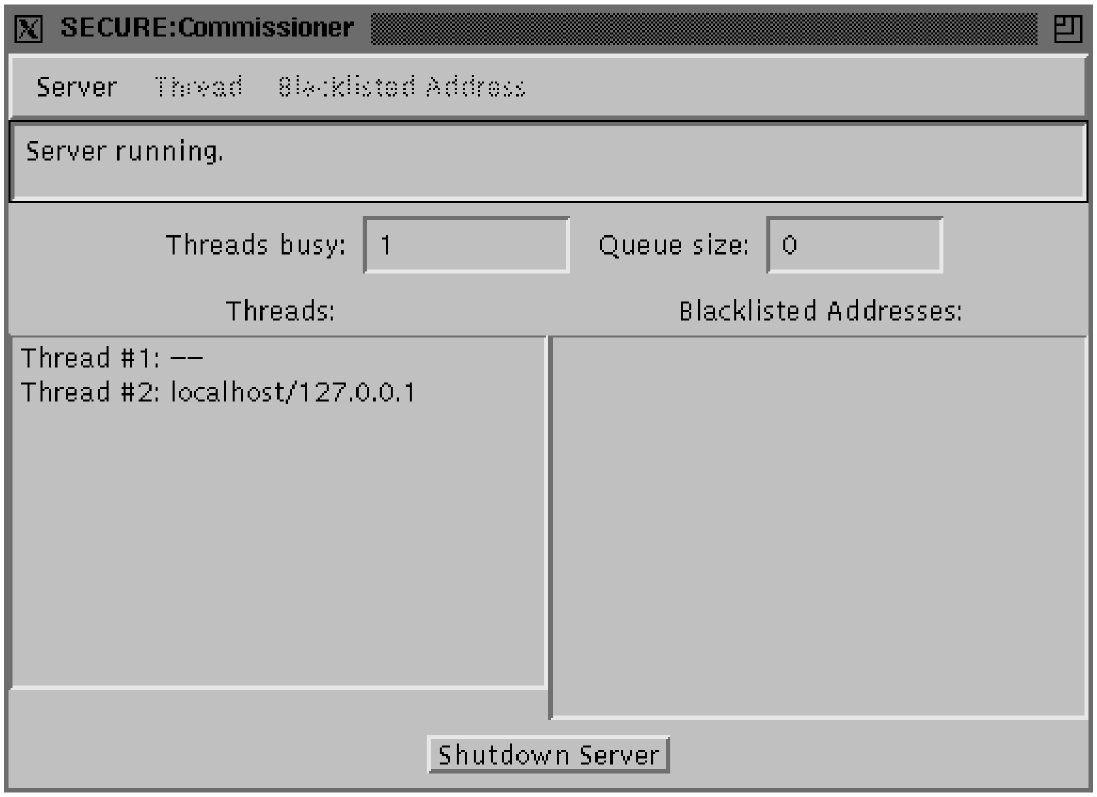
Choice 3 of 3: 1998

Figure 4.3: Election Builder Answer Card for Question Number 2

4.7 Servers

4.7.1 Server Interface

All servers have some basic functionality in common. This is a Java interface, inherited by all servers, which defines their general properties.



Server running.

Threads busy: 1   Queue size: 0

Threads:

Thread #1: --
Thread #2: localhost/127.0.0.1

Blacklisted Addresses:

Figure 4.4: Server Interface

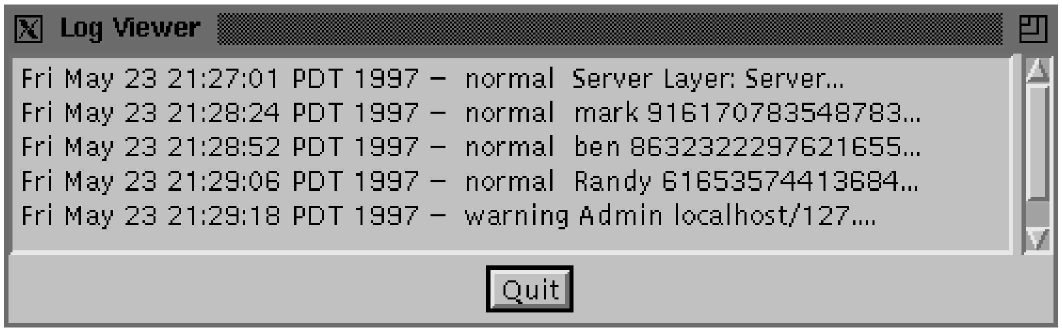
Figure 4.4 shows a typical server, in this case the commissioner server. All servers are window based. The top box indicates its status, currently “Server running.” We see that the server runs two threads, one of which, thread number two, is busy taking a connection from the localhost. There are currently no connection requests waiting on the queue nor are there any blacklisted IP addresses (see below).

Information on the threads and blacklisted addresses can be viewed using the menu bar at the top of the window. Thread information might include a list of connections it handled. Blacklisted Addresses could contain a record of all attempts made from an address. At this time, we do not store either of those records. However, there is currently an option to remove an address from the blacklist.

All servers support both secure and regular connections. One attack to which servers are vulnerable is flooding. That is, the members a group wishing to disrupt the election all sit down at terminals and keep requesting connections to the servers (or have an program to do this for them). Every thread will repeatedly waste time dealing with these bogus requests, similar to anarchists who keep walking into a polling station and asking to vote. If 10,000 people continually did this at a single polling station, they would greatly impede the system from performing.

To protect against this sort of attack, after a set number of connections to a server, an IP address is blacklisted, an idea first suggested by Ben Adida, and connections from that IP address will no longer be accepted. While a request to connect must still be dealt with, it is blocked at a lower level, requiring less of the server's time. Larger, more complex systems, may require a more complex blacklist. For instance, certain "public" terminals (defined as having more than one possible user) such as those at libraries or colleges may be allowed many more requests than a private PC at a home or office. Still, with appropriate numbers for apportioning voting districts, the number of public terminals per cluster (e.g. building), and the number of connections before being cut off, this should not be a problem, unless early one morning the attackers run around from terminal to terminal and blacklist every physically local IP address. It is reasonable to assume, however, that public terminals are monitored to some extent, and the attackers cannot access private ones. To this end, districts can set up special computer polling stations, which would simply replace the current mechanical booths and paper ballot boxes. (People could still vote using their private PC, of course.)

.



```
Fri May 23 21:27:01 PDT 1997 -  normal  Server Layer: Server...
Fri May 23 21:28:24 PDT 1997 -  normal  mark 916170783548783...
Fri May 23 21:28:52 PDT 1997 -  normal  ben 8632322297621655...
Fri May 23 21:29:06 PDT 1997 -  normal  Randy 61653574413684...
Fri May 23 21:29:18 PDT 1997 -  warning Admin localhost/127....
```

Figure 4.5: Sample Log (from the Administrator)

All servers maintain a log file. There are two types of entries to this log. The first is an entry which will eventually published, as per the protocol. The second is an error entry. Figure 4.5 shows the record log of the administrator. Double clicking on an entry pops up another window with the specific information about the entry. In this case, clicking on a name would bring up a window with the blinded vote and its signature. Double clicking on the “warning” entry would give a description of the complaint sent to the commissioner. The errors are recorded because a well-prepared attacker might cut off the communications lines to the commissioner (or the commissioner might be corrupt).

The values to be published (e.g. committed vote) are all simply byte arrays. We cast these objects into *BigIntegers* to display them. Part of the first of the two numbers recorded by the administrator can be seen is the main window shown in the figure above.

Finally, all servers will have their keys generated beforehand. On start-up, each server will have its key set, and all public keys of other servers hardcoded within them.

4.7.2 Administrator Server

The administrator server is responsible for verifying the voter's right to vote, and authenticating the ballot. It must sign at most one vote per legitimate voter who requests it to do so.

The server has access to all the registration files, which need to have been securely transferred to it ahead of time. (One option is to run the administrator server on the same machine used for registration.) The server runs a number of threads, each of which will respond to secure connection request from a voter applet, confirm the voter ID and password, and sign the committed, blinded vote, and then marks the voter as having voted. There is also a queue of finite size should the server receive more requests than it can handle. A voter applet trying to connect to the server when the queue is full will be refused. At this point, a complaint may be sent to the commissioner (if enough complaints are sent, the commissioner may suspect foul play and investigate.)

The protocol should work as follows:

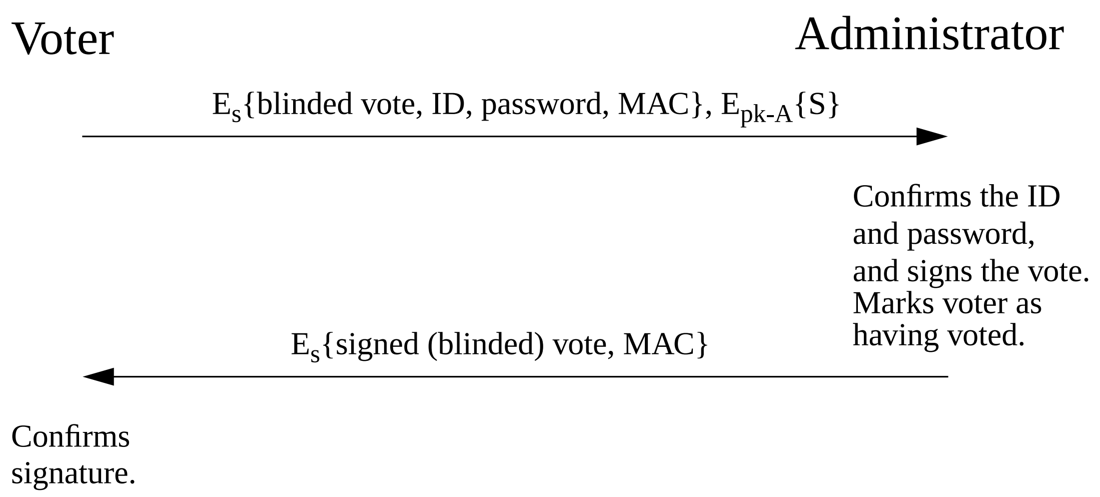

Voter
Administrator
E_s{blinded vote, ID, password, MAC}, E_pk-A{S}
Confirms the ID and password, and signs the vote. Marks voter as having voted.
E_s{signed (blinded) vote, MAC}
Confirms signature.

Figure 4.6: Voter-Administrator Communications

Here S is the Blowfish session key and pk-A is the public key of the administrator.

If the administrator finds that the message is not valid (e.g. bad MAC, missing certain components, the voter is not listed, incorrect password), it will inform both the voter applet and the commissioner server of the problem. The voter/applet may decide to try again, give up, and/or complain to the commissioner.

4.7.3 Anonymous Server

Fujioka et al. calls for an anonymous channel. This is one of the largest gaps between theory and practice. A few simple anonymous channels were considered in which a middle man strips off the headers. Prof. Rivest noted that they all suffer from the same type of timing attack. Namely, eavesdroppers can record the times and sizes of network traffic on both sides of the channel.

Our approach was to use an anonymous server, somewhat similar to an anonymous remailer. The high-level model is as follows. The voter applets connect to the anonymous server (aka the anonymizer), at various times, which will see their IP addresses, but not their votes. The anonymous server then strips off the headers, with their identifying information, and just after the voting deadline, randomly re-orders the votes, and sends them off, en masse, to the counter at a specified time. The counter sees only the votes, without knowing their point of origin.

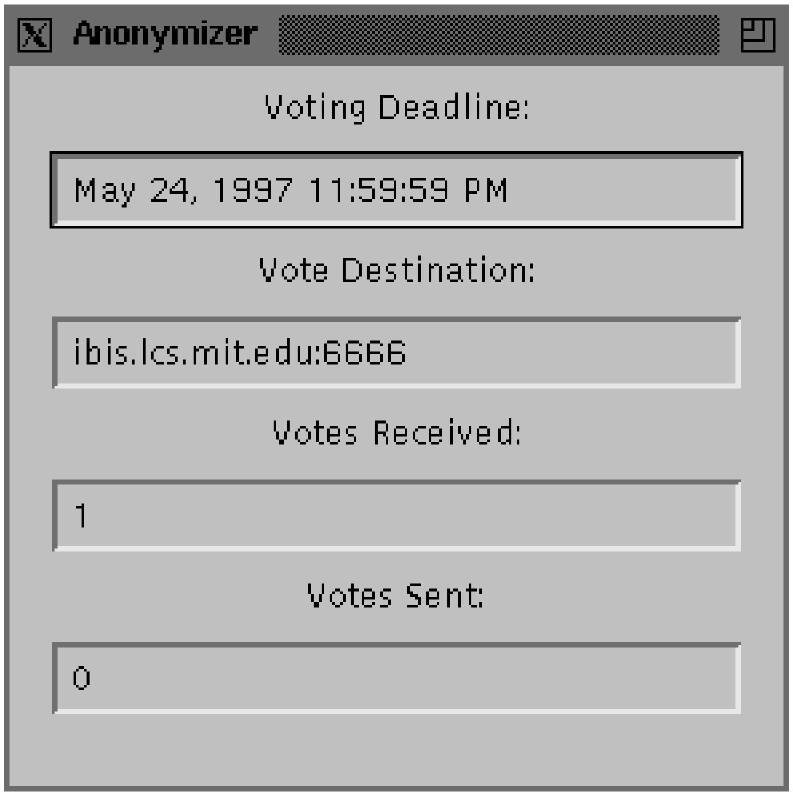
Connections are handled much like they are by the administrator server. The only difference is in the message itself, what gets sent, and how it is handled. The voter applet creates a secure connection with the counter (remember, there is no handshaking, so the applet can initiate this protocol with no interaction from the counter). Two object are used in the secure connection to the counter, the message, keys, and MAC, all encrypted using Blowfish, and the Blowfish session key encrypted with the counters public key (as in Figure 3.1). Together, these two objects become the new message, and are wrapped in a second layer of secure connection. This outer layer is used as a connection between the applet and the anonymous server. The vote is then sent to the anonymizer. Upon receiving this message, the anonymizer strips off the outer layer (as well as the headers) and records the vote, which is still encrypted so that only the counter can see it, in a file. At the appropriate time, the anonymizer reorders all votes and sends them on to the counter.

Suppose we did not double encrypt the message from the voter to the counter with two layers of secure connection. Suppose the outer layer were left off. Although the anonymous server prevents timing attacks, anyone (especially the counter) tapping the anonymous server's communication lines could defeat the scrambling, because the earlier message will include unencrypted header info, and the body of the message could be matched against the server's outgoing messages.

Looking at it the other way, if we left off the inner layer of encryption, the anonymizer would see the plaintext votes. Knowing the sources of these votes, it can correlate voters with votes. (This, of course, would be almost the same as having the anonymizer be the counter with no anonymous channel.)

Under the protocol listed above, the message received by the anonymous server has been "doubly encrypted" (not in the sense of triple-DES, but rather in the one defined above). The message sent from the anonymous server to the counter is encrypted just once. The only variation in vote size at this stage comes from the length of the write-in names (if any). Traffic analysis attacks and their prevention are discussed in section 6.3.2)

.



Voting Deadline:

May 24, 1997 11:59:59 PM

Vote Destination:

ibis.lcs.mit.edu:6666

Votes Received:

1

Votes Sent:

0

Figure 4.7: Anonymous Server Record Window

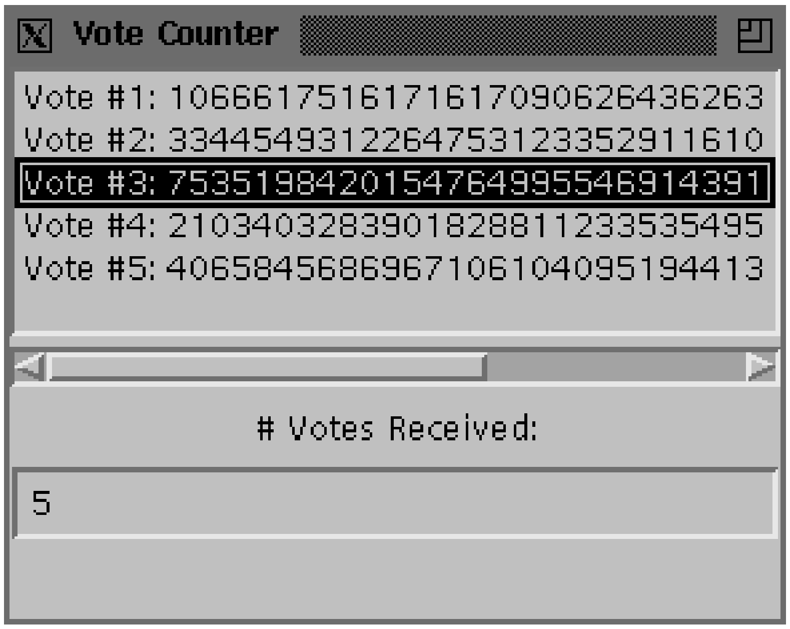
In addition to the server interface listed in section 4.7.1, the anonymizer has a second window. This window lists the status of the votes. The Anonymizer shown in Figure 4.7 shows that one vote has been received, and none have been sent on to the counter. When the deadline passes, the votes will be sent to port 6666 on ibis.lcs.mit.edu. If “Votes Sent” does not match “Votes Received” after they have all been sent off, the log file can be checked to find the votes that were not sent.

4.7.4 Counter Server

The counter server is relatively simple. At one or more designated times, it receives encrypted votes from the anonymous channel. As noted above, the votes have been encrypted with a session key, and this session key is encrypted with the counter's public key.

Upon decryption, the counter will confirm that each vote has a valid signature from the administrator. Moreover, it checks that the committed vote matches the commitment of the keys and plaintext vote. Any discrepancies cause the vote to be excluded from the tally, and are reported to the commissioner. The counter publishes the plaintext vote, the commitment keys, the committed vote, and the administrator's signature of the committed vote.

The votes are then tallied. Any voter can confirm that his vote is on the list. Additionally a voter, or any other party, can confirm that all votes there are valid (by checking their hashes and signatures) and counted correctly.



```
Vote #1: 1066617516171617090626436263
Vote #2: 3344549312264753123352911610
Vote #3: 7535198420154764995546914391
Vote #4: 2103403283901828811233535495
Vote #5: 4065845686967106104095194413
```

```
5
```

Figure 4.8: Counter List (Vote 3 of 5 selected)

Figure 4.8 shows the counter's listing of votes. Each vote number is listed along with the byte arrays (cast to *BigIntegers*) which are the vote. This interface works exactly like a log file, such that double clicking on a vote will bring up a window recording information for that specific ballot, as shown in Figure 4.9.

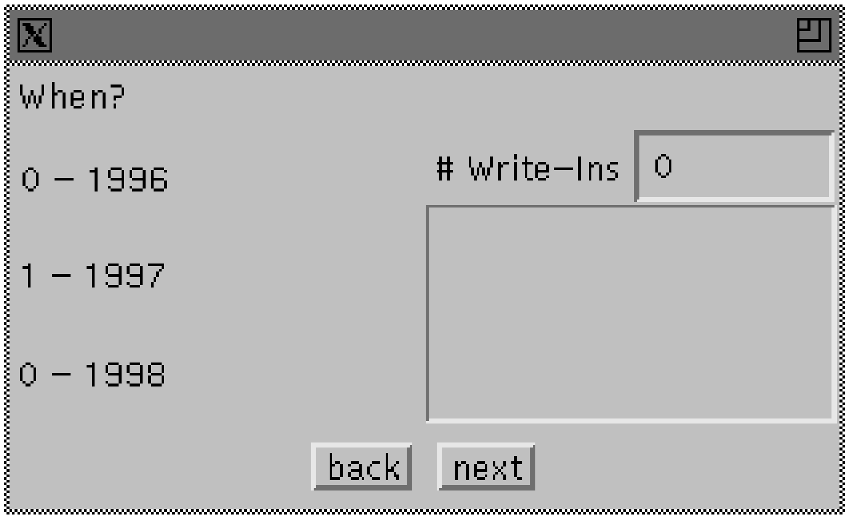
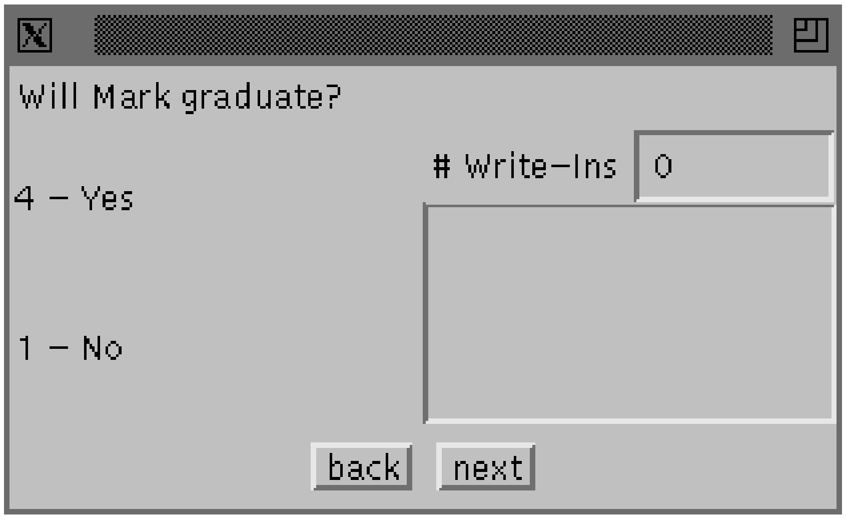


Figure 4.9: Vote Number 3, Question 2

Note that in the figures above, not all information is listed in the windows. That is, Figure 4.8 did not list the committed, signed and plaintext vote, as well as the commitment keys. Nor are those numbers explicitly given in the window shown in Figure 4.9. This was done only for the sake of clarity and could certainly be included in the display.

A typical way in which an interface like this may be used is for the votes to be listed, having been sorted by the number which is the commitment of the vote. Clicking on the actual vote listing would perform a consistency check and, if it passed, pop up the window above.

Finally, a tally is generated by clicking the "count" button on the server's vote list (not shown). When clicked it will open another window, which looks like a specific vote, but with the final tallies and all write-ins displayed. The tallies for our sample election are shown in figures 4.10 and 4.11.

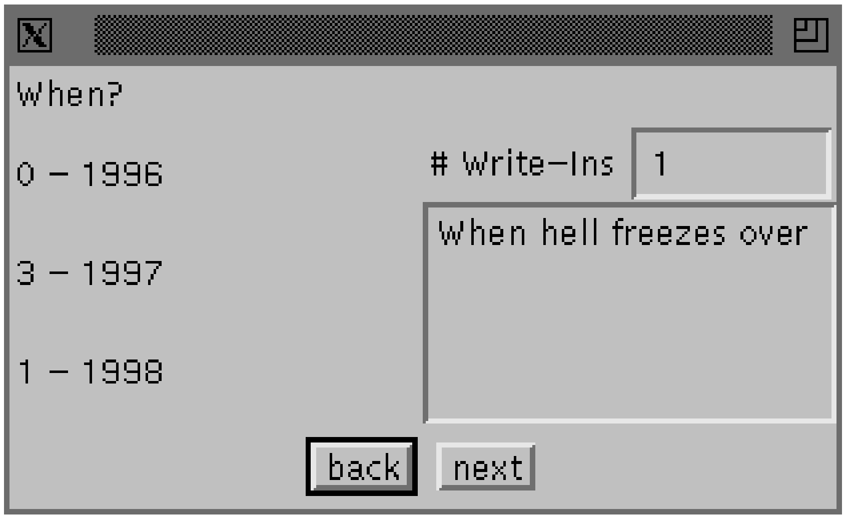

Will Mark graduate?

4 – Yes

1 – No

Write-Ins 0

Figure 4.10: Question 1 Final Tally


When?

0 – 1996

3 – 1997

1 – 1998

Write-Ins 1

When hell freezes over

Figure 4.11: Question 2 Final Tally

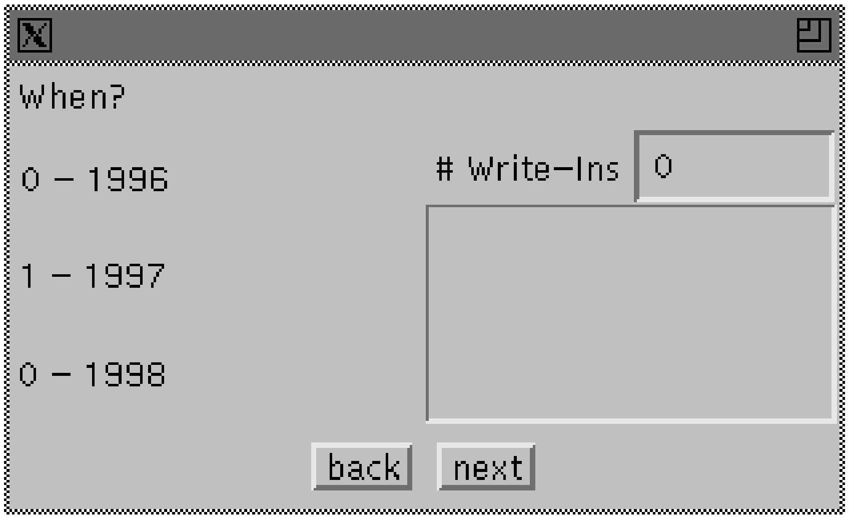


Figure 4.12: Vote 3, Question 2

4.7.5 Commissioner

The commissioner is the party responsible for overseeing the entire election process. The commissioner's job is generally a passive one. Most of the time, it will sit and wait for complaints. The following is a list of possible complaints. Listed in parenthesis are the party or parties who would send such a message.

1. Connection Error (Any)
2. Bad MAC (Any)
3. Bad Message Format (Any)
4. No Keys to Uncommit (Counter)
5. Voter Not Registered (Administrator)
6. Invalid Password (Administrator)
7. Vote Already Signed (Administrator)
8. Vote Already Committed (Voter)

9. Vote Already Uncommitted (Counter)

10. Bad Signature (Voter)

11. File Error (Any Server)

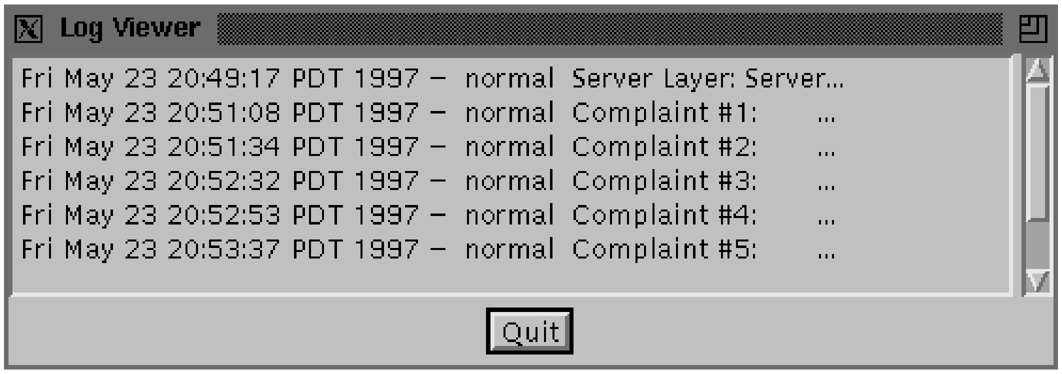
12. Math Error, when performing cryptographic calculations (Any)

13. Unknown Error, used when no other case applies (Any)

14. Vote Received after the Deadline (Anonymizer)

15. Voter Already Voted (Administrator)

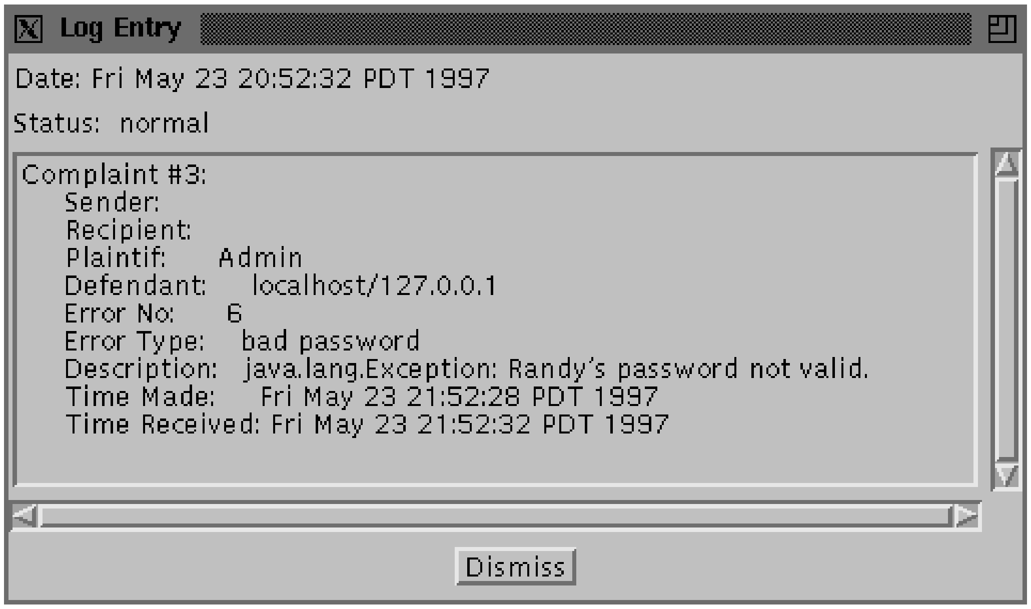
Anytime one of the above cases occurs, one or both parties will contact the commissioner (via a secure connection) and note the problem, parties involved, and time this occurred (i.e. time complaint was sent). The commissioner will note the time, and from whom the complaint was received (IP address). A commission of humans can then sort through the complaints and take appropriate action.



```
Fri May 23 20:49:17 PDT 1997 -  normal  Server Layer: Server...
Fri May 23 20:51:08 PDT 1997 -  normal  Complaint #1:      ...
Fri May 23 20:51:34 PDT 1997 -  normal  Complaint #2:      ...
Fri May 23 20:52:32 PDT 1997 -  normal  Complaint #3:      ...
Fri May 23 20:52:53 PDT 1997 -  normal  Complaint #4:      ...
Fri May 23 20:53:37 PDT 1997 -  normal  Complaint #5:      ...
```

Figure 4.13: Commissioner’s Complaint Log

As with all logs, double clicking on an entry gives more detailed information about it. For example, suppose someone tried to vote in Randy’s place, but did not know his password. In this example, the error log of which is shown in Figure 4.14, the administrator complained to the commissioner that someone using Randy's name tried to vote but did not provide a valid password. The attempt was made from the localhost at 11:52 PM on May 23. (Note: unfortunately, typos made by legitimate votes will be considered malicious attempts. It is up to the human commissioners to use their judgement in cases like this.)



```
Date: Fri May 23 20:52:32 PDT 1997
Status:  normal
Complaint #3:
    Sender:
    Recipient:
    Plaintif:     Admin
    Defendant:    localhost/127.0.0.1
    Error No:     6
    Error Type:   bad password
    Description:  java.lang.Exception: Randy's password not valid.
    Time Made:    Fri May 23 21:52:28 PDT 1997
    Time Received: Fri May 23 21:52:32 PDT 1997
```

Figure 4.14: Complaint Number 3. Someone tried to vote in Randy's name, but did not know the password.

Finally, the commissioner must check the record of each server to confirms that all votes passed from one stage to the next, and votes were neither added nor lost.

A server is given to automate the complaint process. However, most of the commissioner's job depends very much on the specific instance of an election, and how those who are running it wish to handle such complaints. Consequently, the commissioner's responses to complaints have been left unspecified.

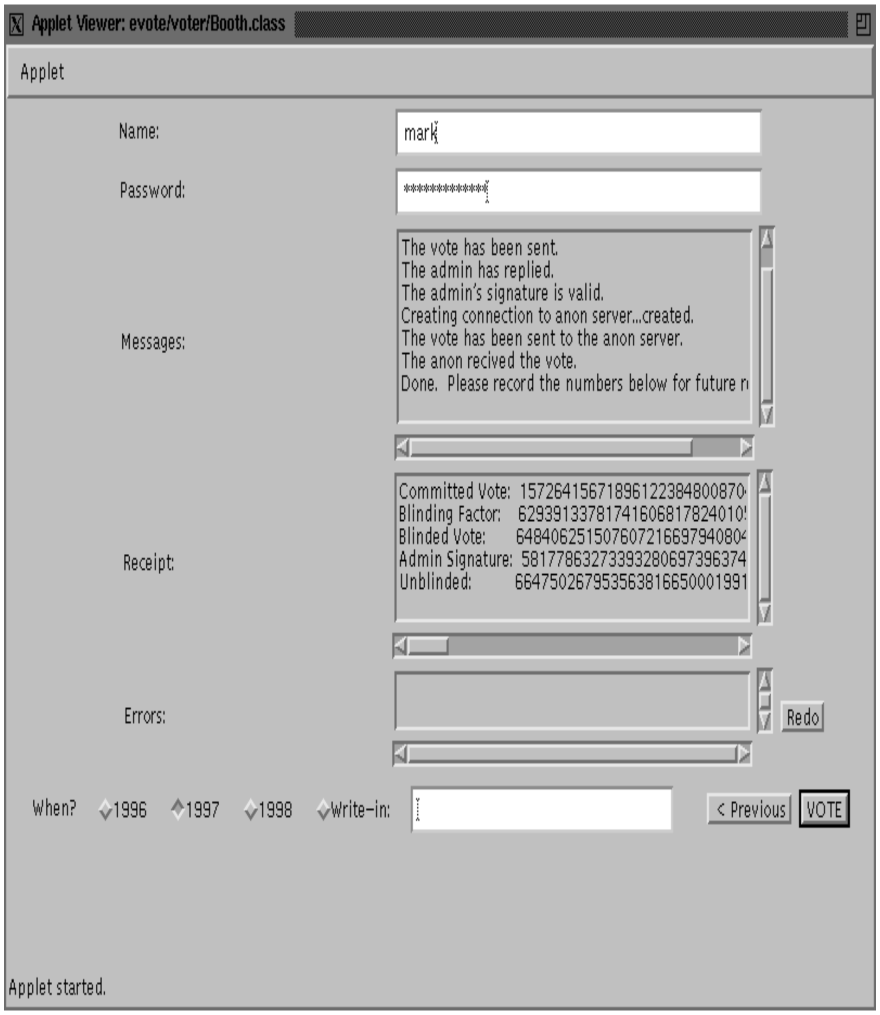
4.8 Voting Applet

The voting applet is the program used by the voter. It is run using a web browser that supports Java. Browsers give different classes of applets different permissions. This applet needs permission to open connections to multiple addresses, and ideally should have file accesses, too (see below). The best way to achieve this is to use the JDK signed applet feature, to allow safe downloading of the appropriate applet (and not some trojan horse), so the granted functionality won't be abused, and can be limited to this trusted applet only. Unfortunately, JDK1.1 only partially supports this feature and *applet signing is not yet available (but expected shortly)*.

Once the applet has been downloaded, the voter simply needs to click on the appropriate choices, enter his name and password and click on the "Vote" button. After that everything should be done automatically, including the filing of most complaints, should something go awry. At the end of the applet's execution, it will print the voter's receipt to the screen, which includes his committed vote, blinded vote, signed blinded vote, and signed unblinded vote, as well as any complaints.

If the applet has access to the local file system, it could be extended to write this information (and any complaints) to a file. Taking this idea further, it could be made to read in the voter's personal registration file so the user would not need to remember a password. The voter could get a copy of his registration file on disk when he registers and upload that in an appropriate place in the local file system for the applet to use.

Of course, automating the applet in this way is dangerous. However, if the applet is signed, there should be no problem. The applet byte codes can be downloaded, verified, and then reverse-compiled to generate the source code. The code can then be confirmed to be innocuous. Or, more mundanely, the source code could be widely distributed, and seen to be safe. Then anyone wishing to do so can compile the code, and compare it with the signed applet byte code.



Applet Viewer: evote/voter/Booth.class

Applet

Name: mark

Password: ************

Messages:

```
The vote has been sent.
The admin has replied.
The admin's signature is valid.
Creating connection to anon server...created.
The vote has been sent to the anon server.
The anon recived the vote.
Done.  Please record the numbers below for future r
```

Receipt:

```
Committed Vote:   157264156718961223848008704
Blinding Factor:  62939133781741606817824010
Blinded Vote:     64840625150760721669794080
Admin Signature:  5817786327339328069739637 4
Unblinded:        6647502679535638166500019 91
```

Errors:

Applet started.

Figure 4.15: Sample Applet

This is just one of many possible layouts of the applet interface. In this case, two fields are used to take in the voters name and password (which is not displayed in the clear). The messages text area reports the status of the vote as it is processed. The receipt field lists the appropriate numbers. If the applet cannot access the filesystem, the user must cut and paste the numbers to record them. The third box is used for error messages. If anything goes wrong during the execution of the protocol, it will be reported here to alert the voter. Finally, at the bottom of the applet is the ballot.

Chapter 5

Testing

This project has been tested on small elections. It ran and worked fine when there was no foul play. By altering our code and/or physically manipulating the data files, we tested the following cases, alone and in combination.

1. Voters tried to vote repeatedly.

2. Unregistered voters tried to vote.

3. Bad passwords were used.

4. The administrator gave invalid signatures.

5. Votes were lost by the anonymous server.

6. Votes were lost by the counter.

7. Duplicate votes were given to the counter.

8. Invalid votes were given to the counter.

9. The administrator shut down before scheduled.

10. The anonymizer shut down before scheduled.

11. The counter shut down before scheduled.

12. The commissioner shut down before scheduled.

In all cases tested, the code followed the protocol and worked correctly.

It should be noted however, that security problems are a negative goal. Where other programs can show they achieved something, we cannot conclusively demonstrate security short of testing every possible input combination (e.g. different ballots, different number of voters, different voters trying different attacks in different orders at different times, etc.).

We do believe we have tried a reasonable set of test vectors to give us some indication of the security of the system, in addition to our own analysis of the code. However, further testing and tuning is necessary before the system can be used for meaningful purposes.

We were unable to test transmission errors (say, due to “corrupted wires”), as we did not have the hardware available. However we believe the protocol will hold work.

We did not explicitly “test” cases in which the various servers “lied” and needed to be caught. We skipped this family of tests for two reasons. First, this is an issue of policy, not cryptography. Second, in most cases with a single server, the cheating party itself cannot be uniquely determined (see section 6.3), only the error corrected. To actually catch a party, multiple servers are needed. Although we did not “test” this case, we confirm all data is printed correctly and so the votes can be recovered.

Perhaps most importantly, the testing was not performed under true operating conditions. Resource limitations allowed us only one machine on which to work. During the election, all servers, as well as the applets, were run on the same host (this is why “localhost” is listed as the machine address in many of the figures). Additionally, the applet was run under the JDK1.1 appletviewer because currently no browsers support Java 1.1.

Further testing is recommended and planned.

Chapter 6

Limitations, Their Solutions, and Further Extensions

The system is still in its infancy. E-Vox is currently a working prototype of what promises to be a tremendous tool for democracy. Currently, there are a number of limitations to E-Vox. Fortunately all of them can be solved without too much effort by more coding, internet advances, and policy choices.

6.1 Code Improvements

6.1.1 Election Instantiation

The first type of limitation suffered by E-Vox is the naivete of the code. There are no macro programs by which to set up the system. The only automated part is the election builder, allowing us to substitute different ballots in different elections. Ideally, we would like to automate more of the code. The parameters (e.g. key size) can only be set by editing a specific file, whereas in the future we hope to have a master program which will do this for us. The same is true for the specific encryption protocols (the library itself should certainly be expanded). Even the counting, as it stands, is done by explicit code in the counter server. The system cannot currently support, say, a preferential balloting system, without a nontrivial effort. It could, but the low level code itself would need to be modified, as opposed to being able to select a back-end counting plug-in from a pool of tabulation plug-ins.

The master program, would ask for all system parameters, including key sizes, redundancy among servers and architecture layout, server addresses, ballot, tallying system, cryptographic building blocks, and deadlines. It would then set all system wide variables and create the appropriate structures. At this point the server code could be moved to independent machines, and the public keys for the servers can be generated.

6.1.2 Public Postings

The protocol calls for public listings of the votes from various servers. Given our decision to use a web browser as the voting booth, a posting on a web page seems like a logical continuation. A simple (scripting) program can be execute by the servers after the deadline has passed which will cause HTML pages to be created. These pages can then be viewed and their contents checked for validity by anyone wishing to do so.

6.2 Protocol

The E-Vox architecture, as specified in section 3.3, has a number of single points of failure. Any of the servers can attempt to disrupt the protocol. A good example of this is a problem found both in Fujioka et al. and our system. Although a lost vote can be recovered, the corrupt party often cannot be uniquely determined. (Fujioka et al. refers to an example of this as an illegal key problem.) We propose a number of solutions to this problem.

6.2.1 Receipts

Because every server has a RSA public key set, we can use that to create receipts, allowing for certified message passing. Recall that secure connections have two components, a Blowfish encrypted message and an encrypted Blowfish session key.

The message Alice sends to Bob is just as before. Now, Bob takes the (unkeyed) hash of the contents of the first part of the communication, that is, the encrypted message, keys and MAC, and signs it to create a receipt, H. This receipt is then returned to Alice. Note that the receipt does not need to be encrypted, because the hashing has removed any connection between the receipt and the message, but it can be sent using a secure connection just like any other message. If Alice’s message is lost, she can prove Bob received it because she can rehash it to get H, and then and show that the receipt from Bob is his signature of H.

.

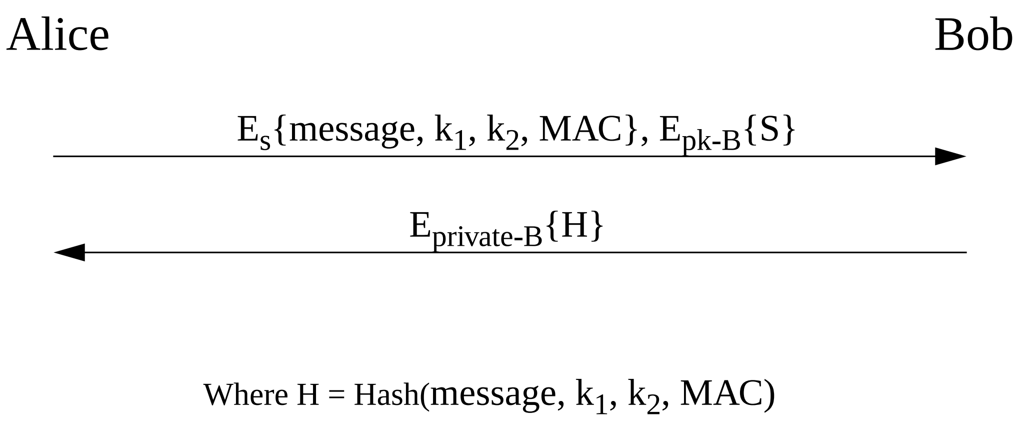

Alice
Bob
$E_s\{\text{message}, k_1, k_2, \text{MAC}\}, E_{pk\text{-}B}\{S\}$
$E_{private\text{-}B}\{H\}$
Where $H = \text{Hash}(\text{message}, k_1, k_2, \text{MAC})$

Figure 6.1: Secure Connection with a Receipt

This does, in some sense, only shift the problem. Bob can refuse to send a signature to Alice, making her think he never received the message. He can later claim he did send her the message, but that Alice lost it in an attempt to discredit him. Still, this is an improvement over the previous case.

The lack of receipts may not be a significant problem. It seems reasonable to assume every voter is motivated to cast valid vote. A voter, then, would have no interest in not sending his vote. A malicious voter may wish to lose his vote, and claim it was the server’s fault in order to frame the server. However, it would take a coordinated effort among many voters to truly cast suspicion in this manner. That is, if only one or two voters, among thousands do this, the server will most likely be considered trustworthy and they would be suspected of foul play. If, on the other hand, a few hundred voters did this, then the server would appear faulty. The possibility of sufficiently large conspiracy among voters, however, seem unlikely. (The point at which the server does become suspected is a matter of policy.)

6.2.2 Communications as a Single Point of Failure

The system tends to put all its eggs in one basket. Ideally the vote should be locked away in a vault (committed) and the keys to the vault should be sent separately. This way if one of the two messages is compromised, the anonymity of voter, and/or the integrity of the vote, is not. By reducing the protocol to only two stages, we send the keys with the vault. This does create a single point of failure.

For example, all messages are subject to a dictionary attack. That is, the encrypted messages can be recorded during the election process. Then, off-line, the attacker can decrypt the first part of the secure connection message by trying every possible Blowfish key to decrypt the message. This attack is hindered by the use of large keys.

We could hinder this type of attack by splitting up our message, and using multiple anonymous channels, each with its own, independent secure connection from the applet. Now instead of sending the message, M, in the clear, which includes the plaintext vote, committed vote, and commitment keys, we encrypt it using another random encryption key, Q. Q is then sent through the second anonymizer. Breaking the encryption of the first message will only yield the attacker another encrypted message, which must then be broken using the same computationally expensive attack of brute force. Breaking the second message reveals even less, as it is only an encryption key to an unknown message.

Figure 5.2 shows an example of this type of distributed anonymous channel. Note that the inner secure connection, that is, the one between the applet and the counter, is not shown in this diagram.

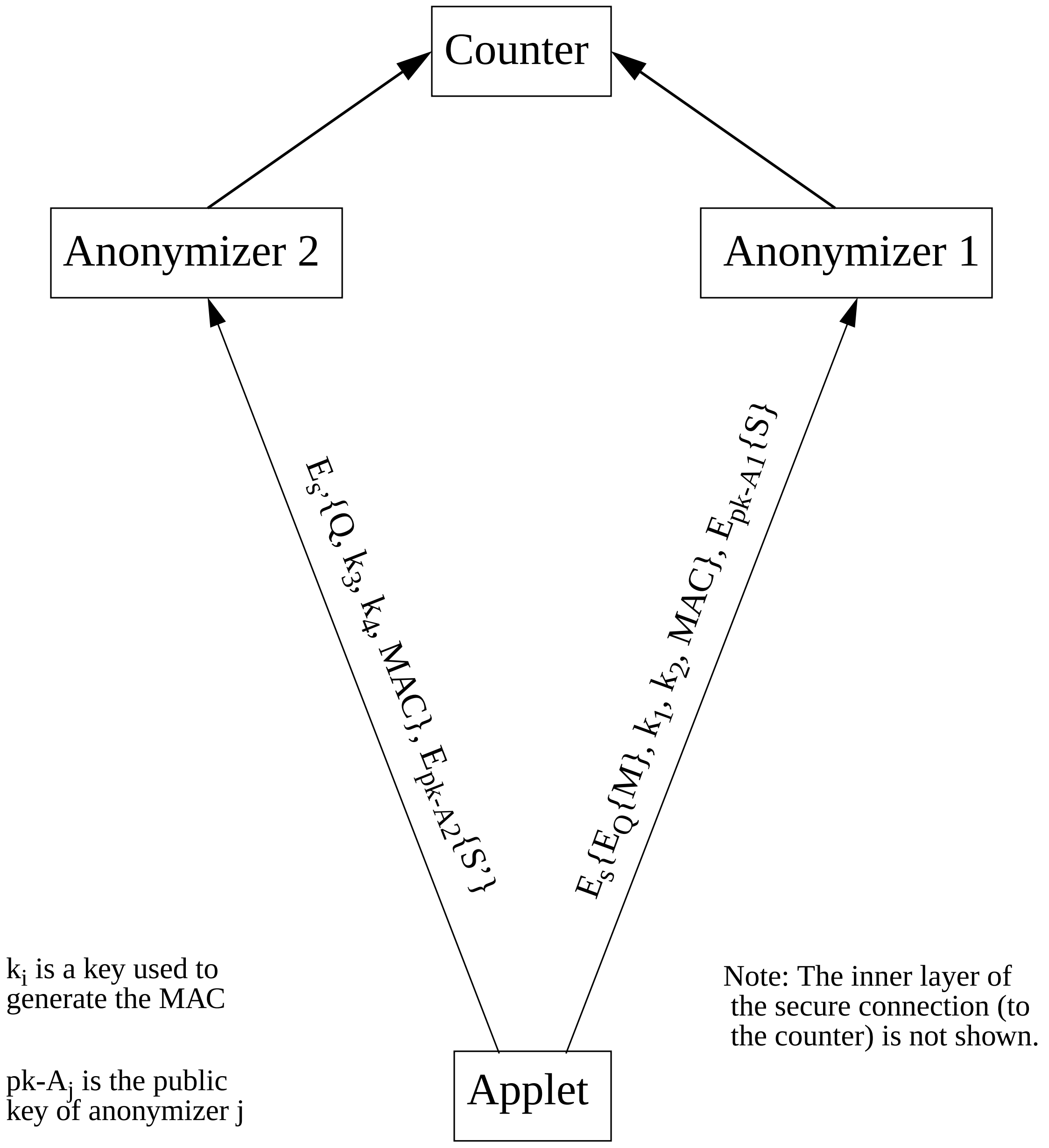

Counter
Anonymizer 2
Anonymizer 1
$E_{s'}\{Q, k_3, k_4, MAC\}, E_{pk\text{-}A2}\{S'\}$
$E_s\{E_Q\{M\}, k_1, k_2, MAC\}, E_{pk\text{-}A1}\{S\}$
k_i is a key used to generate the MAC
$pk\text{-}A_j$ is the public key of anonymizer j
Note: The inner layer of the secure connection (to the counter) is not shown.
Applet

Figure 6.2: Split Message Passing Through the Anonymous Channel

The two messages must somehow be paired. this is easily achieved through index numbers (also not shown in Figure 6.2). The index numbers can be generated by the applet, or by requesting a number from one of the servers. The use of sufficiently large, randomly generated index numbers makes collisions unlikely in the former case (or the latter case with redundant servers).

6.2.3 Cryptographic Attacks

In general, a cryptographic attacker can do worse than just eavesdropping. Often protocols are designed to withstand a specific type of attack. In our case, however, no more than two messages are ever known to be encrypted with the same Blowfish key. And the encrypted Blowfish keys themselves, being random, do not easily lend themselves to an attack on the private keys of the servers.

There is one potential cryptographic weakness in our system, which is that the plaintext messages are often partially known. Certainly the attacker can expect correlations within a vote, as people vote party lines. More to the point, the Java serialized object has a well defined pattern. [Cor97] Knowing simply how the messages are laid out, the attacker knows some of the bits in the encrypted message.

As to whether this is of any use is open for debate. There are some attacks on reduced round Blowfish encryptors. Schneier, the author of the encryption scheme, knows of no attacks on the full 16 round implementation; he notes that there are some weak keys for Blowfish, but “they seem impossible to exploit.”

6.2.4 Administrator Voting

The administrator is the single most powerful entity. Its signature alone validates a vote. This leaves the administrator a few avenues of deceit.

First the administrator can simply create false ballots. All signed ballots are listed by the administrator with a voter’s name. However, this is not a problem. Five minutes before the deadline, if the administrator notices Alice didn’t vote, he can create a vote in her name. If she didn’t bother to vote, she probably won’t bother to check the lists either.

Alternatively, the administrator can create a vote in anyone's name. If, later, that person tries to vote, or claims the administrator cheated, the voter cannot prove it. The administrator can claim the voter voted before, and is simply pretending not to have done so.

Digital signatures alone, like those used in Fujioka et al. do not solve the problem. Because the vote is committed and blinded, it looks random. If a public key digital signature system like RSA is used, the administrator can create a random object, S, and encrypt it with the voter's public key, e and n, to create another object, M, such that M= S^e mod n. Now it appears as though, the voter signed M. If the voter claims this is garbage and cannot be uncommitted, the administrator replies that the voter simply lost they keys to uncommit on purpose.

The above problem, we believe, is a fundamental one faced by any system, like registration. There are some solutions to guard against it, though.

The first solution is "one-time passwords." When a voter registers, his password is not recorded, but rather, the registrar records the hash of it. When voting, a voter only sends the hash of the password to the administrator who can verify it is correct. At the end of the protocol, the original password is sent to the counter, who got a list of all hashes from the registrar. The password itself is hashed and confirmed to be on the list. The administrator cannot reverse the hashing function and so cannot find the password. An alternative to this approach is to use secret sharing. [Sha79] Instead of saving the hash of the password, the encryption of it, using a public key, can be stored. The private key used for decrypting is shared among a number of parties (perhaps in a threshold scheme) so that a reasonable subset must come together in order to be able to see the actual password. Both attacks are vulnerable to a dictionary attack by the registrar or administrator. This attack can be hindered with the use of salt.

Finally, Davenport, et al. [Dav96] suggest a practical approach to help deter the administrator form voting. If the voting is being used on a system where all the registered voters have email accounts, the commissioner can notify all voters that their votes have been tabulated by emailing everyone on the administrator's published list of voters. Again, an abstainer may be so passive as to not care, and the administrator can try to intercept the email, but this approach has the right idea in that the attacker must perform more work, and the victim can recognize the attack with less work.

6.3 Architecture

Another approach to the corrupt server problem, one suggested in Fujioka et al., is to create redundant systems. To guard against dropped votes, parallel servers at any given level will all (or some subset of them) get the message. If all receivers but one have the message, the fault lies on the receiving end. On the other hand, if only one receiver has any record of a transmission, the sender may have tried to incriminate the servers by sending it to only one. These cases rely on a "preponderance of the evidence" approach to catching the malicious party.

6.3.1 Redundant Administrators

As noted earlier, when voters are not looking, a polling official can drop a few extra ballots in the box. This is often prevented by having members of adversarial parties all watching the ballot box together.

One possible solution to the problem of the malicious administrator is the natural analog of this. That is, provide a number of parallel administrator servers, say *n*. For a vote to be accepted by the counter, it must have *t* of *n* valid signatures, from separate administrators. Therefore, *t* of *n* administrators would need to be corrupt in order to create false ballots. Clearly *t* must be greater than *n*/2 (rounded up), to prevent a voter from voting under two disjoint subsets of administrators (or two subsets with one corrupt administrator in common).

This system however, is not as efficient as the original. Still, it may be good enough. Testing is recommended.

6.3.2 Anonymizer-Counter Additional Message Passing Security

The messages may be sent to the counter in any number of ways, at any reasonable interval (long enough to insure a significant number of votes to have been scrambled). To further avoid the above problem with differing message lengths, a block of votes could be grouped together, and a secure connection established between the anonymizer and counter server, so that the individual votes cannot be viewed (in their encrypted form) by an eavesdropper.

Of course, the counter still sees the individual votes, so this will not help against a corrupted counter which compares message lengths. The only variation in length comes from write-in votes. These are likely to be rare and not significantly different in length from a regular ballot (one with no write-ins). Still every message from the applet, to the anonymizer, to the counter, can include padding to insure that they are all the same length (it assumes write-ins are all bound to some reasonable size). The communications between the applet and the administrator is of a fixed size because of the hashing.

6.3.3 Redundant Anonymizers

Another potential problem is that the anonymizer could simply drop the message from the voter. Again, the lost vote would not be detected until after the election has finished and the counter has published the results. To protect against this, multiple anonymous servers can be used in parallel, such that only one needs be reliable to pass the vote through the channel.

An orthogonal problem occurs when the anonymous server keeps record of the message origins, or does not randomly reorder them. If the anonymizer works with the counter, granted, this collusion is contrary to our assumption, anonymity is lost. The problem can be circumvented by using chained anonymous servers (a la the nym anonymous remailer system [Nym]). The vote would be sent from the voter to the first anonymous server in the chain. It would then be passed along the chain, with the final server in the chain forwarding it on to the counter. Only one server need reliably strip off the header and randomly scramble votes for the channel to be anonymous.

6.3.4 Web Spoofing

The World Wide Web is a rather untamed area. A number of attacks exist against anyone who ventures into it. Relevant to our purposes, there is a spoofing attack which can learn votes without the voter knowing.

The attack is really that of a man-in-the-middle. Suppose Alice tried to download a web page, or Java applet. Bob can create a special interface applet and trick her into downloading that instead. Bob's applet will mimic the one Alice was originally looking for, without even knowing which one she wanted ahead of time.

Bob will intercept Alice's request and download his applet to her instead. Bob's applet is a "shell" in which the applet Alice really wants is run. Any data Alice gives, or receives from the applet she wants to run, will really be given to Bob's applet. Bob's applet records the information, and then mimics the behavior of the original applet by passing the data on to it, and returning the results to Alice. Or, Bob's applet could simply impersonate the one Alice wants, and won't really contact it.

Attacks have this nature have been studied by Felten et al [Fel97]. A (humorous) example of this type of web spoofing is the Zippy Filter [Zip].

Such an attack is prevented by signed applets. As long as the key used for verifying the signature is publicly known, a web browser can confirm the downloaded applet is the correct one before running it.

6.4 Policy

6.4.1 Time

An issue for any distributed system is that of time coordination. A common approach is to create time beacons which can be used as a universal standard (to within a reasonably small error). The use of beacons would work here, too, but it is most likely unnecessary.

In current, real world elections, if a voter shows up to the polls late, and claims his watch was slow, he forfeits his right to vote. It is the responsibility of the voter to keep track of the time. We require the same of a user voting electronically.

The only case where unsynchronized time may come into play is during a complaint. The times of complaints may be useful in diagnosing the problem. If everyone has their own definition of time, the time stamp becomes meaningless. Fortunately, the commissioner marks the time of receipt of the complaint. Assuming a sufficiently small delay between the act about which the complaint is made, and the receipt by the commissioner, the unsynchronized time should not be an issue. (Redundant commissioners would need to coordinate time, but they can be easily synchronized at start-up.)

6.4.2 Registration

The Registrar can create ghosts. That is, it can register non-existent voters and later cast votes under those names. The prevention of ghosts is a policy issue, and not one for cryptography. A practical solution is to have adversarial parties oversee the registration process, to make sure the dead do not rise to vote again.

6.4.3 Key Distribution

As noted earlier, we face the fundamental problem of key distribution. The servers must securely share public key information. Additionally, if the applet is signed, this information, too, must be publicly dispersed.

Chapter 7

Conclusion

Only within the last few years has electronic voting moved from the realm of papers to actual computer implementations. Although E-Vox is not quite the first implementation, we believe it is the first secure, user-friendly, stand-alone system.

Our model is only a prototype, and further work needs to be done. Most importantly, the system needs to be run and attacked. We have suggested a number of extensions in Chapter 6. All of these increase security at the cost of simplicity. To optimize our design, we must find which attacks are the most cost-effective for the attacker and protect against those types of attack.

As the world wide web continues to become more integrated in our daily lives, we believe future protocols will follow our design and use web-based servers and applets. Additionally, projects such as digital cash, which face similar design problems, can build on our work. (When voting becomes electronic, is it any surprise that cash is not far behind?) Actually, Prof. Rivest has pointed out that voting can be considered a form of spending special coins.

While smart cards may be years in the future, digital certificates are becoming commonplace. Such certificates, could effectively provide a general authentication system throughout the world wide web, making protocols like ours even easier to implement.

Given all these trends, work in this area is certain to continue. We look forward to learning from the further expansion of the field.

Appendix

Electronic Voting and Its Effects on Society

A full investigation into the effects of electronic voting on elections, and society in general is far beyond the scope of this paper. Nonetheless, as scientists and engineers, we have a social responsibility to inform society of the power of what we build, and the effects it might have.

Kirby [Kir95] has both surveyed the literature in this area, as well as conducted some limited testing as to the effects of electronic voting at MIT. Although MIT is a very biased population form which to sample, it is a good start.

By the very nature of it being computer based, we might expect to see both a technical and monetary bias. Voting will have become easier for the people who can afford or even know how to use a (public) computer.

On the other hand, a distrust of computer security could cause people to doubt the outcomes of elections in which computer voting was popular. This might cause politicians to back away from such schemes. On the other hand, E-Vox and other computer based systems allow every voter to verify the election returns for himself.

Studies of the effects of the physical ballot layout have been done for both paper ballot, and DRE systems. Voting over the web offers a much more complex ballot. For instance, hypertext links could be on the ballot, linking the voter to multimedia web pages of the candidates, parties, and special interest groups. Web pages of this sort will be carefully created in an attempt to give the voter that final push, while at the ballot box!

We look forward to further research in this area.

References

[Anon] Web site http://www.stack.nl/~galactus/remailers/

[Cha82] D. Chaum, “Blind Signatures for Untraceable Payments,” Advances in Cryptography: Proceedings of Crypto 82, Plenum Press, 1983, pp 199-203.

[Cor97] G. Cornell and C. S. Horstmann, *Core Java*. Sunsoft Press, Mountain View, CA, 1997.

[Cra96] L. F. Cranor and R. K. Cytron, “Design and Implementation of a Practical Security-Conscious Electronic Polling System.” WUCS-96-02, Washington University Department of Computer Science, St. Louis, January 23, 1996. Taken from http://www.ccrc.wustl.edu/~lorracks/sensus/

[Dav96] b. Davenport, A. Newberger, and J. Woodward, “Creating a Secure Digital Voting Protocol for Campus Elections,” Princeton University, 1996. Taken from http://www.princeton.edu/~bpd/voting/paper.html

[DeM82] R. DeMillo, N. Lynch, and M. Merritt, “Cryptographic Protocols,” Proceedings of the 14th Annual Symposium on the Theory of Computing, 1982, pp. 383-400.

[Dif77] W. Diffe and M. E. Hellman, “New Directions in Cryptography,” I*EEE Transactions on Information Theory*, v. IT-22, n. 6, Jun., 1977, pp. 74-84.

[Fel97] E. W. Felten, D. Balfanz, D. Dean, and D. S. Wallach, “Web Spoofing: An Internet Con Game,” Technical Report 540-96 (Revised Feb. 1997), Department of COmputer Science, Princeton University.

[Fuj93] A. Fujioka, T. Okamoto, and K. Ohta “A Practical Secret Voting Scheme for Large Scale Elections,” *Advances in Cyptology* - AUSCRYPT ‘92.

[Gen96] R. Cramer, R. Gennaro, and B. Schoenmakers, “A Secure and Optimally Efficient, Multi-Authority Election Scheme,” MIT, Nov 6, 1996. A preliminary version of which was submitted anonymously to the *Security in Communications Networks* workshop, Sep. 16-17, Amalfi, Italy, 1996.

[Kil90] J. Kilian, *Uses of Randomness in Algorithms and Protocols*, MIT Press, 1990.

[Kir95] J. P. Kirby, “Electoral Method Effects of Decentralized Electronic Voting.”

[Kra97] M. Krawczyk, M. Bellare, and R. Canetti, “HMAC: Keyed-Hashing for Message Authentication,” RFC 2104, Feb. 1997. Taken from http://src.doc.ic.ac.uk/computing/internet/rfc/rfc2104.txt.

[Nur91] H. Nurmi, A. Salomaa, and L. Santean, “Secret Ballot Elections in Computer Networks,” *Computers & Security*, v. 10, 1991, pp. 553-560.

[Nym] Web site http://www.cs.berkeley.edu/~raph/n.a.n.html.

[Riv78] R. L. Rivest, A. Shamir, and L. M Adleman, “A Method for Obtaining Digital Signatures and Public Key Cryptosystems,” *Communications of the ACM*, v. 21, n. 2, Feb. 1978, pp. 120-126.

[Sak94] K. Sako “Electronic Voting Schemes Allowing Open Objection to the Tally,” *Transactions of the Institute of Electronic, Information, and Communication Engineers*, v. E77-A, n. 1, 1994, pp. 24-30.

[Sch96] B. Schneier, *Applied Cryptography.* John Wiley & Sons, New York, 1996.

[Sha79} A. Shamir, “How to Share a Secret,” Communications of the ACM, v. 24, n. 11, Nov 1979, pp. 612-613.

[Zip] Web site http://www.metahtml.com/apps/zippy/welcome.mhtml.