Abstract

This paper describes an implementation of an Internet-based voting scheme based on the work of Atsushi Fujioka, Tatsuaki Okamoto and associates. This algorithm is notable for its use of blind signatures, which allow an election to provide privacy as well as security. The implementation described is a complete, working system used for collegiate elections.
# Table of Contents

**Abstract**  
i

**Acknowledgement**  
iii

**List of Figures**  
iv

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Thread 1: Requirements Analysis and External Design</td>
<td>2</td>
</tr>
<tr>
<td>2.1 Generalized Voting System Requirements</td>
<td>2</td>
</tr>
<tr>
<td>2.2 Adapting Voting System Requirements to the LionVote Situation</td>
<td>3</td>
</tr>
<tr>
<td>2.3 Areas Where Requirements Are Eased</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Survey of e-voting research part I – Broad Research</td>
<td>7</td>
</tr>
<tr>
<td>2.5 Survey of e-voting research part II – Focusing on the FOO algorithm</td>
<td>8</td>
</tr>
<tr>
<td>2.6 Construction of Entities in the System</td>
<td>10</td>
</tr>
<tr>
<td>2.7 Detailed Description of Protocol</td>
<td>13</td>
</tr>
<tr>
<td>2.8 Differences between LionVote Proposal and FOO+ Scheme</td>
<td>15</td>
</tr>
<tr>
<td>2.9 Ballot Generation Issues</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. Thread 2: Internal Design and Coding</td>
<td>17</td>
</tr>
<tr>
<td>3.1 Requirements Softening</td>
<td>17</td>
</tr>
<tr>
<td>3.2 Development Timeline</td>
<td>20</td>
</tr>
<tr>
<td>3.3 Revised Detailed Description of Protocol</td>
<td>26</td>
</tr>
<tr>
<td>3.4 Deployment</td>
<td>29</td>
</tr>
<tr>
<td>3.5 Comparison of LionVote with Other Student Voting</td>
<td>30</td>
</tr>
<tr>
<td>3.5.1 Comparison of LionVote with Other Student Voting Implementation</td>
<td>30</td>
</tr>
<tr>
<td>3.6 Software Engineering Lessons Learned</td>
<td>33</td>
</tr>
<tr>
<td>3.7 Future Enhancements</td>
<td>34</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. Conclusion</td>
<td>35</td>
</tr>
</tbody>
</table>
Acknowledgement

I would like to thank, in particular, Dr. Null for her patience and encouragement. I would also like to thank the other members of the faculty that I have learned much from: Dr. Wahls and Dr. Bui. Thanks to all faculty reviewers including Dr. Naumov and Dr. Ding. Thanks to my family for dealing with the whining and to Mary Satorie for her affectionate support and proofreading.
**List of Figures**

1. LionVote Design Phases ................................................................. 20
2. LionVote Implementation Phases ..................................................... 21
3. Sample XML ballot specification ..................................................... 22
4. HTML rendering of ballot specification ........................................... 25
5. High-level overview of implementation protocol ............................... 27
6. LionVote and voting criteria ............................................................ 32
1 Introduction

The need for this project derives from a desire for improvement in campus voting procedures at Penn State Harrisburg. Currently, student voting is done through pencil and paper balloting at an on-campus voting center and faculty voting is done through either e-mail or physical mail. Each approach has its pluses and minuses. On-site paper ballot voting requires significant paperwork and volunteered time for voting supervision and tallying. An electronic voting system can alleviate or eliminate the numerous problems in a paper ballot as witnessed in our most recent presidential election. For a commuter school, limiting voting to on-site locations also can lower turnout, since many students, especially graduate students, attend classes one or two evenings a week. Nevertheless, the current paper student election does satisfy many of the criteria for a desirable voting system. The current faculty election process is more problematic. It enhances one criterion, convenience, at the expense of several others by allowing a voter to be linked to the vote during the tallying phase and allows votes to be forged. The faculty elections are chosen as the primary model, over student elections, due to the facts that faculty elections already support a form of mobile, computer voting and that faculty voting has several major flaws mentioned above.

A secondary objective of this paper is to provide a case study in Software Engineering. This paper closely follows the confusion of a real life software project: features were planned for but abandoned, other features were planned for but left for future enhancements, and others were not planned for but hastily added. This confusion typifies the volatility, uncertainty, and pressures of expediency that are inherent in software development. It is also important to leave the discussion of the original design intact as a potential source of future student enhancements.

This paper is structured to have two main threads that mirror development: discussion of the specifics of the system design and a discussion of software engineering issues raised in the development process. The first thread of this paper is dominant in the requirements analysis and early design phase where most of the academic research and technology choices were done. The FOO+ algorithm [OhkM] was chosen from a variety of academic papers as being a well-regarded, practical approach to a voting system. The second thread of this paper is dominant in the later design phase and coding phases. Many compromises were made based on time, money,
and analysis of what customers actually want. To make things explicit, there is a working voting system, referred to as LionVote (named after the Penn State mascot, the Nittany Lion) that requires expert setup to prepare but runs a faculty election. This second thread concludes with a comparison with other student electronic voting projects and possibilities for future enhancements to LionVote.

2 Thread One: Requirements Analysis and External Design

2.1 Generalized Voting System Requirements

Numerous papers list criteria for an ideal voting system, but one of the most cited examples is that provided by Lorrie Faith Cranor in Sensus: A Security-Conscious Electronic Polling System for the Internet [CraC]. Here are the seven criteria used from this paper:

1. **Accuracy** is comprised of three components: A vote can’t be changed after it has been made, a correct vote once made must be counted, and a vote that wasn’t made correctly shouldn’t be tallied.

2. **Convenience** is easy to define but highly subjective in how individuals regard a system. Convenience mandates the system should involve as little time as possible with voter registration and polling, the system should be as simple in polling as possible, and the system should be intuitive through reuse of common interface techniques. The last point, intuitiveness can be problematic for computer-based voting. This is because any interface techniques that involve use of a computer can be extremely inconvenient to those who have never used one or those who don’t like to use computers.

3. **Flexibility** refers to the types of balloting questions that can be raised. This includes options such as supporting the ability to vote for more than one candidate or position on a topic where appropriate or to actually write in an entry. For a system to be reusable, ballot generation by voting administrators should allow a ballot to be generated for any election.
4. **Invulnerability** ensures that eligible voters are given the opportunity to vote during the time frame that their election is open, eligible voters can’t vote multiple times, and non-eligible voters aren’t given the opportunity to vote. Invulnerability is described here as having three characteristics while [CraC] omits the first characteristic. This first characteristic is added because any particular computer solution, especially one that relies on any network, should carefully consider typical network bandwidth and server processing time requirements to ensure that a system provides adequate availability.

5. **Mobility** refers to the ability of voters to pick their voting location freely. This is where solutions such as the Internet and telephone or modem-based solutions can offer improvements.

6. **Privacy** includes the obvious requirement that other parties should not have the ability to correlate a voter with his or her vote. An interesting and more challenging requirement pointed out by [CraC] is that a voter shouldn’t be able prove how he or she voted. This variation of privacy is known as receipt-freeness. This eliminates any possibility of coercion or vote buying. This second requirement is contradictory to the criteria of mobility. Any scheme that allows a voter to select the voting location completely, such as vote-by-mail must introduce the possibility of coercion or vote buying.

7. **Verifiability** ensures that anyone can verify all votes, or in a weaker form, that voters can verify that their particular votes were counted. If anyone can verify all votes, this is called universal verifiability.

In addition, [OhkM] presents another significant election criterion:

8. **Fairness** means that no entity can analyze voting output prior to the completion of the voting phase of the election. This criterion forbids even voting authorities such as an administrator or tallier from having this knowledge.

### 2.2 Adapting Voting System Requirements to the LionVote Situation

The first and biggest design decision was between stand-alone DRE (direct register electronic) or Internet-based solutions. This choice has obvious implications for the mobility and privacy criteria as well as the invulnerability and accuracy criteria. It is clear from the faculty
policy of allowing voting through ordinary e-mail (without use of digital signatures or a multi-step process that attempts to address privacy, accuracy, and invulnerability concerns) that mobility and convenience are valued over privacy and invulnerability (particularly through the risk of faked votes on behalf of eligible voters). If faculty elections were based on stand-alone DRE, there would be a loss of mobility and convenience. This factor led to adoption of an internet-based model that can be a “win-win” situation, because mobility and convenience can be maintained or improved slightly, and privacy, accuracy, and invulnerability can be improved.

However, this is not an improvement for one category of voters – the computer-averse. LionVote assumes that there is a homogeneous system. All voting occurs through computers and uses the same authentication and voting mechanism. Of course, it would be possible to use a hybrid voting system that would allow either paper or computer balloting. A hybrid system adds additional system requirements to synchronize various aspects of a dual system: ballot forms, lists of eligible voters, lists of voters who have already voted, and tallying integration. However, for an academic election, with less at stake and a more computer literate electorate, a homogeneous system should be adequate. An academic election reduces the possibility for lawsuits and a public outcry. In the worst case, an academic election that is botched could have a do-over. A do-over could be very hard to accomplish in a public election as some of the legal analysis over the 2000 Florida presidential election suggested.

Having established that a “win-win” improvement in faculty elections can occur, there are still some disadvantages of an internet-based solution relative to a stand-alone DRE that need to be addressed. It is possible for a denial-of-service attack to reduce server availability to the point of disenfranchising voters. It is also possible that a server could actually become compromised, leading to virtually any negative consequence relating to accuracy and invulnerability. Avi Rubin [Rub] offers numerous reasons why Internet voting isn’t safe, including the potential for spoofing. In one spoofing scenario, a DNS attack technique such as cache poisoning can cause a user’s correct entry of a DNS name to be redirected to another site. For this implementation, the risk of malicious intervention has been de-emphasized because of the lower stakes involved in an academic election, and the larger security flaws that exist in the current faculty voting system.

The specific Internet protocols used by LionVote rely on the Hypertext Transfer Protocol (HTTP), or its secure variant (HTTPS). The voters vote using web browsing clients of their
choice, which interact with several web servers. No restrictions are made for the voting client to have a known IP address, although such a restriction could be made through access control using the web server or a firewall. The actual implementations are discussed later.

Another general decision pertained to user identification. This decision was constrained by the decision to support voter mobility in the form of Internet voting. Having mobility precludes the ability to have a voter authenticated through an external identification mechanism such as a voting authority comparing the picture on a photo-id. Another promising technology is to have smart cards that maintain key information. This could combine well with a photo-id. However, it would require the prevalence of card swiping devices, which currently aren’t a standard PC feature. Key distribution is a relatively challenging problem due to user ignorance as to what a Public Key Infrastructure (PKI) is and what they need to do to obtain and maintain a key plus the inherent insecurity involved in the PCs that people use. This means that using SSL, or other technologies that rely upon PKI, for client as well as server authentication is pre-mature. Using SSL for only server authentication is a technology worth considering today.

Having ruled out several user identification options, what is left? For identification, name or PSU access ID should be used. Since access IDs are a unique public piece of information associated with all eligible people they have been selected. Since access IDs are public knowledge, a verifying piece of information should be used as well, such as a Social Security Number, access ID password, or one-time only password. Since SSN information can be obtained in a variety of ways, this will be used only as a last resort. Another viable option is a one-time only password. It would have to be sent in the mail well in advance. This would introduce problems with people misplacing or never receiving their passwords. The final verifying option is the access ID password. The feasibility of integrating the voting system into the PSU Kerberos authentication mechanism is uncertain. MIT student elections have successfully used Kerberos using the Pericles system [Her]. If Kerberos were feasible for LionVote, this would be the ideal solution for voting. The voting system would (and should) know only the access IDs, not their passwords.

The voting system criterion of convenience requires careful consideration in voting system development. In any election, there certainly will be computer-averse people; although, in a university setting, the average voter will be computer-savvy. However, it is possible to overstate the difficulty of computer systems relative to pen and paper systems. For a pen and
paper system, it was shown in the 2000 presidential election that a minority of voters doesn’t understand the balloting options. For example, a surprising number of people inadvertently overvoted in a specific pattern. One of these patterns is that people will vote for their preferred presidential candidate and then write-in the running mate. This invalid vote must be dropped despite the voters’ wishes being clear. Computer voting can allow ballots to be structured so that completed ballots are validated to eliminate both undervoting and overvoting. Explicitly providing an abstain/no vote option allows undervotes to be eliminated. In elections allowing multiple choices under a single ballot item, it would still be possible to prevent undervotes by having multiple no-vote options choices. Here is an example of how that might work. Consider a three-candidate election item where up to two candidates may be chosen. To prevent undervotes there would be choices for each of the three candidates plus two no-vote options. To completely ignore this item it would be necessary to select two no-vote options. A potential drawback to this approach to eliminating undervoting is that this could be confusing to the user.

There is a role in this election for a human voting authority that sets up and runs an election. This person will need some tools to do this. These tools must include interfaces to:

- Start and stop the voting process
- Begin the tallying process
- Determine tallying threshold encryption levels
- Identify the location of talliers
- Generate election-unique keys for the talliers and the administrator
- Split the talliers’ private key and distribute it

Clearly, this list is just a general requirements specification, not a detailed design. For security reasons, the user administrator isn’t permitted to perform administrative duties from any location. It is assumed that carefully secured computers with few users allowed any access are used to administer LionVote.

2.3 Areas Where Requirements Are Eased

The generalized voting system requirements are strenuous and on some points contradictory. Based on PSU’s academic environment and the limitations placed on the scope of this project due to time constraints some further restrictions have been placed. One restriction is
that eligibility lists are considered outside the scope of the system. A simple eligibility scheme will be used that doesn’t incorporate a database itself and doesn’t interact with any existing PSU databases that maintain information pertinent to determining voter eligibility.

Another restriction is that connection-oriented anonymous channels are not utilized. An anonymous communication channel is one where the sender and receiver are not visible. An author-based anonymous channel focuses on hiding the author of a published document, whereas a connection-based anonymous channel focuses on making the actual transfer anonymous [WalRC]. LionVote doesn’t provide for connection-oriented sender or receiver anonymity. It is possible for an intermediary, such as a packet sniffer, to monitor transmissions. Based on an initial identification of the voting servers it would be feasible to ascertain who voted and when.

### 2.4 Survey of E-voting Research part I – Broad Research

Ron Rivest, in his presentation for the March 2001 Caltech/MIT Voting Technology Conference, had a very brief introduction to electronic voting that mentions three main points: smart cards, blind signatures, and homomorphic encryption [Riv]. Smart cards represent a potential backbone of a public key infrastructure. Blind signatures, which is the most important technique used in this project, and homomorphic encryption are two ideas first introduced in the 1980’s that play a role in many subsequent proposals. Berry Schoenmakers describes homomorphic secret sharing as a way to combine many secret messages into one message that remains secret [Scho]. In a voting context, this allows votes to be aggregated while still keeping the contents of the votes secret. This technique, which was developed by Josh Cohen Benaloh [Ben], has been used by Berry Schoenmakers, Ron Cramer, and associates for their research [Scho]. Benaloh himself has offered several e-voting proposals that have an interest from an academic perspective but are considered impractical. Impractical issues in his proposals have included voter verification requiring college-level math and restrictions to single bit outcomes in elections. Homomorphic techniques are also relied upon by one of the pioneering e-voting companies, votehere.net.

Another widely used technique in e-voting is threshold encryption. Threshold encryption is due to the pioneering work of Adi Shamir who invented this technique in his paper How to Share a Secret [Sha]. He used the mathematics of LaGrange polynomials to demonstrate how any secret can be shared among $n$ parties so that a threshold $m$ of these parties must pool their
polynomial coefficients to reconstruct the secret. Recombination is impossible with less than \( m \) shares. The polynomial used must be of degree \( m-1 \). Threshold encryption is used in the design section of paper but is omitted from the implementation section.

Several interesting voting schemes have been developed that use different criteria from the ones outlined earlier in the paper. In electronic jury voting, privacy is still important and an additional restriction is made to prevent the final voting tally from being disclosed. It is merely revealed whether a desired threshold of votes has been met or not. Electronic jury voting uses homomorphic El Gamal encryption [HevK]. Another scheme, the Big Brother Ballot, has privacy during the voting process but reveals the way people voted afterwards [Lee].

### 2.5 Survey of E-voting Research Part II – Focusing on the FOO Algorithm

After evaluating several e-voting schemes and implementations, the decision was made to use the FOO (a.k.a. FOO92) or FOO+ (a.k.a. FOO99) algorithm as opposed to developing one from scratch or relying on the work of authors such as Josh Cohen (also known as Josh Benaloh) [Ben] or Berry Schoenmakers [Scho]. There are still plenty of gray areas and design decisions even with a reasonably detailed description such as provided by FOO. The algorithms provided in academic papers tend to aim at persuading, regarding the underlying math and high-level protocol, without getting into implementation details. The FOO92 article is one of the foundation papers in electronic voting as well influencing related concepts such as electronic commerce. Several successful academic projects for college elections have already used a variant of this algorithm. These include the Sensus [CraC] and EVOX ([Her], [Dur]) systems.

The FOO92 algorithm consists of the following. There will be three roles in the voting scheme: many voters, one administrator, and a tallier. The administrator is responsible for ensuring invulnerability. It makes sure that the only eligible voters can vote and that they can only vote once. Privacy is assured in the voter-administrator interactions by using blind signatures, which were invented by David Chaum [Cha]. In a blind signature, the voter blinds (multiplies by a random number) a vote before having the administrator sign it. Thus, the administrator never knows how the voter voted. Upon receiving the reply from the administrator, the voter unblinds the signed ballot (multiplies by the multiplicative inverse of the
earlier blinding) to obtain a signed, non-blinded ballot. This allows the voter to pass along a signed ballot (thus authenticated) to the tallier without providing identification information. This algorithm is complicated by the bit-commitment scheme that is mentioned earlier. Bit commitment provides fairness by forbidding the tallier from examining votes during the voting and tallying phases, but at the considerable expense of requiring the voter or a proxy remain active throughout the entire voting period.

The major criticism, pointed out by numerous critics, of the FOO92 algorithm is that voters are required to remain active during the entire voting and tallying phases. Numerous critics have pointed this weakness out. Jong-Hyeon Lee, in *The Big Brother Ballot*, makes a representative criticism “It is impractical. In practice, voters are able to give up at any phase, and it is another possible representation of their preference” [Lee].

This criticism has lead the FOO authors to provide a response to this criticism. In it, they eliminate bit-commitment and the underlying need for the voter to remain active. They had originally used bit-commitment to achieve fairness, whereas their revised algorithm maintains the fairness criteria through threshold encryption. Instead of having a single tallier that, in the absence of bit commitment, can peek at votes in the voting phase, multiple talliers are used. These talliers individually lack the secret key to interpret votes. The secret key is divided, prior to the election, into pieces for each tallier. They must combine with a certain threshold of other talliers to reconstruct their secret key and thus interpret votes. This threshold could be any percentage desired, even requiring all talliers to join. However, if all talliers were required then failure of even one tallier would mean that it would be impossible for anyone to ever recover votes.

Another strength to the FOO+ algorithm is its limited requirement for public key infrastructure. The administrator needs to have a key pair and the talliers collectively need one. It is possible to implement the FOO+ algorithm using public key cryptography for each voter. However, with Kerberos, the voter doesn’t need a public key pair, or even a secret key or password specific to the election. Voters already possess a general purpose password that can be reused for LionVote participation. This is important because, as discussed previously, a public-key infrastructure is lacking for widespread use. Each voter, and specifically the voting component acting as a proxy for the user, merely needs to identify the public key for the administrator and tallier.
Since PSU has a Kerberos infrastructure, this would be an appealing modification to the protocol. The existing PSU authentication server would receive the PSU access ID and password and then grant a voter a Ticket Granting Ticket. This TGT is subsequently used to obtain DES session keys that are used to communicate with the Administrator. One potential disadvantage to using Kerberos is suggested by Mark Herschberg, one of the EVOX authors. He suggests that using Kerberos limits the widespread utility of a voting system, since it isn’t widely available outside academia [Her]. Since Herschberg’s article was written, Kerberos has grown in popularity, with the spread of Linux and the reliance upon Kerberos in Windows 2000 and XP. In addition, the availability of a Kerberos infrastructure within PSU outweighs this disadvantage.

2.6 Construction of Entities in the System

The biggest architectural decision involved the choice of languages and communication model involved. One option was to use pure Java for the client and server using either Java Server Pages (JSP) or Remote Method Invocation (RMI) as the communication means. Another option was to use Perl for the server with CGI for the communication protocol. A third option was to use Perl and CGI but to have most of the substantive logic in supporting C++ programs.

The pure Java option was discarded due to cryptographic toolkit support being weak or available only as a proprietary product. Sun has put out many standards in recent years including Swing, Java Cryptographic Extensions (JCE), and Java Authentication and Authorization Service (JAAS). The JCE standard has various implementations from third-party vendors such as Cryptix. These various standards and libraries can be confusing to work with. Another factor is that the security programs that Sun has put out to demonstrate their standards are primitive. Sun seems to have left much of the hard work for third-party vendors to resolve. The RSA BSAFE library for Java [RSAS] is an appealing alternative but one whose cost and licensing is prohibitive for academic use.

The pure Perl option was discarded more reluctantly, after much time was spent working with Perl. Free Perl IDE and code library tools are inadequate. The only free GUI debugger available was a beta version of the ActiveState Komodo IDE that was itself bug-laden and unsuitable for use as a debugger. ActiveState has more recent non-beta versions of Komodo (available at www.activestate.com), but this has unfortunately become a commercial product.
Cryptographic support within the basic distribution of Perl isn’t that strong. The author has since learned of the availability of the net::SSLeay module. The name of this module is misleading since the SSLeay library is obsolete and no longer maintained. SSLeay has evolved into the OpenSSL library; it is the OpenSSL library which net::SSLeay supports, not SSLeay. The Perl modules supporting cryptography have a bias towards Unix only compatibility. The only major Perl cryptographic support came from Cryptix, which had already ceased Perl development in favor of Java development. Since the Windows NT operating system family (which includes Windows 2000 and Windows XP) is used for development and development these factors doomed the pure Perl option.

As frustrating as the process of evaluating Perl was, a fair amount of expertise in Perl was gained. Perl is a powerful language that also offers ease of use in creating dynamic web pages. Perl still has a place in this system as a glue language that pieces together the various parts of the system. The pieces that are glued together are largely based on C++ code.

The drawback to this design is the voting client and its need to perform blinding and RSA encryption. The advantage to pure Java client implementations is that no additional code needs to be downloaded. These approaches can be considered a thin-client approach. With a compiled C++ approach the client must either install an executable image locally, rely on a server to perform computation on its behalf, or use some form of local-remote processing (e.g. RPC, CORBA, Java-RMI, Soap). Since C++ binaries are operating system and hardware architecture dependent, C++ installations would reduce the mobility of users, who would be restricted to using only certain kinds of machines. Any remote processing creates the potential for a privacy violation. Even if the server application is trustworthy itself, a hacker may be able to examine virtual memory or hard drives to obtain information about votes.

The design problem of client-side processing is one shared by the Princeton Perl [DavNW] and Sensus [CraC] projects. In the Princeton Perl project, it is assumed that the voting client is really a web server and is trusted by the client. No safeguards are providing against a malicious or untrustworthy voting server. SSL is recommended by the authors to prevent eavesdropping on the communication. Sensus goes in the other direction by not having a web browser interface and relying on a compiled C++ based executable.

RPC was chosen as the communication mechanism between the voting proxy and the administrator. RPC offers a higher-level view on communicating by taking care of the actual
communication protocol and by marshalling abstract data types into a mutually agreed-upon (between client and server) format. It is convenient to have a structure that can denote an RSA public key, for example, and just pass an element of that structure.

After C++ was chosen as the primary implementation language, the next step was selection of a cryptographic library. Four libraries were researched: Cryptlib, RSARef, Crypto++, Libeay/OpenSSL. These libraries, although free, have some rough edges that show they are not professional quality.

Cryptlib [Gut], developed by Peter Gutmann, is the one analyzed most carefully. It has been in existence for a while and Gutmann is well respected, particularly for his work on pseudo-random number generators. Cryptlib also offers abundant support for certificate and key management. The NIST has certified a product that builds on Cryptlib for several algorithms. Cryptlib also offers excellent cross-platform documentation that demonstrates a nice set of APIs that range from simple but inflexible to complicated and very flexible.

Despite all these positives, Cryptlib has some negatives. There is only one developer on the project and he has completed his graduate studies and moved into the commercial world. This does not mean that the software has been abandoned. In addition, the sample code provided is limited. The most important limitation is that it appears that public key manipulation cannot be performed at a low enough level. For example, the rsaEncrypt and the rsaDecrypt functions aren’t exposed as part of the API. The cryptCreateSignature and cryptCheckSignature seem to be the only public-key functions allowed (or encouraged) and they are high-level.

RSARef is another package that was considered ([SHO],[CraC]). This package has the advantage of being developed by RSA, but it is unfortunately no longer publicly available. This library was a very popular library at one time. RSA currently markets the commercial BSAFE API and codebase which is the replacement to RSARef. Versions of RSARef can still be found, including the version that Sensus uses.

Crypto++ by Wei Dai is a package that wasn’t examined as closely as the others were [Dai]. It is a cross-platform solution that reuses some parts of other code libraries. It provides an object-oriented wrapper for those libraries.

OpenSSL, and its predecessor SSLeay, were originally developed by Eric A. Young [VMC]. It is most noticeable for being a cross-platform solution that is used as the basis for Apache SSL support. It can also be using as a general-purpose cryptographic toolkit, since it
offers good Big Number libraries as well as good RSA and DES access. It doesn’t offer as much overall functionality as Cryptlib, which actually uses some of the OpenSSL code, but it offers a more convenient low-level API. Unfortunately, the documentation, as the maintainers of OpenSSL admit, isn’t very good. In addition, as with Cryptlib, the original author has moved on to bigger and better things. Eric Young now works for RSA. OpenSSL was the final choice in the design phase, although use of some Cryptlib functionality was planned as well.

2.7 Detailed Description of Protocol

Hitherto, the algorithm used by LionVote has only been alluded to. Here is a formal description of the protocol used for the design of the system. It is significantly more complicated than the protocol used for the implementation of LionVote. The implementation protocol is discussed later in this paper.

Step 1: Voter authenticates using Kerberos and marks ballot.
1. Voter enters access ID and password to obtain $K_{i,a}$ which is used as the DES session key between the voter and administrator. Both the client and server are reliably aware of the other’s identity (the voter can’t claim to be another voter) under Kerberos. Note that this step omits the details of how Kerberos works and merely takes the output of the process.
2. Upon successful Kerberos authentication, user is taken to the voting page, which is a CGI-generated HTML page.
3. Voter marks ballot ($v_i$) and chooses to submit ballot. The ballot data is transferred using CGI and HTTPS.

Step 2: Voter transmits encrypted ballot to Administrator for blind signature.
1. Voting page communicates with Administrator. RPC is the designated communication mechanism. Voting page encrypts ballot with talliers’ public key. This yields $x_i$.
2. Voter blinds ballot with random blinding factor $r$ (which must be chosen so that the greatest common denominator of it and the public modulus of the administrator’s public key is 1, i.e., they are relatively prime). Blinding takes $x_i$ and multiples it by $r^e$, taking the remainder mod $n$. 

13
where $e$ is the public key component of the administrator’s key and $n$ is the public modulus of the administrator’s key. This yields $e_i$.

3. Voter encrypts $e_i$ with $K_{i,a}$ to yield $s_i$.

4. Access ID is encrypted with $K_{i,a}$ to ensure that administrator has successfully decrypted voter’s transmission. This yields $a_i$.

5. $s_i, a_i$ and access ID are finally transmitted.

Step 3: Administrator validates and processes Voter’s encrypted ballot.
1. Administrator verifies that $a_i$ decrypted with $K_{i,a}$ yields the access ID. If not, the Administrator returns an invalid key error.

2. Administrator looks up access ID in voter rolls for this election. If voter isn’t eligible, the Administrator returns a not eligible error. If voter has already voted the Administrator returns an already voted error.

3. Administrator decrypts $s_i$ using $K_{i,a}$ to yield $e_i$.

4. Administrator signs $e_i$ using its private key component, $d$. This yields $d_i$. This is the pivotal step because $(e_i \cdot r)^d \mod n$ is equivalent to $(e_i^d \cdot r) \mod n$. This means that the voter needs only to remove $r$ from this to obtain a signed, unblinded ballot.

5. Administrator transmits $d_i$. It isn’t necessary to encrypt with $K_{i,a}$ since voter can already verify the Administrator’s signature by analyzing $d_i$.

6. Administrator finally publishes a list of access ID and $e_i$ pairs.

Step 4: Voter completes voting.
1. Voter checks to see that $d_i$ decrypted with $e$ (the administrator’s public key component) yields $e_i$.

2. Voter unblinds $d_i$ by multiplying it by $r^{-1}$ and taking the remainder mod $n$ (the administrator’s public key component). This yields $y_i$, the signed, unblinded ballot.

3. Voter posts $y_i$ and $x_i$ to the bulletin board.

Step 5: After all voting has completed, talliers recover key.
1. Using the LaGrange polynomial method, rejoin the talliers’ private key. Once the key has been rejoined, a primary tallier must be determined. To accomplish this, each tallier is assigned
before voting starts a unique number to arbitrarily order them. If all talliers agree, the lowest numbered tallier is considered the primary tallier.

2. In the event that joining fails (i.e. a tallier is malfunctioning), talliers will repeat the joining process by eliminating one more tallier. This iterative process attempts to eliminate the malfunctioning tallier(s) from the rejoining and counting steps. For each elimination, all combinations of talliers are tried until success is had. If the eliminations continue to the point that there are fewer talliers than the threshold, $k$, then the election is declared a failure. Since this threshold can be set to any desirable level failure should be an extremely unlikely occurrence.

Step 6: A primary tallier performs counting.

1. The tallier iteratively retrieves each entry from the bulletin board. $y_i$, the signed, unblinded ballot, is decrypted with the Administrator’s public key, which should yield $v_i$, the original ballot. $x_i$ is decrypted with the talliers’ private key and should also yield $v_i$. If there is a match then the ballot data is added to the totals.

2. The tallier publishes the final results. It should be noted that, unlike the implementation algorithm, receipt numbers are not given to voters and therefore not published at this stage.

### 2.8 Differences between LionVote Proposal and FOO+ Scheme

There are several significant distinctions between the FOO+ scheme and the LionVote proposal that were mentioned previously. One is that LionVote doesn’t include an anonymous communications channel. Another distinction is the type of user key used. FOO+ does not stipulate a specific type of public or symmetric key scheme. The design for LionVote calls for the use of Kerberos for voter keys. However, the blind signature scheme dictates that the public keys for the Administrator and the Talliers must remain.

Another significant divergence is in the counting phase. The primary purpose of multiple talliers is to ensure the fairness criterion. Once the voting has been completed, fairness will be satisfied. There is no need to have an unnecessarily complicated tallying protocol for this project. Under normal circumstances, the talliers will unanimously rejoin their shared private key. It is only necessary for one tallier to perform the counting act. Having redundant counting would be a nice feature but is unnecessary for an election of this magnitude.
2.9 Ballot Generation Issues

An important decision is in regard to granularity of an election. Granularity refers, in this case, to what an election is. The simplest coarse-grain election scheme is that an election is comprised of multiple items for which each voter has full access to the entire election slate or no access at all. By uniquely naming each election, it would be possible to have multiple concurrent elections running. A finer grain approach would entail a set of election items having their own grouping based on differing voter eligibility. For example, faculty elections might have a classification of Fall ’01 but also have further breakdowns such as all faculty, computer science and engineering faculty only, and department heads only.

The choice of granularity affects both the voting interface and election administration. A finer granularity adds an extra step in the voting process. Due to the simplicity of a coarse-grain election and the author’s experience as a voter with only coarse-grain elections this option is favored. However, any system should be tailored to what the administrators and users want, not the other way around. Therefore, if faculty election requirements favor a finer granularity then this will be provided.

Ballot generation is assumed to use a simple text-based specification. This specification determines the number of election items. Text describing each item is included along with the choices for each item, including write-in. Each item must have a maximum number of choices that can be selected. XML is the design-time choice for the ballot specification language. An option to consider is freeform text choices for a survey type question. Freeform text questions can’t be tallied normally and must only be gathered.

It is assumed that a voting authority directly enters the XML specification. A nice extension to this project would be to write a program that presents an easier front-end to the ballot specification. Such a front-end would eliminate the difficulties involved with correcting an improperly specified ballot.

Once generated, the ballot specification has uses for both the voter and tallier. The voting proxy must lay out the ballot and ensure that ballots are marked correctly. The tallier must double-check that a vote is valid and use the specification to tally correctly.
3 Thread 2: Internal Design and Coding

3.1 Requirements Softening

Requirements softening, aimed at providing what the user really needs with as few man-hours as possible, has caused a major change in this project. The two biggest general areas of requirement softening are in the broad areas of privacy and security. It is a cliché, but one that has truth behind it, that the security of a system is only as strong as its weakest link. This makes security concerns very appealing from an intellectual standpoint involving lots of arguments, proofs, and mathematical reasoning. However, this can be overkill for a faculty election with relatively little at stake compared to corporate elections and government elections. The current voting scheme used by the faculty does offer convenience and mobility but provides no privacy or safeguards ensuring an accurate election. This implementation makes tradeoffs in favor of ease of use and administration against security and privacy concerns.

The current attitude towards what are claimed to be “secure online transactions” is a reflection of the conventional wisdom of software providers and software users. Currently, many commercial web sites offer transactions using SSL. Most users have no grasp of the complexities involved in SSL. For example, there are numerous versions of SSL including, in increasing recency of their establishment, version two, version three and Transport Layer Security (TLS). Version two is supported by many browsers but is considered to be flawed by most security experts [VieMc]. SSL is also extremely vulnerable to man in the middle attacks in which the server is not who the client thinks it is. Most users don’t understand PKI, don’t know how to configure the security settings in their web browsers, and don’t know how to scrutinize a certificate offered by a site. In essence, security is taken on a reluctant blind faith by most users.

For privacy, there is also a similar lack of knowledge by users but, worse than that, there is an ignorance, and even disdain, on the part of software providers for privacy issues. Over the past several years, there has been an awakening within the industry to security concerns. Microsoft is the perfect example of a company that has a new understanding of security as functionality that people will pay for. Having all developers spend a month on security training may have been done in part as a cynical public relations move, but it has contributed to a great increase in security-related fixes. However, Microsoft initiatives, such the authentication infrastructure initiative called Passport and the Digital Rights Management (DRM) operating
system codenamed Palladium, show that Microsoft has yet to grasp users’ desire for privacy. Users may not understand the complexities necessary behind privacy, just as they lack the understanding of security, but they definitely do desire both criteria. The Passport system, ironically codenamed HailStorm, was met with a hailstorm of criticism by people who hated a system in which Microsoft would be responsible for providing a repository of personal user information including credit card numbers. Passport has been de-emphasized by Microsoft due to the widespread criticism and hostility on the part of users. A form of authentication infrastructure, which Passport was one attempt at, is a prerequisite for significant expansion in online commerce. However, Microsoft placed too much emphasis on simplicity for end users and trusted in companies’ ability to handle sensitive information appropriately. Privacy was not adequately considered in Passport. With Palladium, Microsoft again is attempting to make Big Business happy in the area of multimedia purchasing and usage by restricting the ways in which people can copy or modify information. Here again, the demands of users for privacy are not being adequately considered in requirements analysis.

Having established why security and privacy requirements have been softened, here are the concrete ways in which that softening has occurred. One way is in the strength and situations in which cryptography is used. Another way is the absence of safeguards in voting administration. Finally, the most concrete way is an elimination of one the most important privacy safeguards within the algorithm- multiple talliers.

Cryptography usage is a detailed design issue that is largely beyond the scope of the high-level algorithm description earlier. One obvious issue is key size. With many cryptography algorithms, the time it takes to crack a message varies directly with the size of the encryption key. For the sake of speed in testing, 512 bit keys for RSA have been used when security experts are now recommending 2048 bit keys or greater [VieMc]. Changing key size isn’t a major architectural change. However, it would be significant to change the key size in combination with ensuring that a certain response time can be given to users based on an average and/or maximum transaction rate. With a high volume of concurrent users, there is a heavy burden placed on the servers. It is much more difficult to do something reasonably efficiently than to do it in a straightforward or naïve way.

Another softening is the decision to simplify things by limiting usage of safeguards for the administrator. It is a security truism that a good key should consist of something you know
AND something you have. It would be nice to have some physical key to be one component of administrative security. Using a Java Cryptographic iButton, one cheap form of a smart card, would satisfy this need. A group of Stanford students developed a prototype using the iButton for all voters [ClPR]. Another component of the key should be a password required for entry so that someone couldn’t just use the physical key in an unauthorized way. However, the decision has been made to make things somewhat easier for the administrator (and much easier for the programmer!) by limiting the usage of this kind of safeguard.

Finally, a major change to the LionVote algorithm was the consolidation of the multiple talliers into one. Remember, multiple talliers provided redundancy in the case of hardware or other failure and provided the fairness criterion. Fairness is an important criterion, but we can provide a modicum of this by not deliberately designing the software to have back doors and by having the tallier tally only after voting has been completed. In the case of LionVote, tallying is done when the tallying software is shut down by a user.

Another important change to the algorithm is the replacement of Kerberos encryption with a do-it-yourself password scheme. The role of Kerberos was to provide authentication of a Penn State access ID. The voting software would then only need to determine which access IDs were authorized for a given election. Potential difficulties in making Kerberos work were too high of a risk for such an essential election component. If there were deployment difficulties with it, the whole voting system would be unworkable. In addition, to integrate Kerberos into LionVote would require working closely with the Penn State bureaucracy. Another drawback to Kerberos is that setting up a Kerberos testbed is a difficult task.

An alternative to Kerberos is designing the Administrator module to perform authorization and authentication functions. The human administrator is responsible for generating a list of eligible voters for each election. The system then would use a dictionary to generate unique passwords for each user. These passwords would be good for one election only. A password would be an easily remembered sequence of words and alphanumeric characters such as mork7elf. Password distribution would be done through campus snail mail.

The online bulletin board system is another requirement that has been softened. This bulletin board was intended to display encrypted ballot information in a mix network manner. A mix network mixes the displayed order of information so that a voter cannot be connected as easily with the ballot he or she casts. It was a design choice not to have intermediate results
available, as many online polls do. Intermediate results might either discourage voters from voting or encourage voter coercion. Since no user really needs to have encrypted ballot information visibility, and because there is now only one tallier, a simpler protocol is used. The tallier now provides a receipt number to each user and publishes a list of receipt numbers after the election. A receipt allows each user a small degree of confidence that his or her ballot has been counted; although he or she can’t verify that his or her vote has been accurately counted. It is not possible for any one person to ascertain if the entire election has been run accurately. Nor is it possible for anyone to prove how he or she voted, which means that this election is technically still receipt-free by the definition of the voting literature.

### 3.2 Development Timeline

The timeline of the project can be broken down at a broad level into two threads: requirements and tools analysis resulting in a high-level design, and detailed design and coding which resulted in a working voting system. Thread one was fundamentally exploratory and open-ended. A vast number of approaches in the computer science literature to electronic voting were analyzed in this period. Also at this time, a variety of programming languages including C++, Perl, and Java were explored. Thread two, except for the requirement softening described earlier, used the major decisions made in thread one. It is possible to break the development process down even further into nine phases of active development that are illustrated in figures 1 and 2.

![Figure 1: LionVote Design Phases](image-url)
The first four phases constitute thread one of development. Phase one involved doing untargeted research into the existing literature for electronic voting and the tools that could be used to implement a voting system. The basics of web server development and cryptography were also concurrently explored. Four months were spent in this phase and three months in the next phase, a more targeted research. In this second phase, the algorithm to be used, the FOO and FOO+ approach, was selected from the most widely cited papers. In the third phase, which took two months, an initial design was made. At this time, the mechanics of how to implement blind signatures and Shamir secret sharing were also explored. In the fourth phase, the external design was finalized and a design specification was prepared for faculty review.

The next five phases constitute thread two of development. Phase five was development of a voting prototype program. Phase six involved use of the OpenSSL cryptographic toolkit to develop all cryptographic functions needed for the voting system. Phase seven involved development of what could be considered an alpha release of the complete voting system, which is labeled Version 1.0. Phase eight involved removing all restrictions on the size of the ballot that the system can process. Phase nine, the current phase, involves adding features and smoothing rough edges in the program to the point where the program is considered worthy of release.

The prototype project in phase five, which is labeled Bootstrap, established a framework to which features were incrementally added. This prototype initially involved two separate components that didn’t merge until phase seven. One component was ballot specification and rendering of this specification as a web page. For this component, the earlier decision to use XML was the beginning. A format was created that identified a unique election name, allowed for a variable number of questions with a variable number of possible answers (see Figure 3).
Each question also has a number of election answers that must be selected. Having a fixed number of selectable answers eliminates potential problems with undervoting and overvoting. A flexible XML format was used that only requires the ballot specification to be well-formed, not having the additional structure, which comes from validating the content of the document. Validation is done within the Voter and Tallier software itself. Using a flexible format does allow for the possibility of the administrative user creating an invalid ballot specification. In the future, the plan is to have a wizard-based program actually generate the XML specification for the user to prevent them from making mistakes.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<ballot_questions Election_ID = "FEB-28-2002">

<ballot_question ID = "Who's the boss?">  
<choices>2</choices>  
<answer>Tony Danza</answer>  
<answer>Alyssa Milano</answer>  
<answer>The Mom (Judith Light?)</answer>  
</ballot_question>

<ballot_question ID = "Who's in charge?">  
<choices>1</choices>  
<answer>Charles (Scott Baio)</answer>  
<answer>Charles' Goofy friend - played by Willie Ames</answer>  
<answer>One of the Kids</answer>  
<answer>One of the parents</answer>  
</ballot_question>

</ballot_questions>

Figure 3: Sample XML ballot specification

XML proved to have several strengths. Recent Web browsers have a native display of XML pages that illustrates the treelike hierarchy necessary for a well-formed XML pages. In
addition, there are numerous free XML parsers available that eliminate most of the work in parsing. Programming done using a parsing library can deal at a higher level with tags rather than at a lower level with bytes in a file. Parsers also tell if a document is well-formed or not and if not the location of where the parsing error occurred. This eliminates much of the error-handling for files.

The other component in phase five was RPC interfaces for the voter proxy, tallier, and administrator modules. The initial version was just an RPC-based hello world set of programs with a stab at the RPC interface that was intended to be used. This provided a framework to build on and forced the author to keep distinct, from a very early stage, the roles that each voting component played.

There was some overlap in time between phase five and six, in which cryptographic functionality was developed. Phase six turned out to be the most difficult phase due primarily to the choice in cryptographic libraries. In the design phase, OpenSSL had originally been chosen over Cryptlib. OpenSSL was poorly documented and had a messy low-level API whereas Cryptlib had a high level API and appeared well documented. Unfortunately, a bad decision was made in phase six to turn back to Cryptlib due to the lure of the seemingly well-documented API. Peter Gutmann, the author and documenter of Cryptlib, can be a very articulate gadfly on security issues and is an expert on sources of randomness but isn’t that strong on writing documentation that illustrates how simple tasks can be completed.

After several months of having little success with Cryptlib, a return was made to the OpenSSL library. In the intervening year since initial research, OpenSSL documentation had improved significantly. Some parts of the API such as file I/O for keys are still undocumented but other parts, such as their invaluable generalized I/O abstraction, the BIO, have now been documented. The undocumented APIs also tend to lack consistency in organization and naming conventions for functions. In addition to recent improvements within the OpenSSL documentation itself, a variety of OpenSSL books have come out, most notably Network Security with OpenSSL ([VMC]). These books came out a little late in my learning curve but even without them OpenSSL proved to be much less daunting than it appeared to be at first.

An important aspect of cryptography that is often overlooked is pseudo-random number generation. People often use only the weak pseudo-random number generation that is provided with most operating systems. This weak generation typically uses linear congruential functions.
With such functions, it can be possible to discern, based on a few consecutive samples, what the forthcoming sequence of random numbers will be. One feature of Cryptlib that was successfully utilized is the well-regarded Windows-specific random number generation functionality. Cryptlib uses, as sources of randomness, information about the Cryptlib-using program as well as system information such as I/O and network-related information [Gut]. Anyone spending much time with Windows can attest to the fact that Windows is non-deterministic!

Once the cryptographic code was written a working alpha-level version of the software was developed in approximately a month. This quick development process was made possible by the use of several programming libraries. The cgi.pm Perl module and some very good example programs from Lincoln Stein were used to help develop the web part of the voter proxy [Ste]. The proxy first renders an HTML page that primarily is a form consisting of the ballot specification (see Figure 4). Once the user fills out and submits the form, the cast ballot is put into XML format. The cast ballot, access ID, and one time only password are passed to the C++ based program which interacts via RPC with the Administrator and Tallier modules. After all of the encryption and decryption is done, the Tallier uses the ballot specification and the cast ballot XML file to add entries to the totals. For XML writing the XML::writer module of Perl was used and for XML reading the Expat non-validating parser library in both Perl and C++ was used. An additional library that was helpful was the hash functionality of OpenSSL. This hash library was used for the Administrator role in determining voter eligibility and checking to see if he or she had already voted.
Phase eight, support for any size ballot specification, proved challenging. This phase was made more challenging by failed to develop a clear plan. This violates an important programming rule – have a good plan of what to accomplish and how to go about implementing that before beginning coding. The typical consequence of failure to have a plan is software that is messy in various ways. In this case, structures, global variables, and functions were used rather than taking an object-oriented approach that would have provided abstraction and data hiding. Classes can make dynamically memory management, which this project uses heavily, much easier through destructors. The internals of LionVote require a multi-level approach to memory deletion that requires a detailed understanding of the data structures used. An even more egregious offense of the LionVote implementation is use of two similar modules to accomplish the same task, chunking a piece of data that may be any size into a group of even-sized blocks. These modules could have been consolidated into one.

The specific goal in this phase was to take RSA encryption, which normally accepts as input as many bits of data as its key size, and extend it into a block-oriented encryption model.
As explained in the Handbook of Applied Cryptography ([MeOV]), RSA can be readily extending using the Electronic Code Book (ECB) or Cipher Block Chaining (CBC) modes. Electronic Code Book mode treats each block as individual unit without chaining it to the content of other blocks. This approach is conceptually simple but is generally more susceptible to a variety of attacks, including statistical attacks on identical blocks that recur over time [Schn]. ECB is used for LionVote since simplicity is valued over strong security. For RPC marshaling, numbers were converted into variable-length strings. Since a block-oriented model is necessary, each step of the process is turned into a for loop, for each block of the message. Then, after a voting component completes a step of processing, an array of Big Integers is passed via RPC. Since Big Integers must be marshaled, an extra level of complexity was involved.

An additional detail to maximize security is padding of messages. Padding ensures that blocks with short messages aren’t easier to decrypt than blocks that are used fully. Padding uses transformative functions that are designed to be very difficult to invert. OpenSSL’s implementation of Optimal Asynchronous Encryption Padding (OAEP) to pad messages was used. OAEP for OpenSSL is an undocumented low-level API that they don’t expect most programmers to use directly. It isn’t planned to use this padding on versions prior to and including the first actual faculty election, in order to avoid additional complexity.

3.3 Revised Detailed Description of Protocol

The design version of the protocol for LionVote was described earlier in the paper. This implementation modifies the design version based on the requirements softening topics discussed earlier. Important modifications include substitution of a password generation scheme for use of Kerberos and consolidation of multiple talliers into one. One enhancement, based on user feedback, was added to the role of the tallier. This is the added function of providing an immediate receipt number to voters, which they can use after the election to gain confidence that their votes were counted. Figure 5 demonstrates a simplified version of the implemented protocol. This figure illustrates the interactions between the different server components: WWW server, voter proxy, administrator, and tallier.
Figure 5: High-level overview of implementation protocol
Step 1: Voter marks ballot and enters authentication information.
1. User is taken to the voting page, which is a CGI-generated HTML page consisting of the ballot and fields to enter access ID and unique per-election password.
2. Voter marks ballot (\(v_i\)) and chooses to Submit ballot. The ballot data is transferred using CGI and HTTPS.

Step 2: Voter transmits encrypted ballot plus authentication information to Administrator for blind signature.
1. Voting page communicates with Administrator. RPC is the designated communication mechanism. Voting page encrypts ballot with tallier’s public key. This yields \(x_i\).
2. Voter blinds ballot with random blinding factor \(r\) (which must be chosen so that the greatest common denominator of it and the public modulus of the administrator’s public key is 1, i.e. they are relatively prime). Blinding takes \(x_i\) and multiples it by \(r^e\), taking the remainder mod \(n\) where \(e\) is the public key component of the administrator’s key and \(n\) is the public modulus of the administrator’s key. This yields \(e_i\).
3. To keep symbols consist between original and revised protocol \(e_i\) is equivalent to \(s_i\). In the original protocol, \(e_i\) and \(s_i\) were distinct based on encryption using a Kerberos key.
4. Access ID is encrypted with Administrator’s public key to ensure that no eavesdropper can identify the voter. This yields \(a_i\).
5. \(s_i, a_i\), and the Voter’s password are finally transmitted.

Step 3: Administrator validates and processes Voter’s encrypted ballot.
1. Administrator uses its public key component to retrieve the access ID from \(a_i\).
2. Administrator looks up access ID in voter rolls for this election. If voter isn’t eligible, the Administrator returns a not eligible error. If voter has already voted the Administrator returns an already voted error.
3. Administrator verifies that the eligible voter’s password is the correct one. The administrator has a list of the per-election passwords for each voter.
4. Administrator signs \(e_i\) using its private key component, \(d\). This yields \(d_i\). This is the pivotal step because \((e_i * r^e)^d \mod n\) is equivalent to \((e^d * r) \mod n\). This means that the voter needs only to remove \(r\) from this to obtain a signed, unblinded ballot.
5. Administrator transmits $d_i$.

6. After the election is complete, Administrator publishes a list of access ID and $e_i$ pairs.

Step 4: Voter prepares submission to tallier.

1. Voter checks to see that $d_i$ decrypted with $e$ (the administrator’s public key component) yields $e_i$.
2. Voter unblinds $d_i$ by multiplying it by $r^{-1}$ and taking the remainder mod $n$ (the administrator’s public key component). This yields $y_i$, the signed, unblinded ballot.

Step 5: Voter sends signed ballot to tallier and receives immediate receipt.

1. Voter communicates $y_i$ and $x_i$ to the tallier
2. The tallier adds this information to the list of ballots to be tallied. The receipt number is immediately provided to the user.

Step 6: Upon completion of election, Tallier tallies votes and publishes results and list of receipt numbers used for the election.

1. The tallier iteratively retrieves each entry from the bulletin board. $y_i$ is decrypted with the Administrator’s public key, which yields $v_i$, the original ballot. $x_i$ is decrypted with the tallier’s private key and also yields $v_i$. If there is a match then the ballot data is added to the totals.
2. The tallier publishes the final results and the list of voter receipt numbers that was generated during the course of the election.

3.4 Deployment

Deployment of any software system has plenty of unknowns, and LionVote was no exception. Foremost of the unknowns was how the system will actually perform. It is always an accomplishment to make software behave in a stable manner on the testbed, but it is difficult to foresee all of the things that could go wrong. LionVote’s implementation is relatively inflexible due to processing that occurs based on non-persistent memory-based storage. The lack of persistent memory made ensuring software reliability even more important.

Another area that is harder to foresee in deployment is installation hassles with the OpenSSL library, Expat library, RPC run-time, or Apache web server. One common problem
with Windows programs is “dll hell”. If there are multiple dlls versions of a code library and the wrong one is chosen or there is only version but it is the wrong one then the program will crash upon startup. This can be a hassle with OpenSSL in particular since many programs, including Apache, require a specific version of the OpenSSL library. There can be setup issues involving which accounts programs are run under and whether those accounts have appropriate security permissions. It can also be a bureaucratic struggle to get human voting system administrators the necessary permissions. Another issue is that several modules that may not be in the base Apache configuration, mod_env, mod_perl, and mod_ssl are either recommended or required for the voting system.

In addition to setup and reliability related deployment issues there are non-essential features requested by the user. Of course, if it were always clear what is and isn’t essential then users and software developers would never be in conflict. As time permits, tweaks will be added that will make future elections easier for end users to administer.

3.5 Comparison of LionVote with Other Student Voting Implementations

There are three student papers that were studied in-depth during development of LionVote. Each system uses a variant of the FOO algorithm. Princeton has a partial implementation done by three students for an undergraduate project in 1995. Washington University of Saint Louis has the Sensus project, which was implemented between 1996 and 1997 [CraC]. The Sensus project was developed by one person and was used for mock elections only. At MIT, the EVOX project was developed with the help of grants from DARPA and NTT [Her]. The MIT EVOX project was done by a team of students and led by Professor Ron Rivest. Their software has been successfully used to run their 1999 MIT campus elections. The source code is available for all systems discussed above except EVOX.

The Princeton project was developed by Ben Davenport, Alan Newburger, and Jason Woodard [DavNW]. It is a well-designed idea that ran into the end of the semester. Their use of cryptography doesn’t use any cryptography toolkit. Instead, the students rolled their own cryptographic functionality using Perl’s Math::BigInt library. Their project failed to support blinding for a reasonable key size within even an hour of processing time. Failure to support
blinding can be tied to a naïve approach to modular exponentiation. Using advanced methods such as the Montgomery method modular exponentiation can be greatly speeded up. To be fair to these students it must be understood that using Perl in 1995 was a cutting edge thing to do. In many cases, as in this one, being on the cutting edge means going with support of library functionality. At that time, the only major cryptographic toolkit that the author is aware of was the RSARef C library. Perl does support the capability for modules to rely on code written in other languages but no one had made such a module publicly available. In 2002, Perl is a mature, well-documented language with OpenSSL functionality available via the Net::SSLeay module.

Washington University in Saint Louis has the Sensus project, implemented by L.F.C. CraC. It is the college voting implementation that has received the most press. The FOO+ article examines this algorithm to be a “direct implementation of the FOO92 scheme (without the anonymous channel).” As such, the project has a weakness in the convenience voting criterion because software representing voters are required to remain active during the entire election. In the FOO92 article, and Sensus, voters reveal their ballot, via bit commitment, during the tallying phase. It is a well-documented system that is reasonably close to being suitable for an election. One liability it has is that it creates its own ballot specification language. This language is well designed, but parsing any language is a serious task and a difficult task to hand-code. Trying to parse a homemade language manually can also be difficult to do. Well-known languages have many examples that help one gain familiarity with the language. Tools such as Lex and Yacc can eliminate much of the pain of writing a parser. A solution such as XML, which LionVote uses, is preferable because it is both widely understood and supported by a wide variety of robust parsers. As with the Princeton project, this is in some ways not an entirely fair criticism since XML didn’t exist at the time of the Sensus project. The Sensus project also has some items hard-coded that would need to be cleaned up to be reusable or a new executable would have to be generated for each election. Another criticism of Sensus is that the voting system consists of client software that is Unix-specific. This affects the criteria of mobility, because only Unix computers may be used, and convenience, because the software must be installed on the system used by each user. LionVote, in contrast, allows most Internet-connected users to access the system from their own computer.
MIT has a project mentored by Ron Rivest called EVOX that implements a variant of the FOO scheme [Her]. This implementation is criticized in FOO+ for having the voting client send its bit commitment to the tallier during the voting phase, which eliminates the purpose of the bit-commitment—providing fairness by keeping all votes secret during the voting phase so that no intermediate results could be known [OhkM]. That criticism is certainly applicable but bit commitment, while intellectually interesting, is the most impractical aspect of the FOO scheme. The secret sharing approach of the FOO+ article with multiple talliers is much more practical. Another criticism of EVOX is that election results are not publicly posted, as LionVote does. LionVote publishes lists of both voter receipt numbers and the election totals. It should be understood that the EVOX project did continue after the Herschberg article. Therefore, criticism of this version of the system may not accurately reflect more recent versions of the software, for which comparable documentation and/or source code is unavailable.

The Penn State project can be compared favorably in some ways to each of these projects. It is ready for use in an election, which is a slight improvement over Sensus and a big improvement over the Princeton project. It would appear from the information available that the MIT project is clearly superior, which is natural since it is significant larger in ambition and resources devoted to it. The design, but not implementation, of LionVote does include a notable improvement to all of these projects in that fairness is assured via use of multiple talliers without inconveniencing voters by mandating that they continue to participate in the election.

The following chart summarizes the voting criteria that LionVote meets.

<table>
<thead>
<tr>
<th>Voting Criteria</th>
<th>LionVote design</th>
<th>LionVote implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Convenience</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Invulnerability</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mobility</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Privacy</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Verifiability</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Fairness</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

*Figure 6: LionVote and voting criteria*
3.6 Software Engineering Lessons Learned

One of the biggest software engineering lessons learned involved time management and making hard decisions. Working on a project intermittently and with only self-imposed deadlines has resulted in a software engineering scheduling nightmare. Spending numerous hundreds of hours over two and half years to produce the LionVote system would get any real world development team fired! Without any money or demanding customers involved, the goals of the project were really quite different though. The learning experience was paramount. Many things learned were intangible and can’t be directly illustrated in the final system. However, this emphasis on learning was simply a crutch at times. It was used to make excuses for taking tangents that were clearly dead-ends, particularly the time spent with the Cryptlib library. In this section, several concrete techniques that rectified problems with decision-making and time management are discussed.

One of the most valuable techniques used in the writing this software is the development of honest to-do lists broken down into various categories. One to-do list might include features being considered for longer-term development and another might be for very near term bugs and kluges. Keeping things in ones head can work reasonably well when on a project every day full-time. However, with this voting project being done on weekends and evenings it is possible for weeks to go by without any work being done. After a return to the LionVote project after a hiatus, without an accurate picture in writing of project status, it can take added time for progress to resume.

Another technique that is relevant to use of to-do lists is to feature prioritization. Many open source projects, notably Mozilla, are a good example in this regard. For a given release, features can be categorized as essential, too complicated for the present, or not essential. Once it is clear what the essential features are, it is easier to develop a schedule on what to code. With prioritization information available, critical path (CPM) analysis can be performed. CPM can help identify things that cannot be put off until later. Use of project management software, such as Microsoft Project is highly recommended, to do CPM and other types of analysis. LionVote didn’t use project management software, although that certainly would have improved the process.

Feature prioritization and to-do lists were helpful in developing another technique that proved helpful – a blog, or web-based log, that offered regular status updates on the project.
This was useful in communicating with the faculty supervisor. Even more importantly, it helped to keep momentum in the project. In a part-time project, where the availability of time for development was intermittent it was extremely helpful to have a timeline of where the project had been. This helped make it clear where things needed to go next.

Another significant lesson learned was the need to use version control software regularly. After having a few problems keep tracking of the most recent version of files, the practice was adopted of putting changed files into Microsoft Visual SourceSafe every day or two of development. With the availability of free, open source programs such as CVS no software worth writing should be written without version control. This practice has the added benefit of allowing you to discover where mistakes were introduced into the process. In addition, code tends to be cleaner when version management is used because programmers no longer feel obligated to keep every observation, regardless of its current relevance, in the active baseline.

Another thing that goes hand in hand with version control is backups. Having a version control archive isn’t as reliable if you don’t periodically back up the entire archive. Backups also apply to an installation of software required by the voting system and of software used to develop the voting system. Any hard drive will eventually fail, the only question is when.

3.7 Future Enhancements

Many enhancements that could be made to this system by future student researchers. Notable enhancements could include better administrative tools, smart cards for administrative access, database support, online bulletin board system, Kerberos integration, a purist approach to web services, and an installer program.

A more pure web service concept is a topic that hasn’t been discussed before. LionVote has a thin web-based veneer, which runs a series of Windows specific programs running a relatively old-fashioned RPC model. Over the past several years, several higher-level remote code invocation models have been created such as XML-RPC and Soap. These interfaces can eliminate the need for programs that are reliant on Windows, or other specific operating systems. Unfortunately, SOAP and XML-RPC currently have limited support for C++ on Windows; Java is the best-supported language for these models. Another approach to a less platform-dependent system is to use the Java language’s Remote Method Invocation (RMI) feature. As mentioned
earlier, the lack of a free, widely used, mature cryptographic library eliminated Java as a possibility.

An installer program would be another valuable addition to LionVote as a product. Improvements in recent years in installation ease from the open source software community make software much easier to use. In the past couple of years, Apache has really matured, with distributions such OpenSA and their own binary installer distribution that makes Windows installation a snap. The system would be much more reusable if a Windows installer making program was used to create an easy to use installer.

4 Conclusion

The primary objective of LionVote was providing a fully functional e-voting system that improved on the very poor security and privacy of the current faculty voting system while keeping most of the mobility and convenience of the current system. The project used a modified version of the FOO+ algorithm for its design. For the actual implementation, the design was simplified to allow a one-person project to give the user the things they absolutely needed without the inclusion of inessential and time-consuming features. The differences between design and implementation illustrate the secondary objective of being a representative case study in software engineering. We hope that the LionVote system developed will be put to use by faculty and that the software engineering lessons learned can be applied by other students, perhaps even to an enhanced version of the voting software.
References


Michels, Markus and Horster, Patrick “Some remarks on a receipt-free and universally verifiable Mix-type voting scheme”, Advances in Cryptology ASIACRYPT ’96, Springer-Verlag.


Rivest, Ron PowerPoint presentation for Caltech/MIT Voting Technology Conference March 30-31, presentation was found at www.vote.caltech.edu.


Shostack, Adam, “RSAREF: Don’t Use RSAREF”, http://www.homeport.org/~adam/crypto/rsaref.phtml

