Performance Optimization of a Split-Value Voting System

by

Charles Z. Liu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

Author .....
Department of Electrical Engineering and Computer Science
June 8, 2015

Certified by .....
Prof. Ronald L. Rivest
Thesis Supervisor

Accepted by .....
Prof. Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Performance Optimization of a Split-Value Voting System

by

Charles Z. Liu

Submitted to the Department of Electrical Engineering and Computer Science
on June 8, 2015, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

TODO: ABSTRACT

Thesis Supervisor: Prof. Ronald L. Rivest

Acknowledgments

TODO: ACKNOWLEDGEMENTS

Contents

Chapter 1

Introduction

As a cornerstone of democratic governments, elections are the subject of intense scrutiny. Voters expect that the systems and procedures put in place are easy to follow, efficient, and inexpensive. However, history tells us that in practice, this may not always be the case.

Perhaps the most well-known example of the failure of voting systems in the current generation is the 2000 Presidential Election in the United States, most notably the series of recounts in Florida that delayed the nationwide election result by several weeks and opened cases in the Florida Supreme Court and the United States Supreme Court. Even in the aftermath of the election, studies were inconclusive as to whether the county-specific recounts or a statewide recount would have swung the results in Al Gore's favor. The problem was exacerbated by the media's premature declaration of the election outcome, George W. Bush's narrow margin of victory (around 500 votes), and controversy surrounding the "butterfly ballots" that were used in Palm Beach for the first time.

Studies conducted after the 2000 Election in Florida reveal many issues that appear to be quite subtle in the larger picture of a Presidential election: the public's familiarity with a new ballot layout and instructions, the timing of media releases and exit polls, and even the degree to which paper ballots are cleanly marked or punctured by the voter. In the aftermath of the chaos, Congress passed the Help America Vote Act, which attempted to help states upgrade their election infrastructure to prevent similar issues four years later. Many states turned to electronic voting systems. Unfortunately, these systems have been met with criticisms over their accessibility, relability, and security. Furthermore, many of these proprietary systems have reached or are approaching the ends of their lifetimes,

prompting another wave of research and development on secure, verifiable, efficient, and reliable electronic voting systems.

Elections are not only thwarted by the failures of the mechanisms used in them. Reports of manipulated or lost ballots, coerced voters, and simply corrupt election officials are no surprise in certain parts of the world even today. In decades past, voters in Italian elections often had the option of selecting ranked preferences among a long list of candidates. The clever "Italian attack" reportedly used by the Mafia involved coercing a voter to vote for the coercer's first choice candidate, followed by a sequence of low-probability candidates in some predetermined order. With a long list of candidates, the chance that another ballot contains the exact same sequence in the same slots is very low; thus, if the votes are publicly broadcast following the election, the sequence becomes a signature for the coerced voter that the coercer can check, even if all other identifiable information is stripped.

Clearly, there is much more to be done in advancing election systems. In this section, we briefly survey the history of voting systems, including electronic voting systems, and conclude with a discussion of desirable properties in electronic voting systems.

1.1 A Brief History of Voting Systems

The first recorded elections occurred in the sixth century BC in ancient Greece, where adult males were charged with the responsibility to vote at meetings of the assembly. In the beginning, voting was typically done by a raise of hands in a public space. This developed into the use of colored stones, where voters would select a colored stone out of two possible colors and place it into a central jar. Although we use secret ballots today without a second thought, the first "secret ballots" were used in the ancient Greek and Roman civilizations, where records exist of contraptions that allowed a voter to hide the stone as it was deposited into the jar. By the end of the nineteenth century, most Western countries, including the United States, Australia, and France, demanded the use of a secret ballot at elections, and voter privacy was a non-negotiable property of official elections.

As populations grew and election officials looked for ways to make elections more efficient, mechanized voting machines were developed. Lever machines were introduced in New York during the 1892 Election. Punch card systems were introduced in 1965 and were also widely used until they drew sharp criticism during the 2000 Election in Florida. Lever

machines, punch cards, and other mechanical and electrical systems such as optical scanners provided different tradeoffs in cost, convenience, efficiency, usability, accessibility, and security. However, in the past decade, growth in both the power and popularity of digital technology has enabled a shift towards electronic voting systems.

1.2 The Rise of Electronic Voting

In the aftermath of the 2000 Election, the Help America Vote Act instituted a series of requirements on the states which aimed to replace punch card and lever machines, establish standards on voter privacy and accessibility for voters with disabilities, and improve the usability of voting systems. The number of direct-recording electronic (DRE) voting machines grew in response. DREs have a number of advantages over their mechanical counterparts.

- They enable rapid, accurate tallying of votes.
- They may include numerous accessibility functions, notably those for the visually impaired.
- They may include many languages.
- They may catch errors such as undervoting and overvoting before the vote is cast, preventing invalid ballots from being discarded.

However, DREs are often complex systems containing proprietary hardware and software, making it difficult to verify their correctness and security properties; indeed, studies have shown the existence of numerous security vulnerabilities in many commercially available electronic voting systems. Because of the lack of a paper trail, any errors in the system are unrecoverable.

A solution to the lack of verifiability in DREs was proposed in 1992 by Mercuri [CITE THIS], creating a voter-verified paper audit trail (VVPAT) system. In the simplest VVPAT system, a DRE machine is augmented with a printer. In addition to submitting the vote electronically, the machine prints out a human-readable copy of the voter's selections. The voter may then confirm that the paper record is indeed correct before depositing it into a ballot box. In case of a disputed election, recount, or malfunctioning machines, the paper records may be consuled. VVPAT systems may be extended so that the voters may keep

a printed receipt; however, care must be taken to ensure that the printed receipt does not reveal the voter's vote to anyone else.

Today, VVPAT systems are the most common type of electronic voting system; 27 states in the United States require a paper audit trail by law and an additional 18 states use them in their elections.

1.3 Desirable Properties in Electronic Voting Systems

The two main propeties desired in an election are *privacy* and *verifiability*.

Privacy concerns the ability of an adversary to gain information on a voter's choices in an election. A lack of privacy results in the possibility of coercion or bribery as seen in the earlier example of the "Italian attack". Note that the adversary is not limited to an outside third party; privacy is broken if a corrupt election official may associate any vote with the corresponding voter. As voting systems have become more complex and electronic voting systems have become more popular, the notion of privacy has evolved.

The simplest form of privacy is *ballot privacy*, which stipulates that the contents of the ballot must remain secret. With typical paper ballots such as those used today in the United States, ballot privacy guarantees that there can be no association made between a voter and her ballot, thereby providing privacy to the voter. However, with electronic voting systems, a stronger definition of privacy is desired. *Receipt freeness* or *incoercibility*, first introduced in [BT94], concerns the inability of the voter to later prove how she voted. Equivalently, it is impossible to effectively coerce a voter by forcing her to reveal her vote following the election. The concept of receipt freeness grew in importance as electronic voting systems often leave a receipt that the voter may bring out of the polling site.

In certain areas of the United States and other countries including Switzerland and Estonia, Internet voting has become commonplace. In Estonia, for example, voters may scan their government issued ID cards to authenticate themselves into a voting website. Many more states and countries have vote-by-mail, in which ballots may be mailed ahead of the election date. A stronger notion of *coercion resistance* due to the work of [JCJ05] identified the need to prevent adversaries from jeopardizing voter privacy in these cases, for example, by stealing or forcing a voter to reveal her authentication credentials.

Verifiability concerns whether individual voters or the public as a whole may verify

that the election outcome faithfully represents the selections of the voters. In particular, *individual verifiability* refers to the ability of a voter to verify that her vote was recorded in the election. *Universal verifiability* refers to the ability of anyone in the public to verify that the declared election outcome matches the set of votes that were recorded in the election.

Although today's paper ballots provide good privacy, they do so at immense cost to verifiability. Despite thorough documentation of best practices concerning how ballots should be handled, who should touch them, how they should be sealed, etc. [ACE Electoral Network], it remains practically impossible for a voter to know for certain that her ballot was not lost in transit. Since ballots are not released to the public, voters must trust that the outcome declared by election officials represents the accurate tallying of all ballots. As seen in the 2000 Presidential Election example, good verifiability is not a strength in the systems used in the majority of the United States.

For electronic voting systems, however, we may achieve verifiability while preserving privacy by using cryptography where appropriate. For example, a vote may be encrypted before being cast; the encrypted votes may then be added homomorphically to yield an encryption of the final election outcome, enabling the election officials to compute the result of an election without revealing any individual votes. Threshold encryption may strengthen this. For example, by requiring that 3 out of 5 election officials be present to obtain the decryption key, we reduce the chance of a malicious election official revealing individual votes.

By introducing a secure, publicly viewable bulletin board, we may achieve verifiability at every stage from when the vote is cast until the election result is revealed.

Definition 1.3.1. An *end-to-end verifiable* voting system is one that satisfies the following properties.

- *Cast-as-intended.* It may be verified that the encryption or other representation of the vote is indeed a representation of the proper vote. Methods to ensure cast-as-intended behavior include Chaum's work [XXX] or the challenge/repsonse method by Neff [YYY].

- *Recorded-as-cast.* It may be verified by the voter that her vote is properly taken into account. For example, voters may be given identifiers, and a public bulletin board may contain a listing of voter identifiers and corresponding encrypted votes. We must

ensure that this step does not reveal the plaintext vote to an adversary.

- *Tallied-as-recorded.* It may be verified by anyone that the final election result is correct given the individual votes. For example, a list of encrypted votes would be homomorphically added, and then the election officials would provide a zero-knowledge proof that the election result corresponds to a decryption of the tallied votes without revealing the decryption key.

1.4 Conclusion

In this chapter, we presented an overview of voting systems: their history, implementations, and downfalls. We discussed the various notions of privacy and verifiability and introduced the concept of an end-to-end verifiable voting system. In the remainder of this thesis, we survey the relevant technologies used in end-to-end verifiable voting systems in Chapter 2, introduce the split-value system and its implementation in Chapter 3, present performance optimizations to the system in Chapter 4, discuss opportunities for future work in Chapter 5, and conclude in Chapter 6.

Chapter 2

Electronic Voting Systems and Technologies

We now turn to an overview of existing electronic voting systems and technologies, focusing on systems that provide end-to-end verifiability. First, we discuss common design patterns found in end-to-end verifiable voting systems. Then, we present an overview of the cryptographic techniques used to achieve verifiability while maintaining voter privacy. We conclude with studies of voting systems that have been implemented in practice.

2.1 Design of End-to-End Voting Systems

Recall from [REF PREVIOUS CHAPTER] that an *end-to-end verifiable* voting system is defined by three characteristics: cast-as-intended, recorded-as-cast, and tallied-as-recorded. The first two conditions must be verified by the voter herself; our desire for incoercibility requires that no third-party may verify these properties on behalf of the voter. The last condition may be verified by anyone in the public, including someone who did not participate in the election. By verifying all three conditions, any voter may verify that her vote was counted properly and that the declared election outcome is honest.

A typical election consists of two phases.

1. *The voting phase.* In the voting phase, a voter indicates her choices for a given election by casting a ballot. The voter's ballot is typically encrypted; we must provide a proof to the voter that the encryption of the ballot faithfully represents her choices. For the

purposes of this discussion, we will assume that the electronic ballot is accompanied by a paper trail and the voter is given a receipt that she may take out of the polling site. We will also assume that privacy is guaranteed inside the voting booth; even with the cooperation of the voter, no one else may observe the voter's actions inside the voting booth.

2. *The tallying phase.* In the tallying phase, the election officials tally the set of encrypted ballots to produce an election outcome. We require that they do so without compromising the privacy of any individual voter; otherwise, a corrupt election official can act as a coercer. We also require that the election officials provide a proof that the announced election outcome is honest.

In this section, we examine designs that are used to ensure each of these conditions of an end-to-end verifiable voting system is met.

2.1.1 A Secure Bulletin Board

Central to the design of end-to-end verifiable voting systems is a publicly viewable secure bulletin board. There may be additional restrictions imposed on the actions that may be taken on it; for example, the bulletin board may be append-only.

In general, encrypted votes may be posted to the secure bulletin board along with identifiable information about a voter as long as the ciphertext provides no information about the underlying plaintext to anyone, including the voter herself. This means, for example, that the voter cannot know the randomization values used in the encryption; otherwise, a voter could prove her vote to a coercer.

The secure bulletin board also contains a proof of the election outcome, thereby allowing anyone to verify it. This can be accomplished by listing the plaintext votes after removing the associations between the plaintexts and voter IDs or by posting a zero-knowledge proof of correctness for the decryption of the tallied votes.

2.1.2 Cast-as-Intended Behavior

To ensure cast-as-intended behavior, we use the *"cast or challenge"* protocol as described in [CITE NEFF, BENALOH]. After a voter has made her selections, the system encrypts them using some randomness and prints a receipt containing a cryptographic hash of the

ciphertext. The system may additionally print a paper copy of the ballot containing more information which the voter must deposit into the ballot box before leaving.

The voter then has the option to either *cast* the ballot or *challenge* the system.

- The voter chooses to cast her ballot. She deposits the printout into the ballot box.

- The voter chooses to challenge the system. In this case, the system reveals the ciphertext and the randomization values used to produce it. The voter can verify that encryption of her vote with the given randomization produces the desired ciphertext and that the commitment to the ciphertext on the printout is correct. Should the voter choose this option, the ballot is spoiled. She must ask the system for a new randomized encryption of her selection, with which she has the option to cast or challenge again.

In this scheme, the system must output a commitment to some ciphertext before it knows whether the voter will choose to cast or challenge. A skeptical voter may choose to challenge multiple times, thereby increasing the probability of catching a malicious system. Because of our privacy assumptions about the polling site, no one else may observe the voter while she issues a challenge to the system. However, a ballot that the voter challenges must be spoiled because the challenge process reveals how the ciphertext is constructed; a voter could use this information to prove how she voted after leaving the polling site if the encrypted votes are publicly posted.

2.1.3 Recorded-as-Cast Behavior

To ensure recorded-as-cast behavior, we provide the voter with a way to check that her vote is accurately recorded on the secure bulletin board. Using her paper receipt, the voter can find her voter ID on the secure bulletin board and verify that the hash value of the posted ciphertext corresponds to the hash value on her receipt. Combined with the cast or challenge protocol of the previous section, the voter can verify that her selection has been correctly recorded on the secure bulletin board. However, she cannot prove to anyone how she voted with just the ciphertext and corresponding hash value.

2.1.4 Tallied-as-Recorded Behavior

To ensure tallied-as-recorded behavior, the system provides a proof that the encrypted votes recorded on the secure bulletin board correspond to the election outcome. In general, there are two ways to accomplish this: either the ciphertexts are combined to produce the election tally using homormorphic encryption, or the ciphertexts are shuffled and then decrypted so they cannot be traced back to the voters that produced them. Homomorphic encryption allows us to tally the votes without having to decrypt any individual vote, thereby maintaining voter privacy. Mixnets shuffle the input list of votes so that the output list cannot be traced back to the inputs; the output list may then be decrypted and the plaintexts posted on the secure bulletin board so that anyone may verify the election tally. Sections [REF NEXT 2 SECTIONS] describe each scheme in greater detail.

2.2 Homomorphic Encryption Schemes

In a *homomorphic encryption* scheme, a set of operations may be performed on ciphertexts to produce the encrypted result of a (possibly different) set of operations on the underlying plaintexts without the need for decryption. The usefulness of homomorphic encryption is immediately apparent in a cryptographic voting system: ciphertexts corresponding to the selections of individual voters may be tallied to yield an encryption of the election outcome without requiring that the individual votes be decrypted.

2.2.1 Exponential El Gamal

The exponential El Gamal public-key encryption is one example of a cryptosystem that supports homomorphic encryption.

Definition 2.2.1. (Exponential El Gamal)

- **Key Generation.** Given a group G with order q and generator g, which are public parameters, choose the secret key $k \xleftarrow{R} \{1, \ldots, q-1\}$ and the public key $y = g^k$.

- **Encryption.** For each plaintext message m, select $r \xleftarrow{R} \{1, \ldots, q-1\}$ and construct the ciphertext $(c_1, c_2) = (g^r, g^m y^r)$.

- **Decryption.** Given a ciphertext (c_1, c_2), calculate

$$\log_g \left((c_1^k)^{-1} c_2 \right) = \log_g \left((g^{kr})^{-1} g^m g^{kr} \right) = m$$

- **Homomorphic Addition of Ciphertexts.** Given two ciphertexts $C_1 = (g^{r_1}, m_1 \cdot y^{r_1})$ and $C_2 = (g^{r_2}, m_2 \cdot y^{r_2})$, componentwise multiplication yields an encryption of the sum of their underlying plaintexts with randomness $r_1 + r_2$.

$$\begin{aligned} C_1 \times C_2 &= (g^{r_1}, g^{m_1} y^{r_1}) \times (g^{r_2}, g^{m_2} y^{r_2}) \\ &= (g^{r_1 + r_2}, g^{m_1 + m_2} y^{r_1 + r_2}) \end{aligned}$$

Tallying and Proof of Decryption

After the votes have been tallied via homomorphic addition, the resulting ciphertext is decrypted to yield the sum of all the votes. This decrypted sum is then posted to the secure bulletin board and should be consistent with the declared election outcome. Finally, the system needs to prove that the posted decryption is a proper decryption of the ciphertext without revealing the secret key; otherwise, a voter could use the revealed secret key to prove to a coercer how she voted.

In exponential El Gamal, given public parameter g and public key y, we want to show that plaintext m is the proper decryption of ciphertext (c_1, c_2). Recall that a proper encryption of the plaintext m results in ciphertext $(g^r, g^m y^r)$. We divide the second component of the ciphertext by g^m and then construct a proof showing that the result is of the form $(g^r, y^r) = (g^r, (g^x)^r)$. A proof of equality for two discrete logarithms, such as those described in [CHAUM/PEDERSEN] can then be used.

2.2.2 Threshold Encryption

When using a system that employs homomorphic tallying, it is advisable to also use *threshold encryption*, in which a minimum threshold k out of a possible n trustees must agree before decryption can occur. This mitigates the risk that some subset of election officials is corrupt and prevents them from decrypting individual votes. Threshold encryption with El Gamal, such as that described in [CITE DESMEDT/FRANKEL] may be used.

2.2.3 Existing Systems Using Homomorphic Encryption

2.3 Mixnets

Another way of verifying that the election result accurately reflects the set of encrypted votes is to use a *mixnet*. First described by Chaum in [CITE CHAUM] as a method to ensure anonymous communication, mixnets secretly permute the list of inputs so that they cannot be associated with the list of outputs. A mixnet may involve many *mixservers* that are each a part of the overall chain; each mixserver secretly permutes the output of the previous mixserver, and sends its output to the next mixserver to be permuted.

In the context of elections, the input to the mixnet is the list of encrypted votes, and the output is a list of the same votes, encrypted differently and in some different order. We can decrypt the ciphertexts that are output from the mixnet and post the resulting plaintexts to the secure bulletin board along with proofs of correct decryption. This allows anyone to check that the plaintexts produce the election outcome while making it impossible to associate each any plaintext vote with a voter.

The encryption of the votes must change between the input and output; we cannot simply shuffle the encrypted values. Otherwise an adversary could easily compare the input and output lists, find identical ciphertexts, and associate them with the voter IDs which are posted alongside the input ciphertexts on the secure bulletin board.

2.3.1 Types of Mixnets

There are two general types of mixnets which accomplish our goals: *decryption mixnets* and *reencryption mixnets*.

- In a *decryption mixnet*, a message is encrypted with the public key of each mixserver in the reverse order of traversal. In a system with n mixservers, the input to the mixnet for message m is $E_{K_n}(E_{K_{n-1}}(\ldots(E_{K_1}(m))))$, where E_{K_i} denotes encryption using the public key of the ith mixserver. At each mixserver, one layer of encryption is removed and the list of messages is shuffled before being sent to the next mixserver. The mixnet outputs a list of decrypted messages whose ordering cannot be traced back to the input ordering. Decryption mixnets are commonly used to anonymize routing in designs known as "onion routing", where each message is encrypted repeatedly like the

layers of an onion, and each mixserver "peels off" a layer until the plaintext message is reached.

- In a *reencryption mixnet*, a message is encrypted once and input into the mixnet. At each mixserver, messages are reencrypted using a public-key encryption scheme that supports reencryption and then shuffled. An example of a public-key encryption scheme that supports reencryption is exponential El Gamal, as discussed in the [REF PREV SECTION], by homomorphically combining the message to be reencrypted with an encryption of the "zero message". This similarly achieves our goals: the underlying plaintexts do not change through the mixnet, but their ciphertext representations and ordering are changed in a way that makes the outputs untraceable to the inputs.

2.3.2 Verifiable Mixnets

When using a mixnet to anonymize votes, we

Chapter 3

End-to-End Verifiable Voting Using Split-Value Representations

In this section, we present the split-value system enabling end-to-end verifiable voting as described by Rabin and Rivest [CITE] as well as an overview of the implementation as given in [CITE MARCO'S THESIS]. We will focus on the main ideas and components of the design and implementation of the system, and leave the explanation of the details to the respective papers [X, Y].

3.1 The Split-Value Voting System Design

3.1.1 Definitions

Given an election, choose the *representation modulo* M so that any voter's selection, including possible write-in candidates, may be represented by an integer modulo M.

Definition 3.1.1. The *split-value representation* of x modulo M is the pair $\mathrm{SV}(x) = (u, v)$ such that $x = u + v \bmod M$.

Note that for a given x, knowing the value of one of u or v, but not both, reveals no information about x. This key property allows us to construct zero-knowledge proofs of equality for two values using their split-value representations, without revealing the actual values themselves.

Definition 3.1.2. A *commitment* to a value x with randomness r is the value $c = \textsc{Com}(x, r)$ such that c may be "opened" to reveal x and r. We desire that the Com function is both

hiding and binding. That is, given $c = \textsc{Com}(x, r)$, no information is gained about x, and it is infeasible to produce another (x', r') such that $c = \textsc{Com}(x', r')$.

The HMAC function, with the randomness r as the key, is used as the commitment function in our implementation.

Definition 3.1.3. Given $\text{SV}(x) = (u, v)$ and randomness r and s, the *split-value commitment* of x is $\textsc{ComSV}(x) = (\textsc{Com}(u, r), \textsc{Com}(v, s))$.

Using split-value commitments, we may construct probabilistic proofs that two values are equal without revealing the value itself. Suppose that a prover wishes to prove that $x = x'$. Given their split value commitments

$$\textsc{ComSV}(x) = (\textsc{Com}(u, r), \textsc{Com}(v, s))$$
$$\textsc{ComSV}(x') = (\textsc{Com}(u', r'), \textsc{Com}(v', s'))$$

the prover claims that there exists a value t such that $t = u - u'$ and $t = v' - v$. This can be true if and only if $x = x'$. With probability $1/2$, the examiner asks for the opening of the "left" commitments, in which case the prover gives the values u, r, u', and r'. The verifier checks that the values $\textsc{Com}(u, r)$ and $\textsc{Com}(u', r')$ are correct and that $t = u - u'$. With probability $1/2$, the examiner asks for the "right" commitments, in which case the prover sends the other components of the split-value commitment. Thus, if $x \neq x'$, a dishonest prover can win with probability at most $1/2$.

3.1.2 The Verifiable Mixnet

The main component of the split-value system is a verifiable mixnet composed of a 3×3 grid of mixservers. These servers are responsible for receiving votes, obfuscating and permuting them, and producing an output list that cannot be traced back to the input list. Denote the mixserver in row i and column j as $P_{i,j}$.

When the voter's vote w is cast on her device (e.g. a tablet), it is broken into three components x, y, and z such that $w = x + y + z \bmod M$. The device then creates split-value representations of each component. For each component, the split-value representation and its commitment are sent over a secure channel to the first column mixserver of the row corresponding to the component. For example, x and $\textsc{ComSV}(x)$ are sent to $P_{1,1}$, y and

ComSV(y) are sent to $P_{2,1}$, and so on. Note that because each mixserver only receives one out of three components of the vote, no single server may reconstruct any individual vote (and break voter privacy) without leaked information from at least two other mixservers.

Definition 3.1.4. A triplet $S' = (x', y', z')$ is an *obfuscation* of $S = (x, y, z)$ if $x' + y' + z' = x + y + z \bmod M$.

The first column of servers agree upon a triplet of values (p, q, r) such that $p + q + r = 0 \bmod M$ for each vote. The first row then computes $x' = x + p \bmod M$, the second row computes $y' = y + q \bmod M$, and so on. The column of servers then agree upon a random permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$. Each server transmits the permuted obfuscated values to the next column in the same row: $P_{1,1}$ transmits $x'_{\pi(1)}, \ldots, x'_{\pi(n)}$ to $P_{1,2}$, $P_{2,1}$ transmits the permuted y' values to $P_{2,2}$, and so on.

The second and third columns of mixservers repeat this process, and the results of the final obfuscation and permutation are posted to the secure bulletin board. Given a security parameter m, this process of obfuscation and permutation across each column of the mixnet is performed $2m$ times. In our current implementation, $2m = 24$.

To verify the election result, we divide the $2m$ repetitions into two groups of m. The first m repetitions are used to show that the mixnet does not alter the votes. To do so, we reveal the permutations used at each column. For each split-value commitment on the input side of the mixnet, we follow the revealed permutations to arrive at its corresponding split-value commitment on the output side and construct a proof of equality as described in the previous section. These permutations and proofs of equality are posted to the secure bulletin board for anyone to examine.

The other m repetitions are used to prove that the election outcome is correct. All of the commitments to the votes are opened, revealing the votes themselves. These votes are posted to the secure bulletin board along with the election outcome for anyone to examine.

3.2 Implementation Details

The original design of the system included a functioning preliminary prototype written using Python 3.4. In this prototype, mixservers were simulated using Python objects. A simulation of a complete election, including vote generation, mixing, proof construction, and verification, was contained within a single Python process. Python was chosen for its

ease of programming, debugging, profiling, and installation. Care was taken to ensure that included libraries was kept to a minimum, so that the code was easy to run and results easily replicated.

3.2.1 Server Processes and Communication

Subsequent work [CITE] significantly decoupled the system. The mixservers, secure bulletin board, and vote generation each became its own process. An interface was built on top of Python TCP socket servers to facilitate communication, and Python socket handlers were used to handle incoming requests. All data was serialized with standard JSON libraries. The implementation also introduced the notion of a *controller server*, whose responsibilities include the following.

- Serve as the first point of contact for all other servers. All other servers start knowing the IP address and port of the controller.
- Keep track of and broadcast network information, including all other servers, their addresses, and roles, to the network.
- Assign roles to each other server, and assign their positions in the mixnet.
- Synchronize the various phases of the election simulation, including vote production, mixing, proof construction, tallying, and verification.

Because all communication occurs via standard TCP sockets, the implementation may be used on any group of computers running on a local area network. The implementation has been tested on single-core and multicore machines as well as multiple virtual machines running on a virtualized network.

3.2.2 The Implementation of Randomness

One particular detail that deserves special attention is the implementation of randomness. Randomness is used throughout the system: in constructing split-value representations, in constructing commitments, in choosing which component of a split-value representation to reveal during a verification, and so on. Fast, cryptographically secure randomness is essential to the correct functioning of the system.

The utility function library written for this implementation includes a function that generates pseudorandom bytes. Each server maintains a list of randomness sources. When a randomness source is created, its name is used as the initial seed. When a sequence of pseudorandom bytes is desired, the seed is hashed twice using the SHA-256 hash function. The first hash determines the updated seed, which is stored in the place of the original seed. The second hash is of the original seed with a "tweak" string appended. With a good hash function, these two hashes will not appear to have originated from the same seed.

During the production of the proofs, we require a good source of randomness to choose the m repetitions that will be used to verify the mixnet and the other m repetitions that will be used to verify the election result. There are two options for this.

- An *external source of randomness*, such as a dice-roll which occurs at a public ceremony after voting has ended. The benefit to this method is that we obtain "true" randomness that no adversary can predict beforehand. However, it is possible that election officials and the public are uncomfortable with introducing what appears to be games of chance into the election process.

- A *Fiat-Shamir style hash* of the secure bulletin board. With overwhelming probability, a Fiat-Shamir style hash of the secure bulletin board will provide randomness that is good enough to overcome an adversary. There is the small but nonzero chance that an adversary could manipulate the election result by providing both a faulty proof that supports the manipulated outcome and an alternate version of the secure bulletin board that, when hashed, produces randomness that verifies her faulty proof. To compensate for this, we set the number of repetitions (m) to be higher so that the chance of producing an alternate secure bulletin board that passes all of the verifications decreases exponentially.

In the current implementation, we use the Fiat-Shamir method. However, this choice incurs a penalty in performance, as our setting for m could be reduced dramatically by using an external source of randomness, resulting in fewer iterations through the mixnet and a shorter proof. In the next chapter, we will see that the use of an external source of randomness may provide enough of a performance boost to meet certain goals that election officials desire.

3.3 Conclusion

In this section, we introduced the concepts and design behind the split-value verifiable voting system, including the idea of a split-value commitment. We discussed the ways that votes are obfuscated and permuted through the mixnet. Proofs are constructed to verify both the mixnet and the election outcome without revealing associating voters with their votes. Finally, we examined details of the prototype implementation, including the communication interface between servers and the implementation of randomness used in our simulations.

Chapter 4

Performance Optimization of the Split-Value System

After verifying the correctness of the implementation, we turn to performance. The performance of an electronic voting system is an important consideration when selecting among various options. The system must scale to be able to handle hundreds of thousands or even millions of voters per district and perhaps multiple simultaneous races. Furthermore, the media and the public demand fast, accurate reporting of election results. We would like to be able to provide the election results along with a verified proof within hours of the closing of polling sites.

One proposed advantage of the split-value system is its efficiency. Existing comparable systems such as [STAR-VOTE] use relatively time-consuming public key encryption operations such as modular exponentiation, while the split-value system uses much simpler and faster modular addition operations. Compared to other systems, the split-value system pays a higher cost due to the need for $2m$ repetitions. Furthermore, the cost of inter-process and inter-machine communication and data serialization become non-negligible.

The implementation presented in [CITE MARCO], referred to as the *reference implementation* here, contained numerous opportunities for optimization. In this section, we begin with a survey of the performance of cryptographic primitives and data serializers used heavily in the implementation. Then we present the main optimizations that were successfully applied to the reference implementation and their impact and conclude with an analysis of the scalability of the system.

All timings reported in this section, unless otherwise noted, are performed on a 16 core AWS machine using two Intel E5-2666 v3 processors operating at 2.9 GHz and 32 GB RAM. The default settings for simulating an election consist of a single race with two candidates and a third write-in candidate up to sixteen characters, 1000 voters, and $2m = 24$.

4.1 Performance of Cryptographic Primitives

4.1.1 Hashing and Encryption Functions

[TODO: TABLE] shows the performance of various cryptographic primitives, including hash functions (MD5, SHA-256), a block cipher (AES in CBC mode), and a stream cipher (XSalsa20), by running each function ten million times. The built-in Python hash functions and XSalsa20 perform efficiently as expected, and we choose SHA-256 as the hash function used in the implementation as it is both fast and secure.

It is surprising that the Python NaCl and Cryptography third-party libraries, both of which are very popular, significantly underperform compared to the built-in functions. The inefficiency of AES is particularly surprising given the 2010 release of the Intel AES New Instructions Set, which gives hardware support to all standard modes of operation and key lengths. The expanded instruction set gives very good performance; encryption with 256-bit AES-CBC takes 5.65 CPU cycles per byte (256-bit AES-CBC is the most expensive operation) [TODO CITE THIS]. The CPUs used in this benchmark did indeed support the expanded instruction set, but it is likely that the third-party libraries did not use them.

Unfortunately, the original design estimated the performance of AES to be more than two magnitudes faster [CITE THIS] than its Python implementation. Even using the significantly faster HMAC with SHA-256, commitments are more than forty times slower than estimated. It is likely that C code written using the expanded hardware instructions will offer significant performance benefits. For our Python implementation, we use HMAC with SHA-256 as the commitment function.

4.1.2 Pseudorandom Number Generators

As discussed in [LINK TO CH 3], randomness is used throughout the system. The method used in the implementation gives a good pseudorandom generator with three calls to the SHA-256 hash function (once to generate the new seed, and twice to compute the tweaked

hash which becomes the output). However, as seen in [TODO: TABLE], there is a non-negligible overhead likely due to the repeated calls to the `digest` function and the byte-encoding functions used to incorporate the tweaking string.

When benchmarking the two functions side by side, it appears that `os.urandom(n)`, which provides "a string of `n` random bytes suitable for cryptographic use" [CITE PYTHON] is a faster option. However, in the context of the overall system, the improvement is minimal. Furthermore, `os.urandom` relies on the operating system's implementation of the `/dev/urandom` entropy pool, which is not always guaranteed to have enough entropy to produce good randomness. For these reasons, we stick to the method of repeatedly hashing the seed string to generate randomness.

4.2 Performance of JSON Serializers

The JSON format is used to serialize all data sent over the wire and posted to the secure bulletin board; it was chosen because it is compact, very commonly used, and well-supported by all platforms. The library that is used has a signficant impact on the performance of the system.

[TODO: TABLE] shows the performance of three common JSON serialization libraries for Python, tested on a moderately large (164.6 MB) JSON file found in [CITE GITHUB]. The `simplejson` and `ujson` libraries have C extensions that can be compiled for performance boosts; they were included in our tests. Our tests show that the `ujson` library significantly outperforms the other two libraries; the C extensions for `simplejson` do not provide any noticeable performance benefit.

Unfortunately we could not use `ujson` because it does not support integers longer than 64 bits. We started our implementation with Python's built-in JSON serializer; however, we discovered that the structure of the data in our particular system heavily favored `simplejson`. This was not reflected in the data used in the benchmarks in [TODO: TABLE]. Switching libraries produced significant speedups.

4.3 Optimization of the Prototype

To effectively optimize the reference implementation, we begin by collecting timing information on each phase of the simulation. We delve further into the performance of each

function call by using the Python profiler, which tells us how much time is spent in each function, including and excluding the time spent in functions called from it. From there, we generally prioritize optimizing the most time-consuming functions as well examining the most frequently called functions to determine if any unnecessary work is being done.

We focus on the timings of the mix, proof, and verification phases as they are the most time-consuming. In a real election, the vote generation would occur throughout the day rather than all at once, so it is not a bottleneck in the system. The setup occurs before the election, and the tallying is very fast compared to the mix, proof, and verification phases.

[INSERT TABLE] shows the major optimizations that were performed on the reference implementations and the resulting runtimes. The performance gains for each optimization are reported relative to the row immediately preceding it in the table.

In summary, the most effective optimizations involved leveraging the available parallelism in the multiple mixservers and ensuring that commonly used subroutines were as fast as possible by optimizing by hand, using a built-in Python function, or selecting a third-party library optimized for speed. Overall, we observe a tenfold speedup due to our optimizations detailed in this section.

4.3.1 The Python Profiler

TODO: show sample profiler output

4.3.2 Message Passing Over Sockets

As explained in the previous chapter, communication between machines happens via TCP servers communicating over Python's standard socket implementation, which provides an interface for sending and receiving data.

- `socket.recv(bufsize)`. Receive data from the socket up to `bufsize` bytes, and return the data received.
- `socket.send(string)`. Send data through the socket, and returns the number of bytes sent. `socket.send` does not guarantee that the all data is sent; the caller must check the return value and retry sending the remaining data if necessary.
- `socket.sendall(string)`. Send data through the socket, and return `None` on success. Unlike `socket.send`, `socket.sendall` guarantees that all data is sent if the function

returns successfully.

Due to constraints in the operating system, the value for `bufsize` used in the `socket.recv` function is usually 4096 or 8192, corresponding to the maximum length that may be sent via a single call to `socket.send`. Note that there is not a corresponding `socket.recvall` function. If a longer message is desired, we must repeatedly call `socket.recv` until the entire message is received. The reference implementation handled this situation by sleeping for a short amount of time after each call to `socket.recv` before attempting to receive more data from the socket.

Because `socket.recv` blocks if the data is not ready yet (up to some timeout), we do not need to sleep to wait for more data; rather, we just need to know how much data to expect and continuously call `socket.recv` until that length is reached. We define the following higher-level interface for communication over sockets:

- `send(socket, message)`. Prepend `message` with its length and send the resulting bytes over `socket`.
- `recv(socket)`. Receive and return all data from the socket by stripping out the length from the first chunk of the message and repeatedly calling `socket.recv` until the length is met or an error is encountered. We assume that the length of a message will never exceed the limit of a 32-bit signed integer.

This optimization yielded a speedup of 20.9 seconds (68.7%) in the proof phase and 43.5 seconds (43.9%) in the verify phase.

4.3.3 Byte/Integer Utility Functions

Our profiler output revealed that a significant amount of time was spent on the reference implementation's utility functions for converting between Python bytes, integers, and hex strings. The conversions are necessary because different components of the system expect values to be in different forms; for example, the library used for cryptographic hash functions returns a byte array, while our secure bulletin board, to which the results of computing these cryptographic hashes are posted, accepts string inputs.

Specifically, for the `sv.bytes2hex` implementation, which returns the hex string representation of a byte array, we achieved a 15× speedup in the function's runtime by simply replacing the byte-by-byte conversion with the built-in `binascii.hexlify` function.

For the `sv.int2bytes` function, which converts any integer into a bytearray representation, we use Python 3's new `bit_length` attribute, which supports signed integers up to 64 bits long, along with a 64-bit mask and shift. This allows us to process the input eight bytes at a time instead of one, yielding a 1.5× speedup in the function's runtime compared to the reference implementation.

Overall, these optimizations give a 6.1% speedup, mostly in the proof and verify phases.

4.3.4 Introducing Parallelism

No performance analysis is complete without an analysis of the potential for parallelism. Multicore CPUs are now the norm; even a typical laptop that we would expect an election official to own and use in running the system is likely to have at least two CPU cores. Furthermore, in our system, the computation required to mix the votes and construct a proof of the election outcome is distributed across the nine mixsers. We show here that significant parallelism can be leveraged in each of the mix, proof, and verification phases of the system.

Parallelism in the Mix and Proof Phases

In the reference implementation of the mix and proof phases, the controller server issued remote procedure calls (RPCs) to each mixserver sequentially. Each RPC would block on the controller until the coresponding mixserver's RPC response arrived at the controller; only then would the next RPC be issued. Because the RPCs are independent within each part of the proof and verify phases, they may be issued asynchronously and in parallel.

We use Python's implementation of processes in the `multiprocessing` library to accomplish this parallelization. One process is forked off the process running the controller server for each RPC. The process is responsible for issuing the RPC, waiting for the response, and returning to the parent process. Even with a single-core controller server, a process that is waiting for an RPC response may yield the CPU to allow another process to issue an RPC.

Processes are used instead of threads because Python's Global Interpreter Lock limits parallelism between the various threads spawned from a single process. We confirmed that using Python threads instead of processes yielded no noticeable speedup. The built-in `multiprocessing` library does not support process pools as it does thread pools, so we must fork new processes for each RPC instead of relying on existing workers. However,

forking a new process is a constant-time cost relative to scaling the number of voters and our tests show that it is negligible.

Introducing parallelism produced a 12.1 second (194.8%) speedup in the mix phase and 21.5 second (36.2%) speedup in the proof phase.

Parallelism in the Verification Phase

Unlike the mix and proof phases which involve many distinct machines, the verification phase occurs on a single machine. However, all parts of the verification process only depend on the contents of the secure bulletin board; after all, we expect that any member of the public may take the contents of the secure bulletin board and verify the election outcome herself. Although no part of the verification phase depends on any previous part because each part may independently read the secure bulletin board, in practice it is easier to let earlier phases populate a shared data structure for later phases to use. This prevents the need to deserialize the secure bulletin board, which may be a large file and thus a costly process, at the cost of some parallelism.

In the reference implementation, later parts of the verification phase used data that earlier parts had already verified. This makes verification simpler because at no point does the verifier need to worry about inconsistent or unverified data propagating through it. However, this scheme resulted in a complex data dependency graph which made the verifier difficult to parallelize; a process running one part of the verification could be started only when all of its dependencies had completed.

The requirement that data consumed in later parts must itself be verified is stricter than necessary. If any step of the verifier fails, we declare the election to be invalid, so even if bad data propagates through the verifier, at some point that data will itself be checked and an error produced. This check might not happen until after the bad data has been used in other parts of the verifier, but the problem will be caught nonetheless.

Relaxing our requirements on the verifier and introducing parallelism resulted in a 8 second (8%) speedup in the verification phase.

4.3.5 JSON Serialization

While the benchmarks in [LINK TO SECTION ABOVE] showed no benefit in switching to the `simplejson` library, in our system the performance speedup was enormous. By

simply switching the serializer library and eliminating indentation and formatting from JSON strings, a 41.5 second (233%) speedup in the proof phase and 2.9 second (40.5%). The overall runtime speedup for the simulation was 45 seconds (1.26%).

4.3.6 Hashing the Secure Bulletin Board

Hashes of the secure bulletin board are used in generating the cut-and-choose and left-right challenges during the construction of the proof. They are also used in the verification to ensure that the hash value used in the proof are actually proper hashes of the secure bulletin board.

Because the verification process itself makes modifications to the secure bulletin board data structure, causing the hash values to differ, the reference implementation makes a deep copy of the data structure before proceeding with verification. By moving the hashing verification to be before the rest of the data structure is verified, a 1.3 second (22%) speedup was obtained.

In the reference implementation, the mixservers ask the server storing the secure bulletin board for its hash value for the aforementioned reasons. Although the mixservers operate in parallel, the secure bulletin board runs on a single process and therefore serializes the requests for the hash value. Therefore, a mixserver that issues its request later must wait for the secure bulletin board to service all earlier requests, making no progress in the process. In our optimization, the controller server requests the hash value and sends it to each mixserver as part of the body of the relevant requests. Each mixserver then has all the information necessary to construct the entire proof section at the beginning of each phase and wastes no time waiting for the single-threaded secure bulletin board to service other requests. This optimization resulted in a 11.7 second (208%) speedup.

Each request for the hash value of the secure bulletin board requires that the contents of the data structures held in memory be serialized before it can be hashed. As the secure bulletin board grows, this serialization becomes expensive. We can reduce this cost by computing the updated hash value every time anything is appended to the bulletin board. We calculate the new hash value by concatenating the data to be appended with the previous hash value. Because the previous hash value is always of fixed size, the cost to compute the hash value does not grow as the size of the data structures grows. While this results in a relatively small (13.5%) performance gain in the proof phase, it prevents performance from

diminishing as the size of the secure bulletin board increases.

4.4 Performance Scalability

In the previous section, we simulated elections with 1000 voters because it provided useful data for optimization without taking too long to run successive simulations. A real election, however, may have hundreds of thousands or even millions of voters. We are interested in how our system scales with the number of voters.

[TODO:TABLE] shows the performance of the system with 100, 500, 1000, 5000, and 10,000 voters and the corresponding costs per voter. We see that the performance of the system scales roughly linearly as the number of voters, which bodes well for scaling the system to production. In the current prototype and the testing machine, connections begin to time out for 100,000 voters. This is likely an issue with the pool of TCP connections becoming saturated since the simulation is running on a single operating system. However, this remains to be tested rigorously.

[TODO:TABLE] shows the performance of the system with the number of repetitions ($2m$) set to 4, 8, 16, and 24. Although the runtime grows very slightly sublinearly as the number of repetitions, reducing the number of repetitions may reduce the runtime significantly.

4.5 Conclusion

In this section, we examined the performance of the split-value system implementation. Benchmarking functions available for cryptographic primitives led to the usage of the SHA-256 hash function and the HMAC commitment scheme in our implementation. We used the Python profiler to identify areas of the implementation that could yield large performance gains. By leveraging parallelism, reducing the wait time on sockets and requests to the secure bulletin board, and a more efficient JSON serializer, our optimized system runs in less than one-tenth the time of the reference implementation.

With the optimized implementation, an election of one million voters will take between seven and eight hours to mix, prove, tally, and verify. This is a bit more than twice as long as the time generally allotted between the closing of polling sites and the release of election

results in the evening news. We hope that further optimization may yield this factor of two to three improvement in the runtime.

Using properly hardware-optimized AES encryption as a commitment scheme has the potential to reduce runtime significantly; according to Intel's measurements, the estimate of 8 million commitments [CITE PAPER] per second can be achieved even with conservative estimates.

In the current implementation, considerable time is spent on repeating the mixing and proof construction for $2m = 24$ times. As mentioned in [LINK TO PREV CHAPTER], this repetition is necessary due to the reduced security in using a Fiat-Shamir hash of the secure bulletin board instead of true randomness. The process may be sped up at least twofold if election officials adopted a true source of randomness during the construction of the proof.

Overall, it appears that under ideal situations (access to hardware functions, minimal cost in serialization and communication), the split-value system delivers on its promise to provide security without the heavy computation of most existing cryptographic solutions. However, its implementation requires careful optimization and selection of third-party code.