The Implementation of a Split-Value Verifiable Voting System

by Marco Antonio L. Pedroso

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

January, 2015

© Massachusetts Institute of Technology. All rights reserved.

Author: ____________________________________________________________
Department of Electrical Engineering and Computer Science
January 30th, 2015

Certifier by: _________________________________________________________
Prof. Ronald L. Rivest Thesis Supervisor
January 30th, 2015

Accepted by: _________________________________________________________
Prof. Albert R. Meyer, Chairman, Masters of Engineering Thesis Committee
The Implementation of a Split-Value Verifiable Voting System

by Marco Antonio L. Pedroso

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
January, 2015

Abstract

This study provides a proof-of-concept of a newly designed verifiable voting system. The implementation elicits details in the communication and synchronization among servers, unavailable in the original design paper [M. O. Rabin and R. L. Rivest, "Efficient End to End Verifiable Electronic Voting Employing Split Value Representations." in Proc. EVOTE '14, Bregenz, Austria.]. The implemented system was tested on networks of virtual machines in a cluster, and its performance was evaluated based on the computational time and amount of data transferred. The preliminary results simulates elections with up to ten thousand votes. The team will conduct further work to implement the handling of server failures and the secure channels among the servers. This study demonstrates the feasibility of running large elections with more transparent voting systems, by leveraging the split-value representation and simple cryptographic primitives.
Acknowledgments

Firstly, I would like to thank my family for their continuous support throughout my life.

Secondly, I would like to thank my thesis supervisor, Prof. Ronald Rivest, for his expertise and availability. He introduced me to the interesting field of verifiable voting technology and provided me numerous references to learn about the state of the art in the field. He was always open to discuss different design ideas and allowed me to own the project. He guided me to follow an iterative approach, which helped me build a complex system with multiple agents. I am also very grateful for the high level and detailed feedback he provided me on the writing for this thesis. Most importantly, I am thankful for his guidance in defining the scope of this work in such a way that it was feasible, but at the same time challenging.

Finally, I would like to thank MIT and the Computer Science and Artificial Intelligence Laboratory for providing a supportive environment, bringing together the human and physical resources necessary to make all of this possible.
1. Introduction

In the first elections in ancient Greece, where only several hundred people voted in a public space, simple methods such as raising hands or calling out candidates’ names functioned well. However, as the scale and significance of elections grew, many problems emerged. Privacy became a major concern, when voters wanted to express their preferences without fear of retaliation. The size of elections grew exponentially, with 814 million eligible voters in the 2014 elections in India alone [1]. Such change in scale requires the implementation of efficient and cost-effective operations. In addition, given that 15% of the global population is disabled, accessibility has become an issue [2].

Elections have evolved considerably and election officials have addressed some of the pertinent issues. However, as election systems have become more complex they have also become less transparent. The infamous 2000 presidential race in the United States between George W. Bush and Al Gore required multiple recounts, exhibiting the unreliability of the existing voting systems [3]. More recently, in 2014 after the presidential elections in Brazil, motivated by popular support the losing party demanded a special election audit to verify the logs of the voting machines [4]. In the current situation, where even established enterprises like Google and Sony are victims of cyber-attacks, it is unlikely that the budget constrained election systems are not also vulnerable [5]. With many large elections occurring electronically, malicious software may affect the election results much more drastically than old-fashioned ballot stuffing.

Providing verifiability of the election seems to go in the opposite direction of guaranteeing voters’ privacy. Achieving verifiability implies convincing voters that their votes were counted as intended, while the privacy requirement aims to hide the voters’ intent. In order to ensure both, cryptography have appeared as a natural ally. Recently, zero knowledge proofs, homomorphic encryption and mix-nets have been playing an important role as building blocks in the design of voting systems. These cryptographic
tools have been used in actual elections [6, 7]. However, performance limitations and complexity of the designs have been a barrier for wider adoption of such systems.

The purpose of the project is to implement a verifiable electronic voting system designed by Rabin and Rivest [8]. The design of the Split-Value Voting System provides proofs for a voter to verify their cast ballots, while respecting voters’ privacy. The Split-Value Voting System addresses practical limitations such as being efficient, conceptually simple and flexible to accommodate different election rules. For this thesis project, we provide a prototype as a proof-of-concept for such design.

The ability to trust election outcomes is a fundamental part of democracy. The hope is that with this thesis we can shed light in how to make elections verifiable. Work in this area can bring substantial benefits especially to countries that suffer from corruption and use obsolete voting systems.

The thesis is structured as follows. In Section 2, we provide an overview of the design of voting systems including the evolution of voting technology, threats to elections, and the desired goals. In Section 3, we summarize the related work on End-To-End verifiable voting systems including definitions. We also highlight the most notable designs and the most common building blocks used. In Section 4, we describe the core components of the Split-Value Voting System, the reference for our implementation. Section 5 covers the design contributions to the system, including the overall system architecture and details on the communication coordination for each phase of the election. In Section 6, we describe the experimental setup including the implementation details and the setup for the simulated elections. In Section 7, we analyze the preliminary performance results of the simulations and discuss their limitations. In Section 8, we discuss possible future work. Finally, we conclude in Section 9.
2. **Background**

2.1. **Evolution of Voting Technology**

In the beginnings of election history, voting publicly by a show of hands was the most common method for running elections. As election grew in importance and in scale, technologies started to play a role in the procedures. Figure 1 shows various technologies used for elections.

The demand for improved privacy led to the introduction of paper ballots in voting systems. Paper ballots allow a simple mechanism for casting and counting ballots, making them widely used across the world especially for smaller elections. Under proper vigilance, election officials and observers can maintain the ballot boxes unviolated. However, conducting a large election with paper ballots easily becomes a logistical nightmare because of the need to provide translated materials and to conduct centralized tallying.

![Evolution of voting equipment](image1)

**Figure 1:** Evolution of voting equipment over the years, from plain public voting up to internet voting.

As the number of voters increased, election officials adopted lever machines and punch cards to speed up the tallying process. By 2004, many of the American states shifted away from these technologies due to high error margin associated with mechanical failures [9].
Electronic voting machines were the successors to punch card and lever machines. Their great novelty was that they allowed people with disabilities to vote independently, making blind people strong advocates of such machines [10]. The biggest challenge in electronic systems is that their large codebase usually contain over 500,000 lines of code, making them difficult to build, test and audit. Even established open-source software contain several security vulnerabilities per thousand lines of code [11]. In fact, during the period between 2007 and 2009, the California Top-To-Bottom Review revealed major security problems in all commercial voting equipment available on the market at that time [12].

In the following generation, the paradigm of Software Independence was proposed [13]. The Software Independence paradigm requires independent procedures based on digital and physical records to compute the final tally. Most technologies in use today for the presidential election in the United States are either Optical Scanners or Electronic Machines with a paper trail [14]. Generally, election officials compute the tally based on the digital records with additional auditing procedures to ensure consistency of the result with the paper records. In case a mismatch occurs, election officials use the backup plan of hand counting paper ballots. What is missing in these systems, however, is that voters do not have any guarantee about whether or not votes are actually counted. Voters have to rely on an intricate chain of custody of ballots to trust the final tally. The End-To-End voting paradigm described in section 3 can make the process more transparent and can prevent manipulations of cast ballots.

Recently there is growing support for and use of remote voting, either via mail ballot or internet voting. In remote voting, voters have to sacrifice their privacy for convenience. Internet voting, however, has yet many other challenges ranging from malware in machines to insecure channels which make it inappropriate to run an election where the primary goals are the integrity and privacy of the ballot. Drew et al. [15] demonstrate practical attacks to the Estonian Internet Voting System and argue that Estonia discontinue the use of internet voting until there are adequate advances in computer security. New
technological solutions are necessary to prevent vote selling and other threats in a scenario where a large fraction of voters cast their ballot remotely.

2.2. Threats to Voting Security

In 2012, campaign sponsors put over two billion dollars into political campaigns for the US presidential election (Huffington Post [16]). The laws that will pass, the contracts that will be signed, and the future of a country are all heavily dependent on the outcome of an election. As a result there may be multiple adversaries interested in manipulating the outcome.

Among potential adversaries against the integrity of an election one can include voters, election officials, voting machine vendors, lobbyists and foreign nations. Voters may wish to sell their votes, which should be prevented since the goal of elections is to capture honest voter preferences. Election officials, who are often believed to be the trusted parties in the process, may be partisan and have incentives to try to modify an outcome if possible. Some third parties, for example software vendors, may put an invisible Trojan horse inside the system. In addition, foreign powers or large corporations may attempt cyber-attacks, since results of an election may affect them too.

Multiple threats have appeared in the literature. They include voting on behalf of the deceased, ballot-box stuffing, coercion, vote selling, chain voting, miscounting of votes, malicious software and viruses on voting machines. For a more comprehensive list of threats, we suggest reading [10].

2.3. Voting System Goals

Below is a list of the desired properties for voting systems, followed by the descriptions of each of them:
• **Integrity:** Votes cannot be changed or deleted after being cast, and votes cannot be added for people who did not vote or are ineligible to vote for a particular race. In general, the tally should be accurate reflecting the preferences reported by eligible voters who actually voted.

• **Privacy:** To ensure the true preferences are reported in an election, voters cannot provide evidence to third parties that they voted in a particular way, or they may be subject to coercion. The privacy property should hold even if the voter wants to break it, to discourage illegal practices such as vote selling.

• **Assurance:** The outcome provided by a voting system should be verifiable. The mere fear of corruption of a voting system can cause great damage to the support of the government and the democratic image of a country. The outcome of an election must be auditable and the losing parties and voters should be convinced that the final tally is correct.

• **Accessibility:** The system needs to ensure that people with disabilities have the same voting rights as everyone else.

• **Ease of Use:** Both voters and election officials should be able to easily use the voting system to perform their respective tasks.

• **Availability:** The voting system must be robust to failures with appropriate back up plans to ensure elections do not get interrupted.

• **Cost:** The cost of running an election needs to remain close to the minimum, while meeting the other goals.

• **Time:** The time to compute the final tally and to audit an election must respect the deadlines imposed by the voting jurisdiction.

The focus of this project is to expand the design space by providing a new verifiable voting system satisfying the goals above. Our design is compatible with existing accessible voting interface. The voting machine can be off-the-shelf technology making it a cheaper alternative to the existing voting systems in
the market. Furthermore, in comparison to other systems with similar verifiable properties, our solution allows a faster tallying of the votes.

The innovation focuses on the integrity, privacy and assurance requirements where cryptography provides new solutions to ensure privacy while proving correctness.

3. Related Work

3.1. End-To-End (E2E) Definition

The goal of End-To-End verifiable voting systems is to provide high trust in the outcome of an election by detecting errors and frauds at multiple steps of a voting system. In other words, it aims to guarantee assurance of the outcome while preserving privacy of voters’ choices.

Voters do not need to understand all election procedures or how the voting machines work in order to trust the election outcome. In End-To-End voting systems, voters are able to check the correctness of the election based on their input (the vote they cast) and the output (the final tally and additional election data). At first, it ensures the encrypted vote matches the voters’ intention. Secondly, using a receipt generated in the casting phase, a voter can check the public bulletin board to ensure his vote is considered in the final tally. Lastly, the information on the public bulletin board about all the encrypted votes and election metadata provided by election officials make it possible to trust that the final tally is computed correctly. Figure 2 depicts the End-To-End properties.
A key component of such systems is a secure public append-only bulletin board. The most common implementation of a bulletin board is to have each voter’s name and an associated encrypted vote posted on it. For instance, the STAR-Vote design [17] includes posting encrypted votes using the ElGamal public key encryption [18] on a public bulletin board. Besides the encrypted votes, the final election outcome and any election metadata that provide further evidence on the outcome being correct, can be posted in the bulletin board for verification.

A challenging aspect of this design is ensuring that a voter is able to check that her ballot has been properly posted in the bulletin board. At the same time, she should not be able to convince anyone else how she voted from that encrypted representation. To achieve this as intended, we need subtle protocols to prove to a voter that the ciphertext for her ballot actually decrypts her plaintext vote.
3.2. Alternative End-To-End Voting Systems

E2E systems have been used in actual elections. Scantegrity was used in Maryland, United States in 2009 [6], the Wombat system was used in 2012 in Israel [7]. There is also a new design by the STAR-Vote team to be used in Travis County, Texas [17].

The work by Jonker et al. [19] synthesizes the common building blocks used in the construction of E2E voting systems. According to the authors, to achieve privacy the most common designs use a subset of the following tools: homomorphic encryption, re-encryption, blind signatures, zero knowledge proofs, designated verifier proofs, secret sharing, mixnets, communication channels with privacy properties, and tamper-resistant hardware. To complement the list above and ensure verifiability designers often use the following building blocks: secure bulletin boards, ballot auditing, plaintext equivalence tests, and commitment schemes. Further work in End-To-End voting system including surveys, alternative descriptions and comparisons of the proposed designs can be found [20, 21, 22].

As seen above E2E voting system designs are not a new concept in the academic literature. However, they are not widely used in practice. One of the main barriers for adoption of the existing systems is the complexity of the cryptographic protocols used, which require a certain level of mathematical maturity to understand. Therefore finding the most elementary solutions that achieves the goals listed in subsection 2.3 is important when designing a practical system.

The next section describes the design of the Split-Value Voting System [8], for which we provide an implementation. A major advantage of this design over other E2E systems is that it avoids computationally expensive computations such as modular exponentiation and public-key operations often used in other designs. Furthermore, the mix-net approach taken accommodates rules other than simple majority (or plurality) rules that constrain the applications of other designs.
4. The Split-Value Voting System Design

The Split-Value Voting System was proposed by Rabin and Rivest in 2014 [23], and further developed in [8]. To allow the reader to understand the refinements on the original design discussed in Section 5, we focus on presenting here the core components of the design, omitting some details present in the paper.

- **Representation modulo \( M \):** votes for a given race are represented by a number modulo \( M \), where \( M \) is chosen big enough to accommodate any possible voter choice.

- **Split-Value Representation:** let \( x \) be a value modulo \( M \). A split-value representation of \( x \) is any tuple \( SV(x) = (u, v) \), where \( u \) and \( v \) are values modulo \( M \) and \( x = u + v \ (mod \ M) \). Notice that, by picking \( u \) randomly among the values modulo \( M \) and setting \( v = x - u \), one can get a random split-value representation, in which knowing \( u \) or \( v \) alone does not give any information about the value of \( x \).

- **Commitment:** committing a value \( u \ mod \ M \) is the digital equivalent of sealing \( u \) in an envelope. The commitment value \( C = Com(u) \) is computationally hiding, meaning that by looking at \( C \) (the envelope) it is infeasible to gain any information about \( u \) (the content). \( C \) is also computationally binding, the content of the envelope cannot be modified, meaning that nobody can open a commitment in two different ways. Opening a commitment means to reveal the committed value \( u \) together with and any random values used to compute \( C \).

Figure 3 shows how the split-value representation and commitments are used to keep votes secret while providing evidence that they were not modified. Assume that a vote is translated to \( 14 \ mod \ 17 \). The voting system provides evidence that a vote input by the voter is equal to one of the votes used to compute the final tally in the other end. A random split-value representation of the input
vote, let’s say $SV(x) = (11,3)$ where $11 + 3 = 14 \mod 17$, is committed and posted to the Secure Bulletin Board. Similarly, for the other end, the voting system commits to the split-value representation $SV(y) = (8,6)$, together with an offset value $t = (-3,3)$ that relates the two representations. After all values are committed to, a left-right challenge is chosen randomly and either the commitments on the left or right side are opened and the equation is verified. In the example, the left side is randomly chosen one for the half-opening verification. Notice that this verification step only guarantees that an error is detected at least 50% of the times, to achieve better confidence the procedure is repeated with multiple “copies of the vote” given by different split-value representations. The second remark is that the half-opening does not reveal the content of the vote.

Figure 3. A vote translated to the number 14, is initially represented by the split-value pair $(11,3)$. The voting system proves that this vote correspond to a ballot that is counted in the final tally by comparing their split-value representation. In the commitment phase, a commitment to the input and output votes to be compared are posted in the Secure Bulletin Board, together with an offset value relating them. Then, either the left or the right side is chosen to have the votes half-opened, and anyone can verify that the input and output vote differ by the offset.
The second important component of the Split-Value voting system is the way it anonymizes the voters. The design envisions election officials, with possibly observers and party representatives, to distribute the computation of the outcome and corresponding proofs. The adversarial model assumes that agents are honest-but-curious, i.e. they follow the protocol but they may cheat by revealing their private information (we discuss extensions to actively malicious adversaries in the future work section). By distributing the computation, no agent alone can learn the correspondence between voters and votes. Depending on the number of honest-but-curious adversaries who may collude, more servers may be required to ensure privacy. Below, we give some details on how the distributed computation takes place.

- **Randomness for Challenges:** To defend against malicious servers the randomness used for challenges cannot be at the servers’ control. One option is to use a random external source such as a dice-rolling ceremony. For the implementation, we chose to extract the randomness from all data posted on the Secure Bulletin Board in the Fiat-Shamir style [24].

- **Mix-Net:** Servers in a network are used to ensure privacy by detaching voters from their respective ballots. This detachment is achieved by permuting (shuffling) the list of votes several times, and using obfuscations (details on the paper [8]) to keep the permutations hidden. To ensure this process do not modify the values of the votes, the left-right challenges mentioned previously can ensure that the input list of votes and final list of votes are equivalent except for the order.

The way information flows among the servers in the voting system is shown in Figure 4. Votes are inserted using a digital interface such as a tablet in the polling place. The original votes are split and sent to the first column of the mix servers. In the mix-net, each column reshuffles the data and sends it to the following one. In the end, 2m lists are produced by the voting system. Half of these lists are used to ensure these lists correspond to the original votes by performing a half-opening of the split-value representations. The remaining m lists have the output commitments fully opened and are used to
compute the final tally. The choice of which lists are used for checking the input consistency or computing the final tally is determined by random challenges.

Figure 4. In the Split-Value voting system information flows from left to right. First voters use a digital interface in the polling place, such as tablet, to enter the choices. The vote is converted into a split-value representation and different shares are sent to different rows of the first column of the mix-net. The Mix servers shuffle the data received and send it to the next column. The last column outputs 2m lists of votes, where half are used to guarantee consistency with the input votes and the other half to compute the final tally.

The original paper [8] also provided a shared-memory implementation by Prof. Rivest to simulate an election. (See https://github.com/ron-rivest/split-value-voting .) The provided implementation served as stepping stone for the development of this work. The given implementation allows the simulation of election for multiple races and with write-in votes. Moreover, the election simulations generate universally verifiable proofs of correctness of the election outcome. The main limitation in the provided implementation is that a single program
executes the simulation in a shared-memory environment, instead of having different agents communicating among themselves and running the computation in a distributed way.

5. Design Contribution

5.1. System Architecture and Communication Phases

In our prototype implementation, we have multiple servers, each responsible for different components of the voting system. We introduce a Controller Server to coordinate the various phases of the election. We also give details on how to generate challenges for the proofs based at two distinct times, one for the cut-and-choose and the other for the left-right challenges.

First, the Controller is initialized with the election parameters. Then all other servers ping it to learn about the election parameters and about the other servers in the network. Once communication channels are established, the Controller is the one that tells the other servers when to execute the phases of the election such as producing votes, mixing them or tallying them. The introduction of the Controller simplifies the initial setup and the coordination required among servers.

Similar to the Controller, the Secure Bulletin Board (SBB) communicates with all other servers. The SBB is the server where all non-sensitive election data is posted. It is also possible to extract randomness from the posted data, in our design the SBB server can provide randomness by computing a hash of its content.

In the current implementation, the Controller and the SBB are single points of failure in the voting system. However, by adding servers it is possible to accommodate failures, Culname and Schneider presented a possible way to do it in [25].

The Voter Server, or Voter Tablet, represents the machines where the voters make their selections. For this prototype, the casting of votes is happening in a simulated fashion with no user
interaction. Moreover, the receipts that would be given to voters to verify that their ballot were cast as intended are only posted to the bulletin board.

The Mix Servers, also referred as servers in the mix-net, have three important roles. The first role is to anonymize the votes through permutations and obfuscations. The second role is to compute the final tally. The final role is to post evidence (a proof) that the previous two functions were executed correctly. Any server can verify the outcome of the election by reading the proofs of correctness from the bulletin board.

The roles of the servers is presented in Figure 5. The communication channels among servers were shown in Figure 4, with the exception of the channels that connect the Controller and SBB Servers to all other servers.

![Figure 5. Servers and their responsibilities](image)

As mentioned previously, the Controller coordinates the phases of the election. Tables 1 through 4 describe each phase of the election and the communication that is happening in them. The details presented reflect the code implemented and correspond to one way of implementing the Split-Value Voting System design proposed in the paper [8].
Table 1 describes the initialization routine and how channels are established among servers. Before the election, the Controller learns about the election parameters such as the list of candidate for each race. The election parameters also include the desired level of robustness/security which determines the number of servers required in the mix-net. All other servers are initialized simply with the TCP/IP address of the Controller. After all servers finish pinging the Controller, letting it know they are alive, then the Controller decides on roles for each server, such as Mix Server (1,2), and sends them the election parameters. In Phase 3, the Controller forwards to all servers the IP addresses of all other servers, allowing them to establish communication channels. The implementation of secure communication channels is not yet implemented and it is described in Section 8 with other possible future work.

Table 1. Phases of the Election – Initialization Routines and Establishment of Communication Channels

<table>
<thead>
<tr>
<th>Phases</th>
<th>Description</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Initial Setup</td>
<td>-Controller receives election parameters</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-Other servers receive Controller address</td>
<td></td>
</tr>
<tr>
<td>1 Pinging</td>
<td>-Servers are turned on and ping Controller (possibly with specific role requests)</td>
<td>ping</td>
</tr>
<tr>
<td>2 Initialization of Roles</td>
<td>-Controller decides on roles and sends election parameter for initialization of each of them</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-Servers reply with key setup information</td>
<td>role, election parameters</td>
</tr>
<tr>
<td></td>
<td>-Election setup information is posted to SBB</td>
<td>key setup</td>
</tr>
<tr>
<td></td>
<td></td>
<td>election setup info</td>
</tr>
<tr>
<td>3 Broadcast of Network Info</td>
<td>-Controller broadcasts addresses and key setup information to allow for secure channels</td>
<td>addresses and keys</td>
</tr>
</tbody>
</table>
Once the setup phase is over, the election can start. Table 2 describes the vote production and vote distribution phases. In a real election, vote production would require user interaction, but in our implementation of a simulated election, the candidate choices are randomly picked. Moreover, the receipts that would also be given to the voter are only posted to the SBB, so that a verifier can later ensure that each ballot is counted. The vote distribution phase begins by distributing the shares of the split-value representation of votes to the Mix Servers in the first column of the corresponding row. We also have the Voter posting the commitments to the input split-value representation of votes, something that the first column of mix servers could also do.

Table 2. Phases of the Election – Vote Production and Distribution

<table>
<thead>
<tr>
<th>Phases</th>
<th>Description</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Vote Production</td>
<td>-Controller asks Voter Server to produce votes&lt;br&gt;-Voter Server produces votes&lt;br&gt;-Voter creates split value representation and by hashing them generate a receipt&lt;br&gt;-Voter posts voter receipts to SBB</td>
<td>Voter → voter receipts → SBB</td>
</tr>
<tr>
<td>5 Vote Distribution</td>
<td>-Controller asks Voter to distribute votes&lt;br&gt;-Voter distributes split-value representation to first column of Mix Servers&lt;br&gt;-Voter posts cast vote commitments to SBB</td>
<td>Voter → cast vote commitments → SBB → Mix Servers First Column</td>
</tr>
</tbody>
</table>

Tables 3 describes how the Mix Servers process the split value representations to hide the correspondence of voters to their ballots. The first step is to produce multiple copies of the votes using different random split-value representations. In the end of the election half of these copies are used to check for consistency between the input and output through half-openings and the other half used for computing the final tally as described in Section 4. Each column in the mix-net agrees on permutations and obfuscation values. As long as nobody can learn these for all the columns it is impossible to recover the link between a voter and her ballot, guaranteeing her privacy. Once the permutation and
obfuscation values are shared, the input votes are processed across each row by first applying the permutation and then adding the obfuscation values. The information is transferred from the mix server indexed by row i, column j, to the server indexed by the same row i and column j+1.

Table 3. Election Phases – Mixing and Tallying

<table>
<thead>
<tr>
<th>Phases</th>
<th>Description</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>6. 1 Mix Votes: replicate input</td>
<td>-Controller asks Mix Server to mix -First column of Mix Servers replicates input up to 2m copies</td>
<td>Mix Servers First Row</td>
</tr>
<tr>
<td>6. 2 Mix Votes: generate permutations</td>
<td>-First row of mix servers generates permutation and shares them across columns</td>
<td>Mix Servers Other Row</td>
</tr>
<tr>
<td>6. 3 Mix Votes: generate obfuscation values</td>
<td>-First row of mix servers generates obfuscation values and shares them across columns</td>
<td>Mix Servers First Row</td>
</tr>
<tr>
<td>6. 4 Mix Votes: process columns left-to-right</td>
<td>-Mix Server(i,j) mixes split votes (shuffles and obfuscates) and forwards them to Mix Server(i, j+i) in same row but next column</td>
<td>Mix Server (i, j)</td>
</tr>
</tbody>
</table>

In Table 4, we show the phases for the proof production part of the election. In the reference implementation provided with the original paper [8], this phase was based on a single commitment when both the cut-and-choose and the left-right challenges were generated at once. However, in the development of the multi-processor implementation described here, the need to separate the two challenges became clear. The reason was that computing the offset values required sharing the permutations, which would violate voters’ privacy for the list of votes used in the outcome production. Therefore, the proof production was broken into five main parts. First, the commitments to the output are posted. With all the commitments posted, the SBB provides randomness from the committed data to generate the cut-and-choose challenges, which determines the lists used for input comparison. Only for the copies to be used for input comparison, the permutation and offset values used to generate the
output are committed to and posted. Similar to the cut-and-choose challenges, the proof servers then get randomness from the SBB to compute the left-right challenges. The mix servers post half openings for input and output in the input comparison lists based on the left-right challenges. To conclude the proof production phase, the mix servers open fully the commitments to the lists selected by the cut-and-choose challenge for the outcome production. Once the proofs are produced the tally is computed. The servers in the last column receive the output from all rows and compute the final tally. One of the servers is chosen arbitrarily to report the tally to the bulletin board.

Table 4. Election Phases – Proof Production

<table>
<thead>
<tr>
<th>Phases</th>
<th>Description</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Proof Production: Output Commitments</td>
<td>-Controller asks Mix Server to make proofs</td>
<td>Mix Servers Last Column</td>
</tr>
<tr>
<td></td>
<td>-Last column computes output commitments and posts them to SBB</td>
<td>output commitments</td>
</tr>
<tr>
<td>7.2 Proof Production: Cut-and-Choose</td>
<td>-Mix Servers get hash from SBB and compute Cut-and-Choose challenges.</td>
<td>Mix Server (i, j)</td>
</tr>
<tr>
<td>Challenges</td>
<td></td>
<td>request</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hash of content</td>
</tr>
<tr>
<td>7.3 Proof Production: Offset Commitment</td>
<td>-For input comparison lists, Mix Servers share permutations among themselves.</td>
<td>Mix Server (i, j)</td>
</tr>
<tr>
<td></td>
<td>-First column servers share inputs to corresponding last column servers.</td>
<td>input comparison permutations</td>
</tr>
<tr>
<td></td>
<td>-Last Column computes and post offsets for input comparison</td>
<td>Mix Servers Last Column</td>
</tr>
<tr>
<td></td>
<td></td>
<td>input comparison offset values</td>
</tr>
<tr>
<td>7.4 Proof Production: Left-Right Challenges</td>
<td>-Mix Servers get hash from SBB and compute left-right challenges.</td>
<td>Mix Server (i, j)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>request</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hash of content</td>
</tr>
<tr>
<td>7.5 Proof Production: Post-Challenge</td>
<td>-For output production lists, last column posts openings of output commitments</td>
<td>Mix Servers Last Column</td>
</tr>
<tr>
<td>Proofs</td>
<td>-For input comparison lists, first column and last column post either left or right openings of split-values.</td>
<td>Mix Servers First Column</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full or half openings of output commitments</td>
</tr>
<tr>
<td></td>
<td></td>
<td>half openings of input commitments</td>
</tr>
<tr>
<td>8 Computing Tally</td>
<td>-Last column of Mix Servers shares final split values among themselves</td>
<td>Mix Servers Last Column</td>
</tr>
<tr>
<td></td>
<td>-A Mix Server computes tally and posts it to SBB</td>
<td>split values</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tally Mix Server</td>
</tr>
<tr>
<td></td>
<td></td>
<td>computed tally</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SBB</td>
</tr>
</tbody>
</table>
For the last step of the election, any computer, including a voter’s one can verify the election. This verifier gets all the data from the SBB and can verify all the proofs ranging from checking the commitments values to reconstructing the final tally.

5.2. Main Design Points

One could implement the Split-Value Voting System in different ways. Here we highlight some of the design decisions made for this implementation and provide the rationale behind them. The topics discussed range from the data storage solutions to communication tools. We omit further discussion over the two design contributions presented in subsection 5.1, namely the introduction of the Controller and the refinements of the details of the proof production with two separate challenge phases.

For storing the election parameters and the other election metadata, we decided to use nested dictionaries as our databases. The main advantages of using nested dictionaries is that they provide a good abstraction to encapsulate data and they allow for compact storage and data transfer not requiring repetitions of the keys.

We decided to use packages contained in the standard Python distributions for implementing the external communication methods. We use standard sockets and socket servers. We implement each server role defined in section 5.1 as a pair of server and handler classes. The server classes contain the methods that request data from other servers, while the handler classes process incoming requests. Additionally, the server classes keep some state information required for communication to other servers. Alternatively, one could use external packages such as Remote Python Calls (RPyC) or Python Remote Objects (Pyro). The packages contained in the standard distribution were favored for two main reasons. The first reason is that the standard package provides more flexibility in how messages are sent, allowing
communication to happen more compactly. The second reason and maybe more important is that it simplifies the installation requirements to deploy the voting system in new machines.

In subsection 5.1, we already introduced the Controller that helps with the coordination of the other servers. It is also relevant to highlight that the Controller abstract the details on how the tallying, the proof production or other phases of the election occur. It provides ways for the other servers to synchronize phases among themselves. A phase is ended, when all server responsible for it reply to the Controller saying they are done. The Secure Bulletin Board server also has a modular design. This modularity may allow the SBB to be used with other or variant end-to-end designs.

6. Experimental Setup

One goal of this project was to make our experiments reproducible, meaning that other programmers should be able to download the source code, build it and run it.

We ran experiments using the clouding computing infrastructure provided by the Computer Science and Artificial Intelligence Laboratory (CSAIL). CSAIL provides infrastructure as a service based on the OpenStack software suite. With 76 physical nodes and 1,000 physical cores, they currently use Ubuntu 14.04LTS (Trusty) with KVM as the virtualization layer. (See http://tig.csail.mit.edu/wiki/TIG/OpenStack.)

To analyze the performance of our implementation we conducted tests in two different scenarios: using a single virtual machine and multiple virtual machines. In the single virtual machine scenario, we used a 12-core virtual machine, with 48GB of RAM and 64GB of disk. In the multiple virtual machine scenario, we used 12 instances of a 1-core virtual machine, with 4GB of RAM and 64GB of disk each. The implementation was also tested using multiple processes on an Intel Core 2 Duo laptop with 3GB of RAM and that runs Windows 7. However, we do not include the data on these latter simulations since they showed results comparable to the single virtual machine scenario.
The language chosen for implementing the Split Value Voting System was Python version 3.4, to favor easy readability of our reference implementation. No Python packages were used other than those in the standard library. Furthermore, all the communication was done using Python sockets and the JSON serializer, therefore it is possible to replace components of the overall system by pieces written in different programming languages as long as they follow the API’s provided.

For the simulations, we varied the number of voters while keeping the other election parameters fixed. The number of voters was either 100, 1000 or 10000. The simulations presented two races one being a yes/no question and the other being a race between two candidates with a write-in option. For the security parameters, we choose the number of copies to be 24 (as suggested in the original paper[8]) and we allowed up to two potentially evil (gossipy) servers. As such, our simulation involved nine mix servers, for a total of twelve servers or processes. Further details about the default election parameters can be obtained from the file sv_main.py in the codebase.

A snapshot of the code used in the experiments and referred throughout this thesis is available at https://github.com/marco-p/mpc-split-value-voting.

7. Performance Analysis

We used virtual machines with the same images to ensure consistency across the results. In the single virtual machine tests, the focus was in collecting information on the overall election time and storage requirements. The overall election time measures the time from the moment all servers pinged the Controller to the time the tally and proofs are posted. In order to analyze the storage requirement for the simulations, we measured the file size of the bulletin board.

Table 5 shows the results from the single virtual machine experiment. As expected, as the numbers of voters increases, we see a linear increase in the storage size of the bulletin board and the
overall election time. It is important to highlight though that both numbers are preliminary given that digital signatures and parallelization were not yet implemented in the code. In addition, these are only the first results for simulations with multiple agents and the code can be optimized to achieve better performance.

In the original paper [8], the authors estimated the size of the SBB for a 1 million voters and with the same other election parameters to be around 4.5GB. If we extrapolate our results to one million voters we would get a file of 64GB. Our results show the input and output commitments taking 23GB instead of the 4.5GB estimate. Most likely, this difference occurs because we use slightly larger commitment sizes and because of the indentation on the sbb file. The rest of the storage taken in the sbb file is dominated by the posting of the offset values, the outcome check and the output openings.

Table 5. The size of the SBB storage and the overall election for various number of voters

<table>
<thead>
<tr>
<th>Number of Voters</th>
<th>SBB Storage</th>
<th>Overall Election Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>6.8 MB</td>
<td>138.2 sec</td>
</tr>
<tr>
<td>1000</td>
<td>68.2 MB</td>
<td>504.3 sec</td>
</tr>
<tr>
<td>10000</td>
<td>654MB</td>
<td>4325.7 sec</td>
</tr>
</tbody>
</table>

In the experiment with twelve separate virtual machines, we studied bandwidth as a potential bottleneck, by contrasting the scenarios with one virtual machine and the one with various virtual machines. Table 6 shows the results of such experiment. For a fixed number of voters, the time in the two scenarios were very similar. Most likely, the bandwidth provided between Virtual Machines in
the infrastructure provided was not a bottleneck. Further studies should be conducted in a constrained bandwidth scenario, possibly by using physical machines. Another interesting fact was that the 12-virtual-machine simulation for 10000 voters did not run until completion. The simulation halted during the proof production phase, due to an out of memory error, since servers started to reach the 4GB RAM limit. If the linear trend is reflected in the memory usage, we anticipate that running larger elections may require a better management of memory and possibly writing objects to disk.

In the original paper [8], the authors provided an estimate on the computation time of running the cryptographic functions, such as opening commitments and performing private-key operations. However, our preliminary simulations suggest that the computational bottleneck may be in the transferring and parsing of data.

Table 6. The overall election time for the one and many virtual machine scenarios

<table>
<thead>
<tr>
<th>Number of Voters</th>
<th>Single VM</th>
<th>Multiple VMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>138.2 sec</td>
<td>145.3 sec</td>
</tr>
<tr>
<td>1000</td>
<td>504.3 sec</td>
<td>601.2 sec</td>
</tr>
<tr>
<td>10000</td>
<td>4325.7 sec</td>
<td>*Memory Error</td>
</tr>
</tbody>
</table>
8. Future Work

As mentioned previously, the discussion and results presented in this paper are preliminary. The rest of the team, currently Master of Engineering candidate Charles Liu, Professor Rivest and Professor Rabin, will continue the work to provide a full implementation of the original design and to include further extensions to it. The most important aspects of the original design that were not yet implemented are the following:

- **Secure Communication:** Allow servers to communicate in secure channels with encrypted communication and digital signatures. This includes modifying the current network methods and performing the initial key setup in the beginning of the election.

- **Handling Server Failures:** Introduce backup servers who can replace a Mix Server in case one fails.

Besides those two aspects that were presented in the original paper, further possibilities of future work include:

- **Secure Bulletin Board:** Introduce digital signatures to all postings to identify sources and make the storage persistent during election.

- **Parallelism:** Many phases of the election, especially in the communication between the Controller and Mix Servers can happen simultaneously in order to achieve better performance.

- **Extending the Adversarial Model:** one example would be handling malicious servers that could send arbitrary messages.

Lastly, there is some further work that can be done in the experimental aspect of the project. One can simulate the elections on a cluster of physical machines to simulate a real scenario. Moreover, one can provide comparisons to other verifiable voting systems such as Wombat[7] and STAR-Vote[17].
9. Conclusion

We described the multi-dimension complexity of voting systems and identified the Split-Value design as a candidate for providing verifiability while addressing practical requirements. We refined details not fully described in the original paper to implement the system in a multi-processor platform. Our preliminary experiments are able to simulate verifiable elections with up to ten thousand voters. In the simulations, the system performed rapidly and flawlessly. Lastly, we documented the current state of the system development and proposed possible future work to extend the security guarantees.

10. Bibliography


