From Ballots to Bytes: Voting Electronically with Secret Sharing & Homomorphic Encryption

Mikkel Mathias Sieling Katholm: 202107199
Ketil Sylvester Badsberg: 202106758

Bachelor Report (15 ECTS) in Computer Science
Advisor: Sophia Yakoubov
Department of Computer Science, Aarhus University
December 2024

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

Electronic voting systems require robust security mechanisms to ensure the integrity, privacy, and correctness of the voting process. This bachelor project investigates and compares different cryptographic schemes suitable for electronic voting, focusing on secret sharing and encryption methods. We study three secret sharing schemes: Additive secret sharing, Replicated secret sharing, and Shamir secret sharing. We analyze their properties, error tolerance, and suitability for large-scale elections. Additionally, we explore homomorphic encryption, particularly their application in secure voting through a scheme such as ElGamal encryption, and the potential integration of zero-knowledge proofs to enhance voter privacy and verifiability. By comparing these schemes, we highlight their strengths and limitations, presenting a comprehensive overview in tabular form to guide the selection of the appropriate scheme. Our findings suggest that while secret sharing offers perfect secrecy, encryption has several desirable properties in an electronic voting setting. With the compatibility of zero-knowledge proof, encryption has the potential to be used in a great scheme. The key distribution in the encryption scheme is flawed and we outline a solution by secret sharing the key and encrypting the votes. This is a promising approach that ensure enhanced security and practicality in electronic voting systems.

Keywords: *Electronic voting, Secret sharing, Homomorphic encryption, Additive secret sharing, Replicated secret sharing, Shamir secret sharing, Berlekamp-Welch algorithm, ElGamal encryption.*

Mikkel Katholm & Ketil Badsberg
Aarhus, December 2024.

Contents

1 Introduction

A central pillar of today's society is democracy. Each year, millions of people vote on a wide range of topics, everything from what movie to watch in the local film club to presidential elections. Some polls are significantly more sensitive than others, when voting on what movie to watch it might be enough to do a raise of hands, whereas the election of a world leader one may prefer not to publicly share any political preferences. How can we ensure the correctness of a poll without having the voters reveal their votes? Throughout this report we will be looking at several types of cryptographic schemes that can ensure the correctness of an election, while the votes remain private. In today's modern society a large proportion of infrastructure is handled electronically, therefore the schemes we will be analyzing will all be used in an electronic voting setting. Like all technologies there are tradeoffs for each scheme, even though one of the simplest schemes can be proven to uphold the properties needed, it is under the assumption that the participants involved are semi-honest. i.e. they do not deviate from the protocol, but can attempt to learn something about the vote. If a corrupted participant takes part in the process, the scheme has to be able to detect and correct errors.
We will consider a client-server setting where multiple clients send their data to multiple servers. This also applies for our implementation of the schemes. One of the main advantages of using a client-server setting, over a party setting, is a client might be a relatively small computing unit and therefore not have a lot of memory and computing power, whereas a server will have significantly more computational power.
For each scheme within this report we lay out the theory behind the schemes, the algorithms and provide formal proofs of the properties necessary for electronic voting purposes. For any scheme to be used in electronic voting it has to able to combine the votes in such a way such that the output is the result of the vote, without having to reveal any single vote. This property is knows as additive homomorphism.
We present three well known secret sharing schemes: *Additive secret sharing*, a simple but limited scheme that functions as an introduction to secret sharing; *Replicated secret sharing* that extends the previous scheme to provide practical flexibility; and *Shamir secret sharing* (1979) which provides even more flexibility. We also showcase how and under what circumstances error detection and correction can be utilized. This includes simply comparing data and using majority to determine the correct value as well as using the *Berlekamp-Welch* (1986) algorithm in the Shamir secret sharing scheme.

While it is possible to vote electronically using secret sharing it might not always be ideal in a given situation. Encryption is an alternative approach with different benefits as well as disadvantages. We examine the linearly homomorphic encryption scheme *ElGamal* (1985) which is a type of public key encryption, such that voters encrypt their vote and only an authorized participant with the secret key can decrypt the vote. Furthermore we will see how to combine ElGamal encryption with Shamir secret sharing such that the secret key remains private.

The two approaches are applicable for real life elections but in reality paper based solutions are still used throughout the modern world. It may not be the most optimal solution as the manpower that goes into each election is significant since it requires manually counting the votes. In a highly digitalized society such as Denmark, many

wonder why we are not using our already implemented infrastructure to vote electronically. And indeed why? As we will see in this report: It is clearly possible to implement.

Based on the theory presented we provide an implementation of Replicated secret sharing, Shamir secret sharing, and ElGamal Encryption to showcase how they would work in a practical setting. The schemes has been implemented locally and not in a true network setting, nevertheless the code provides an excellent proof of concept and can be extended to work on a distributed network. Besides implementing the schemes, we have added both error detection and correction to further showcase for the schemes can function in a real world setting there some participants might become corrupted.

2 Secret Sharing

The concept of secret sharing refers to the ability to split a secret into multiple parts, called *shares*, which can be distributed to others, called *shareholders*. The shares can later be pooled and combined to recover the secret. This means that a person can share a secret with others without revealing it to them individually. The secret can only be revealed once the *reconstruction threshold* t_r is met, meaning at least t_r shares have to be combined to recover the secret. This number will differ depending on the scheme. Further more the *privacy threshold* t is the maximum amount of shares that reveal nothing about the secret.

2.1 Notation

The following notation is used throughout the report.

Notation

t_r : Reconstruction threshold
t : Privacy threshold
n : Number of shares generated
m : The message/secret
r : The randomness
S : The set of shares
s_i : The i'th share
C : The set of clients
c_i : The i'th client
K : The set of shareholders
k_i : The i'th shareholder
p : Large agreed upon prime
$\mathbb{Z}_p$: Set of $\{0, 1, \ldots p-1\}$

2.2 Algorithms

Any secret sharing scheme utilizes two algorithms: one for generating the shares and one for reconstructing the message.

Algorithmic Definitions

$$genShares(m, r) \rightarrow S = \{s_1, s_2, \ldots, s_n\}$$
$$rec(S) \rightarrow m'$$

The function $genShares(m, r)$ takes a message and some randomness as input and outputs n shares chosen according to the algorithm. The $rec(S)$ takes a given number of shares and runs the appropriate reconstruction algorithm and thus outputs m'.

2.3 Properties

The secret sharing schemes must uphold the three properties correctness, privacy and additive homomorphism.

Correctness

$$
\begin{aligned}
&\forall m, \forall r \\
&\forall Q \subset [1,2,\ldots,n] \text{ s.t. } |Q| \geq t_r \\
&\mathit{genShares}(m,r) \rightarrow S = \{s_1, s_2, \ldots, s_n\} \\
&\mathrm{Rec}(S_Q) \rightarrow m' \\
&\text{such that } m = m'
\end{aligned}
$$

The correctness definition simply says that for a message encoded into shares using the scheme's *genShares* function, the message can be recovered using the corresponding reconstruction function. It is worth noting that this property does not consider shares that has been tampered with and can only be applied to honest shares. In later sections we will cover how to handle shares that has been tampered with.

Privacy

$$
\begin{aligned}
&\forall m, m' \\
&\forall Q \subset [1,2,\ldots,n] \text{ s.t. } |Q| \leq t \\
&S \leftarrow \mathit{genShares}(m,r) \\
&S' \leftarrow \mathit{genShares}(m',r') \\
&S_Q = \{s_i\}_{i \in Q},\ S'_Q = \{s'_j\}_{j \in Q} \\
&\forall x \\
&Pr_r[S_Q = x] = Pr_{r'}[S'_Q = x]
\end{aligned}
$$

In a secret sharing scheme, a given message m is divided into multiple shares S using randomness r. The same is done with with a different message m' and randomness r'. The above definition describes how any set of Q with at most t shares, the shares S_Q and S'_Q are equally likely to be the same as x, which denotes all possible combination of shares. This means that the set S_Q reveals absolutely nothing about m.

Additive Homomorphism

$$
\begin{aligned}
&\forall m, m' \\
&S \leftarrow genShares(m, r) \\
&S' \leftarrow genShares(m', r') \\
&s_i^{\diamond} = s_i \diamond s_i' \\
&S^{\diamond} = \{s_1^{\diamond}, s_2^{\diamond}, \ldots, s_n^{\diamond}\} \\
&S^{fresh} \leftarrow genShares(m + m', r^{fresh}) \\
&\forall x \\
&Pr[S^{fresh} = x] = Pr[S^{\diamond} = x]
\end{aligned}
$$

Where $\diamond$ is a scheme specific operation

The additive homomorphism property claims that for any two messages encoded into shares, the two set of shares can be combined into a new set of shares with the same property as a fresh set of shares. When the combined shares are reconstructed, the output is the sum of the two messages.
The requirement that the combined share must be distributed exactly as a fresh sharing of the two messages summed is a strong version of additive homomorphism. Requiring this ensures that the combined shares uphold the same properties as a fresh set of shares. Additive homomorphism without this strong definition still makes it possible to combine shares such that the sum can be reconstructed, however it does not necessarily uphold the definition of privacy.

3 Secret Sharing Schemes

This report covers three secret sharing schemes: Additive-, Replicated- and Shamir secret sharing. The following sections will show how the schemes algorithms operates as defined in section 2.2 and how they uphold the secret sharing properties defined in section 2.3. The secret sharing schemes throughout this report will all be using some large agreed upon prime denoted as p. Note that the prime must be chosen to be larger than the message, otherwise the modulo operation would wrap the message, thus making it impossible to recover the original message.

3.1 Additive Secret Sharing

One of the simplest way of sharing a secret is by using Additive secret sharing. This protocol is rather simple and upholds the required properties and functions as a secret sharing scheme. However, the scheme lacks practical functionality such as error detection and correction, as we will see later in the report.

Algorithms

genShares$(m, r) \rightarrow S$

$$s_1, \ldots, s_{n-1} \leftarrow \mathbb{Z}_p$$
$$s_n = m - \sum_{i=1}^{n-1} s_i \mod p$$
$$S = \{s_1, s_2, \ldots, s_n\}$$

To generate n shares the algorithm picks $n-1$ numbers uniformly at random from $\mathbb{Z}_p$, these numbers act as the first $n-1$ shares. The final share is computed as the message m subtracted all other shares, thus providing the correlation needed for reconstruction (Cramer et al., 2015, and references therein).

rec$(S) \rightarrow m$

$$m = \sum_{i=1}^{n} s_i \mod p$$

The original message can be computed as the sum of all shares.

Correctness: By applying the definition of $genShares(m,r)$ we argue

$$\begin{aligned} s_n &= m - \sum_{i=1}^{n-1} s_i \mod p \\ -m &= -s_n - \sum_{i=1}^{n-1} s_i \mod p \\ m &= s_n + \sum_{i=1}^{n-1} s_i \mod p \\ m &= \sum_{i=1}^{n} s_i \mod p \end{aligned}$$

Because of the correlation between the first $n-1$ shares and the last share as well as the properties of addition and modulo, the original message can be computed by taking the sum of all n shares

Privacy: There are two types of shares in this scheme: the first $n-1$ shares that are chosen uniformly at random and the last share s_n that are dependent on all other shares. To prove privacy for the first $n-1$ shares, the definition of uniformly randomness are applied. The last share is dependent on the uniformly randomly chosen shares and thus the definition also applies for the last share. In other words for any message m the probability of guessing the missing share is the same for any m because the shares are chosen at random.

Additive Homomorphism: We argue that two encoded messages can be combined in such a way that the reconstruction produces the sum of the messages.

Given any message m and m' and randomness r and r'

$$\begin{aligned} S &= \{s_1, s_2, \ldots, s_n\} \leftarrow genShares(m,r) \\ S' &= \{s'_1, s'_2, \ldots, s'_n\} \leftarrow genShares(m',r') \end{aligned}$$

Applying the scheme specific operation $\diamond$ which in this case is adding the shares.

$$S^{\diamond} = \{s_1 + s'_1, s_2 + s'_2, \ldots, s_n + s'_n\}$$

Applying the definition of $rec(S)$:

$$rec(S^{\diamond}) = \sum_{i=1}^{n} s_i + s'_i = \sum_{i=1}^{n} s_i + \sum_{i=1}^{n} s'_i = rec(S) + rec(S') = m + m'$$

Given that both sets of shares are generated using independent randomness the summation of the shares are still random. Since the individual share reveal absolutely nothing about the message, neither does the summed shares.

3.2 Replicated Secret Sharing

As we saw in section 3.1, one can use Additive secret sharing to share a message encoded as a number. The scheme has minimal flexibility and requires all shareholders to reconstruct the message. Replicated secret sharing takes the protocol of Additive secret sharing and adapts it to make a new scheme where a fixed number of shareholders are required to reconstruct the secret.

Theory

A message m is encoded into n Additive shares by running the $genShares(m,r)$ defined in section 3.1. The shares each shareholder will possess are defined as $n-1$ Additive shares, thus one Replicated share consists of $n-1$ Additive shares. As we saw in section 3.1, the shareholders does not have enough information to infer anything about the message. The major difference from Additive secret sharing is that only two shareholders need to pool their shares, given that each shareholder is only missing one Additive share to reconstruct the message. This ensures that the message can be reconstructed even if some shareholders loose their share. Shareholders simply need to collect two Replicated shares, and thus they have n Additive shares and can run the reconstruction algorithm.
This version of Replicated secret sharing can be generalized such that a custom reconstruction threshold t_r can be set, by adjusting the amount of Additive shares in each Replicated share. All Replicated shares must be constructed so that any t_r Replicated shares combined contain all n Additive shares, such that the message can be reconstructed. Even though it is possible to set t_r to any desired value, we will through the rest of the report only be considering the case where $t_r = 2$.

Algorithms

The algorithms for Replicated secret sharing uses the Additive algorithms which we denote as $ADD.genShares(m,r)$ and $ADD.rec(S)$.

$genShares(m,r) \rightarrow S$

$$
\begin{aligned}
&ADD.genShares(m,r) = \{as_1, as_2, \ldots, as_n\} \\
&s_1 = \{as_2, as_3, \ldots, as_n\} \\
&s_2 = \{as_1, as_3, \ldots, as_n\} \\
&\quad \vdots \\
&s_n = \{as_1, as_2, \ldots, as_{n-1}\} \\
&S = \{s_1, s_2, \ldots, s_n\}
\end{aligned}
$$

A Replicated share contains $n-1$ Additive shares.

$rec(S) \rightarrow m$

$$|S| \geq 2$$
$$m = ADD.rec(S)$$

Given that a Replicated share consists of $n-1$ Additive shares any set of two Replicated shares contains all the Additive shares. Thus by collecting at least two Replicated shares, the message can be reconstructed by running the reconstruction of Additive shares.

Error detection and correction

When encoding a message and distributing its shares among shareholders it is essential to ensure the integrity and reliability of the shares. A corrupt shareholder could deliberately supply incorrect information during the reconstruction process such that the reconstructed message is not identical to the original message. Thus it is essential to know when such an incidence occurs. To check for errors the protocol can be extended so all shareholders that hold a copy of the same Additive share will securely share this with each other and thus they can see if there are any inconsistencies (Cramer et al., 2015). To correct the errors, the shareholders with a copy of the share vote by majority on what the correct share is. While the protocol can correct $\left\lfloor \frac{n-2}{2} \right\rfloor$ corruptions it can however detect $n-2$ errors.

Correctness: Given any message m and randomness r, the share are constructed by $genShares(m,r) \rightarrow s_1, s_2, \ldots, s_n$. The shares are distributed among all shareholders such that each shareholder k_i posses a Replicated share. Observe that the the i'th shareholder does not posses the i'th Additive shares. To reconstruct the shares the shareholders pool their information and runs $rec(S)$ and gets the message.
The argument used for the mathematical proof is equivalent to the correctness proof for Additive secret sharing and can be found in section 3.1: Correctness.

Privacy: Since every Replicated share contains $n-1$ Additive shares and n are required to reconstruct the message, any single Replicated share reveal nothing about the encoded message as per the privacy proof from Additive secret sharing given in section 3.1.

Additive Homomorphism: Given that the only difference from Additive secret sharing is that each shareholder hold more Additive shares, the proof from section 3.1 still holds.

3.3 Shamir Secret Sharing

Shamir secret sharing has several advantages over the previous schemes. One of the major ones are the possibility to set a custom threshold for the amount of shares needed to reconstruct the message, while still being able distribute a single share to each shareholder. The message owner can distribute any given amount of shares to

a shareholder thus giving shareholders with different privileges a different amount of shares. Unlike Additive and Replicated secret sharing where all or at least two shareholders, respectively, can reconstruct the message. Shamir provides high flexibility with this custom threshold such that the reconstruction does not require too many or too few shareholders. Furthermore, Shamir also has the advantage that the message owner can distribute more shares if shareholders cease to exists, looses their shares, or simply if others also needs a share.
The scheme encodes the message at a point on a polynomial of a chosen degree, typically at $x = 0$ and distributes points of the polynomial as shares to shareholders. By pooling a number of shares one can reconstruct the polynomial by utilizing Lagrange Interpolation, and thus compute the value of the message by evaluating the polynomial at the chosen x value.

Theory

The foundation of Shamir secret sharing relies on Lagrange Interpolation for splitting a message into shares and reconstructing the message from these shares. Additionally we use the Berlekamp-Welch algorithm to correct errors when reconstructing the message by locating and removing corrupted shares.

Lagrange Interpolation utilizes data points (x_i, y_i) to construct a unique polynomial L of degree t such that $L(x_i) = y_i$. L is referred to as the Lagrange polynomial. To construct L so-called basis polynomials l_i are made from the data points:

Lagrange Interpolation

Given data points $((x_1, y_1), (x_2, y_2), \ldots, (x_{t_r}, y_{t_r}))$

$$l_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^{t_r} \frac{x - x_j}{x_i - x_j} \mod p$$

$$L(x) = \sum_{i=1}^{t_r} y_i l_i(x) \mod p$$

Where $l_i(x)$ is the i'th basis polynomial and $L(x)$ is the Lagrange interpolating polynomial.

Combining t_r of these basis polynomials results in the unique polynomial L.

Algorithms

Lagrange Interpolation allows for secret sharing by constructing a polynomial P of degree t where a message is encoded at a chosen x value. In this section we will encode at $x = 0$. Shares of the message can be computed as points on P where the set of shares is defined as $S = \{(1, (P(1)), (2, P(2)), \ldots, (n, P(n))\}$. As seen Lagrange Interpolation can be used to reconstruct the polynomial given at least t_r points that are generated on the original polynomial. This allows for the decoding of m by evaluating

the reconstructed polynomial at $P(0) = m$. One of the main advantages of this scheme is it allows for choosing both the reconstruction threshold and the number of shares independently of each other.

genShares$(m, r) \rightarrow S$

Compute P where a_i is chosen uniformly at random from $\mathbb{Z}_p$:

$$a_1, a_2, \ldots, a_t \leftarrow \mathbb{Z}_p$$
$$P(x) = m + a_1 x + a_2 x^2 + \cdots + a_t x^t \mod p$$

Compute shares:

$$S = \{(1, P(1)), (2, P(2)), \ldots, (n, P(n))\}$$

The reconstruction of the polynomial and the message can be done as described in the previous section. The message can also be computed without constructing P by combining the Lagrange basis polynomial and Lagrange polynomial computation:

rec$(S) \rightarrow m$

$$P(0) = \sum_{i=1}^{t_r} y_i \prod_{\substack{j=1, \\ j \neq i}}^{t_r} \frac{x_j}{x_j - x_i} \mod p = m$$

Why we use finite fields

To show why we use finite fields, i.e. modulo arithmetic and not integer arithmetic, we use an example to show why integer arithmetic is not sufficient for privacy and how modulo arithmetic is.

Consider a message encoded into a two-degree polynomial which means that $t_r = 3$. A malicious party Eve obtains $t = 2$ shares and attempts to learn *something* about the message.
Eve does the following, she collect two shares s_1, s_2 and computes two partial second-degree equations, one for each share. By subtracting the two equations from each other and isolating an unknown coefficient, Eve can eliminate one coefficient and isolate the message. The resulting equation does not reveal the message but yields *some* information about it.

$$
\begin{aligned}
&s_1 = (1, 53)\\
&s_2 = (2, 124)\\
&f(x) = a_1 \cdot x^2 + a_2 \cdot x + m\\
&f(1) = 53 \Rightarrow a_1 \cdot 1^2 + a_2 \cdot 1 + m = 53\\
&f(2) = 124 \Rightarrow a_1 \cdot 2^2 + a_2 \cdot 2 + m = 124\\
&f(2) - f(1) = 124 - 53 \Rightarrow (4a_1 - a_1) + (2a_2 - a_2) + (m - m) = (124 - 53)\\
&\qquad\qquad\qquad\qquad \Rightarrow 3a_1 + a_2 = 71\\
&a_2 = 71 - 3a_1\\
&f(1) = 53 \Rightarrow a_1 + 71 - 3a_1 + m = 53\\
&m = 2(a_1 - 9)
\end{aligned}
$$

By knowing only t shares, Eve learns that the message is an even value which significantly narrows down the possible values of the message. This clearly breaks the definition of privacy as all messages are not equally likely.

Applying finite fields solves this issue. Given a prime p, apply modulo p operations to the polynomial. In this example $p = 73$.

$$
\begin{aligned}
&s_1 = (1, 53)\\
&s_2 = (2, 51)\\
&f(x) = a_1 \cdot x^2 + a_2 \cdot x + m \mod 73\\
&f(1) \Rightarrow a_1 \cdot 1^2 + a_2 \cdot 1 + m \equiv 53 \mod 73\\
&f(2) \Rightarrow a_1 \cdot 2^2 + a_2 \cdot 2 + m \equiv 51 \mod 73\\
&f(2) - f(1) \equiv (51 - 53) \mod p \Rightarrow (4a_1 - a_1) + (2a_2 - a_2) + (m - m)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \equiv (51 - 53) \mod 73\\
&\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow 3a_1 + a_2 \equiv 71 \mod 73\\
&a_2 \equiv 71 - 3a_1 \mod 73\\
&f(1) \equiv a_1 + 71 - 3a_1 + m \mod 73\\
&m \equiv 2(a_1 - 9) \mod 73
\end{aligned}
$$

Using the above result Eve can not conclude that the message is even due to the modulo operation's value wrapping property. This behavior prevents any leakage of the message given t shares and thus privacy holds.

Packed Shamir

While standard Shamir provides a high flexibility, we have only seen how to encode one message into a polynomial. Dahl (2017) shows how to modify the generated polynomial such that we can encode multiple messages into a single polynomial without compromising on security. The messages will be encoded along the negative x-axis while the shares to distribute will be points located along the positive x-axis. Note that because of modulo arithmetic the message encoded at $x = -1$ will be located at $-1 \mod p = p - 1$. This can be done by choosing point uniformly at random until there

are enough points for the polynomial to reach the desired degree. Note that for each extra message to encode the degree of the polynomial will increase by one.

genShares$(m,r) \rightarrow S$

Given a set of messages $M = \{m_0, \ldots, m_i\}$ of size $|M| > 0$.
Note for a set of messages the reconstruction threshold t_r increases by $|M| - 1$

$$n \geq t_r$$
$$x_1, x_2, \ldots, x_n = \{1, 2, \ldots, n\}$$
$$x_1^M, x_2^M, \ldots, x_{|M|}^M = \{1 - |M|, 2 - |M|, \ldots, 0\} \mod p$$
$$y_1, y_2, \ldots, y_{t_r - 1} \leftarrow \mathbb{Z}_p$$
$$D = \{(x_1^M, m_{|M|-1}), \ldots, (0, m_0), (1, y_1), \ldots, (t_r - |M| + 1, y_{t_r - |M| + 1})\}$$
$$P(x) = \sum_{(x_i, y_i) \in D} y_i \prod_{\substack{x_i \neq x_j \\ (x_j, y_j) \in D}} \frac{x - x_j}{x_i - x_j} \mod p$$
$$S = \{(x_1, P(x_1)), (x_2, P(x_2)), \ldots, (x_n, P(x_n))\}$$

rec$(S) \rightarrow m$

$$n \geq t_r$$
$$|M| = \text{Number of messages}$$
$$S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$$
$$P(x) = \sum_{(x_i, y_i) \in S} y_i \prod_{\substack{x_i \neq x_j \\ (x_j, y_j) \in D}} \frac{x - x_j}{x_i - x_j} \mod p$$
$$M = \{P(0), P(-1), \ldots, P(-|M| + 1)\}$$

Correctness: To argue correctness, the polynomial given from the interpolation of t_r data points must be unique. This proof is given by Lagrange Interpolation which always produces a unique polynomial that interpolates the n data points (Shamir, 1979).

Privacy: Given just one too few shares and any possible message, there exists a unique polynomial that interpolates the associated points. This means that the adversary's best and only approach is to try and guess a remaining share.

For any message m that is encoded into the below share:

$$S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} = genShares(m, r)$$

For any set $I \subset S$ with too few shares to reconstruct the polynomial, $|I| = t$, any two possible guesses of the message $\forall x, x'$ have the same probability of exposing the message.

$$D = \{(0, x)\} \cup I$$
$$D' = \{(0, x')\} \cup I$$
$$P(x) = \sum_{(x_i, y_i) \in D} y_i \prod_{\substack{x_i \neq x_j \\ (x_j, y_j) \in D}} \frac{x - x_j}{x_i - x_j} \mod p$$
$$P'(x) = \sum_{(x_i, y_i) \in D'} y_i \prod_{\substack{x_i \neq x_j \\ (x_j, y_j) \in D'}} \frac{x - x_j}{x_i - x_j} \mod p$$

Additive Homomorphism: The scheme specific operation that is performed in Shamir is adding the the corresponding shares and using modulo the prime.

$$\forall m, m' \text{ and } r, r' \leftarrow Z_p$$
$$S = \{(1, P(1)), (2, P(2)), \ldots, (n, P(n))\} \leftarrow genShares(m, r)$$
$$S' = \{(1, P'(1)), (2, P'(2)), \ldots, (n, P'(n))\} \leftarrow genShares(m', r')$$
$$S^{\diamond} = \{(1, P(1) + P'(1)), (2, P(2) + P'(2)), \ldots, (n, P(n) + P'(n))\}$$

To prove $rec(S^{\diamond}) \rightarrow m + m'$ inspect the $rec(S)$ function from section 3.3:

$$P^{\diamond}(x) = \sum_{(x_i, (y_i + y_i')) \in S^{\diamond}} (y_i + y_i') \prod_{\substack{x_i \neq x_j, \\ (x_j, (y_j + y_j')) \in S^{\diamond}}} \frac{x - x_j}{x_i - x_j} \mod p$$
$$= \sum_{(x_i, y_i) \in S} y_i \prod_{\substack{x_i \neq x_j, \\ (x_j, y_j) \in S}} \frac{x - x_j}{x_i - x_j} \mod p + \sum_{(x_i', y_i') \in S} y_i' \prod_{\substack{x_i \neq x_j, \\ (x_j', y_j') \in S}} \frac{x - x_j'}{x_i' - x_j'} \mod p$$
$$= P(x) + P'(x)$$

Thus $P^{\diamond}(0) = P(0) + P'(0) = m + m'$

By inspecting the reconstruction function when applied to the combined set of shares $S^{\diamond}$, notice that the summation and product can be separated into two different summations and products which correspond to the definition of $res(S)$ and $rec(S')$. For this reason it is possible to reconstruct the summation of messages.

Error detection

To detect errors in Shamir secret sharing, generate the polynomial and check that all points are on the polynomial. i.e. $\forall(x_i, y_i) : P(x_i) = y_i$, otherwise an error has occurred. Note that more than t_r shares are needed otherwise you would simply generate an incorrect polynomial in which all used points are on. Use t_r shares to generate P, use the additional shares to detect errors.

One disadvantage is prevalent in this error detection; You can detect errors but not locate them. There is no way of knowing if the polynomial was generated with one or more corrupt shares or, if the additional shares, which we use for verification, are corrupted. This means we can not pinpoint which share is causing the error. This problem will fortunately be covered by error correction.
Additionally, there are three cases to consider when shareholders run error detection.

1. There are at least t_r honest shareholders and thus their honest shares uniquely defines the honest polynomial and the message encoded into it. This means that no matter how many corrupted shareholder there are they can not convince the honest shareholders of anything, because the honest polynomial is uniquely determined by known shares.

2. If there are fewer than t_r honest shareholder and more than t_r corrupted shareholders, the corrupted shareholders can interpolate their shares, and learn the polynomial. This subsequently means that they are able to learn the honest shareholders share and allows them to define a new polynomial that interpolates the honest shares but not the message. As a result, allowing them to pick their shares on the new polynomial. This would make it impossible for the honest shareholders to detect any errors.

3. If there are fewer than t_r corrupted shareholders and fewer than t_r honest shareholders, but they together are more than t_r. The corrupted shareholders can choose to delay publicizing their shares, until the honest shareholders have made their shares public. This allows the corrupted shareholders to falsify their shares in such a way that the can learn the honest polynomial and thus falsify their shares in an undetectable manner.

Based on the above cases if either case 2 or 3 applies, there is no reliable way to detect any errors. An adversary can wait to see all the honest shares and use this information to falsify his own share, and thus try to disrupt the outcome of the reconstruction. One approach to patch this flaw is to use a commitment scheme that ensures shareholders can not change their published share once they have shared it. If a commitment scheme is implemented it would make it impossible for the third case to occur. We will not be looking further into commitment schemes.

Error correction: Berlekamp-Welch

We have seen how given just enough shares the original message can be reconstructed, however only if there are no dishonest shares or errors. By sending more shares than needed for the reconstruction, we can run the Berlekamp-Welch algorithm (Welch and

Berlekamp, 1986). By having more shares we can allow for some errors to be present in the data, while still being able to correctly identify the errors and reconstruct the original message. The algorithm operates on polynomial representations. By having defined an error locator polynomial the algorithm can identify and discard corrupted shares, while preserving the integrity of the rest of the shares.

Consider a scenario where a message is encoded into a degree t polynomial and n shares is sent to a number of shareholder, but up to ε shareholders are malicious and thus they can supply incorrect or no information when the message needs to be reconstructed. To ensure the correct message can be recovered, n has to be at least $t_r + 2\varepsilon$. When at least $t_r + 2\varepsilon$ shareholders pool their shares of the secret, where up to ε shares could have been tampered, the Berlekamp-Welch algorithm runs on the collection of shares. The incorrect shares are located and thus not used when the reconstruction algorithm is run.

Feng (2020) provides an excellent explanation of the algorithm that we utilize going forward. Before introducing the algorithm we will define the polynomial and variables used.

Polynomials used in Berlekamp-Welch

The Berlekamp-Welch algorithm relies on two polynomials. The error-polynomial $E(x)$ has the property that $E(x_i) = 0$ if the i'th data point is corrupted and $E(x_i) \neq 0$ if the i'th data point is not corrupted. $Q(x)$ is a polynomial which combines the received shares and the error polynomial. Given that $Q(x)$ is defined to be the product of $P(x)$ and $E(x)$, it has the property that $Q(x_i)$ will evaluate to zero if the i'th x coordinate of the share is an error. Otherwise the product will be some value that is not zero, unless the share itself is zero.

$$E(x) = (x - e_1)(x - e_2)\dots(x - e_\varepsilon)$$
$$Q(x) = P(x)E(x)$$

When the shareholders pool their shares, they can not distinguish between legitimate and corrupted shares, so they can not compute $E(x)$ by the definition above. Instead $E(x)$ and $Q(x)$ can be written as below where e_i and q_i are some unknown coefficient.

$$E(x) = x^\varepsilon + e_{\varepsilon-1}x^{\varepsilon-1} + \dots + e_1 x + e_0$$
$$Q(x) = q_{n+\varepsilon-1}x^{t_r+\varepsilon-1} + q_{t_r+\varepsilon-2}x^{t_r+\varepsilon-2} + \dots + q_1 x + q_0$$

We now know that there are exactly $t_r + 2\varepsilon$ unknown coefficients, and $t_r + 2\varepsilon$ shares. This supplies just enough information to determine each coefficient, by solving the following equation system.

Consider the set of collected shares $S = \{(x_1, y_1), \ldots, (x_{t_r+2\varepsilon}, y_{t_r+2\varepsilon})\}$

$$\begin{aligned} Q(x_1) &= y_1 E(x_1) \mod p \\ Q(x_2) &= y_2 E(x_2) \mod p \\ &\vdots \\ Q(x_{t_r+2\varepsilon}) &= y_{t_r+2\varepsilon} E(x_{t_r+2\varepsilon}) \mod p \end{aligned}$$

This can be expressed in a more exhaustive manner:

$$\begin{aligned} Q(x_1) = y_1 E(x_1) &\equiv q_{t_r+\varepsilon-1} x_1^{t_r+\varepsilon-1} + q_{t_r+\varepsilon-2} x_1^{t_r+\varepsilon-2} + \cdots + q_1 x_1 + q_0 \\ &= y_1 (x_1^{\varepsilon} + e_{\varepsilon-1} x_1^{\varepsilon-1} + \cdots + e_1 x_1 + e_0) \mod p \\ Q(x_2) = y_2 E(x_2) &\equiv q_{t_r+\varepsilon-1} x_2^{t_r+\varepsilon-1} + q_{t_r+\varepsilon-2} x_2^{t_r+\varepsilon-2} + \cdots + q_1 x_2 + q_0 \\ &= y_2 (x_1^{\varepsilon} + e_{\varepsilon-1} x^{\varepsilon-1} + \cdots + x_2 e_1 + e_0) \mod p \\ &\vdots \\ Q(x_{t_r+2\varepsilon}) = y_{t_r+2\varepsilon} E(x_{t_r+2\varepsilon}) &\equiv q_{t_r+\varepsilon-1} x_{t_r+2\varepsilon}^{t_r+\varepsilon-1} + q_{t_r+\varepsilon-2} x_{t_r+2\varepsilon}^{t_r+\varepsilon-2} + \cdots + q_1 x_{t_r+2\varepsilon} + q_0 \\ &= y_{t_r+2\varepsilon} (x_{t_r+2\varepsilon}^{\varepsilon} + e_{\varepsilon-1} x_{t_r+2\varepsilon}^{\varepsilon-1} + \cdots + e_1 x_{t_r+2\varepsilon} + e_0) \mod p \end{aligned}$$

The equations system outlined above is solved by Gaussian Elimination or similar methods to obtain a value for each coefficient. With these values the error polynomial can be defined and evaluated at each x_i of the shares. If $E(x_i) = 0$ an error is located and the share is discarded before running the reconstruction algorithm. The ability to detect and correct errors makes the Berlekamp-Welch algorithm a beneficial addition to Shamir secret sharing.

Shamir Secret Sharing with Berlekamp-Welch Error Correction

1. The message m is encoded into a t-degree polynomial.
2. Generate n shares using $genShares(m, r)$, where $n \geq t_r + 2\varepsilon$, and distribute the shares.
3. Pool at least $t_r + 2\varepsilon$ shares and run the Berlekamp-Welch algorithm. The algorithm outputs the set of legitimate shares S.
4. Run $rec(S)$ to recover the message m.

4 Electronic Voting with Secret Sharing

We have seen how to encode any message and distribute the shares of the encoded message to shareholders. This is however not enough if one wants to use any of the schemes for electronic voting. The protocol of the introduced schemes have to be adapted to handle more than a single message or vote. This can however easily be done as the additive homomorphic property has been proved to hold for all the secret sharing schemes throughout this report. The schemes still have the same properties and limitation as we have seen in their respective section.
In order to adapt the schemes to electronic voting we will introduce what we call intermediate results: Given the shares of multiple encoded messages the i'th intermediate result is defined as the result of the scheme specific operation on all the i'th shares. Henceforth denoted as res_i.

Implementation clarification

Our python implementations of the schemes do not communicate shares between client and servers in a network setting. We decided to focus on implementing the scheme functionalities without diving into the complexity of network communication. Instead we simply store shares locally using arrays and lists. We chose to simulate a client-server over a party setting. Our choice to simulate the clients and server would of course not work in a real life setting, but as a proof of concept it works just fine. Our implementation could be adapted to work in a real electronic voting setting by adding this network layer instead of simulating it.

4.1 Multiple candidates within a single message

There is a way where only one message is encoded into shares where we still have the ability to vote for multiple candidates. To show the fix we will use a small example, consider the following:

Nine clients votes for one of three candidates with the following representations:

$$|C| = 9$$
$$Cand_1 = 0$$
$$Cand_2 = 1$$
$$Cand_3 = 10$$

A client chooses their input corresponding to the numeric representation. Note that the representation of the third candidate is larger then the number of voting clients. Consider the following chosen inputs.

$$m_1 = 0 \;\; m_2 = 1 \;\; m_3 = 10 \;\; m_4 = 10 \;\; m_5 = 1$$
$$m_6 = 0 \;\; m_7 = 1 \;\; m_8 = 10 \;\; m_9 = 10$$

The output of the vote would be the sum of the inputs: 43. This would mean that the third candidate got four votes, the second candidate got three votes and the first candidate got the remaining $9-4-3=2$ votes.

By choosing the representation of the third candidate to be larger then number of voting clients we ensure that there is no way the candidates can be mixed up. If a forth candidate were to be added, its representation would have to be larger than the number of voting clients times the representation of the third candidate. In general the number of votes for each candidate can be computed as the following.

$$\begin{aligned}
Cand_{n_{votes}} &= \left\lfloor \frac{result}{Cand_n} \right\rfloor \\
rem_n &= result \mod Cand_n \\
Cand_{n-1_{votes}} &= \left\lfloor \frac{rem_n}{Cand_{n-1}} \right\rfloor \\
rem_{n-1} &= rem_n \mod Cand_{n-1} \\
&\vdots \\
Cand_{2_{votes}} &= \left\lfloor \frac{rem_3}{Cand_2} \right\rfloor \\
Cand_{1_{votes}} &= |C| - Cand_{n_{votes}} - \ldots - Cand_{2_{votes}}
\end{aligned}$$

The ability to encode multiple candidates into one message is certainly possible but not very efficient. The size of the messages grow exponentially with the amount of candidates. Even though this is possible it is not something we will use or look further into.

4.2 General Electronic Voting using Secret Sharing

Before using secret sharing to vote electronically, there are a number of subjects the parties involved must agree on and be aware of. From a client's perspective they must know how many shares to generate and who they need to send their shares to. They

do of course also need to agree with the servers on what scheme to use as well as the public parameters. i.e. what is the prime and the thresholds. Before the servers can do anything the also have to know some information. Either the servers need to know how many clients there are, if not they would not know when to stop waiting for incoming shares, or they stop waiting for shares once a deadline has been met. Furthermore both the clients and servers needs to have a way of privately sharing information such that no other party can eavesdrop. When this has been agreed upon can the protocol begin.

General Protocol

1. $|C|$ Clients $c_1, c_2, \ldots, c_{|C|}$ chooses their input $m_i \in \{0,1\}$.
2. Each client runs $genShares(m_i, r_i)$ for their respective message.
3. Each client privately distributes their shares to the servers.
4. Each server computes the intermediate result res_i using the scheme specific operation $\diamond$ on the received shares.
5. The servers pool their intermediate results into set $S = \{res_1, res_2, \ldots, res_n\}$ and runs $rec(S)$ to compute the result.

4.3 Properties of Additive Secret Sharing

Additive secret sharing is a great way to introduce the secret sharing to someone who has never seen the concept. Even though the scheme is a completely valid scheme for secret sharing and for electronic voting, it is not very practical to use in a large setting, because of its inability to identify and handle errors. As a result, the scheme is not great for large scale elections but can have applications in smaller settings. Suppose that a small group of people wants to calculate their average salary. The parties involved would simply run the protocol and use their salary as the input and divide the result with the number of participants. One subtle fault is present in this scheme: What if someone chooses a false input? There is no fix for this using only secret sharing, it can however be fixed with the addition of encryption and *zero-knowledge proofs*. Encryption will be covered later in this report. If a malicious party is involved, the scheme might not produce an output that is correct (Cramer et al., 2015). Because of the low flexibility of the scheme, it is best used as an example or if the nature of the vote or computation is not critical to keep honest.

4.4 Properties of Replicated Secret Sharing

As described in section 3.2, the Replicated secret sharing scheme uses the same principle as Additive but with a different reconstruction threshold. This makes it more applicable in practice as all servers are not needed to compute the result of the vote. Due to this threshold, tolerance is added for missing servers and the ability to detect and correct errors under the right circumstances. These properties makes the Replicated scheme a robust and solid choice for electronic voting. However, the amount of data each server must store depends on the reconstruction threshold chosen; if the threshold is $t_r = 2$

each server must store $n-1$ Additive shares which is substantially more compared to other schemes.

Implementation

Our python implementation is divided into two main classes: client and server. The *genShares* algorithm is implemented in the client class, while the *rec* algorithm is implemented in the the server class. The formulas described in the theory section (3.2) are implemented to provide the desired functionality.

We also added error detection and correction functionality. Errors are detected by comparing res_i's between servers. If two res_i are not identical, an error is detected, meaning that either the res_i was corrupted during transmission (which is not the case for our implementation, see Implementation Clarification, section 4) or a malicious server deliberately sent two different versions of res_i. In either case, error correction steps in. Error correction works by collecting the same res_i received from different servers and choosing the res_i with majority.

4.5 Properties of Shamir Secret Sharing

While it is possible to setup the previous schemes with multiple candidates, the Shamir secret sharing scheme does so in a more efficient and elegant way. The size of the messages does not grow as large as shown in section 4.1, however the reconstruction threshold does increase with the amount of candidates. By using packed Shamir it allows for multiple messages to be encoded into the same polynomial as described in section 3.3. This capability coupled with the Berlekamp-Welch algorithm for error correction and even identification of corrupted shares, makes the Shamir secret sharing scheme an excellent choice for electronic voting.

Implementation

Our implementation of Shamir secret sharing primarily consists of the two algorithms defined in section 3.3. They are simply implemented based on the theory and defined protocols in section 3.3. The Berlekamp-Welch algorithm has a major implementation in our code. It takes a list of shares, the maximum number of errors, the degree of the polynomial and the prime p. Much of the functions concerns correctly setting up matrices to solve the equations as described in section 3.3. The function uses Gaussian elimination and reduces to echelon form to obtain the error polynomial $E(x)$. To locate the corrupted shares, all shares are evaluated with the error polynomial and a share is removed if $E(x) = 0$. The remaining share will then be legitimate and are returned.

5 Homomorphic Encryption

An alternative approach to private and secure electronic voting is encryption, where the vote is encrypted using a public key encryption scheme. Such a scheme will need properties much alike secret sharing schemes. We have previously worked with perfect secrecy where no matter how much computing power an adversary has they can not learn anything about the message. This definition does not apply to encryption, instead of the definition of privacy a property called semantic security is defined. This property says that there exists no probabilistic polynomial time adversary that can efficiently break the encryption.
The relevant properties for an encryption scheme are correctness, semantic security and additive homomorphism.

5.1 Notation

The following notations will be used throughout the following sections.

Notation

m : Message	p : Safe prime
CT : Ciphertext	q : Prime based on p: $q = \frac{p-1}{2}$
pk : Public key	g : Generator of order q
sk : Secret key	$\mathbb{Z}_q$: Set of $\{0, 1, \ldots q-1\}$

5.2 Algorithms

A public key encryption scheme has three algorithms: Key generation, encryption and decryption. The key generation algorithm provides the necessary keys and the public parameters for the protocol to work. The encryption algorithm takes a message m and outputs a ciphertext CT. The decryption algorithm takes a ciphertext CT and outputs a decrypted message m'.

Public Key Encryption Algorithms

$$keyGen() \rightarrow pk, sk$$
$$enc(m, pk) \rightarrow CT$$
$$dec(CT, sk) \rightarrow m'$$

5.3 Properties

Correctness

$$\begin{aligned}&\forall m\\&CT = enc(m, pk)\\&dec(CT, sk) = m'\\&\text{s.t. } m = m'\end{aligned}$$

Simply put, the scheme should be able to encrypt a message to a ciphertext and correctly recover the message by decrypting the ciphertext.

Semantic Security

$\forall$ probabilistic polynomial time adversary $\mathcal{A}$

$$\forall m, m' \text{ s.t. } |m| = |m'|$$

The adversary now possesses the two message m, m' and the ciphertext of one of the messages, but the adversary does not know which message the ciphertext corresponds to.

$$\begin{aligned}&pk, sk \leftarrow keyGen()\\&b \leftarrow \{m, m'\}\\&CT = enc(b, pk)\\&b' = \mathcal{A}(m, m', CT, pk)\\&Pr[b' = b] = \frac{1}{2} + \text{negl}\end{aligned}$$

The probability that the adversary can associate the ciphertext to the message is negligible better than guessing.

Additive Homomorphism

$$\begin{aligned}&\forall m_1, m_2, \ldots m_n\\&CT_1, CT_2, \ldots, CT_n = enc(m_1, pk), enc(m_2, pk), \ldots, enc(m_n, pk)\\&CT^{\diamond} = CT_1 \diamond CT_2 \diamond \cdots \diamond CT_n\\&CT^{fresh} = enc(m_1 + m_2 + \cdots + m_n, pk)\\&\forall x\\&Pr[CT^{fresh} = x] = Pr[CT^{\diamond} = x]\end{aligned}$$

where $\diamond$ is a scheme specific operation.

The additive homomorphism property says that the sum of the original messages can be achieved by performing operations on the corresponding ciphertexts and decrypting the combined ciphertext. The combined ciphertext is distributed exactly like a fresh ciphertext and therefore upholds the same properties as a fresh encryption, which is the strong definition of additive homomorphism.

6 Homomorphic Encryption Schemes

Homomorphic Encryption allows for computation to be carried out on data while it is encrypted. It hereby ensures that private information remains private even when the data is being processed. When using homomorphic encryption, data computation can be outsourced without compromising on the security and privacy of the data. More specifically it can be viewed as an extension of public key cryptography, where the computations are done without access to the secret key.

6.1 ElGamal Encryption Scheme

Proposed in 1985 by Taher Elgamal, the public key encryption scheme ElGamal uses the hardness assumption of discrete logarithms: It is believed that there are no polynomial time algorithm to solve the discrete log problem. This assumption provides the security of the scheme. (Elgamal, 1985, p. 2)

Algorithms

The required algorithmic functions for ElGamal are defined below:

$keyGen() \rightarrow sk, pk$

1. Choose large safe prime p and $q = \frac{p-1}{2}$ such that q is also a prime.
2. Define the generator g as a random quadratic residue.
3. Pick $sk \in \{1, 2, \ldots, q-1\}$
4. Compute public key $pk = g^{sk} \mod p$
5. Make p, q, g, pk public.
6. Return sk, pk

The Key Generation algorithm generates the required variables mainly the public and secret key but also the public parameters that are used in the two remaining algorithms.

$enc(m, pk) \rightarrow CT$

Given a message m, encrypt into two ciphertexts ct_1 and ct_2 as follows:

1. Choose random $r \leftarrow Z_q$.
2. Compute ciphertext 1: $ct_1 = g^r \mod p$
3. Compute ciphertext 2: $ct_2 = g^m pk^r \mod p$
4. Return $CT = (ct_1, ct_2)$

The ciphertext in ElGamal consists of two ciphertext ct_1 and ct_2. The first ciphertext is chosen at random while the second is computed with the same randomness and

the public key. This establishes a relation between the keys and the ciphertext which ensures correct decryption.

$dec(CT, sk) \rightarrow m$

Given a ciphertext $CT = (ct_1, ct_2)$, decrypt as follows:

1. Compute $x = ct_1^{sk} \mod p$
2. Calculate $\frac{ct_2}{x} \mod p \equiv g^m \mod p$
3. Recover message by calculating possible values of m and compare to calculated value above: $g^m \in \{1, g\}$

Note that the decrypt algorithm does not immediately recover the message, but g^m. To recover the message it is required to calculate and compare possible values with the recovered value. This means that if the message has many possible values it can take many computations and comparisons to recover the message. With today's computational power however this is mostly insignificant if there are not many possible values to compute.

The Protocol

Given the public parameters pk, g, p.

1. Choose message m.
2. Encrypt the message: $CT = enc(m, pk)$ and send ciphertext to the owner of the secret key.
3. The owner decrypts the message: $m = dec(CT, sk)$

Correctness: The mathematics behind the ElGamal encryption scheme provides the correctness by using the relation between keys and ciphertexts.

Given the public parameters pk, g, p and the secret key sk.

$$CT = (ct_1, ct_2) = enc(m, pk)$$
$$ct_1 = g^r \mod p$$
$$ct_2 = g^m pk^r \mod p$$

Examine the decryption algorithm $dec(CT, sk)$, observe:

$$x = (ct_1)^{sk} \mod p$$
$$\frac{ct_2}{x} \mod p \equiv \frac{g^m pk^r}{(g^r)^{sk}} \mod p \equiv \frac{g^m (g^{sk})^r}{(g^r)^{sk}} \mod p$$
$$\equiv \frac{g^m (g^{sk})^r}{(g^{sk})^r} \mod p \equiv g^m \mod p$$

Semantic Security: We will not be providing a formal proof for semantic security as it is beyond the scope of this report. The security follows from the hardness of *decisional Diffie-Hellman* (Diffie and Hellman, 1976)

Additive Homomorphism: To achieve additive homomorphism a correct operation must be applied to the ciphertexts. For ElGamal this operation is multiplication of the corresponding ciphertexts as shown below:

Given pk, sk from $keyGen()$ and the public parameters g and p.

$$\begin{aligned} &\forall m && \forall m' \\ &CT = enc(m, pk) && CT' = enc(m', pk) \\ &ct_1 = g^r \mod p && ct_1' = g^{r'} \mod p \\ &ct_2 = g^m pk^r \mod p && ct_2' = g^{m'} pk^{r'} \mod p \end{aligned}$$

$$\begin{aligned} ct_1^\diamond &= ct_1 \cdot ct_1' \mod p \\ ct_2^\diamond &= ct_2 \cdot ct_2' \mod p \\ CT^\diamond &= (ct_1^\diamond, ct_2^\diamond) \end{aligned}$$

Observe that the new ciphertext $CT^\diamond$ is identical to a fresh encryption of $m + m'$ with randomness $r + r' \mod q$. In particular, this implies that it decrypts correctly, as we demonstrate below.

Based on the above definition of additive homomorphism, we showcase specifically how the new ciphertext is correctly decrypted:

$$\begin{aligned} x = ct_1^{\diamond sk} \mod p &\equiv (ct_1 \cdot ct_1')^{sk} \mod p \equiv (g^r \cdot g^{r'})^{sk} \mod p \\ \frac{ct_2^\diamond}{x} &\equiv \frac{ct_2 \cdot ct_2'}{(ct_1 \cdot ct_1')^{sk}} \mod p \equiv \frac{g^m pk^r \cdot g^{m'} pk^{r'}}{(g^r \cdot g^{r'})^{sk}} \mod p \\ &\equiv \frac{g^{(m+m')} pk^{(r+r')}}{(g^{r+r'})^{sk}} \mod p \equiv \frac{g^{(m+m')} g^{sk(r+r')}}{(g^{r+r'})^{sk}} \mod p \\ &\equiv \frac{g^{(m+m')} g^{sk(r+r')}}{g^{sk(r+r')}} \mod p \equiv g^{m+m'} \mod p \end{aligned}$$

As mentioned earlier, the decryption does not recover the sum of messages but the sum of messages in the exponent of the generator g.

7 Electronic Voting with Encryption

To vote electronically with an encryption scheme, the vote is encrypted and the ciphertext is used in the computation of the result of the election. The protocol for electronic voting with encryption is outlined below.

7.1 General Electronic voting using Encryption

General Protocol

1. $|C|$ Clients $c_1, c_2, \ldots, c_{|C|}$ chooses their input $m_i \in \{0,1\}$.
2. Each client c_i runs $enc(m_i, pk)$ and gets a ciphertext CT_i.
3. Each client c_i sends their ciphertext to the server(s).
4. Each server computes the final ciphertext $CT^{\diamond}$ using the scheme specific operation $\diamond$ on the received ciphertexts.
5. Each server runs $dec(CT^{\diamond}, sk)$ to compute the result of the vote.

7.2 Properties of ElGamal

Even though the scheme can be used for electronic voting there are still some problems to consider. Primarily, how many servers need to have access to the secret key? Sharing the key with multiple servers subsequently means that there will be many copies of the secret key stored on many servers, increasing the chances of it being leaked. The other possible solution is to share the secret key with only one server, this however results in a single point of failure. If the server is not honest then everything breaks. When a server has access to the secret key, the server can decrypt individual client's ciphertexts and thus learn what a client voted. For these reasons standard ElGamal is not ideal for electronic voting. These problem can however be resolved as we will see in the following section.

7.3 Electronic Voting with ElGamal Encryption with Shamir shared key

In order to avoid revealing the secret key and thus individual votes, the key generation algorithm is used together with Shamir secret sharing.

Setup

1. Run $keyGen() \rightarrow sk, pk$ from section 6.1 to generate the keys.
2. Run $genShares(sk, r) \rightarrow S$ defined in section 3.3 Shamir Secret Sharing.
3. Distribute shares of the key to the servers.

The servers now possess a part of the secret key and not the whole key. It is worth noting that the key distribution must be done by a trusted party since the secret key is

revealed before distributing the shares.

The encryption of client's votes is the same algorithm defined in section 6.1. Since both the ElGamal and Shamir scheme have the additive homomorphic property the alternative scheme described here has the same property.

dec$(CT, sk) \rightarrow m$

1. Each server computes $ct_1^{\diamond} = ct_{1,1}ct_{2,1}\dots ct_{|C|,1}$ and $ct_2^{\diamond} = ct_{1,2}ct_{2,2}\dots ct_{|C|,2}$
2. Each server k_i computes $d_i = ct_1^{\diamond s_i} = (g^r)^{s_i} \mod p$ and distributes d_i
3. Each server computes all Lagrange basis polynomials $\{\Lambda_1, \Lambda_2, \dots, \Lambda_n\}$
4. Each server computes:

$$d = \prod_{i=1}^{t_r} d_i^{\Lambda_i} \equiv \prod_{i=1}^{t_r} (g^{r \cdot s_i})^{\Lambda_i} \equiv \prod_{i=1}^{t_r} (g^{r \cdot f(x_i)})^{\Lambda_i} \equiv \prod_{i=1}^{t_r} g^{r \cdot f(x_i)\Lambda_i}$$
$$\equiv g^{r \cdot \sum_{i=1}^{t_r} f(x_i)\Lambda_i} \equiv g^{r \cdot f(0)} \equiv g^{r \cdot sk} \mod p$$

5. Each server decrypts $ct_2^{\diamond}$:

$$ct_2^{\diamond} \cdot d^{-1} = (g^m pk^r)(g^{r \cdot sk})^{-1} = g^m g^{r \cdot sk} g^{-r \cdot sk} = g^m \mod p$$

6. Recover the message by calculating possible values of m and compare to calculated value above: $g^m \in \{1, g\}$ for one vote. Note that for each extra vote the possible outcome increases by one, i.e. for $|C|$ voters g^m can take any value in $\{g^0, g^1, g^2, g^3, \dots, g^{|C|}\}$

where m is the sum of messages.
Note that all computations in step 5 are modulo p

By supplying the server with shares of the key instead of the whole secret key, we ensure that no individual server has access to the key. Therefore they can not individually decrypt either individual votes or the sum of the votes. In order to decrypt, the servers compute $d_i = ct_1^{\diamond s_i}$ and thus never directly reveal their share of the secret key. By distributing the key with Shamir, the secret key can be reconstructed directly in the exponent during the decryption stage. Because the secret key is distributed using Shamir it is possible to set the reconstruction threshold to any desired value. Given that the shareholders do not directly publish their share of the secret key, it makes it impossible for an adversary to use the transmitted data to reconstruct the secret key efficiently. Furthermore, it makes it possible to run error detection on the secret key. This is achieved by comparing the received share with what the polynomial evaluates to at the given x value, directly in the exponent.

8 Discussion

8.1 Discussion of Secret Sharing schemes

The secret sharing schemes examined in this report have different characteristics, that make them more or less ideal for electronic voting, depending on the setting. As seen in section 4.3 the Additive secret sharing scheme is far from ideal when used in a large scale election due to its inability to tolerate errors. Replicated secret sharing makes up for many of the Additive secret sharing's shortcomings with its ability to both handle errors and missing servers. These properties allows Replicated to be used in a more realistic setting where errors are a possibility. There are however still some shortcoming, the amount of data to send is entirely dependent on the threshold of the scheme. The lower the threshold is, the larger the Replicated share becomes. This unfortunate property is where Shamir excels due to ability to choose a custom threshold while maintaining a constant share size. This provides a higher flexibility compared to the rest of the schemes in this report. This property makes Shamir an ideal choice for electronic voting. Furthermore with the addition of Berlekamp-Welsh error correction, it is possible to independently choose how many corrupted servers the system should be able to tolerate based on the threat model.
While we have assumed that voters are honest and chose their inputs without deviating from the protocol, this is still a complication to consider. If this not the case, a client might be corrupted and thus have the capability of deviating from the protocol and disrupt the outcome of the election by choosing the input to be a large number and not either 0 or 1. As stated earlier there is no fix for this using only secret sharing, for it to be fixed encryption is needed.

8.2 Discussion of Homomorphic Encryption

The alternative approach to electronic voting we presented in this report is homomorphic encryption. Instead of dividing a message into shares, it encrypts the message and produces a ciphertext that can only be decrypted by whoever possesses the secret key. This type of scheme has several advantages, first and foremost there is no need to worry about keeping the ciphertext hidden from others as they can not learn what is encrypted. This also makes it possible for a client to broadcast the ciphertext instead of establishing a private connection to each server.

Encryption schemes provides less security compared to secret sharing schemes, as can be seen in the definition of semantic security in section 6.1 compared to the definition of privacy in secret sharing in section 2.3. The difference is that in semantic security no adversary can efficiently extract information from the ciphertext, while perfect secrecy says that the an inadequate amount of shares reveal absolutely nothing about the secret. These are major downgrades from secret sharing which leads to the question: Why choose encryption over secret sharing? The answer lies in a subject that we have not cover in this report: Zero-knowledge proofs. This protocol makes it possible for a party, the *prover*, to prove they have a piece of information to another party, the *verifier*, without the prover revealing the information to the verifier. In the context of electronic voting, this would provide the ability to prove that a voter is honest without revealing

what they voted. This is a major complication that secret sharing suffers from that encryption can handle.

As we have seen, the distribution of the secret key can be problematic. For electronic voting purposes standard ElGamal encryption is not sufficient for secure voting. One way to obtain a suitable system for electronic voting purposes, is to combine encryption with secret sharing to ensure the entire secret key is not stored on any system.

8.3 Navigating the schemes

In the above discussion we have compared the schemes and their properties. It can be difficult to keep track of all of these concepts, and as mentioned earlier, it essential to choose the right scheme for a given situation, as some polls are more sensitive than others. The tables below gives a comprehensive overview and opportunity to compare the schemes and make the right choice. The first table concerns the three secret sharing schemes and the relevant properties.
Below m is defined to be the size of the modulus.

Secret Sharing Comparison Table

	Additive	Replicated	Shamir
Servers needed for rec(S)	n	2	t_r
Missing server tolerance	0	$n-2$	$n-t_r$
Max corrupted servers: error detection	0	$n-2$	$n-t_r$
Max corrupted servers: error correct	0	$\lfloor \frac{n-2}{2} \rfloor$	$\lfloor \frac{n-t_r}{2} \rfloor$
Server needed memory	m	$m \cdot (n-1)$	m

In order to compare secret sharing with encryption we provide the below table that highlights additional properties to consider when choosing a scheme. When using encryption there is no need to set up private peer to peer (P2P) channels and instead broadcasting can be used, thus lowering the complexity of the communication channels. Note that the values provided below represent the highest possible data transmissions.

Electronic Voting Scheme Comparison Table		
	Secret sharing	Encryption with Secret shared key
Data received (Server)	$m \cdot (n-1)$ bits	$\|CT\|$ bits
Data sent P2P (Client)	$m \cdot (n-1) \cdot n$ bits	0 bits
Data sent P2P (Server)	0 bits	0 bits
Data sent Broadcast (Client)	0 bits	$\|CT\|$ bits
Data sent Broadcast (Server)	m bits	m bits
Requires private transmission	Yes	No
Requires setup by trusted party	No	Yes
Zero-knowledge proof compatible	No	Yes

8.4 Future Work

While our project has provided a detailed analysis of both secret sharing and encryption schemes that can be used for electronic voting, there are several direction for future work that could enhance the viability of these systems.

Zero-knowledge proofs: The addition of zero-knowledge proofs would provide verification of individual votes while they remain private. This ensures that no voting client can disrupt the outcome of the election by choosing a dishonest input. Zero-knowledge proofs has profound implications for electronic voting and warrants further study.

Secret sharing with encryption: Another combination of secret sharing and encryption is where the shares are encrypted before sharing them. This makes it possible for a server to prove the integrity of the share using a zero-knowledge proof. Additionally, it allows for broadcasting of shares instead of relying on private channels. Such a scheme merits further study due to its ability to leverage the strengths of both secret sharing and encryption, while remaining compatible with zero-knowledge proofs. Furthermore it would be worth while to study how a client could prove the integrity of their vote.

Error correction on d_i: In section 7.3 we briefly mentioned the possibility to perform error detection on d_i's shared between servers. Another valuable property would be the addition of error correction with the Berlekamp-Welch algorithm. To determine whether this is a possibility the future work would involve the study of solving equation systems in the exponent.

Acknowledgments

We thank our advisor, Assistant Professor Sophia Yakaubov, who has always provided constant support and mentorship throughout this project. Her ability to explain complex concepts on a Friday to two puzzled bachelor students has played a valuable role, and we are genuinely grateful for her guidance and thorough feedback.

References

Cramer, R., Damgård, I., and Nielsen, J. B. (2015). *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press.

Dahl, M. (2017). Secret sharing, part 1: Distributing trust and work. https://mortendahl.github.io/2017/06/04/secret-sharing-part1/.

Diffie, W. and Hellman, M. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654.

Elgamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472.

Feng, G. (2020). The berlekamp-welch algorithm: A guide. https://gfeng2001.github.io/assets/pdfs/cs70/BWGuide.pdf.

Shamir, A. (1979). How to share a secret. *Commun. ACM*, 22(11):612–613.

Welch, L. R. and Berlekamp, E. R. (1986). Error correction for algebraic block codes.

A Implementation code

Our code for the implemented schemes is available here:
https://gitlab.au.dk/au702308/bachelor-project

Additionally, we made a clone of the project and put it on our personal Github page for the sake of preservation.
https://github.com/MikkelKatholm/Bachelor_Project_From_Ballots_to_Bytes