Electronic Voting using Homomorphic Encryption

Morten Erfurt Hansen
201608514

Johannes Ernstsen
201605841

Mathias Søby Jensen
201608824

June 17, 2019

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Johannes Ernstsen 201605841
Morten Erfurt Hansen 201608514
Mathias Søby Jensen 201608824

Abstract

In this paper we present an implementation of a secure electronic voting system, using homomorphic ElGamal encryption and proofs of knowledge.
This system is secret-ballot and guarantees correctness, and voter anonymity. However, the system only supports boolean type votes, enabling usage in referendums, but prohibiting usage in multi candidate elections. The correctness property is achieved using zero-knowledge proofs for all votes and partial decryptions. Partial decryptions are created by a number of decryption authorities, with each holding a share of the secret key.
The system was implemented through three iterations. The first iteration built an initial system where all parts had complete trust. The second iteration removed trust from the voters by introducing zero-knowledge proofs to ensure that each vote is legal. The third iteration removed the trust in the decryption party, introducing decryption authorities, each partially decrypting the result, and making public their partial results and a proof of decryption.

Acknowledgments

A special thanks to our advisor Peter Scholl, who has been a tremendous help throughout the entire project, and also to Ivan Damgård for guidance in choosing the project direction.

Contents

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Johannes Ernstsen 201605841
Morten Erfurt Hansen 201608514
Mathias Søby Jensen 201608824

1 Introduction

For an electronic voting system to be usable it needs to have all the most important properties of an analogue one. Voters needs to remain anonymous, votes needs to be counted, and the correct winner must be announced in the end. The biggest challenge when it comes to electronic voting is keeping the system safe from attempts to tamper with the results and rig the election.
The first part of the report will introduce theory used in creating a secure cryptosystem using homomorphic encryption and zero-knowledge proofs. After the theory has been introduced, the three iterations of creating the system will be described. The purpose of the first iteration was to create the initial system, and the other iterations were to increase the security of the system, by removing trust.

2 Preliminary

2.1 Public Key Cryptography

Public key cryptography, also known as asymmetric cryptography, is a type of cryptographic scheme, in which both a public and a secret key are generated and used.
The public key, as the name suggests, is available to everyone, and does not need to be kept a secret. It is used for encrypting messages, which should only be readable by those who hold the secret key. The secret key can be used to decrypt such messages and therefore should be kept as secret as possible, often only being known by one entity of a system.
In order to do this, there has to be some sort of relation between the two keys. For the system to be considered secure, it has to be infeasible to find this relation.
If it is feasible to identify this relation, an adversary could simply derive the secret key from the public key, rendering the scheme insecure.
For a cryptographic scheme to be classified as asymmetric, it needs to contain at least the following three basic algorithms; a key generation algorithm, an encryption algorithm and a decryption algorithm.

Key Generation A key generation algorithm generates a pair of keys; a public key and a secret key, denoted (pk, sk). These keys needs some sort of relation, as mentioned above.

Encryption An encryption algorithm generates a ciphertext c given a public key pk and a plaintext m. Usually the encryption algorithm uses randomization, so that the same (pk, m) pair will rarely generate the same c. This prevents an adversary from detecting if the same plaintext m is sent multiple times.

Decryption A decryption algorithm uses a secret key sk, to recover the original plaintext m from a given ciphertext c, encrypted under the corresponding public key pk. [DNO18, p. 98-99]

2.1.1 Homomorphic Cryptographic Schemes

The mathematical term homomorphism is a structure preserving mapping, from one algebraic structure, into another. This property is useful in cryptography, and is used in creating homomorphic encryption schemes.

These schemes enable computations on data, where only their encrypted values are known.
I.e. Let m be our message, and c the ciphertext. A relation between m and c needs to exist; m has to be a group under the $\oplus$-operation and c has to be a group under the $\otimes$-operation.
If the above is satisfied, there exists homomorphic encryption schemes, where an encryption E outputted by this, with ciphertexts $c_1 = E_{pk}(m_1)$ and $c_2 = E_{pk}(m_2)$, has the property that $c_1 \otimes c_2 = E_{pk}(m_1 \oplus m_2)$.

Homomorphic encryption schemes can be used in the construction of election protocols. E.g. being able to add all votes together to obtain the result of an election, without having to decrypt a single vote, thus preserving confidentiality of the vote. [CGS97, p. 7]

2.2 Groups

A pair $(G, \circ)$ where G is a set and $\circ$ is a composition, is a group, when each member satisfies the following four properties; *closure, associativity, identity* and *inverse* [Lau09, p. 51].

Closure if $A, B \in G$ then $A \circ B \in G$

Associativity if $A, B, C \in G$ then $A \circ (B \circ C) \in G$

Identity if $A, I \in G$ then $A \circ I = I \circ A = A$, where I is the identity element

Inverse if $\forall A \in G : \exists A^{-1}$ then $A \circ A^{-1} = I$

2.2.1 Cyclic Groups

A cyclic group G is a group containing an element g, called the generator of the group. Every element in G is generated by applying the groups composition on g an arbitrary amount of times, such that

$$G_q = \langle g \rangle = \{g^i \mid 0 \leq i \leq q-1\}$$

where q is the order of the generator g. [Lau09, p. 72-74]

2.2.2 Subgroups

A non-empty $H \subseteq G$ is a subgroup of the group G, if H is a group with the same composition as G [Lau09, p. 61].

2.2.3 Finding Groups $\mathbb{Z}_p^*$ and G_q

To Find the groups $\mathbb{Z}_p^*$ and G_q, the following steps need to be performed:

1. Primes p and q such that $p = 2q + 1$ need to be found
2. Pick random elements $\alpha \in \mathbb{Z}_p^*$ until $\alpha^{\frac{p-1}{q}} \neq 1$ and $\alpha^{\frac{p-1}{2}} \neq 1$
3. α is a generator for $\mathbb{Z}_p^*$ [Dam17, p. 7]
4. α^2 is a generator for G_q [Dam17, p. 9]

2.3 Security Strength

The security strength for cryptographic protocols often rely on the difficulty of solving specific mathematical problems. This is for instance the case for the El-Gamal cryptographic scheme and proofs of knowledge using Σ-protocols. Both of these examples rely on the discrete logarithms problem.

2.3.1 Discrete Logarithms Problem

Computing the discrete logarithm for a value is believed to be hard for certain groups, as for instance $\mathbb{Z}_p^*$ when p is a random prime. If $p-1$ consists of only small prime factors, it is possible to compute the discrete logarithm with the Polig-Hellman algorithm.

Formally, the discrete logarithms problem is the problem of finding an integer a, such that

$$\alpha^a = \beta$$

when given a group G, a generator α and $\beta \in G$. [Dam17, p. 2-3]

2.3.2 Decisional Diffie-Hellman Problem

The decisional Diffie-Hellman problem can be reduced to the discrete logarithms problem, as it makes stronger assumptions than the discrete logarithms problem. In this problem, three discrete logarithms are given; α^a, α^b, α^c, and the problem is to decide if c is random or if $c = ab$.

Formally, the decisional Diffie-Hellman problem is the problem of deciding if

$$c = ab$$

or

$$c \text{ is uniformly randomly chosen from } \mathbb{Z}_t$$

when given a group G, a generator α and α^a, α^b, α^c, where a and b are uniformly randomly and independently chosen from $\mathbb{Z}_t$. [Dam17, p. 3]

2.4 El-Gamal

El-Gamal is a public key cryptographic scheme. It is based on the Diffie-Hellman key exchange, which is a method for securely exchanging cryptographic keys via a public channel [Dam17, p. 4-5].
For the rest of this report, El-Gamal will refer to a modified version of the original, which uses homomorphic encryption and decryption algorithms.

An El-Gamal ciphertext is given by $(c, d) = (g^r, mh^r)$, where m is the message to encrypt, g is a generator for the cyclic group G_q, $h = g^{sk}$ where sk is the secret key, and r is a random element in $\mathbb{Z}_q$. The message m is chosen as g^b, where $b \in \mathbb{Z}_q$ and is the actual secret value to encrypt. This makes the cryptographic scheme homomorphic, as given two ciphertexts $(c_1, d_1) = (g^{r_1}, g^{b_1}h^{r_1})$ and $(c_2, d_2) = (g^{r_2}, g^{b_2}h^{r_2})$, componentwise multiplication of these results in the following:

$$(c_1c_2, d_1d_2) = (g^{r_1}g^{r_2}, g^{b_1}h^{r_1}g^{b_2}h^{r_2}) = (g^{r_1+r_2}, g^{b_1+b_2}h^{r_1+r_2})$$

As $m = g^b$, the message space is not G_q with multiplication modulo p as composition, but is instead $\mathbb{Z}_q$ with addition modulo q as composition. The ciphertext space is $G_q \times G_q$ with multiplication modulo p as composition for each component. [CGS97, p. 4-5]

The scheme consists of four components; a key generator, an encryption algorithm, a decryption algorithm and a homomorphic addition algorithm.

Key Generation The key generation algorithm for El-Gamal generates a secret key sk and a public key pk. The secret key sk is a random number in $\mathbb{Z}_q$, where q is a prime number, and it is the case that $2q + 1$ is also prime. The public key pk is a 3-tuple $pk = (G_q, g, h)$, where g is a generator for the cyclic group G_q, and $h = g^{sk}$. Finding the group and its generator is based on group theory, which is explained in section 2.2.3.

Encryption The encryption algorithm generates a ciphertext (c, d) given a public key pk and a number to encrypt, b. From this we define $m = g^b$ where g is the generator from pk. To avoid having multiple encryptions of the same value be the same ciphertext, a random number $r \in \mathbb{Z}_q$ is chosen and used in creating the ciphertext. The formula for the encryption is as follows:

$$E_{pk}(b, r) = (c, d) = (g^r, mh^r) = (g^r, g^bh^r)$$

Decryption The decryption algorithm uses a secret key sk, to recover the original plaintext m from a given ciphertext (c, d), encrypted under the corresponding public key pk.
To decrypt we need to compute

$$D_{sk}(c, d) = (c^{sk})^{-1}d = c^{-sk}d = m$$

where $^{-1}$ is the modular multiplicative inverse.
The message $m = g^b$ and the discrete logarithm is believed to be infeasible to compute for large values of p and q. Thus the correct value of b needs to be obtained by comparing $g^b = g^i$ for $i = 0, \ldots, max$, where max is the maximum value encrypted, bounded by $q - 1$. As a consequence of this, it is only efficient for smaller values of b. When there is a match, the current value of i is the same value as b.

Homomorphic Addition The homomorphic addition algorithm generates a ciphertext containing the sum of two plaintexts, given their ciphertexts, by multiplying the two ciphertexts, which gives us:

$$E_{pk}(b, r) \cdot E_{pk}(b', r') = E_{pk}(b + b' \bmod q, r + r' \bmod q)$$

[Dam06, p. 2]

Security The security of El-Gamal is dependent on which groups are used [Dam17, p. 3].
The cyclic groups $\mathbb{Z}_p^*$ and G_q subgroup of $\mathbb{Z}_p^*$, where $p = 2q + 1$ and both p and q are large primes, i.e. p is a safe prime and at least 2000 bits, are useful groups for El-Gamal. [Dam17, p. 8 & 10].
If these cyclic groups are used, then decrypting an El-Gamal ciphertext without the secret key is a hard as solving the discrete logorithm problem, and El-Gamal is CPA (Chosen-plaintext attack) secure under the assumption that the Decisional Diffie-Hellman Problem is hard. [Dam17, p. 2-3, 6 & 8].

2.5 Proof of Knowledge

Proofs of knowledge are proofs, which enables a party (the prover) to prove knowledge of a value satisfying a certain relation, without revealing the particular value, to one or more other parties (verifiers).

2.5.1 Σ-Protocols

A Σ-protocol is a special case of proof of knowledge. A protocol $\mathcal{P}$ is a Σ-protocol for a relation R if the following criteria are met

- $\mathcal{P}$ has completeness i.e. if a prover and verifier P, V follow the protocol on input x and private input w to P where $(x, w) \in R$, V always accepts
- $\mathcal{P}$ is on the form:
  1. P sends a message a
  2. V responds with a random t-bit string e
  3. P replies with z and V decides to either accept or reject based on x, a, e, z
- It must uphold the special soundness property
  i.e. from any x and any pair of accepting conversations on input x $(a, e, z), (a, e', z')$ where $e \neq e'$ one can efficiantly compute w such that $(x, w) \in R$
- There must exist a polynomial-time simulator M, which on input x and a random value e outputs an accepting conversation of the form (a, e, z), with the same probability distribution as the conversations between the honest P, V on input x.

[Dam10, p. 3]

Johannes Ernstsen 201605841
Morten Erfurt Hansen 201608514
Mathias Søby Jensen 201608824

Schnorr Protocol The type of Σ-protocol used for this project is derived from the Shnorr protocol. The Schnorr protocol is used for proving knowledge of some value w satisfying the relation R, meaning that (g, h) is publicly known, and w is only known to the prover. This gives the following relation for R:

$$R = \{(g, h), w \ : \ g, h \in G, \ h = g^w\}$$

The protocol works as follows

1. P chooses r at random in $\mathbb{Z}_q$ and sends $a = g^r \ mod \ p$ to V
2. V responds with a random challenge denoted $e \in \mathbb{Z}_{2^t}$, where $2^t < q$
3. P responds with $z = r + ew \ mod \ q$.
4. V then verifies that $g^z = ah^e \ mod \ p$ and that p, q are prime, and g, h has order q. If all is true, V accepts

A cheating prover, who does not know w has probability $\frac{1}{2^t}$ of guessing it, so with a t value high enough, the probability for a cheating prover to provide a correct proof becomes neglible. [Dam10, p. 1]
A diagram showing the execution of the protocol can be seen below in figure 1.

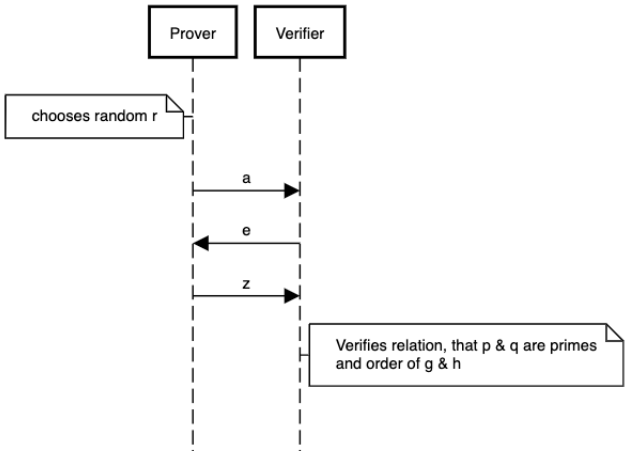

Figure 1: Schnorr Protocol

OR-Proof In some cases it is not enough to prove that one thing is true, but that it is one of two things. This is the case for this voting system as the vote should either be 0 or 1. For this we use an or-proof, or a disjunctive proof, which proves that $(x_0, w) \in R \ \vee \ (x_1, w) \in R$ without revealing which one is the case.
To do this we assume that we are given a Σ-protocol $\mathcal{P}$ for R and that P, V receives common input x_0, x_1, being the two legal and publicly known inputs. We also assume that we have a value w that is private to the prover, P, where $(x_v, w) \in R$, and $v \in \{0, 1\}$.

The protocol works as follows

1. P computes first message a_v using $\mathcal{P}$ with input (x_v, w)
   P chooses a random value for e_{1-v}, and runs the simulator M given with $\mathcal{P}$, on the input (x_{1-v}, e_{1-v}), and gets output $(a_{1-v}, e_{1-v}, z_{1-v})$.
   P sends (a_0, a_1) to V
2. V responds with a randomly chosen t-bit string denoted s
3. P calculates $e_v = s \oplus e_{1-v}$ and computes z_v using $\mathcal{P}$ from the challenge e_v, with input (x_v, a_v, e_v, w). (e_0, z_0, e_1, z_1) is sent to V
4. V verifies that $s = e_0 \oplus e_1$ and that $(a_0, e_0, z_0), (a_1, e_1, z_1)$ are accepting conversations for the protocol $\mathcal{P}$

[Dam10, p. 8-9]

Disjunctive Chaum-Pedersen Proof The Chaum-Pedersen proof is a Σ-protocol for proving the equality between discrete logarithms, i.e. the relation $log_g(x) = log_h(y)$. This protocol gets its properties from the Schnorr protocol, described in section 2.5.1.
In this protocol, the prover is able to prove that he knows a privately known witness; $w \in \mathbb{Z}_q$, which satisfies that $x = g^w$ and $y = h^w$. Here the publicly known values are (x, y, g, h). This gives the following relation for R:

$$R = \{(x, y, g, h), w \ : \ x, y, g, h \in G, \ h = g^w\}$$

OR-proofs, which are also known as disjunctive proofs, can be used together with Chaum-Pedersen proofs, resulting in a Disjunctive Chaum-Pedersen proof (DCP).
In the following protocol for the DCP, we have that $(c, d) = (g^w, g^v h^w)$, w is the witness that P wants to show knowledge of, and v is a vote, that the prover wants to prove is either 0 or 1.

1. P chooses y, z_{1-v}, e_{1-v} at random in Z_q
2. P computes:
   - $a_{1-v} = g^{z_{1-v}} c^{e_{1-v}} \mod p$
   - if $v = 1$ then $b_{1-v} = h^{z_{1-v}} d^{e_{1-v}} \mod p$
   - if $v = 0$ then $b_{1-v} = h^{z_{1-v}} (dg^{-1})^{e_{1-v}} \mod p$
   - $a_v = g^y \mod p$
   - $b_v = h^y \mod p$
3. P sends a_0, b_0, a_1, b_1, c, d to V
4. V responds with a random challenge $S \in Z_q$
5. P computes:
   - $e_v = S - e_{1-v} \mod q$
   - $z_v = y - we_v \mod q$
6. P responds with the proof (e_0, z_0, e_1, z_1)
7. V then verifies the following:
   - $S \stackrel{?}{=} e_0 + e_1 \mod q$
   - $a_0 \stackrel{?}{=} g^{z_0} c^{e_0} \mod p$
   - $b_0 \stackrel{?}{=} h^{z_0} d^{e_0} \mod p$
   - $a_1 \stackrel{?}{=} g^{z_1} c^{e_1} \mod p$
   - $b_1 \stackrel{?}{=} h^{z_1} (dg^{-1})^{e_1} \mod p$

If the above are all true, then V accepts the proof. [CGS97, p. 8]

2.5.2 Fiat-Shamir Heuristic

The Fiat-Shamir heuristic is used to make Σ-protocols non-interactive, by replacing the challenge received from the verifier V, by a hashing of certain values. This means that the prover P does not need to interact with the verifier before sending the proof.
The interactive Σ-protocols introduced, are all honest verifier zero-knowledge proofs, meaning that as long as the verifier is honest, the proofs are zero-knowledge. A proof of knowledge can be zero-knowledge if P can prove that he knows a secret value, with a specific relation, without revealing said value to the verifier. When using Fiat-Shamir in a protocol, it becomes zero-knowledge in the random oracle model. [Dam10, p. 17-19]
It's important when implementing Fiat-Shamir, to use the strong version, where one hashes all important information, like ciphertext, prover identification, and public key. If this is not the case, then it becomes possible for an adversary to cheat the system. I.e. if the ciphertext is not included in the hash, then a ciphertext can be computed from the proof, and is not guaranteed to be either 0 or 1. [BPW12, p. 2] [LPT19b, p. 5-6]

Fiat-Shamir heuristics are used in the Σ-protocols in iteration 2 and 3, by hashing important values that are needed in order to prove knowledge.

2.6 Shamir's Secret Sharing Scheme

Shamir's secret sharing scheme is a (k, n) threshold scheme, meaning that n parties each holds a share of a secret, and any subset of the parties of size $\geq k$ can combine their shares in order to reveal the actual secret. In Shamir's secret sharing scheme, a polynomial $f(x) = a_0 + a_1x_1 + \cdots + a_{k-1}x^{k-1}$ of degree $k-1$ is used. Coefficients of $f(x)$ are chosen uniformly at random from $\mathbb{Z}_q$, where q is a prime number. Each party holds a share of the secret $s_i = f(i) \bmod q$. The secret $s = f(0) = a_0$.
It's possible to construct $f(x)$ from at least k (i, s_i) pairs using Lagrange interpolation. As it's usually only interesting to get $s = f(0)$ and not the entire polynomial $f(x)$, there exists a more efficient implementation of Lagrange interpolation for this case, which is explained in the next section. [Sha79, p. 612–613]

2.6.1 Lagrange Interpolation

Lagrange interpolation makes it possible to construct a polynomial $f(x)$ of degree $k-1$ by having at least k $(x_i, f(x_i) = y_i)$ pairs. Computing the interpolating polynomial is done as follows

$$f(x) = \sum_{i=0}^{k} y_i \lambda_i(x)$$

where

$$\lambda_i(x) = \prod_{\substack{0 \leq j \leq n \\ i \neq j}}^{k} \frac{x - x_j}{x_i - x_j}$$

is called the Lagrange coefficient. [Lev10]

When using Lagrange interpolation in the context of Shamir's secret sharing scheme, computing $f(x)$ is not important. Only computing $s = f(0)$ is. It's therefore more efficient to do it like this

$$s = f(0) = \sum_{i=0}^{k} s_i \prod_{\substack{0 \leq j \leq n \\ i \neq j}}^{k} x_j (x_j - x_i)^{-1} \bmod q$$

where $^{-1}$ is used to emphasize that it's modular multiplicative inverse and not regular division.

2.7 Security Policy

When building a system, one needs to first define the objectives of such a system. This is done by creating what is called a security policy. A security policy can have many forms. It specifies the system as a whole, while describing which parts of the system could be unsafe. It could be modelled as a set of rules saying what each component is allowed to do, and where data flows from and to. The objective is to make sure that one can distinguish events that one does not want to happen in the system, from those that are okay. The objective is also to preserve confidentiality, integrity, and availability of the system.
It is very important, that when creating a security policy, one needs to be as precise and concrete as possible. One can also add, to the security policy, which strategies will be used in order to ensure that the objectives are reached. [DNO18, p. 5-6]

2.8 Threat Model

Attacks can come from many directions, and they can be difficult to anticipate. With the correct tools it is possible to identify and protect against most attacks. One of these tools is the threat model.
A threat model specifies which types of attack the system should protect against, and which the system designer should not concern himself or herself with.
The models used in creating a threat model in this report are STRIDE, EINOO and X.800 which will be descibed in this section.
EINOO is used for describing the effects, or the goals, of a possible attack.

2.8.1 STRIDE

STRIDE is an acronym for Spoofing Identity, Tampering, Repudiation, Information disclosure, Denial of service and Elevation of privelege, which are the subjects to be analyzed when using STRIDE.

Spoofing Identity is the act of assuming an identity that is not your own. In most systems a user will need to identify itself to be able to use the system. Identity spoofing can occur when the user verification is lacking, for some reason.

Tampering is the act of making changes. For tampering to be a threat in a system it needs to be undetected, so the system is unable to defend itself. An example could be transmitting votes through an open channel witout message authentication, letting an attacker intercept the packet and changing the vote, without the server's knowledge.

Repudiation is the act of denying something that is true. It could be whether a user has cast it's vote, or for the server to be able to deny the correct voting result.

Information Disclosure is more commonly known as a leak, and is when an attacker is able to see information that should have been undisclosed.

Denial of Service is the act of denying users access to the system, e.g. keeping others from casting their vote, upload their document, etc.

Elevation of Privilege is when an attacker is able to gain privileges, by convincing the system that it is a user.

2.8.2 EINOO

Besides looking at what effects attacks might have, it is also interesting to look at who might attack the system, and where these attacks might occur. This is what EINOO is used for. Like STRIDE, EINOO, is an acronym whichs stands for External attackers, Insiders, Network attacks, Offline attacks and Online attacks. The first two points are describing who might want to attack, while the rest are concerned with how.

External Attackers are attackers who does not take part in the system.

Internal Attackers are attackers who are already part of the system. It is from this kind of attackers that elevation of privelege attacks occur.

Network Attacks are attacks carried out over the network. These attacks are limited to what is included in the X.800 model, which is described in section 2.8.3. As these attacks occur over the internet it is very hard to avoid them, but encryption and message authentication can neutralize most of the threats

Offline Attacks are attacks where the attacker gains access to some kind of persisted storage. Key here is that the system does not need to be running at the time of the attack. Targets could be secret encryption keys, a password database or similar.

Online Attacks are attacks where the attacker is able to observe the system while it runs. This could be while the system decrypts vital information, where the attacker could read or substitute the information, during execution. These attacks are extremely difficult to carry out.

2.8.3 X.800

As mentioned in the section about EINOO, X.800 is used for analyzing means of attacks, mainly over internet connections. The X.800 standard is created by the CCITT, which is the Consultative Committee for International Telephony and Telegraphy.
The attacks are divided into passive and active attacks.

Passive Attacks are attacks where the attacker does not interfere. X.800 describes two types:

- Eavesdropping, where the attacker listens in and reads information sent. Encryption makes this hard
- Traffic analysis, where the attacker looks at the packages sent and analyzes amount and direction of traffic for information. Encryption is not an easy solution for this, as packages still needs to be sent

Active Attacks are attacks where the attacker interferes. There are three types:

- Replay, where the attacker sends one or more packages again in an attempt to make the receiver think that the sender made multiple requests.
- Blocking, where the attacker stops messages from arriving. This can be random messages or packages, but they can also be strategically chosen e.g. to believe that a specific server has died.
- Modification, where the attacker modifies data sent or received. This is easily solved by message authentication, but can be very harmful when successfully executed.

With these three models, a proper analysis of threats can be made, making it easier to construct a secure system.[DNO18, p. 6 & chp. 14]

3 Iteration 1: A Two Server Solution

In the first iteration, the system is built in the most basic way. It is a two server solution with privacy for voters, but no security against malicious attacks. I.e. there are full trust in voters and both of the two servers.
A vote is either 0 or 1, and can be cast by anyone who is allowed to vote. 0 will represent an "against"-vote, while 1 represents a "for"-vote. Thus by summing all of the votes, its possible to determine if the majority has voted for or against in the following way:

1. If $sum\ of\ votes < \frac{total\ votes}{2}$, then the result is: Against
2. If $sum\ of\ votes = \frac{total\ votes}{2}$, then the result is: Draw
3. If $sum\ of\ votes > \frac{total\ votes}{2}$, then the result is: For

All votes are posted encrypted by the individual voters to a server called the public server. The public server is able to sum all votes without decrypting them, and send the encrypted sum of votes to another server called the key server. The key server is the only one able to decrypt votes, and are responsible for decrypting the sum of votes, and sending the result back to the public server, which can then publish the result.

The system is defined in more detail, in the following security policy and threat model.

3.1 Security Policy

Purpose The purpose of this security policy is to create a set of guidelines for a secure electronic voting system. The policy will outline the general structure of the system, and describe the measures taken to ensure security. The policy will also explicitly list the assumptions which the system's security builds upon.

System Description The system will consist of two servers and n clients, namely the voters. The two servers will have different jobs. One will manage all public information, e.g. who voted, adding up votes and publishing the result of the votes. This server will be referred to as the *public server* from now on.
The other server will hold and use the secret key. It will receive and decrypt the homomorphic sum of votes, and return the result to the public server. This server will be referred to as the *key server*.

Trust For the two server model, we will assume honesty and non-corruption of all voters and both servers, which implies that only third parties can be adversaries. This also means that the servers are assumed to be live and accessible through the entire voting process, and free of byzantine corruption. At most one server can leak information, i.e. an adversary needs access to both servers in order to control the entire voting process.

System Functionality For the system to be able to function, it needs to have a set of features and limitations.
For the public server the following criteria must be upheld

- Must not have access to the secret key
- Must know the identity of each voter, to determine whether the voter has already cast its vote, in order to avoid replay-attacks
  - If votes are received multiple times from the same voter, all but the first is ignored
- Must be able to provide the key server with the encrypted sum of all votes
- Must be able to supply the public key to anyone who requests it

- Data about who cast which vote is not allowed to be kept or revealed to the key server
- The period in which voters can cast their vote must be limited, in order to avoid waiting forever for the result
- Must use TLS for connections with clients

For the key server the following criteria must be upheld

- Must be able to generate a secret and a public key for El-Gamal encryption
- The key server must be able to receive the encrypted sum of all votes given by the public server
- Must be able to decrypt the result r encrypted under it's El-Gamal keypair where $0 \leq r \leq n$, and n is number of voters
- Only communication with the public server is allowed

For all clients the following criteria must be upheld

- Each client must retrieve a public key from the key server before the vote is cast
- The votes must be encrypted by the public key supplied by the key server
- Each client must cast a maximum of one vote
  - When casting the vote, it must be supplied as a two-tuple: $(ID, E_{pk}(vote))$ where ID matches an id known to the public server
- The votes must be $v \in \{0, 1\}$

3.2 Threat Model

Based on the security policy, a threat model was created to identify the threats that the system should be able to handle. To do this, the STRIDE, EINOO and X.800 models were used.

3.2.1 STRIDE

STRIDE is an acronym for Spoofing identity, Tampering, Repudiation, Information Disclosure, Denial of service, Elevation of privilege. We used this model to determine the types and damages an adversary can inflict on our system.

Spoofing of Identity: Spoofing of identity is possible if valid IDs can be guessed, extracted from the public server in an attack.
Votes are sent using SSL connections, making it very unlikely to have IDs extracted other ways than the aforementioned. If an adversary manages to get hands on a valid ID, it will be possible to cast a vote $(ID, E_{pk}(vote))$ using the ID and public key. As it is the case that only the first vote is saved, an adversary has no use of an ID already received by the server.

Tampering: As SSL connections are used for sending votes, the encryptions used by SSL needs to be broken in order to tamper with the votes. We consider this highly unlikely.

Repudiation: Repudiation refers in this case to authenticity of the voting result. As long as a party is only communicating with the public server, the result of the voting will be authentic, as the servers are assumed to be honest and an SSL connection is used. If the result needs to be distributed by a third-party, the voting result needs to be signed using a signature scheme, which is not the case in this implementation.

Information Disclosure: The system does not give any information to a user before the voting process has been terminated. Any information disclosure is a result of either network analysis or an offline or online attack on the public or key servers.

Denial of Service: The system assumes that all voters vote only once. Thus an adversary would by dropping packets only prevent this specific user from voting, but not prevent the voting process from terminating. An adversary could also prompt the public server for the public key a great number of times, thereby prohibiting the public server from answering voters. This would corrupt the voting process as voters would be unable to cast their vote. To increase strength against these attacks, a distributed public server would be needed.

Elevation of Privilege: The system contains only two levels of privilege: Those who can vote but has not done so yet, and those who have already cast their vote. The only elevation of privilege attack possible would be to assume identity of someone who is yet to vote, which means that the threat is equal to the one described in "*Spoofing of Identity*"

3.2.2 EINOO

To supplement the results from the STRIDE analysis EINOO was used.

External: The only external attackers relevant to the system, are those with access to the actual servers, and more importantly those with access to network interference.

Internal: We do not include any internal attackers in the threat model in this iteration, as the system is built on the assumption that all participants are honest.

Network: For analyzing network threats the model specified in the X.800 standard was used:

- Passive: There are two types of passive attacks; eavesdropping and traffic analysis. As the system uses encrypted channels, the only information available through eavesdropping is whether a person has voted. Traffic analysis can be used to determine who votes, but as with eavesdropping no useful information can be extracted as the information is encrypted.

- Active: Active attacks are divided into three categories. Replay, blocking and modification. Replay attacks are attacks where an adversary saves one or more packets sent over a connection and re-sends them multiple times, tricking the server into thinking the request was sent multiple times. This is avoided as the public server is to ignore all votes cast by someone who has already voted, according to the security policy. Also SSL protects against this type of attack.
Blocking attacks are already described in the Denial of Service part of the STRIDE analysis. Modification attacks are also described in STRIDE, in the Tampering section.

Offline Attacks: The likelihood of offline attacks depends on the server-setup which is omitted from the scope of this threat analysis. If carried out on the key server, an adversary can gain access to a memory dump, containing the secret key, thus enabling him to decrypt all votes and thereby destroy anonymity.

Online Attacks: Similar to offline attacks, the probability of online attacks are determined by the environment the system lives in rather than the system itself. An online attack can compromise the integrity of the entire system. If the attack hits the public server it can dump all votes and send some generated by itself to the key server. If the attack hits the key server it can ignore the ciphertexts and output a result defined by the adversary.

3.3 Implementation

3.3.1 Cryptography

To facilitate homomorphic encryption using El-Gamal, methods have been implemented for key generation, encryption, decryption and summing a list of encrypted votes.
Additionally, a method for finding a prime modulus which is a safe prime $p = 2q + 1$, and a method for finding a generator g for the group G_q have been implemented. It has also been made possible to choose whether the El-Gamal implementation should use the generated p and g or if it should use the standard ones chosen by the Internet Engineering Task Force (IETF).
Java's BigInteger implementation has been used to represent numbers for the aforementioned methods, as p must be at least 2048 bits long in order for El-Gamal to be sufficiently secure.

Finding Prime Modulus The method `KeyGenerationParametersImpl#findPrimes`, returns a `PrimePair` object containing primes p and q, where $p = 2q + 1$ is the prime modulus. Both p and q are primes with probability $1 - 2^{-50}$.

Finding Generator Given a `PrimePair`, the method `KeyGenerationParametersImpl#findGeneratorForGq` returns a generator g for the group G_q. This is done according to the steps explained in section 2.2.3.

Key Generation The method `ElGamal#generateKeys` can use the prime modulus and generator found using the aforementioned methods, which is done according to section 2.4 under Key Generation. Alternatively the prime modulus and generator specified by the IETF can be used. The method returns a `KeyPair` object containing a secret key and a public key.

Encryption Encrypting a vote is done using the method `ElGamal#homomorphicEncryption`, which returns a `CipherText` object, containing integers c and d. This is done according to section 2.4 under Encryption.

Decryption Decrypting a ciphertext is done using the method `ElGamal#homomorphicDecryption`, which returns the decrypted plaintext, or an `UnableToDecryptException` if it was not able to decrypt. Decryption is done according to section 2.4 under Decryption.

Homomorphic Addition In order to tally votes without decrypting, the method `ElGamal#homomorphicAddition` is used. The method multiplies two `CipherText` objects; c_1 and c_2, and returns a new `CipherText` object. This is done according to section 2.4 under Homomorphic Addition.

3.3.2 Client and Server

As previously mentioned, in the security policy, the system is made up by a set of voters and two servers, a public server for receiving votes and a key server for decrypting them.
The project contains a single runnable file which can be configured to carry out any of these tasks, by providing it with command line arguments.

Both servers use Jersey, on a jetty server, for hosting REST endpoints, and the clients uses Jersey-client for sending requests to the endpoints.
As the main focus for this project is not server/client structure it will only be described briefly. This is also the reason that these resources were chosen; they are both open-source and relatively easy to configure.

The client needs an ID to be able to vote, for the system to act in according to the security policy. These IDs are generated on the public server, and is then given to the voters through a secure channel out of scope for our system. They can be distributed through mail, e-mail, text messaging or any other separate channel.

API For proper server configuration an API needs to be defined. The API for the key server is shown in Table 1, and the API for the public server is shown in Table 2. While the key server only uses 200 and 500 as return-codes, the public server also uses $204, 403, 404$. 404 is returned when a public key is not present, 403 is returned when a vote has already been registered with given ID, and 204 is returned upon successful vote.

Request Public Key	
Type	GET
Path	/publicKey
Params	None
Resp	Hexadecimal public key
Code	200/500

CalculateResult	
Type	POST
Path	/result
Params	None
Body	Encrypted sum of all votes
Resp	Sum of all votes
Code	200/500

Table 1: Key Server API

Request Public Key	
Type	GET
Path	/publicKey
Params	None
Resp	Hexadecimal public key
Code	200/404/500

Get Result	
Type	GET
Path	/result
Params	None
Resp	result as formatted HTML
Code	200/500

Generate Voters	
Type	GET
Path	/generateVoters
Params	None
Resp	newline seperated ids(HTML)
Code	200/500

Vote	
Type	POST
Path	/vote
Params	None
Body	Tuple: $(ID, E_{pk}(vote))$
Resp	Empty
Code	204/403/500

Table 2: Public Server API

Communication Sequences The flow of the processes are depicted in figures 2 and 3. Figure 2 depicts the communication between voters and the public server, whereas the communication sequence between the public server and the key server is depicted in figure 3.

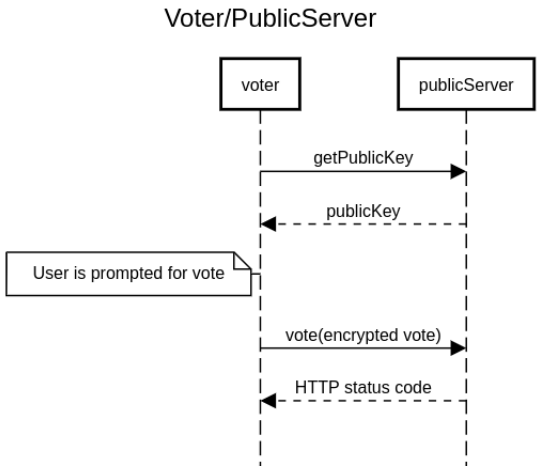

Figure 2: Sequence diagram of voter/public server communication

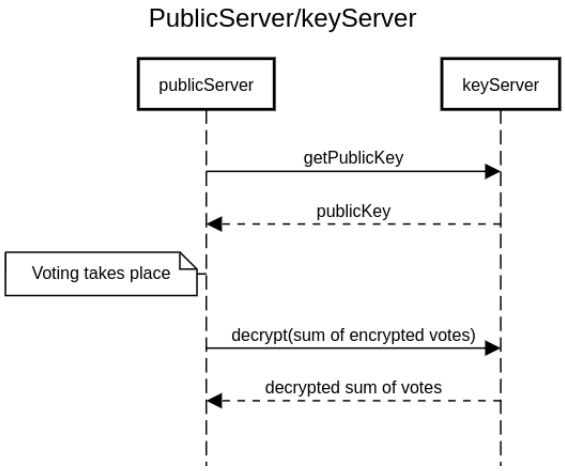

Figure 3: Sequence diagram of public server/key server communication

3.3.3 SSL

To secure the connection between the servers and the clients casting votes, the HTTPS protocol is used, by establishing an SSL connection. This is done using a `ServerConnector` with an `SslContextFactory`, and a `HttpConfiguration`. A self-signed certificate *keystore.jks* is created and stored in */certs/*.
In order for testing on localhost, the following code is needed

```
HttpsURLConnection.setDefaultHostnameVerifier(
(hostname, sslSession) -> hostname.equals("localhost")).
```

For the client and public server to be able to talk to the key server, they use an `SSLContext` with a `TrustManagerFactory` that loads the certificate. The `SSLContext` is then added to the builder, that creates the client.

4 Iteration 2: Security Against Malicious Users

In the second iteration of the project, trust in users are removed. This means that the assumption that votes are always either 0 or 1 is no longer valid. To ensure that malicious users can't rig the election by posting a vote $v \notin \{0, 1\}$, all votes are required to be accompanied by a proof of being either 0 or 1, but without revealing which one is the case. Voter's id, encrypted vote and proof of vote being 0 or 1 is posted by voters to the public server. The public server publishes this information, such that anyone can see it and verify that the votes are valid. Invalid votes will not be included in the sum of votes by the public server.

For the system to stay secure, updates are made to both the security policy and the threat model.
For simplicity only changes from the last iteration to this one has been included. I.e. the information from the prior iteration is still valid where nothing new is specified in the following security policy and threat model.

4.1 Security Policy

Trust Where the first iteration had complete trust in all, this iteration has complete trust in the public server and the key server, while not trusting the voters. The votes needs to be $v \in \{0, 1\}$

System Functionality To accommodate the change in trust, system functionality needs to be updated. To ensure that votes are $v \in \{0, 1\}$, a zero-knowledge proof is used.
The public server has had the following changes:

- The public server must be able to verify a non-interactive zero-knowledge proof, and only accept votes if their proof is verified
- The public server must now also act like a bulletin board i.e. it must allow the public to view the encrypted votes and their proofs

The reason that the public server must now also act like a bulletin board is to let anyone see the registered votes and proofs, and thereby enable them to verify the proofs.

The client has had the following changes:

- The client must be able to generate a non-interactive zero-knowledge proof, proving that the vote is either 0 or 1
- The vote cast is no longer $(ID, E_{pk}(vote))$, but are now $(ID, E_{pk}(vote), proof)$, where $proof$ is a non-interactive zero-knowledge proof, proving that the vote is either 0 or 1

4.2 Threat Model

STRIDE: Information Disclosure With the public server acting as a bulletin board, some information is disclosed during the voting process. The information that is disclosed is tuples containing ID, vote and proof. We assume that the vote is secure, as it is encrypted, and that ID does not contain sensitive information. The only information one can get from this, is therefore which voters are dishonest, which does not endanger the systems integrity.

EINOO: Internal Attackers With trust in voters removed, internal attackers are relevant for the threat model. Internal attackers can only be voters, as both public server and key server are still trusted.
The only possible attack unique to inside attackers is faking votes. As votes in the second iteration has to be coupled with both ID and a zero-knowledge proof, the system is secure, as long as the zero-knowledge proof is. The proof implementation is described in section 4.3.2

EINOO: Offline Attacks As the public server is now also acting like a bulletin board, the system is slightly more vulnerable to offline attacks as the ciphertexts can be downloaded by anyone from the public server. As the votes are encrypted by the homomorphic El-Gamal cryptographic scheme, it will be infeasible to find the values of the votes, assuming the keylength is big enough.

4.3 Implementation

To match the system description, as described in detail in the new security policy and threat model, a few implementations are needed.
The public server has to be modified into working as a bulletin board, and the voters need to create a proof, which the public server should verify, as described below in the Zero-Knowledge Proof section.

4.3.1 Bulletin Board

To implement bulletin board functionality, all the information the public server contains, needs to be available to the public.
The public server as we know it from the first iteration has the following information:

- Public key
- Received votes
- Recognized voters

The public server is not going to be a complete bulletin board, as the list of recognized voters cannot be made public, and it still has responsibility of validating and summing votes. In an ideal scenario this will not be kept here, but assuming trust of public server it is safe to keep it here for now.
The public key is already available for everyone through the API, meaning only the received votes needs to be made available for the bulletin-board functionality we want to be complete. This is done by adding the endpoint seen in figure 4 to the API.

CalculateResult	
Type	GET
Path	/getVotes
Params	None
Resp	List of $(ID, vote, proof)$
Code	200/404/500

Figure 4: REST endpoint for getting posted votes from server

4.3.2 Zero-Knowledge Proof

As trust in voters has been removed, it is necessary to ensure that voters only vote one of the allowed options; 0 or 1. In this system, it is done by using zero-knowledge proofs.
El-Gamal ciphertexts are on the form $(c, d) = (g^r, g^v h^r)$, where v is the vote. This means that voting 0 results in a ciphertext $(c, d) = (g^r, h^r)$ and voting 1 results in a ciphertext $(c, d) = (g^r, gh^r)$.
By using a combination of Σ-protocols for proving equality of discrete logorithms and the OR-proof, it is possible to prove whether a voter has voted 0 or 1. Hence proving that $log_g(c) = log_h(d)$ or $log_g(c) = log_h(g^{-1}d)$. This is also called a disjunctive Chaum-Pedersen proof.
As this iteration includes implementation of a bulletin board, it enables anyone to verify the legitimacy of the votes. The disjunctive Chaum-Pedersen proof is an interactive proof and therefore needs to be made non-interactive in order to support the usage intended by the bulletin board. Making the disjunctive Chaum-Pedersen proof non-interactive can be done by applying the Fiat-Shamir heuristics to the proof.

When a voter casts a vote, it needs to be accompanied by the zero-knowledge proof in figure 5. The proof (e_0, z_0, e_1, z_1) which is computed by the voter, is then sent to the public server, which posts the vote on the bulletin board and later tries to verify it. Verification of a zero-knowledge proof is done as in figure 6, where the prover wants to prove knowledge of some witness `w`, which in this case is the `r` from an El-Gamal ciphertext.

Prover

$$
\begin{aligned}
y, z_{1-v}, e_{1-v} &= \text{random} \in \mathbb{Z}_q \\
a_{1-v} &= g^{z_{1-v}} c^{e_{1-v}} \bmod p \\
\text{if } v = 1: \; b_{1-v} &= h^{z_{1-v}} d^{e_{1-v}} \bmod p \\
\text{if } v = 0: \; b_{1-v} &= h^{z_{1-v}} (dg^{-1})^{e_{1-v}} \bmod p \\
a_v &= g^y \bmod p \\
b_v &= h^y \bmod p \\
S &= H(a_0, b_0, a_1, b_1, c, d, ID) \bmod q \\
e_v &= S - e_{1-v} \bmod q \\
z_v &= y - re_v \bmod q
\end{aligned}
$$

Figure 5: Zero-knowledge proof

Verifier

$$
\begin{aligned}
a_0 &= g^{z_0} c^{e_0} \bmod p \\
b_0 &= h^{z_0} d^{e_0} \bmod p \\
a_1 &= g^{z_1} c^{e_1} \bmod p \\
b_1 &= h^{z_1} (dg^{-1})^{e_1} \bmod p \\
S &= e_0 + e_1 \bmod q \\
S &\stackrel{?}{=} H(a_0, b_0, a_1, b_1, c, d, ID) \bmod q
\end{aligned}
$$

Figure 6: Verification of zero-knowledge proof

The ciphertext is included in the hash, as to ensure that an adversary cannot use a correct zero-knowledge proof and compute a ciphertext from this. The ID is included as well, prohibiting adversaries from using other voters' vote as their own, as mentioned in section 2.5.2.

The zero-knowledge proof has been implemented using Javas implementation of `BigInteger`, for the same reasons that made this the case for the El-Gamal encryption scheme. The hashing used is `SHA256` provided by Bouncy Castle.
Upon voting, a client uses `SecurityUtils#generateVote` to generate a ciphertext and a corresponding proof, from a vote and an ID, which is then sent to the server.

Using the Proof To be able to use the proofs that are generated, the API needs to be updated to include proofs when voting.
To achieve this the voting endpoint from iteration 1 is replaced by the one seen in figure 7.

CalculateResult	
Type	POST
Path	/vote
Params	None
Body	$(ID, vote, proof)$
Resp	Empty
Code	204/403/500

Figure 7: REST endpoint for voting with proof included

The proofs are verified when the poll is closed. Before summing up the votes, they are filtered to exclude those with unverifiable proofs, as some votes might not be correct. Then the flow continues like in iteration 1, as described in section 3.3.2.

5 Iteration 3: Distributed Key Server

In the third iteration, trust in the key server is removed, and decryption is changed from being executed by a single key server to a group of decryption authorities, using a distributed secret key. This resolves the issue of having a single point of attack in form of the key server. Generation and distribution of shares of the secret key is now done by a trusted dealer. This forms a new single point of attack, but will only exist in the initialization phase.
The public server becomes a bulletin board, meaning that all functionality, except keeping a state, is removed.

As in the last iteration, updates are made to both the security policy and the threat model, in order for the system to stay secure,.
Only changes from the previous iterations has been included, meaning that the information from the prior iterations are still valid where nothing new is specified in the following security policy and threat model.

5.1 Security Policy

System Description The system will consist of a number of servers and a group of clients, the voters. One server acts as a bulletin board, meaning that any participant can post and read its content, and three servers acts like decryption authorities. The servers will be denoted BB and DAs, respectively.
A BB is a server instance that can have data uploaded to it, which from that point is assumed publicly known. All data posted to a BB can be fetched by anyone, participants and non-participants alike.
DAs will replace the key server. They each hold parts of a secret key, and will upon completion of the vote do a partial decryption of the result, which will then be posted to the BB.
During system initialization there will be a trusted dealer TD, which is trusted to generate key-pairs, where the secret and public key is now split into parts. These parts will be denoted as secret and public values. The TD will distribute the secret values to the DAs, and post the public values to the BB, all signed with the dealers own signature. It is assumed that the TD's public key is publicly known prior to the system initialization. This functionality is out of scope, but can be achieved through a certificate authority or distribution through some secure channel.

Trust For this iteration we trust the BB to correctly display all information it has received during its lifetime, and to stay alive for the entire voting process. We trust that at most $t < \frac{n}{2}$ of the decryption authorities has been compromised, which is $t < \frac{3}{2}$ as we use three DAs. Furthermore we trust that the TD generates and distributes keys honestly, and securely.

System Functionality Updated system functionality. As voters' functionality is unchanged, their functionality is the same as in last iteration. Public server and key server no longer exists and are replaced by the BB and DAs described here.
In this iteration it is assumed that anyone with access to the system is allowed to vote. This means that voter identities are no longer within the scope of this system. Handling voter identities is described in section 7.4.

The BB must implement the following properties

- The BB must be able to receive the following information:
  - Votes containing $(ID, vote, proof)$
  - Public keys for all DAs
  - Voting results from DAs containing $(ID, partial\ decryption, (c, d), amount\ of\ valid\ votes)$, where (c, d) is the ciphertext containing the sum of all valid votes
  - Time of poll termination

- All information received by the BB of the aforementioned types must be publicly visible
- All information posted to the BB must be paired with a timestamp by the BB itself, ensuring that votes posted after termination are not included when verifying results
- When initialized, the BB must contain information about when the voting terminates. This information must also be publicly visible

The DAs must implement the following properties. There are three DAs, which ensures that the correct result of the voting can be computed, even if one of the DAs are compromised.

- A DA can only hold a share of the secret key
- DAs must fetch the termination time for the poll from the BB. When this point in time has been reached, the authority must do the following:
  1. Fetch all votes from the BB
  2. Verify the votes
  3. Sum the verifiable ones
  4. Decrypt the sum, using its own part of the secret key
  5. Generate a proof of discrete log equality for the result
  6. Post $(ID, partial\ decryption, (c, d), amount\ of\ valid\ votes)$ to BB

The TD is responsible for initializing the DAs. For this to happen it must implement the following properties:

- The TD creates a random polynomial $f(x)$ of degree t, where $f(0)$ is the secret key
- The TD sends a point $(x, s_i = f(x) \bmod q)$ on the polynomial to each of the DAs, along with g, q from the public key. s_i is a share of the secret key
- The TD posts all partial public values, $h_i = g^{s_i}$, along with its own signature to the BB
- The TD terminates, and are no longer present on the network

5.2 Threat Model

STRIDE - Spoofing of Identity: Besides voter identity spoofing there is spoofing the identity of the TD to worry about. The TD uses its RSA key to sign a number of informations posted to the BB during the initialization phase. If an adversary were able to spoof this identity, which would mean getting its RSA secret key, the adversary would be able to post fake information to the BB, and could for instancecause votes to be encrypted using a public key not matching the secret key distributed to the DAs.

STRIDE - Tampering: The TD will create local files to be read from, and it will use RSA to sign it's values to prove that they came from the TD. As the public values will be posted on the BB, which allows for any number of public keys to be posted, the signature can be used to verify which ones are correct. This also means that every client will know the corresponding key to verify the TD at initialization.

STRIDE - Repudiation: As mentioned, for anyone who wants to check the result once voting is over, a proof of discrete logarithm equality is provided alongside the partial decryption. The public key needed to verify the proof is available on the BB, and is signed by the TD, thus anyone can be sure it is authentic.

EINOO - Internal Attackers: With the trust in the DAs removed, internal attackers become even more relevant. We remove trust in the DAs, meaning they can be show malicious behavior e.g. by decrypting before time or providing wrong decryption information. To ensure that DAs will not be able to have wrong voting results accepted, shares of the secret key are distributed among all DAs, and decrypting the votes will happen as a partial decryption, which will be posted to the BB alongside a corresponding proof.

X.800 - Offline Attacks: Offline attacks has not become more probable in this iteration. However they have become way more dangerous. By gaining offline access to the TD before execution one can get its RSA secret key, and then execute an attack by spoofing the identity of the TD.

5.3 Implementation

In order to match the system description, security policy and threat model of this iteration, changes in the servers' functionality is needed. These changes includes key generation, server structure and the servers' responsibilities. The new flow is depicted in figures 8 and 9

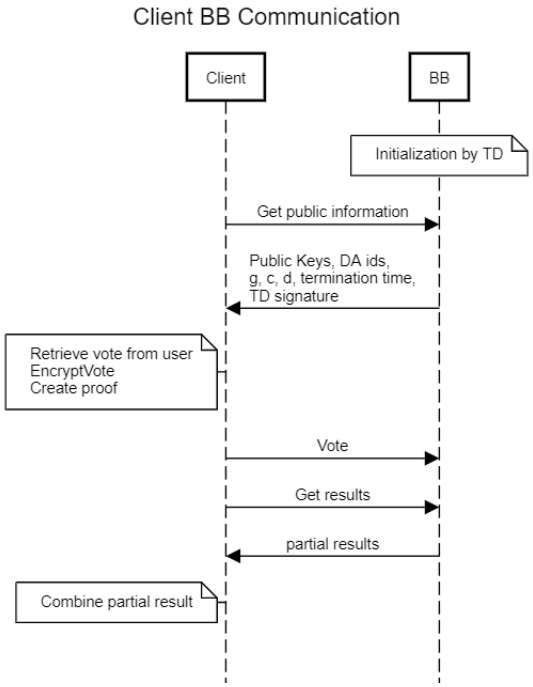

Figure 8: Sequence diagram of Client/Bulletin Board communication

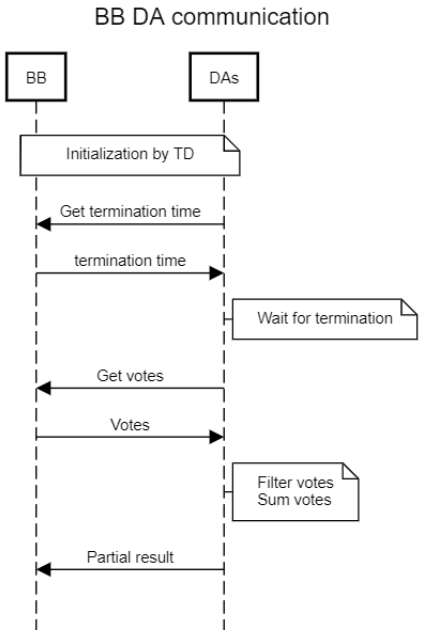

Figure 9: Sequence diagram of Bulletin Board-/Decryption Authority communication

5.3.1 Public Server to Bulletin Board

In the earlier iterations, the public server has acted as an interface between clients and the key server. Its responsibilities have been to make sure only eligible voters with valid votes were accepted, summing up all valid votes, sending the sum of votes to the key server and publishing the result when received from the key server.

In this iteration the public server will become an actual bulletin board. This means that all functionality besides being able to receive and serve data, has been removed from the server.
The data which the BB is responsible for keeping and serving is votes, partial public keys, partial decryptions of the result, and the time of closing of the poll.
To ensure that clocks will not become an issue, the BB adds a timestamp to all votes, which is the one used when comparing to the point in time when voting is no longer permitted. This ensures that all votes are handled equally.

5.3.2 New Secret and Public keys

DAs only possess a share of the secret key, which is generated using Shamir's secret sharing scheme. This means that the secret value $s_i \in \mathbb{Z}_q$ for DA_i is computed as $s_i = f(i) \bmod q$ where the function $f(x)$ is a random polynomial over $\mathbb{Z}_q$, of degree t, and $f(0) = sk$ is the secret key.

The method `SecurityUtils#generatePolynomial` computes the polynomial $f(x)$, and takes as parameters the desired degree, and a prime q. The function returns a BigInteger array representing the polynomial where index i corresponds to coefficient i in the polynomial, and all values are chosen randomly in $\mathbb{Z}_q$.

The method `SecurityUtils#generateSecretValues` updates a `HashMap` with the secret values, mapping each value to the corresponding decryption authority who is to receive them. The values are generated by looping from 0 to the length of a given polynomial, and accumulating $authorityIndex^j \cdot polynomial[j]$, where $authorityIndex$ is the `HashMap` key for the corresponding decryption authority. The returned map can then be used by the TD to distribute the secret values.

`SecurityUtils#generatePublicValues` then uses the secret values to generate the public ones. This is done by for every key i in the secret values map, the secret value s_i is used to calculate the corresponding public value as $h_i = g^{s_i} \bmod p$. Then the public values, h_i's, can be posted on the BB by the TD.

5.3.3 Trusted Dealer

The TD is a newly introduced role, with a very short lifespan.
As described in the security policy, the TD is used during initialization of the system. The TD generates a random polynomial, used to generate keysets, as described in section 2.6. Each partial secret key is saved to a file, with the public key, and the prime modulus p.
The TD also collects data relevant for the rest of the vote, which it posts to the BB, with a RSA signature. The data is the IDs of the decryption authorities, their partial public keys, (h_i, g, q) and the time at which the voting ends.
It is assumed that all participants are in possession of the RSA public key for the TD.
After the data has been posted to the bulletin board, and partial secret keys has been distributed, the TD terminates and are no longer a participant of the system.

5.3.4 Key Server to Decryption Authorities

In earlier iterations, the key servers only job was to decrypt values when queried by the public server. As the public server has been changed, into a bulletin board, this is no longer the case. To make the change clear, the key server has been renamed decryption authority, denoted DA.
The system has multiple DA's, which all holds a secret value, used for decrypting. The DA's parameters are initialized by the TD, which distributes the keys between the authorities, with the result that they have no knowledge of the keys held by the other authorities.
They have the responsibility for querying the BB for votes after the poll has closed, which is calculated using the clock of the BB compared to the clock of the DA. When doing so, it filters away all the votes with incorrect zero-knowledge proofs, summing up the rest. The DA then posts its partial decryption back to the BB, together with a proof of knowledge, the newly calculated ciphertext, the ciphertext containing the sum of all valid votes, and the total number of valid votes counted.

5.3.5 Partial Decryption and Corresponding Proof

To decrypt ciphertexts, of the form (c, d), one computes c^{sk}, where sk is the secret key. Once the voting has terminated, each DA will post $c_i = c^{s_i}$, which is denoted as a partial decryption, to the BB, together with a proof of discrete logarithm equality, e.g. that $log_c(c_i) = log_g(h_i)$.

Prover (DA)

$(c_i, h_i) = (c^{s_i}, g^{s_i})$

$y =$ random $\in \mathbb{Z}_q$

$a = c^y \bmod p$

$b = g^y \bmod p$

$e = H(a, b, c_i, h_i, A_{ID}) \bmod q$

$z = y + s_i \cdot e \bmod q$

send (z, e)

Figure 10: The proof that $log_c(c_i) = log_g(h_i)$

Verifier (Anyone)

$a = c^z c_i^{-e} \bmod p$

$b = g^z h_i^{-e} \bmod p$

$e \stackrel{?}{=} H(a, b, c_i, h_i, A_{ID}) \bmod q$

Figure 11: To verify that $log_c(c_i) = log_g(h_i)$

The methods `DLogProofUtils#generateProof` and `DLogProofUtils#verifyProof` implement the above protocol, similarly to how it was implemented in iteration 2.

The partial decryption is computed in `SecurityUtils#computePartials`, which simply returns $c^{s_i} \bmod p$.

To combine the computed partial decryptions, one needs to compute Lagrange coefficients. This is done using the `SecurityUtils#generateLagrangeCoefficient`, which computes

$$\lambda_j = \prod_{l \in \Lambda \setminus \{j\}} \frac{l}{l - j} \bmod q$$

where Λ is the set of x-coordinates in the polynomial which is used to generate secret values, and j is the current DA's index. Λ only contains x-coordinates of DA's who have posted valid partial decryption. λ_j is the Lagrange coefficient for DA_j.

Now `SecurityUtils#combinePartials` can combine the partials by computing

$$c^{sk} \bmod p = \prod_i c_i^{\lambda_i} \bmod p = c^{\sum_i s_i \lambda_i \bmod q}$$

for the verified partial decryptions.

5.3.6 Security of DAs

DAs only have a share $s_i = f(i) \bmod q$ of the secret key $sk = f(0)$. As a consequence of this, at least $t+1$ (i, s_i) pairs are needed to construct $f(x)$ using Lagrange interpolation (where t is the degree of $f(x)$). This ensures that a single corrupt DA cannot decrypt votes alone unless $t = 0$. If a DA could decrypt votes alone, the anonymity of voters would vanish, as voters id and vote are publicly accessible at the BB. To make sure that DAs can not just make a false decryption, which would be accepted, each DA posts a proof of discrete log equality, where they prove that $log_c(c_i) = log_g(h_i)$. The proof is shown in figure 10 and the verification is shown in figure 11. As c is part of the ciphertext containing the sum of all valid votes, it's publicly accessible at the BB. $h_i = g^{s_i}$ is also public information available at the BB, and verifying that $log_c(c_i) = log_g(h_i)$ is the same as verifying that DA_i has used the correct s_i when computing c^{s_i}.

In this implementation, the DAs post the encrypted sum of votes (c, d) together with the proof of discrete log equality, the partially decrypted value c^{s_i} and the amount of valid votes to the BB. Clients only computes c, d and count the amount of valid votes themselves if some of the DAs disagree in one or more of these three values. This is checked when clients want the result of the voting. A consequence of this is that if all DAs are corrupt and agreeing on fake c and d values, the client will think that the voting ended with a different result than what it actually did. However, since all votes are publicly accessible at the BB, it would easily be discovered if all DAs were corrupt. The implementation allows clients to decide to compute these values themselves, which would disclose the corrupted DAs trick. The only reason for not automatically computing the sum of all votes at each client is that it's very computationally heavy when there are many votes.

5.3.7 Clients

In this iteration the clients need to do a few more computations than they have been doing earlier.
When a client is initialized, it will fetch the public information from the BB, and make sure this is signed by the TD's RSA key. The client will then combine the partials of the public values, to create the public key used for encrypting their vote. This is done using `SecurityUtils#combinePartials` to combine the h_is by computing

$$h = g^{sk} \bmod p = \prod_i h_i^{\lambda_i} \bmod p = \prod_i g^{s_i \lambda_i} \bmod p = g^{\sum_i s_i \lambda_i \bmod q}$$

When voting has been terminated, and the DAs have posted $(ID, \textit{partial decryption}, (c, d), \textit{amount of valid votes})$ to BB, each client can then fetch these. If at least $t+1$ of the partial decryptions have valid proofs, then the client can decrypt and get the voting result.
If not all partial decryptions, encrypted sum of all valid votes, and amounts of valid votes posted on the BB match, or at least one partial decryption has an invalid proof, then the client will fetch all the valid votes from the BB and compute the encrypted sum of all valid votes themselves. This check is performed in `ResultFetcher#decryptionAuthoritiesAgrees`.
The client then proceeds in the decryption process with it's own encrypted sum of all valid votes to verify which proofs of partial decryption are valid. If $t+1$ proofs are valid, the clients can get the result of the voting. This is possible, even if the proof is correct, but the provided c, d, and amount of valid votes are wrong, due to the fact that d is not used in the proof, and should the proof be accepted, then the c used to check it must be the one used to create the proof, thus both c and d would be correct.

The option for the client to calculate the encrypted sum of all valid votes and use this when computing the voting result can be specified by running the client with parameters `--client --read=true --forceCalculations=true`.
The entire process described above of getting the voting result as a client is implemented in `ResultFetcher#run`.

6 Implementation and Evaluation

This section will be a brief description of the practical components of the system, including the testing and benchmarking thereof.
The system has been created using the Java programming language. For dependency management Apache ant and Apache ivy has been used.
To develop the system, testing was used as an important tool, and to optimize fragile points of the system, the `JMH` tool was used for benchmarking.

6.1 Testing

During the development of the system, and especially the parts involving cryptography, automatic testing has been a crucial factor for success. The automatic tests were written using the `JUnit` testing framework, asserting correct behavior of key parts of the system, before being implemented into the final system. Both unit tests and integration tests as well as both test supposed to succeed and tests supposed to fail has been written. This was done in order to disclose as many potential errors in the system as possible.

6.2 Benchmarking

During development and testing of the system multiple scalability issues were discovered. The most prominent being summing all the encrypted votes.
The naive, and initial, implementation of this was summing all the votes synchronously, utilizing only one processor core. By introducing a new method for doing the calculations concurrently multiple processor cores can be utilized. Both these methods takes linear time, relative to the number of votes, but the concurrent method has a smaller gradient, dependent on the number of processor cores. This makes it possible to scale the number and size of processors in the computers doing vote summation, and thereby increasing effectiveness, and not limit scalability to only processor speeds.
The same concept was applied to filtering votes based on both time and vote validation. All results can be seen in appendix B. It is seen that on the particular computer used in B.1 the asynchronous methods performs better by a factor of approximately 6, for all three benchmarked tasks. In the benchmark seen in B.2 the performance gain is limited by the 10 threads being spawned in the asynchronous benchmarking methods. The performance gain is approximately 10 for each of the methods, letting us conclude that the overhead for the concurrency is negligible, and that it scales with the maximum number of threads being executed.

7 Comparisons and Ideas for Future Work

Throughout the three iterations described, a secure system was created, based on two assumptions; we have a bulletin-board acting according to the security policy for iteration three[1], and a trusted dealer also acting according to the description and distributing information through a secure channel.

7.1 Bulletin Board

The bulletin board has very few responsibilities. It needs to be able to have data posted to it, and serve the data back.
This functionality can be achieved reliably in a number of ways. A likely solution would be using a replicated state machine, acting like the bulletin board used in this project. By having many independent servers running the RSM it will be resistant to most attacks. As no verification takes place on the bulletin board, identity spoofing and attacks falls outside it's area of responsibility.

[1] Described in section 5.1

7.2 Trusted Dealer

The trusted dealer is a security issue, as it acts as a single point of failure, even though it only exists briefly in the lifetime of the system.
A solution could be simply running the trusted dealer locally, and then uploading the generated files to servers running decryption authorities using a secure connection[2], and then proceed to delete all files. However, if the security on the computer executing this fails, it will break anonymity for the entire system, as an adversary will have access to all parts of the distributed secret key.
To fully utilize the security of distributed decryption, multiple parts of the distributed secret key can never be in the same place.

7.3 Distributed Key Generation

To completely remove the single point of failure that is the TD, one would change the system so that all players share the secret key, but so that no minority has enough information to compute this key. This means that there has to be a change in how the authorities behave. There will be authorities A_i, $i = 1, \ldots n$ which will each choose a random polynomial $f_i(x)$, used in creating the secret values, and distributing $f_i(j)$ to authority A_j, $j = 1 \ldots n$.
In order to protect against a corrupt player, who all information from some other players, the A_i's would have to make their secret value public, if any A_j would reject their public value. This makes it easier to disqualify corrupt authorities, by checking their secret values, which would already be known to any corrupt player. And if A_j is the corrupt authority, then A_j would have chosen the secret value itself, and thus the adversary learns nothing new, as it already had the correct value from A_i. This way all players know the correct values by ignoring those disqualified.

7.4 Voter Identities

When implementing a voting system, an important issue will always be to determine who is allowed to vote. In the system created in this project, voters send a string as identification. If the system is to be used in an actual election or referendum this will need to be strengthened.
One solution could be using certificates and signatures, with each certificate signed by a trusted third party. If it were to be used in Denmark the signing service of DanID would be an obvious choice. The client would then use the DanID API for having the citizen sign his/her vote, and then post the vote and the signature to the bulletin board. Anyone wanting to verify the votes could then simply verify that the signature using the public DanID certificate.
This would place all trust at DanID, or whoever is chosen for this responsibility.

7.5 E-Voting using Mixnet

Tallying votes using homomorphic encryption is not the only way to create a secure e-voting system. There are systems out there, like Helios, which is based on Benaloh's Simple Verifiable Voting protocol [Ben06]. This type of protocol separates the ballot preparation and casting, where voters are authenticated when the ballots are cast.
This system still involves using a bulletin board, where the votes are displayed with either voter ID or name. When voting, the voter receives a hashed ciphertext of their selected answers. The voter can at any time at this part in the process choose to see the ciphertext, including which randomness was used to create it. This way the voter can verify the encryption themselves. If this option is chosen, the voter will be prompted to generate a new encryption of their choice.

The biggest difference between this system, and the homomorphic system, is that this one wishes to preserve individual ballots. This is achieved using the Sako-Kilian protocol [SK95], which is the first provable mixnet

[2]SSL, SFTP, FTPS or similar

based on El-Gamal. All inputs in the Sako-Kilian mixnet, are El-Gamal ciphertexts. The ciphertexts are fed to a mix server, which re-encrypts and permutes them according to a random permutation. The server also creates a fake "shadow mix". A verifier then challenges the mix server, by having it reveal the permutation and re-encryption factors for either the shadow mix or the shuffle that would transform the shadow mix outputs into primary mix outputs. This ensures that a correct mix server can always answer the challenge, and a corrupt server can be caught with at least a .5 probability. Having the server create t shadow mixes, results in a probability $1-2^{-t}$ that the primary mix is correct, if it can successfully respond to all challenges. As t increases, the probability becomes very close to 1.
These proofs are honest-verifier-zero-knowledge proofs, which are made non-interactive using Fiat-Shamir. To verify the decryption of the El-Gamal ciphertext, the Chaum-Pedersen proof is used. Both are used in exactly the same was as they were in the homomorphic encryption system. [Aid08]

7.6 Pitfalls in Electronic Voting

Electronic voting has been attempted by multiple parties in recent history, and implementation of these systems has revealed some pitfalls.
One example of a system revealing some of these pitfalls is the Swiss Post e-voting system, which was developed by the company Scytl, and intended for voting online in the Swiss Cantons.
The system claims to have a property of complete verifiablity, which means that it can be verified that everything has gone as it should, and no adversary has meddled with data. This is implemented using proofs similar to those used in EVHE.
However, a research team has found that these proofs had issues. They found that the implementation of the Fiat-Shamir transform was faulty, making it possible for a compromised decryption service to replace some votes with garbage and then providing a verifiable proof, thus making the system enter a clearly illegal state, with no apparent reason.[LPT19b]
The other problem was proving that the mixnet servers shuffled the votes correctly. The systems mixnet uses a trapdoor commitment scheme. The research team found that given the trapdoor values it is possible to provide a verifiable proof that the votes has not been altered, while actually altering the votes.[LPT19a]
Another electronic voting system already out there is Helios. A research team found that they also had problems with their implementation of the Fiat-Shamir transformation. The team found that this meant that a malicious prover were able to make the election termination run indefinitely or even tamper with the outcome.[BPW12]

7.7 Usage in Practice

This report has described how an electronic voting system can be created in a secure way, under a number of assumptions. Some of these assumptions have been changed during the report, but as is often the case, not every important aspect has been included in the threat model.
The security policies and threat models used in creating this system has ended where the system did, not taking users into account. Several issues besides the technical ones are worth taking into consideration. As described in section 7.6 there has been other systems attempting this with some serious security concerns. Will the public be able to trust a new system, without understanding the underlying technology? Will it exclude elder citizens from voting, as they lack understanding of the workings of an electronic voting system? How big is the risk of malware taking control during the process of casting a vote, and then casting a vote different from that of the user? Will coercion, where a person forces another to vote against their will become a problem?
All these questions are outside the scope of this report, and also outside the scope of computer science, one could argue, but unfortunately this does not make them irrelevant.

7.7.1 Multi-Candidate Elections

For the system to be used in practice it should also be able to handle general elections, and not just referendums as it is currently limited to.
One way of achieving this is to use k different generators, denoted G_i, one for each k possible candidates. When counting the votes they are accumulated separately for each candidate. For the limitation that a voter can vote for only one candidate to be true, the proof of knowledge must be altered to prove that

$$log_g c = log_h \left(\frac{c}{G_1}\right) \vee \cdots \vee log_g c = log_h \left(\frac{c}{G_k}\right)$$

where g is the generator used. As this proof can only be generated for a maximum of one generator G_i it guarantees that the voter has voted for only one candidate, if it is successfully verified. This does however introduce problems regarding computing the final result, which will not be discussed in this report.
[CGS97, sec. 4].

8 Conclusion

The final version of the system consists of a trusted bulletin board, a number of decryption authorities, a trusted dealer, and a number of voters.
The bulletin board is publicly accessible, and is used to store publicly accessible information. It is trusted to be available and keep data un-corrupted, at all times.
The decryption authorities utilizes a share of the secret key, in order to partially decrypt valid votes, when voting is no longer allowed. Validity is determined by a zero-knowledge proof, which is part of each vote. The partially decrypted result is posted to the bulletin board with a zero-knowledge proof, proving its correctness.
The trusted dealer is responsible for distributing partial secret keys to the decryption authorities, and posting partial public keys to the bulletin board. It also determines the termination time of the poll.
Voters posts their desired vote, a zero-knowledge proof that their vote is legal, and their ID, to the bulletin board. When the decryption authorities has posted their partial results, voters are able to compute the result of the poll.
The system can be used for polls with a boolean result, assuming a non-corruptable bulletin board, guaranteeing anonymity of votes, and correctness of the result.

Johannes Ernstsen 201605841
Morten Erfurt Hansen 201608514
Mathias Søby Jensen 201608824

9 Appendix

A Code

All code can be found in a github repository found at:
https://github.com/Ernstsen/EVHE

A guide to running the system is available in the project wiki at:
https://github.com/Ernstsen/EVHE/wiki/Running-EVHE

B Benchmarks

Benchmarkings were executed using the Java Microbenchmark Harness framework. Three jobs were benchmarked using both an asynchronous method, utilizing multi-threading, and a synchronous method. `BenchmarkTimeFilter` measures the time taken to filter a number of votes from their timestamps. `BenchmarkSum` measures the time taken to sum a number of votes, and `BenchmarkVoteValidation` measures the time taken to filter a number of votes based on the validity of their proof of knowledge.
All the tests were executed with `100`, `500` and `1000` votes, as to make eventual scalability apparent.
The scores represents the average time it takes to execute the benchmark once, while the count represents the number of batches that was run for each test. A batch means that the test was executed 10 times, so for a Cnt of 5 the method was measured 50 times. The error represents the maximum deviation from the score.

B.1 Octa-core Workstation

Benchmark	(size)	Mode	Cnt	Score	Error	Units
BenchmarkTimeFilterAsync	100	avgt	5	0.012	± 0.002	ms/op
BenchmarkTimeFilterAsync	500	avgt	5	0.015	± 0.003	ms/op
BenchmarkTimerFilterAsync	1000	avgt	5	0.017	± 0.002	ms/op
BenchmarkTimerFilterSync	100	avgt	5	0.001	± 0.001	ms/op
BenchmarkTimerFilterSync	500	avgt	5	0.004	± 0.001	ms/op
BenchmarkTimerFilterSync	1000	avgt	5	0.010	± 0.001	ms/op
BenchmarkSumAsync	100	avgt	5	947.618	± 47.678	ms/op
BenchmarkSumAsync	500	avgt	5	4644.938	± 209.005	ms/op
BenchmarkSumAsync	1000	avgt	5	10073.145	± 329.977	ms/op
BenchmarkSumSync	100	avgt	5	5998.197	± 154.070	ms/op
BenchmarkSumSync	500	avgt	5	30833.575	± 1918.372	ms/op
BenchmarkSumSync	1000	avgt	5	61708.432	± 2946.481	ms/op
BenchmarkVoteValidationAsync	100	avgt	5	990.264	± 115.479	ms/op
BenchmarkVoteValidationAsync	500	avgt	5	4900.370	± 1127.479	ms/op
BenchmarkVoteValidationAsync	1000	avgt	5	9223.729	± 1038.233	ms/op
BenchmarkVoteValidationSync	100	avgt	5	5956.034	± 275.060	ms/op
BenchmarkVoteValidationSync	500	avgt	5	29942.303	± 872.636	ms/op
BenchmarkVoteValidationSync	1000	avgt	5	59357.987	± 1142.725	ms/op

The benchmarking was executed on a desktop computer using 16gb of `2366MHz DDR4 memory` and using an `Intel 9700K` with 8 cores and no hyperthreading. The benchmarking process was not the only process utilizing the processor.

B.2 24-core Google Cloud Server

Benchmark	(size)	Mode	Cnt	Score	Error	Units
BenchmarkTimerFilterAsync	100	avgt	5	0.054	± 0.002	ms/op
BenchmarkTimerFilterAsync	500	avgt	5	0.062	± 0.001	ms/op
BenchmarkTimerFilterAsync	1000	avgt	5	0.067	± 0.002	ms/op
BenchmarkTimerFilterSync	100	avgt	5	0.002	± 0.001	ms/op
BenchmarkTimerFilterSync	500	avgt	5	0.007	± 0.001	ms/op
BenchmarkTimerFilterSync	1000	avgt	5	0.015	± 0.001	ms/op
BenchmarkSumAsync	100	avgt	5	378.486	± 14.735	ms/op
BenchmarkSumAsync	500	avgt	5	1919.546	± 23.081	ms/op
BenchmarkSumAsync	1000	avgt	5	4009.460	± 27.862	ms/op
BenchmarkSumSync	100	avgt	5	4147.061	± 160.497	ms/op
BenchmarkSumSync	500	avgt	5	21013.228	± 187.056	ms/op
BenchmarkSumSync	1000	avgt	5	42856.794	± 246.977	ms/op
BenchmarkValidationAsync	100	avgt	5	351.240	± 9.155	ms/op
BenchmarkValidationAsync	500	avgt	5	1677.393	± 30.080	ms/op
BenchmarkValidationAsync	1000	avgt	5	3336.254	± 37.120	ms/op
BenchmarkValidationSync	100	avgt	5	4104.897	± 20.543	ms/op
BenchmarkValidationSync	500	avgt	5	20540.662	± 82.285	ms/op
BenchmarkValidationSync	1000	avgt	5	41102.553	± 249.034	ms/op

The benchmark was executed on a Google Cloud instance with 24 vCPUs and 21.75 GB of memory. The benchmark had the processors to itself. Note that the benchmarking process at most spawns 10 threads, limiting the optimization to a factor of 10.

Johannes Ernstsen 201605841
Morten Erfurt Hansen 201608514
Mathias Søby Jensen 201608824

Bibliography

[Aid08] B. Aida. Helios: Web-based open-audit voting. 2008.

[Ben06] J. Benaloh. Simple verifiable elections. 2006.

[BPW12] D. Bernhard, O. Pereira, and B. Warinschi. How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to helios. 2012.

[CGS97] R. Cramer, R. Gennaro, and B. Schoenmakers. A secure and optimally efficient multi-authority election scheme. 1997.

[Dam06] I. Damgård. On electronic voting schemes. 2006.

[Dam10] I. Damgård. On sigma-protocols. 2010.

[Dam17] I. (2017) Damgård. Discrete log and lwe based cryptosystems. 2017.

[DNO18] I. Damgård, J. Nielsen, and C. Orlandi. *Secure Distributed Systems*. 2018.

[Lau09] N. Lauritzen. *Concrete Abstract Algebra*. 2009.

[Lev10] D. Levy. Introduction to numerical analysis. pages 25–26, 2010.

[LPT19a] S. J. Lewis, O. Pereira, and V. Teague. Ceci n'est pas une preuve. 2019.

[LPT19b] S. J. Lewis, O. Pereira, and V. Teague. How not to prove your election outcome. 2019.

[Sha79] A. Shamir. How to share a secret. communications of the acm, 22 (11). 1979.

[SK95] K. Sako and J. Kilian. Receipt-free mix-type voting scheme: A practical solution to the implementation of a voting booth. 1995.