Building an ElectionGuard Verifier in Go (Udarbejdelse af ElectionGuard Verifier i Go)

Andreas Skriver Nielsen, 202005664
Niklas Bille Olesen, 202006523
Hans-Christian Kjeldsen, 202004086

Bachelor Report (15 ECTS) in Computer Science

Advisor: Diego F. Aranha

Department of Computer Science, Aarhus University

2023-06-06

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

ElectionGuard has the potential to improve the integrity and efficiency of the electoral process by utilizing zero-knowledge proofs, thereby enabling verifiability of election results for independent verifiers. Therefore, as ElectionGuard becomes increasingly adopted, it emphasizes the importance of technical material that supports building independent verifiers. This thesis presents the development of an ElectionGuard verifier in Go, which can serve as a reference for future implementations of ElectionGuard verifiers, thus increasing the integrity of the ElectionGuard voting system. To achieve this, the thesis gives a detailed analysis of ElectionGuard and the cryptographic foundations, such as zero-knowledge proofs and the exponential ElGamal cryptosystem over the integers. Our verifier serves the purpose of verifying the proofs to prove the integrity of the election. We have built the verifier in Go, focusing on analyzability and efficiency. We benchmarked our Go verifier and further parallelized it to achieve a faster running time than the MITRE verifier by a factor of approximately 12 on an i9-10850K CPU. Lastly, suggestions for improvements to ElectionGuard's performance have also been covered by examining the benefits of elliptic curve cryptography. This showed convincing speed and storage improvements; however, with an undesirable increase in complexity and dependencies for verifiers.

Andreas Skriver Nielsen, Niklas Bille Olesen, and Hans-Christian Kjeldsen,
Aarhus, 2023-06-06.

Contents

Chapter 1

Introduction

Elections are a fundamental aspect of any democratic society, allowing citizens to participate in selecting their representatives and leaders. Electronic voting has the potential to revolutionize the way elections are conducted by increasing accessibility and accuracy. However, security, transparency, and privacy concerns have hindered its widespread adoption. To address these challenges, Microsoft has designed ElectionGuard to provide end-to-end verifiability. This means that voters can verify that their votes were correctly recorded by the system and included in the final tally. Utilizing cryptography and zero-knowledge proofs, ElectionGuard offers a secure and transparent way to record and count votes while protecting individual voters' anonymity.

The contributions of this thesis are the following:

- An easier-to-understand and more complete description of the ElectionGuard 1.1 specification, particularly tailored for individuals new to cryptography.
- An independent verifier written in Go that can be used as a reference for future independent verifier implementations.
- A brief introduction to elliptic curve cryptography and an experimental evaluation of exponential ElGamal over elliptic curves as an alternative to exponential ElGamal over the integers.

Chapter 2 covers the literature we have chosen to use within this thesis. This chapter also introduces the general terminology, structure, and protocols of ElectionGuard. Additionally, the theory of exponential ElGamal and zero-knowledge proofs are covered.

Chapter 3 covers key generation protocols within ElectionGuard. This includes the generation of an ElGamal keypair for each guardian, along with a Schnorr proof that the guardian knows the secret key belonging to the public key. The generation of a joint election public key is also explained, along with the threshold encryption scheme, which allows only a quorum k of n guardians available to decrypt using Shamir's secret sharing.

Chapter 4 describes the proof of an encryption being either *zero* or *one*, this is done to prove the correctness of a ballot. This is divided into two proofs; an encryption of *zero* and an encryption of *one*. The approach combines two Chaum-Pedersen proofs using Cramer-Damgård-Schoenmakers

such that a disjunction can be proven without revealing which disjunct is true. This is followed by a proof that the encrypted ballot satisfies the selection limit.

Chapter 5 covers how all encrypted ballots cast are aggregated into tallies and how these are verifiably decrypted. Since a quorum k out of n guardians is required for decryption, this chapter explains how Shamir's secret sharing, combined with Lagrange, enables threshold decryption without revealing the secret keys. Additionally, a Chaum-Pedersen proof is covered to validate the correctness of each partial decryption.

In Chapter 6, we provide guidance for implementing an ElectionGuard verifier. We begin by presenting an overview of the necessary steps to verify an ElectionGuard election. Next, we discuss our implementation of a verifier using the Go programming language, which we chose due to its speed and straightforward syntax. We also examined the trade-off between *analyzability*[1] and *running time*. Our verifier can be found on GitHub. Additionally, we compare our Go-based verifier to MITRE's Julia-based verifier, demonstrating that our implementation is approximately 12 times faster on an x86-based i9-10850K CPU and two times faster on an ARM-based M1 CPU. The *running time* is a crucial characteristic to consider when building a verifier because a slow verifier could significantly delay the election results, potentially leading to voter frustration and mistrust in the voting process.

Chapter 7 considers if it's possible to increase the performance of ElectionGuard, namely the encryption/decryption speed and the ciphertext size. Therefore we delve into the theory of elliptic curves, followed by an explanation and experimentation of exponential ElGamal over elliptic curves as a potential alternative to exponential ElGamal over the integers. We show that exponential ElGamal over elliptic curves is significantly faster than exponential ElGamal over the integers. Additionally, the ciphertext size can be reduced by a factor of four while preserving the same 128 bit security strength.

Lastly, Chapter 8 serves the purpose of concluding the work and summarizing the thesis findings. This chapter briefly covers future work, especially considering the idea of publishing ballots during the election for the verifier to verify rather than afterward.

[1] Analyzability is the ability to *understand* software (Christensen, 2020, p. 49-50)

Chapter 2

Preliminaries

Throughout this thesis, the main literature used is the ElectionGuard official specification (Benaloh & Naehrig, 2018). This specification, written by Microsoft researchers/employees, covers the structure of ElectionGuard, the mathematical properties, and the verification steps necessary to successfully and adequately implement a verifier. We critically evaluated the material and included additional literature to support the ElectionGuard specification.

This chapter will cover the basics of ElectionGuard. This includes the fundamental phases of ElectionGuard, the encryption ElectionGuard uses, and the zero-knowledge protocols ElectionGuard employs to ensure the integrity of an election.

The first proposed protocol is the Schnorr protocol (C.P., 1990), which allows the holder of a secret key s to prove possession of s without revealing s. Furthermore, we will also need to prove that a value decrypts to a particular value without revealing the nonce nor the secret key by using an interactive Chaum-Pedersen proof (Chaum & Pedersen, 1993). Using the Chaum-Pederson proof with the Cramer-Damgård-Schoenmakers technique (Cramer et al., 1994), one can prove that an encryption is either a *zero* or a *one*. This is beneficial when verifying that a vote is correctly formed. The above proofs are initially interactive. However, ElectionGuard applies the Fiat-Shamir heuristic (Fiat & Shamir, 1986), to make the proofs non-interactive. The other material used serves the purpose of providing a better understanding of the core material covered above and an understanding of exponential ElGamal (ElGamal, 1984).

2.1 ElectionGuard

There are three main phases of the ElectionGuard process. The *pre-election key generation* phase, the *ballot encryption* phase, and the *post-election decryption* phase. To understand how ElectionGuard works, it is essential to understand the various terms and protocols introduced during these phases.

Pre-election key generation phase

The primary purpose of the key generation phase is to construct a public key for the election. In ElectionGuard, the responsibility of generating this public key lies upon the system's guardians.

A guardian is a trusted entity whose job is to generate its own individual public-secret key pair. All guardians must be available for the public key ceremony where they join their public keys to form the joint public key as described in Section 3.1. This joint public key is used to encrypt individual ballots and therefore protect the confidentiality of the voter.

Ballot encryption phase

The ballot encryption phase uses the joint election public key to encrypt the individual ballot of voters. A ballot consists of at least one, but possibly many contests. A contest is a list of options, where the voter will give each option the value of either *zero* or *one* (Benaloh & Naehrig, 2018, p. 5). Therefore, an encrypted ballot consists entirely of encryptions of *zeros* and *ones*. The encryption of each option must also be accompanied by proofs, stating that each value is an encryption of either *zero* or *one*. This is covered in Section 4.1. Each contest is also given a selection limit, which specifies how many options a voter is allowed to select. To prevent a voter from exceeding the limit, the homomorphic property described in Section 2.2.2 is utilized. This makes it possible to generate an encryption of the contest sum, thus detecting that the amount of selections does not exceed the selection limit (Benaloh & Naehrig, 2018, p. 15-16). A detailed explanation is given in Section 4.2.

Voters are also allowed to submit undervotes and null votes. A null vote is a blank vote where none of the options are selected. An undervote is a more general term that occurs if the number of selections in a contest is less than the selection limit (Benaloh & Naehrig, 2018, p. 21). When aggregating the options, placeholder options are introduced to accommodate the issue of being able to differentiate between undervotes and regular votes. If the selection limit is one, the placeholder option can be thought of as a "none of the above" option. When the selection limit is greater than one, the amount of placeholder options must be equal to the selection limit. For a vote to be valid, the selected options, including the placeholder options, must therefore always equal the selection limit (Benaloh & Naehrig, 2018, p. 15-16).

Post-election decryption phase

Once the election is complete, the homomorphic property is used to aggregate all cast ballots. Thus, creating the encrypted tallies. To decrypt a tally or a spoiled ballot, multiple secret keys are required. It is also the responsibility of the guardians of the system to decrypt the tallies. Each guardian will therefore do a partial decryption of the encrypted tally. These partial decryptions are used to form the full decryption. In addition to the partial decryptions, each guardian also produces verifiable proof that their decryptions are correct, which is covered in depth in Section 5.1.

In ElectionGuard, only a quorum k of the total n guardians is needed to be available for the decryption to take place (Benaloh & Naehrig, 2018, p. 11-12). When generating the joint public key, all guardians also publish shares of their secret key to all other guardians. From these key shares, the k guardians can form the partial decryption of the missing guardians. This ensures that the aggregated tallies can always be decrypted by any quorum set of k guardians. This is covered in Section 5.3.

Spoiled ballots

When a voter has submitted their encrypted ballot, they need to be confident that their selections have been correctly encrypted. To accommodate this, ElectionGuard allows voters to either cast or spoil their submitted ballot (Benaloh & Naehrig, 2018, p. 23). If the voter spoils their ballot, they are given the possibility to submit a new ballot. At the end of the election, all spoiled ballots are verifiably decrypted. A spoiled ballot does not influence the result of the election.

If a voter chooses to spoil a ballot and verifies that it was encrypted correctly, it does not guarantee that their next vote will be correctly encrypted. In practice, if a voter discovers the decryption of their spoiled ballot to be different from their intended vote, the machine accountable for the encryption of that ballot will be replaced.

For the current ElectionGuard version i.e., 1.1, it is impossible to verify the encryption of a spoiled ballot before the post-election decryption phase. This will be resolved with the release of version 2.0[1].

2.2 Encryption in ElectionGuard

ElectionGuard utilizes exponential ElGamal over the integers for encrypting ballots due to its homomorphic additive property. This property allows for ballot aggregation without the need to decrypt individual ballots. The security of exponential Elgamal depends on the hardness to solve the discrete logarithm problem, for which there is no efficient algorithm if the group is chosen carefully (Damgård, 2022, p. 111). Additional information regarding the discrete logarithm can be found in Appendix A. To construct the generator used for encryption, ElectionGuard utilizes Schnorr groups (Damgård, 2022, p. 116-117). ElectionGuard also makes use of NIZK proofs, which have the property of being able to prove to be in procession of some secret x without revealing the secret x nor the nonce R used for encryption.

2.2.1 Schnorr groups

The Schnorr group, also known as $\mathbb{Z}_p^r$, is a order q subgroup of $\mathbb{Z}_p^*$ (Wikipedia, 2023). The process of generating this group involves first generating three values: p, q, and r. These values are selected such that $p = qr + 1$, where both p and q are prime. Then any value h within the range $1 < h < p$ is selected, such that the following congruence is fulfilled

$$h^r \not\equiv 1 \pmod{p}.$$

The generator

$$g = h^r \mod p$$

is then a generator of order q. ElectionGuard uses Schnorr groups in the exponential ElGamal encryption scheme. This is due to its efficiency since q is chosen to be a small prime compared to p, which makes exponentiation fast in exponential ElGamal.

[1] https://github.com/microsoft/electionguard/discussions/305

2.2.2 Exponential ElGamal over the integers

ElGamal is a public key cryptosystem. It is based on the mathematical problem of solving discrete logarithms and provides confidentiality (ElGamal, 1984). ElectionGuard uses an exponential form of ElGamal to encrypt ballots. This gives the additive homomorphic property that is desired when aggregating ballots.

To encrypt messages using the ElGamal encryption scheme, we must define q, p, r, and g. These four values are defined in ElectionGuard (Benaloh & Naehrig, 2018, p. 8-9) and follow the definition of Schnorr groups described in Section 2.2.1. To generate an ElGamal public-secret key pair, one must first pick a random value $s \in \mathbb{Z}_q^*$ as the secret key, where the public key is $K = g^s \bmod p$ (Benaloh & Naehrig, 2018, p. 4).

When encrypting a message M, using the exponential form of ElGamal, a nonce R is selected such that $R \in \mathbb{Z}_q$. The following ciphertext tuple can then be generated and sent to the receiver

$$(\alpha, \beta) = (g^R \bmod p, g^M \cdot K^R \bmod p).$$

Let $a \equiv_p b$ be a more compact notation for $a \equiv b \bmod p$. The tuple can then be decrypted by the user who holds the secret key s corresponding to the public key K as

$$\frac{\beta}{\alpha^s} \equiv_p \frac{g^M \cdot K^R}{(g^R)^s} \equiv_p \frac{g^M \cdot (g^s)^R}{(g^R)^s} \equiv_p \frac{g^M \cdot g^{Rs}}{g^{Rs}} \equiv_p g^M. \tag{2.1}$$

It's important to note that the receiver has not recovered M from the ciphertext but has instead recovered $g^M \bmod p$. However, when the set of options for M is limited, it is easy to recover M by linear search, a pre-computed table lookup, or Shank's baby-step giant-step (Shanks, 1969).

The additive homomorphic property, $E(M_1) \cdot E(M_1) = E(M_1 + M_2)$, holds for exponential ElGamal since

$$\begin{aligned} E(M_1) \cdot E(M_2) &= (g^{R_1}, g^{M_1} \cdot K^{R_1}) \cdot (g^{R_2}, g^{M_2} \cdot K^{R_2}) \\ &= (g^{R_1+R_2}, g^{M_1+M_2} \cdot K^{R_1+R_2}) = E(M_1 + M_2). \end{aligned}$$

In ElectionGuard, guardians publish ElGamal public keys along with a Schnorr proof, which proves knowledge of the corresponding secret key.

2.3 Zero-knowledge proofs

A zero-knowledge proof, also called a ZK proof, is a method for proving knowledge of some secret without revealing the secret itself (Cramer et al., 1994, p. 176-177). There are two roles in a ZK proof; a verifier and a prover. The requirements of a ZK proof are soundness and completeness. For the proof to be sound, the verifier must be able to reliably determine whether or not the prover is actually in possession of the information. For the proof to be complete, the prover must be able to exhibit knowledge of the relevant information to a high degree of probable certainty. A proof is zero-knowledge if it achieves both soundness and completeness without revealing the secret.

2.3.1 Schnorr protocol

A Schnorr proof is a specific *interactive* zero-knowledge proof that can prove knowledge of any discrete logarithm. The original Schnorr proof (C.P., 1990) is an interactive proof, meaning that the prover has to interact with the verifier in real time.

In order for the prover P to prove knowledge of x in $y = g^x$, the following interaction with the verifier V takes place (Kogan, 2019):

1. P commits to a nonce r such that $r \in \mathbb{Z}_q$, and sends $h = g^r \mod p$ to V.
2. V receives h and responds with a challenge $c \in \mathbb{Z}_q$.
3. When P has received c, P sends the response $u = r + cx \mod q$.
4. V checks that $g^u = hy^c$.

Completeness holds if $u = r + cx \mod q$ because

$$g^u \equiv_p g^{r+cx} \equiv_p g^r \cdot (g^x)^c \equiv_p h \cdot y^c. \tag{2.2}$$

When arguing for soundness, it is actually possible for the prover P to maliciously provide a valid Schnorr proof, without knowing the corresponding secret. If P can predict c, it can be done in the following way:

1. P commits to $h = g^r \cdot y^{-c} \mod p$.
2. V responds with $c \in \mathbb{Z}_q$.
3. P responds with $u = r$.
4. V verifies that $g^u \equiv_p h \cdot y^c$.

Which holds because

$$g^u \equiv_p g^r \equiv g^r \cdot y^{-c} \cdot y^c \equiv_p h \cdot y^c$$

Here it is evident that P can construct a valid Schnorr proof, thus violating soundness. However, given that the prime q is large enough, then the probability of this succeeding is negligible. Exploiting this will only succeed with probability $\frac{1}{|\mathbb{Z}_q|}$ assuming that c is random, meaning that soundness holds.

It is also possible that if V can guess the random $r \in \mathbb{Z}_q$, then V is able to derive x from u. The verifier V has computed $u = r + cx \mod q$. When x is the only unknown value, it is evident that this value is computable. Meaning that the probability of leaking the secret is the same as the probability of P maliciously constructing a valid proof without knowing x. It's important to ensure that the size of $\mathbb{Z}_q$ is sufficiently large to prevent guessing r with a non-negligible probability.

2.3.2 Chaum-Pedersen protocol

The Chaum-Pedersen protocol (Chaum & Pedersen, 1993) is, like the Schnorr protocol, an *interactive* zero-knowledge proof. The protocol is used to prove the equality of exponents of

two modular exponentiations with different bases. In ElectionGuard, this is used to prove that an encryption decrypts to a particular value.

In the following, P refers to the prover, and V refers to the verifier. Furthermore, we define the following two equations; $h = g^x$ and $z = m^x$ for generators g and m both in $\in \mathbb{Z}_p^r$. The prover P would like to prove that $\log_g(h) = \log_m(z)$; therefore, the prover and verifier begin the following interaction.

1. P commits to a nonce $u \in \mathbb{Z}_q$ and sends the tuple $(a,b) = (g^u \bmod p, m^u \bmod p)$ to V.
2. V chooses a random challenge $c \in \mathbb{Z}_q$ and sends this to P.
3. P sends $v = u + cx \bmod q$ to V.
4. V checks that $g^v \equiv_q a \cdot h^c$ and that $m^v \equiv_q b \cdot z^c$ holds.

Here it is evident that the equations in step 4 hold due to

$$g^v \equiv_p g^{u+cx} \equiv_p g^u \cdot g^{cx} \equiv_p g^u \cdot (g^x)^c \equiv_p a \cdot h^c$$
$$m^v \equiv_p m^{u+cx} \equiv_p m^u \cdot m^{cx} \equiv_p m^u \cdot (m^x)^c \equiv_p b \cdot z^c.$$

The Chaum-Pedersen protocol can be converted into a non-interactive proof using the Fiat-Shamir heuristic (Fiat & Shamir, 1986).

2.3.3 Converting an interactive proof to a non-interactive proof

A *non-interactive* ZK proof, also called a NIZK proof, is a ZK proof where the communication between P and V consists of a single message from the prover to the verifier. The Fiat-Shamir heuristic (Fiat & Shamir, 1986) converts an *interactive* proof to a *non-interactive* proof. This can be used to convert both the Schnorr protocol and Chaum-Pedersen protocol into a *non-interactive* zero-knowledge proof.

The key idea behind the Fiat-Shamir heuristic is to replace the challenge c that was generated by V. This challenge is replaced with the value of a hash function with the input of all of the previous values sent to V (Kogan, 2019). With respect to the Schnorr proof, this could, for instance, be

$$c = H(g,h,y). \tag{2.3}$$

The proof is conducted the same way as the *non-interactive* version. The key difference is that V does not send c to P, but V instead verifies that equation 2.3 holds. P sends (h,c,u) to V, and V can verify c by computing the hash value itself. Soundness and completeness hold since P has to commit to a specific nonce r before computing c and u. One crucial property of hash functions is their non-invertibility, which means that finding the corresponding input for a given challenge is difficult. This feature ensures that the prover cannot predict the challenge. Ultimately, it is this property that guarantees soundness for the *non-interactive* protocol.

Chapter 3

Key generation protocol

The key generation phase serves the purpose of generating both the joint election public key and the unique key shares. The key shares are used if not all guardians are present at the post-election decryption phase. Along with the key share, each guardian must provide a NIZK proof for each coefficient in the polynomial related to Shamir's secret sharing (Shamir, 1979).

3.1 Joint public key generation

In the election, there are n guardians that all produce independent ElGamal public-private key pairs. This is done by generating a random secret $s_i \in \mathbb{Z}_q$ for each guardian G_i and then creating a public key $K_i = g^{s_i} \bmod p$, as explained in Section 2.2.2. Each of the guardians must provide a Schnorr proof to prove possession of their corresponding secret key, as described in Section 2.3.1.

To avoid a single guardian decrypting individual ballots, the public keys will be multiplied into a joint public key

$$K = \prod_{i=1}^{n} K_i \bmod p.$$

The joint public key K is used to encrypt individual ballots. The encryption scheme has the property that the encrypted ballots can be homomorphically added into a ciphertext tally without having to decrypt the individual votes. This is covered in Section 2.2.2. The secret key, of each guardian, can now be used to compute a partial decryption for each of the ciphertext tallies since each guardian has proven possession of their secret key. Assuming all guardians are available, they'll partially decrypt the ciphertext tally, which is explained in Section 5.2. However, a procedure is needed in case of missing guardians.

3.2 Missing guardians

We need a procedure to compute some partial decryption without revealing the corresponding secret key. This is to ensure that the ciphertext tally and spoiled ballots can be successfully decrypted, even though some of the guardians are unavailable. To successfully decrypt, k out of the total n guardians must be available (Benaloh & Naehrig, 2018, p. 25-26).

To ensure that at least k guardians are needed for decryption, ElectionGuard makes use of Shamir's secret sharing (Shamir, 1979) along with Lagrange coefficients. Generally, a polynomial with k coefficients is of degree $k-1$, meaning that k points will uniquely characterize the polynomial. Therefore, each guardian G_i constructs their polynomial

$$P_i(x) = a_{i,k-1}x^{k-1} + \ldots + a_{i,2}x^2 + a_{i,1}x + a_{i,0},$$

where $a_{i,0} = s_i$, and the rest of the coefficients are random values in $\mathbb{Z}_q$ (Benaloh & Naehrig, 2018, p. 12-13). For each of the coefficients in $P_i(x)$ guardian G_i publishes commitments

$$K_{i,j} = g^{a_{i,j}} \bmod p. \tag{3.1}$$

The commitment

$$K_{i,0} = g^{a_{i,0}} \bmod p = g^{s_i} \bmod p$$

is the public key of G_i. Along with the commitments, G_i publishes encryptions $E_\ell(P_i(\ell))$ such that the receiving guardian G_ℓ is able to decrypt this share (Benaloh & Naehrig, 2018, p. 13-14). The encryption and decryption of key shares are covered in Appendix B. Each commitment is accompanied by a Schnorr proof to prove possession of the corresponding coefficient.

3.2.1 NIZK proof of polynomial coefficients

The guardian G_i issuing the commitments $K_{i,j}$ has to prove that the guardian knows each of the coefficients $a_{i,j}$. This is proved using the Schnorr protocol described in Section 2.3.1, along with the Fiat-Shamir heuristic described in Section 2.3.3. Guardian G_i creates such Schnorr proof for *each* of the k coefficients in polynomial $P_i(x)$.

3.2.2 Verifiability of commitments and share of secret key

Each of the commitments $K_{i,j}$ for guardian G_i include the coefficients of the polynomial in the exponent as seen in equation 3.1. This means that the commitments can be used to verify the share of the secret key (Benaloh & Naehrig, 2018, p. 14). It can be computed as

$$g^{P_i(\ell)} \equiv_p g^{\sum_{j=0}^{k-1} a_{i,j}\ell^j} \equiv_p \prod_{j=0}^{k-1} g^{a_{i,j}\ell^j} \equiv_p \prod_{j=0}^{k-1} \left(g^{a_{i,j}}\right)^{\ell^j} \equiv_p \prod_{j=0}^{k-1} K_{i,j}^{\ell^j}. \tag{3.2}$$

The guardian G_ℓ receiving the share $P_i(\ell)$ can verify that the point $(\ell, P_i(\ell))$ lies on the polynomial without knowing the polynomial itself.

If guardian G_ℓ reports that this does not verify, the guardian G_i that sent the point must publish the value of $P_i(\ell)$ and the nonce $R_{i,\ell}$ that was used to encrypt in the first place. This is covered in Appendix B. The published value must match the encryption under the public key K_ℓ. Furthermore, this value must satisfy equation 3.2; if this is not the case, then guardian G_i is excluded from the election, and the key generation protocol will restart with a new guardian (Benaloh & Naehrig, 2018, p. 14). In the case that the guardian G_i is able to reveal the nonce and

the plaintext fitting the encryption and equation 3.2 holds, then the key-gen protocol can continue (Benaloh & Naehrig, 2018, p. 14).

Given the establishment of the requirement that only k guardians need to be present during decryption, ElectionGuard can now proceed to the next phase, which is the encryption of ballots and proving that the ballots are correctly formed.

Chapter 4

Proof of ballot correctness

Proving the integrity of ballot encryptions consists of two parts. First, a proof that each encrypted option is an encryption of either *zero* or *one*. Second, the sum of all ballot entries in a contest must be equal to the selection limit of that contest. The encryption of ballots is covered in Section 2.2.2.

4.1 Proof of either zero or one

By utilizing the Cramer-Damgård-Schoenmaker technique (Cramer et al., 1994), it becomes possible to prove a disjunction interactively while keeping the true disjunct concealed. In ElectionGuard, the two disjuncts are Chaum-Pedersen proofs. This ensures that each encrypted selection is an encryption of either *zero* or *one*. Each of the two Chaum-Pedersen proofs is covered in Appendix C.

From the prover's point of view, the proof is divided into two cases depending on the value being encrypted. However, the verifier has no way of telling the difference between the two approaches, thus ensuring confidentiality for the voter. As covered before, an encryption using exponential ElGamal over the integers is defined as

$$(\alpha, \beta) = \left(g^R \bmod p, g^M \cdot K^R \bmod p\right)$$

which will be used in the proofs below.

4.1.1 Construction of proof in case of an encryption of zero

If a selection is an encryption of *zero* then the prover P will select random u, v, and w in $\mathbb{Z}_q$ and commit to (Benaloh & Naehrig, 2018, p. 17):

$$(a_0, b_0) = (g^u \bmod p, K^u \bmod p), \tag{4.1}$$

$$(a_1, b_1) = (g^v \bmod p, g^w \cdot K^v \bmod p), \tag{4.2}$$

such that the challenge c can be computed.

$$c = H\left(\overline{Q}, \alpha, \beta, a_0, b_0, a_1, b_1\right).$$

When the prover P has computed the challenge then P has the freedom to pick a c_0 and c_1 such that the following holds

$$c = c_0 + c_1 \bmod q. \tag{4.3}$$

When having an encryption of *zero*, c_1 is chosen to be equal to the random w picked at the beginning of the protocol, and then the value c_0 immediately follows (Benaloh & Naehrig, 2018, p. 18)

$$\begin{aligned} c_0 &= (c - w) \bmod q, \\ c_1 &= w, \\ v_0 &= (u + c_0 \cdot R) \bmod q, \\ v_1 &= (v + c_1 \cdot R) \bmod q. \end{aligned}$$

This has to satisfy the property from equation 4.3.

$$c_0 + c_1 \equiv_q (c - w) + w \equiv_q c. \tag{4.4}$$

Furthermore, it allows the verifier to verify that the following equation holds

$$(a_0\alpha^{c_0}, b_0\beta^{c_0}) \equiv_p (g^{v_0}, K^{v_0}). \tag{4.5}$$

This holds due to

$$(a_0\alpha^{c_0}, b_0\beta^{c_0}) \equiv_p (g^u \cdot (g^R)^{c_0}, K^u \cdot (K^R)^{c_0}) \equiv_p (g^{u+c_0\cdot R}, K^{u+c_0\cdot R}) \equiv_p (g^{v_0}, K^{v_0}).$$

Lastly, it can be checked that the following holds

$$(a_1\alpha^{c_1}, b_1\beta^{c_1}) = (g^{v_1}, g^{c_1}K^{v_1}). \tag{4.6}$$

This holds due to the following.

$$\begin{aligned} (a_1\alpha^{c_1}, b_1\beta^{c_1}) &\equiv_p (g^v \cdot (g^R)^{c_1}, g^w \cdot K^v \cdot (K^R)^{c_1}) \\ &\equiv_p (g^v \cdot (g^R)^{c_1}, g^{c_1} \cdot K^v \cdot (K^R)^{c_1}) \\ &\equiv_p (g^{v+c_1\cdot R}, g^{c_1} \cdot K^{v+c_1\cdot R}) \\ &\equiv_p (g^{v_1}, g^{c_1}K^{v_1}). \end{aligned}$$

In other words, to verify the NIZK proof, one should check that the equations 4.4, 4.5, and 4.6 hold. Notice that the above approach, seen from the prover's perspective, only works for an encryption of *zero*.

4.1.2 Construction of proof in case of an encryption of one

The prover should also be able to provide a NIZK proof of an encryption of one. However, these two proofs should be indistinguishable from one another from the verifier's point of view (Benaloh & Naehrig, 2018, p. 17). This will be clear in the following.

The values (a_0, b_0) and (a_1, b_1) is computed like previously in equations 4.1 and 4.2, but in the reversed order

$$(a_0, b_0) = (g^v \bmod p, g^w \cdot K^v \bmod p),$$
$$(a_1, b_1) = (g^u \bmod p, K^u \bmod p).$$

The challenge c is simply the hash as seen in the previous NIZK proof. When having an encryption of *one*, c_0 is chosen to be equal to the random $q - w$, and the value of c_1 is

$$c_0 = (q - w) \bmod q,$$
$$c_1 = (c + w) \bmod q,$$
$$v_0 = (v + c_0 \cdot R) \bmod q,$$
$$v_1 = (u + c_1 \cdot R) \bmod q.$$

This still has the property of separating the challenge c into two challenges c_1 and c_2 for which the prover can fix c_0 when it is an encryption of *one* (Benaloh & Naehrig, 2018, p. 18). The pseudo challenge c has to have the same property as before

$$c_1 + c_2 \equiv_q (q - w) + (c + w) \equiv_q c + q \equiv_q c.$$

The following equations have to hold for the proof to verify.

$$(a_0 \alpha^{c_0}, b_0 \beta^{c_0}) \equiv_p \left(g^v g^{c_0 R}, g^w K^v \left(gK^R\right)^{c_0}\right) \equiv_p \left(g^v g^{c_0 R}, g^w K^v g^{q-w} K^{c_0 R}\right) \equiv_p (g^{v_0}, K^{v_0}), \quad (4.7)$$

$$(a_1 \alpha^{c_1}, b_1 \beta^{c_1}) \equiv_p \left(g^u g^{c_1 R}, K^u \left(gK^R\right)^{c_1}\right) \equiv_p (g^{v_1}, g^{c_1} K^{v_1}). \quad (4.8)$$

The interesting case to cover here is the right value of the tuple in equation 4.7 in the last equality. This can be rewritten as

$$\left(g^v g^{c_0 R}, g^w K^v g^{q-w} K^{c_0 R}\right) \equiv_p \left(g^v g^{c_0 R}, K^v g^q K^{c_0 R}\right) \equiv_p \left(g^v g^{c_0 R}, K^v K^{c_0 R}\right).$$

When we have a generator of order q then we know that $g^{|G|} \bmod p = g^q \bmod p = 1$ (Damgård, 2022, p. 17). At this point, it is clear why the equality holds.

4.1.3 Summary

Verifying the proof of an encryption of one is the same procedure as verifying an encryption of zero, meaning that having this format, we are only able to tell that it is either an encryption of zero or a one from the verifier's perspective. It becomes clear when comparing equations 4.5 and 4.6

to the equations 4.7 and 4.8. In addition, we will need a method to detect overvotes, undervotes, and null votes since the approach above is only related to the correctness of individual options.

4.2 Satisfying the selection limit

To prove that a contest vote does not exceed the selection limit, the encrypted options must be aggregated and proven to be an encryption of the selection limit L. An encryption can be either of *zero* or *one* and has the following encryption

$$(\alpha_i, \beta_i) = \left(g^{R_i} \bmod p, K^{R_i} \bmod p\right),$$
$$(\alpha_i, \beta_i) = \left(g^{R_i} \bmod p, g \cdot K^{R_i} \bmod p\right)$$

Using this definition one can compute $(\bar{\alpha}, \bar{\beta})$ as (Benaloh & Naehrig, 2018, p. 21)

$$(\bar{\alpha}, \bar{\beta}) = \left(\prod_i \alpha_i \bmod p, \prod_i \beta_i \bmod p\right) = \left(g^R \bmod p, g^L \cdot K^R \bmod p\right). \tag{4.9}$$

Here L represents the selection limit, and R represents the aggregated nonce. If this holds, then a voter has not overvoted. ElectionGuard uses NIZK proofs to prove that $(\bar{\alpha}, \bar{\beta})$ is an encryption of L given the knowledge of R such that L is equal to the selection limit specified in the manifest (Benaloh & Naehrig, 2018, p. 21).

4.2.1 NIZK proof for encryption of selection limit

The aggregated encrypted selections must be proven to be an encryption of L, given the knowledge of the aggregated nonce R. Proving this can be done using the Chaum-Pedersen protocol, which is covered in Section 2.3.2. The main difference is the computation of

$$(a, b) = \left(g^U \bmod p, K^U \bmod p\right),$$

where $U \in \mathbb{Z}_q$. ElectionGuard computes the pseudo challenge C as covered in Section 2.3.3 and defines V based on the knowledge of the aggregated random nonce R.

$$C = H(\overline{Q}, \bar{\alpha}, \bar{\beta}, a, b),$$
$$V = (U + CR) \bmod q.$$

The value V is published such that it can be checked that the following holds

$$g^V \equiv_p a\bar{\alpha}^C, \tag{4.10}$$
$$g^{LC}K^V \equiv_p b\bar{\beta}^C. \tag{4.11}$$

Equation 4.10 holds due to the following

$$a\bar{\alpha}^C \equiv_p g^U \cdot (g^R)^C \equiv_p g^U \cdot g^{RC} \equiv_p g^{U+CR} \equiv_p g^V.$$

Equation 4.11 holds due to the following

$$b\bar{\beta}^C \equiv_p K^U \cdot (g^L \cdot K^R)^C \equiv_p K^U \cdot g^{LC} \cdot K^{RC} \equiv_p g^{LC} \cdot K^{U+RC} \equiv_p g^{LC} \cdot K^V.$$

The verifying party has to check that the equations 4.10 and 4.11 hold. Since this proves that the contest does not exceed the selection limit given the knowledge of the random nonce R and we know from Section 4.1.1 that they can only be an encryption of *zero* or *one* meaning that if the two proofs validate then ElectionGuard can be sure that each contest is correctly formatted without revealing its content.

In the case that an *overvote* is detected by ElectionGuard, then all selectable options are set to an encryption of *zero*, and all placeholder options are set to an encryption of *one* for the integrity of the ballot to be valid. Lastly, this fact is encrypted and added to the contest data field for the corresponding contest in the ballot (Benaloh & Naehrig, 2018, p. 21). The contest data field is then decrypted if the voter chooses to spoil the ballot; this is done to maintain the integrity of the election. If ElectionGuard detects fewer selected non-placeholder options than the selection limit, an *undervote* has occurred. If no options among the non-placeholder options have been selected, then a *null vote* has occurred. When an *undervote* or a *null vote* is encountered, the contest data field is labeled accordingly. It is important to mention that the contest data field is decrypted for spoiled ballots for the voter to confirm that the machine behaves correctly.

Chapter 5

Aggregation and decryption of ballots

At the beginning of the post-election decryption phase, all encrypted ballots, and their associated proofs, are published in the election record (Benaloh & Naehrig, 2018, p. 23). To compute the tally of the election, all cast encrypted ballots must be aggregated and decrypted. Since only a quorum k out of n guardians is required to be available for decryption, additional procedures must be introduced to allow decryption with missing guardians.

This chapter focuses on the decryption of the aggregate ballot, though the same approach for decryption also applies to individual spoiled ballots.

5.1 Aggregating ballots

The aggregated ballot is the homomorphically-combined representation of all selections made for each option in every contest in the election. This means that the aggregation (A,B) must be calculated for every selection based on all cast ballots

$$(A,B) = \left(\prod_i \alpha_i \bmod p, \prod_i \beta_i \bmod p \right). \tag{5.1}$$

This can be done due to the homomorphic additive property described in Section 2.2.2. When the ballots have been aggregated, the guardians can start the decryption ceremony.

5.2 Decryption of aggregated ballot

The threshold encryption scheme ensures that each guardian G_i can participate in the decryption without revealing their secret key s_i. The approach for decryption of exponential ElGamal over the integers is covered in equation 2.1. In summary, one must compute $\frac{\beta}{\alpha^s}$ to recover $M = g^t \bmod p$. In threshold exponential ElGamal decryption is done the same way; however, α^s is calculated without revealing s.

The encryption tuple is defined in equation 5.1, thus decryption can be carried out by

$$M = B \cdot \left(\prod_{i=1}^{n} M_i \right)^{-1} \bmod p, \tag{5.2}$$

where

$$M_i = A^{s_i} \bmod p. \tag{5.3}$$

Equations 5.3 and 5.2 makes the following clear

$$\prod_{i=1}^{n} M_i \equiv_p \prod_{i=1}^{n} A^{s_i} \equiv_p A^{\sum_{i=1}^{n} s_i} \equiv_p \left(g^R\right)^{\sum_{i=1}^{n} s_i} \equiv_p \left(g^{\sum_{i=1}^{n} s_i}\right)^R \equiv_p K^R, \tag{5.4}$$

where R is the nonce of the aggregated selection, K is the joint public key, and M_i is published by guardian G_i. Hence, equation 5.2 correctly computes $M = g^t \bmod p$ where t is the aggregated selections for the specific option. As stated in 2.2.2, the value of t can be found and published. Along with each of the partial decryptions, a NIZK proof is published to prove that the decryption was carried out correctly.

5.2.1 NIZK proof for partial decryption

Each guardian G_i must prove possession of s_i for which $M_i = A^{s_i} \bmod p$ and $K_i = g^{s_i} \bmod p$ to prove that the decryption share is computed correctly (Benaloh & Naehrig, 2018, p. 24). To prove this, a version of the Chaum-Pedersen protocol, described in Section 2.3.2, is used. The main difference is the computation of

$$(a_i, b_i) = \left(g^{u_i} \bmod p, A^{u_i} \bmod p\right),$$

where $u_i \in \mathbb{Z}_q$. The pseudo challenge c_i, which is covered in Section 2.3.3, and given the knowledge of s_i the prover can construct and publish v_i without revealing the value of s_i itself

$$c_i = H\left(\overline{Q}, A, B, a_i, b_i, M_i\right) \qquad\qquad v_i = (u_i + c_i s_i) \bmod q.$$

Then the verifying party will be able to verify that $v_i \in \mathbb{Z}_q$, $a_i \in \mathbb{Z}_p^r$, $b_i \in \mathbb{Z}_p^r$, and that c_i is computed correctly (Benaloh & Naehrig, 2018, p. 24). Furthermore, the following two equations have to be verified

$$g^{v_i} \equiv_p \left(a_i K_i^{c_i}\right) \qquad\qquad A^{v_i} \equiv_p \left(b_i M_i^{c_i}\right). \tag{5.5}$$

The argument for why equation 5.5 hold, is already covered in section 2.3.2. It is not always the case that all guardians are present at the point of decryption; thus, this has to be handled as well.

5.3 Decryption of aggregated ballot with missing guardians

When guardians are absent, it should be possible for k of the n guardians to reconstruct the partial decryption M_i for missing guardian G_i without revealing s_i. Each of the k guardians G_ℓ can compute a share $M_{i,\ell}$ (Benaloh & Naehrig, 2018, p. 25) as

$$M_{i,\ell} = A^{P_i(\ell)} \bmod p.$$

Guardian G_ℓ can compute $M_{i,\ell}$, due to possession of $P_i(\ell)$, from the key-generation ceremony, as described in Section 3.2. Each $M_{i,\ell}$ is used to reconstruct M_i as

$$M_i = \prod_{\ell \in U} M_{i,\ell}^{w_\ell} \bmod p,$$

where U is the set of all guardians participating and w_ℓ is the Lagrange coefficients (Farmer, 2018) which is defined as

$$w_\ell = \frac{\prod_{j \in (U - \{\ell\})} j}{\prod_{j \in (U - \{\ell\})} (j - \ell)} \bmod q = \prod_{j \in (U - \{\ell\})} \frac{j}{j - \ell} \bmod q. \tag{5.6}$$

Equation 5.6 is a special case of Lagrange coefficients since it is evaluated at zero rather than being expressed as a function of x. Lagrange polynomials are defined as

$$L(x) = \sum_{\ell \in U} y_\ell f_\ell(x), \tag{5.7}$$

where the coefficients $f_\ell(x)$ are defined as

$$f_\ell(x) = \frac{(x - x_0)}{(x_\ell - x_0)} \cdots \frac{(x - x_{j-1})}{(x_\ell - x_{j-1})} \frac{(x - x_{j+1})}{(x_\ell - x_{j+1})} \cdots \frac{(x - x_k)}{(x_\ell - x_k)} = \prod_{j \in (U - \{\ell\})} \frac{x - x_j}{x_\ell - x_j}, \tag{5.8}$$

and y_ℓ corresponds to $P_i(\ell)$.

The Lagrange polynomial $L_i(x)$ is equal to the original polynomial $P_i(x)$ since a polynomial of degree $k-1$ is uniquely determined by k points. Therefore, it is desirable to evaluate this Lagrange polynomial at zero since the intersection with the second axis corresponds to the guardian's secret key s_i. Because the Lagrange polynomial is defined by the Lagrange coefficients, we would like to evaluate these at zero as well. This can be expressed as

$$f_\ell(0) = \prod_{j \in (U - \{\ell\})} \frac{-x_j}{x_\ell - x_j} = \prod_{j \in (U - \{\ell\})} \frac{x_j}{x_j - x_\ell} = w_\ell,$$

which corresponds to equation 5.6.

In ElectionGuard, equation 5.7 is implicitly calculated in the exponent of A by using the rules of exponentiation. Having computed M_i, without guardian G_i participating, corresponds to the original definition of M_i since $P_i(0) = s_i$. ElectionGuard can then continue under the assumption

of having all of the M_i and decrypt as described in Section 5.2.

Even though the k guardians can jointly compute M_i, the secret key s_i is never revealed. This means that the calculations above must be carried out for every selection that has to be decrypted in the case that a guardian is missing.

5.3.1 NIZK proof of correctness of partial decryption share

Guardian G_ℓ has to prove knowledge of $P_i(\ell)$ which is used both for the share $M_{i,\ell} = A^{P_i(\ell)} \bmod p$ and for $g^{P_i(\ell)} \bmod p = \prod_{j=0}^{k-1} K_{i,j}^{\ell^j} \bmod p$. This is a Chaum-Pedersen proof as covered in Section 2.3.2, where g and A are used as the base. The guardian commits to the tuple

$$(a_{i,\ell}, b_{i,\ell}) = (g^{u_{i,\ell}} \bmod p, A^{u_{i,\ell}} \bmod p),$$

followed by the computation of the challenge $c_{i,\ell}$ (Benaloh & Naehrig, 2018, p. 25)

$$c_{i,\ell} = H\left(\bar{Q}, A, B, a_{i,\ell}, b_{i,\ell}, M_{i,\ell}\right).$$

The guardian then computes $v_{i,\ell}$ as

$$v_{i,\ell} = (u_{i,\ell} + c_{i,\ell} P_i(\ell)) \bmod q$$

in response to the challenge to prove knowledge of $P_i(\ell)$. Then the verifying party has to verify that $v_{i,\ell} \in \mathbb{Z}_q$, $\alpha_{i,\ell}, \beta_{i,\ell} \in \mathbb{Z}_p^r$, that the challenge $c_{i,\ell}$ is computed correctly and that the following two properties hold (Benaloh & Naehrig, 2018, p. 25-26)

$$g^{v_{i,\ell}} \equiv_p a_{i,\ell} \cdot \left(\prod_{j=0}^{k-1} K_{i,j}^{\ell^j} \right)^{c_{i,\ell}} \tag{5.9}$$

$$A^{v_{i,\ell}} \equiv_p b_{i,\ell} M_{i,\ell}^{c_{i,\ell}}. \tag{5.10}$$

Equation 5.9 holds due to relation from equation 3.2 such that the polynomial can be reconstructed in the exponent from the commitments and the guardian number ℓ. This holds since

$$\begin{aligned} a_{i,\ell} \cdot \left(\prod_{j=0}^{k-1} K_{i,j}^{\ell^j} \right)^{c_{i,\ell}} &\equiv_p g^{u_{i,\ell}} \cdot \left(\prod_{j=0}^{k-1} K_{i,j}^{\ell^j} \right)^{c_{i,\ell}} \equiv_p g^{u_{i,\ell}} \cdot (g^{P_i(\ell)})^{c_{i,\ell}} \\ &\equiv_p g^{u_{i,\ell}} \cdot (g^{c_{i,\ell} P_i(\ell)}) \equiv_p (g^{u_{i,\ell} + c_{i,\ell} P_i(\ell)}) \equiv_p g^{v_{i,\ell}}. \end{aligned}$$

Equation 5.10 holds due to

$$b_{i,\ell} M_{i,\ell}^{c_{i,\ell}} \equiv_p A^{u_{i,\ell}} (A^{P_i(\ell)})^{c_{i,\ell}} \equiv_p A^{u_{i,\ell}} (A^{c_{i,\ell} P_i(\ell)}) \equiv_p A^{u_{i,\ell} + c_{i,\ell} P_i(\ell)} \equiv_p A^{v_{i,\ell}}.$$

When all of the above properties hold, the verifying party can be sure that guardian G_ℓ knows the value of $P_i(\ell)$ and that the partial decryption share is correct without revealing $P_i(\ell)$.

Chapter 6

Building a verifier in Go

The role of an election verifier is to ensure that the election procedure is fair and transparent and that the results of the election accurately reflect the choice of the voters.

There are different types of election verifiers, depending on the electoral system. For example, in a manual voting system, election verifiers can be poll watchers who observe the voting process to ensure that everything is conducted in accordance with the law. However, in ElectionGuard, the verifier's job is that individual voters can verify that their votes have been accurately recorded, and voters and observers can verify that all recorded votes have been accurately counted.

This chapter will outline how we have generated data for benchmarking our verifier, how we have implemented our verifier, and a comparison to another verifier.

6.1 Data and benchmarking methodologies

Throughout the upcoming sections, we heavily rely on data to ensure accurate benchmarks. This data has been generated using the ElectionGuard SDK and is available on GitHub (Data). We generated sample elections for the Hamilton-General election, which includes two contests. The first contest has four non-placeholder selections with a selection limit of two, which results in six total selections. The second contest has two non-placeholder selections with a selection limit of one, which results in three total selections. We generated elections with 10, 25, 100, 200, and 500 ballots, with the spoiled ballot rate being fixed at 30% of the total number of submitted ballots.

For our Go verifier, we implemented the benchmarking ourselves in order for this to be an integrated part of the project. This can be enabled using the `-b` flag, which takes an integer specifying the number of runs. For the MITRE Julia verifier, we used the library BenchmarkingTools. It is important to note that when benchmarking, we do not include deserialization of the election record since this could not be included in the Julia verifier. To measure the performance for a specific sample election, we ran the verifiers 30 times in order to reduce variance and used the mean for comparison. The standard derivation is not present in the plots due to this being negligible. To generate the graphs, we have used Python and Matplotlib. Additionally, the colors have been carefully picked using *Colorbrewer in print: A catalog of color schemes for maps* (Brewer et al., 2003), such that they support color blindness.

6.2 Verification steps

This section highlights two of the steps that need to be verified for any election held using ElectionGuard. These steps are taken from the ElectionGuard specification (Benaloh & Naehrig, 2018, p. 52-71). An overview of all verification steps can be found in Appendix D.

2) Guardian public key validation
An election verifier must confirm the following for each guardian G_i and for each $j \in \mathbb{Z}_k$.
(2.A) The challenge $c_{i,j}$ is correctly computed as $c_{i,j} = H(K_{i,j}, h_{i,j}) \bmod q$.
(2.B) The equation $g^{u_{i,j}} \bmod p = h_{i,j} K_{i,j}^{c_{i,j}} \bmod p$ is satisfied.

Verification step 2 ensures that the two values $c_{i,j}$ and $g^{u_{i,j}}$ have been computed correctly in the Schnorr protocol. This is covered in Section 3.2.1.

7) Correctness of ballot aggregation
An election verifier must confirm for each (non-placeholder) option in each contest in the election manifest that the aggregate encryption (A, B) satisfies
(7.A) $A = \left(\prod_j \alpha_j\right) \bmod p$,
(7.B) $B = \left(\prod_j \beta_j\right) \bmod p$,
where the (α_j, β_j) are the corresponding encryptions on all cast ballots in the election record.

Verification step 7 ensures that the ballots have been aggregated correctly by multiplying all α_j and all β_j. This is covered in Section 5.1.

6.3 Structure of our verifier

The verifier is responsible for verifying ballots, tallies, and NIZK proofs. It is important that the verifier is independent and that the voters and observers have multiple different options when deciding which verifier to use.

Our main goal, when building our verifier, was to ensure that the code was as *analyzable*[1] as possible. The reasoning for this was for our verifier to be able to serve as a reference for other people building a verifier. When building the verifier, we found our original implementation to be quite slow, which we found troublesome. If a verifier takes too long to complete its verification, it could delay the election results and create uncertainty and doubts about the integrity of the election. Therefore, the *running time* of our verifier became a focus as well.

The source code for our verifier can be found on GitHub (Verifier).

6.3.1 Analyzability

For our verifier to be a reliable reference and to establish its correctness according to the specification, it must be analyzable. Analyzability is the ability to understand software (Christensen, 2020,

[1] Analyzability is the ability to *understand* software (Christensen, 2020, p. 49-50)

p. 49-50). If people cannot analyze our verifier implementation, they can not use it as a reference. We increased analyzability by creating a method for each verification step. This leads to reduced efficiency as some steps may iterate over the same data and could potentially be combined into fewer for-loops. This is a trade-off between *analyzability* and *running time*.

Another way analyzability has been increased is by using clean code principles (Martin, 2009) and patterns (Christensen, 2020). Clean code principles have been applied to ensure that the naming of variables, files, and methods reflects their intended behavior.

6.3.2 Running time

In order to optimize the running time of our verifier, we have used parallelization to fully utilize the available CPU resources. This is done by concurrently verifying the steps from 1 through 19. In our implementation, we have experienced that steps 4, 5, 12, and 13 and have the most significant impact on the overall running time, which can be observed in Figure E.1. Therefore we have taken action to further optimize the running time of these steps. For these steps, we have chosen to partition the list of ballots and assigned a separate thread to handle each partition. The amount of partitions depends on the number of logical cores on the machine running the verifier. For example, if the machine has 20 logical cores steps 4, 5, 12, and 13 will each spawn 20 threads. This is done in order to fully utilize the CPU for the whole verification process. In Figure E.1, it can be seen how these partitions have reduced the running time of the individual steps.

The graph below illustrates the performance gain achieved by concurrently verifying all steps and employing the ballot partitioning technique mentioned above. This graph has been generated using the data and method described in Section 6.1.

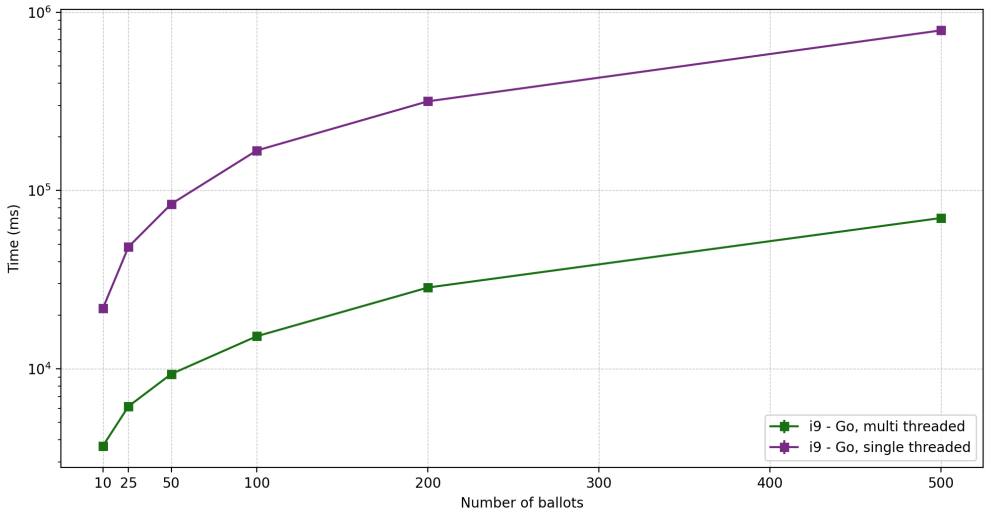

Figure 6.1: Comparison of total running time for Go verifier with i9-10850K

As seen in Figure 6.1, the use of parallelization has improved the running time by a factor of ten when running the verifier on an i9-10850K which has 20 logical cores.

6.4 Comparison with MITRE verifier

To evaluate the performance of our implementation, we compared our verifier with the MITRE ElectionGuard Verifier. The MITRE verifier is an independent ElectionGuard verifier implemented by MITRE in the Julia programming language, which is a functional language. Their verifier utilizes parallelization as well; thus, a direct comparison to our Go verifier can be made.

We experienced that the verifier's running time varied across different computer systems, so we compared it on two machines: an x86-based i9-10850K CPU and an ARM-based M1 CPU. This graph has been generated by using the method and data described in Section 6.1.

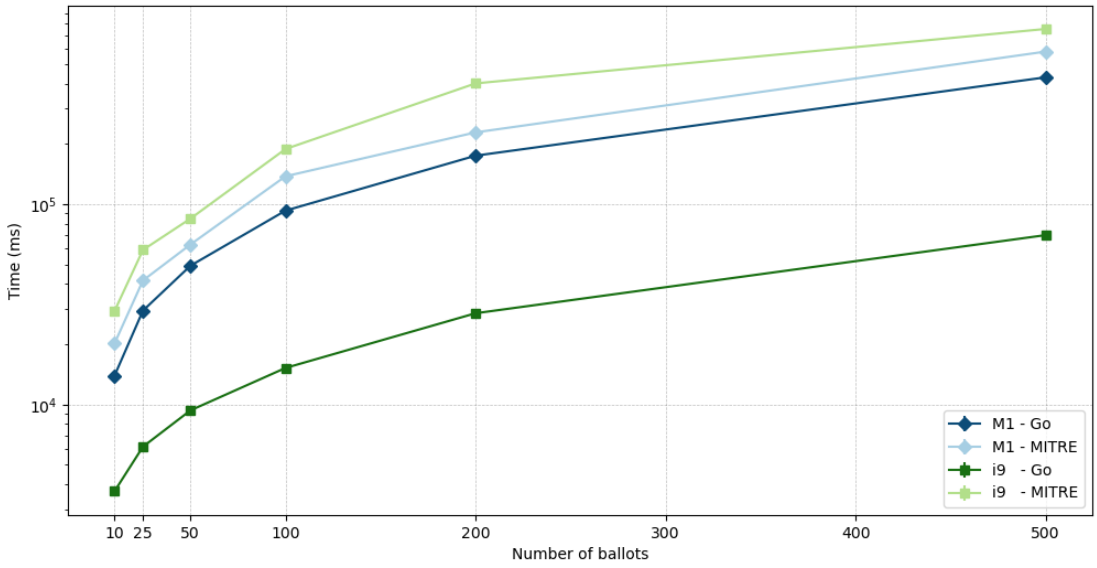

Figure 6.2: Comparison of MITRE and Go verifiers with i9-10850K and M1

As seen in Figure 6.2, lighter colors are the MITRE verifier's performance, and darker colors are our Go verifier's performance. Our benchmarks indicate that our implementation outperforms the MITRE verifier on both the M1 and the i9-10850K. It is clear that the performance difference between MITRE and Go is most significant for the i9-10850K with a factor of approximately 12. It seems like the i9 processor benefits from having more logical cores, which explains the performance difference between the M1 and i9. Furthermore, it is clear that the running time, as a function of the number of ballots, scales more or less the same. They only seem to differ by a constant factor. Furthermore, our comparisons indicate that the running time of the system seems to scale linearly with the number of ballots for both verifiers.

Chapter 7

Exploration of elliptic curves

ElectionGuard uses exponential ElGamal over the integers rather than exponential ElGamal over elliptic curves, even though elliptic curves are known to be more efficient (Hankerson et al., 2003, p. 15-19). This is done to simplify the creation of election verifiers without the need for specialized tools and dependencies (Benaloh & Naehrig, 2018, p. 7).

This chapter delves into the elliptic curve theory, followed by a short explanation of key generation, encryption, and decryption for ElGamal over elliptic curves. Lastly, we have implemented and benchmarked ElGamal over both the integers and elliptic curves with a focus on how ElectionGuard uses ElGamal.

7.1 Elliptic curves

This section is based on *Elliptic curves: Number theory and cryptography* (Washington, 2008) and *Guide to elliptic curve cryptography* (Hankerson et al., 2003) and will briefly explain what elliptic curves are, and how they can be used for cryptography.

7.1.1 Weierstrass Equation

An elliptic curve E over a field $\mathbb{F}$ is the solutions (x,y) to an equation of the form

$$y^2 = x^3 + Ax + B, \tag{7.1}$$

which is the Weierstrass equation, where A and B are constants in some field $\mathbb{F}$. A field is a set equipped with two operations; addition and multiplication. Both have to satisfy associativity, commutativity, distributivity, and the existence of both inverses and the identity element. To ensure distinct roots of this cubic, it should hold that the discriminant $4A^3 + 27B^2 \neq 0$.

When defining points with coordinates in some field $\mathbb{F}$ the notation $E(\mathbb{F})$ is used and corresponds to the following set:

$$E(\mathbb{F}) = \{\infty\} \cup \left\{(x,y) \in \mathbb{F} \times \mathbb{F} \mid y^2 = x^3 + Ax + B\right\}. \tag{7.2}$$

7.1.2 Group law

Points in $E(\mathbb{F})$ form an additive finite abelian group with ∞ as the identity element. This implies that if P_1 and P_2 are any points in $E(\mathbb{F})$, then $P_1 + P_2 = P_3$ where P_3 also belongs to $E(\mathbb{F})$. This addition of points is not the same as adding the coordinates of the points. Instead, this is achieved through a series of geometric operations that involve drawing a line through P_1 and P_2, finding the third point of intersection with the curve, and reflecting this third point across the x-axis. For specific elliptic curves, this process can be formally described as:

Let $P_1 = (x_1, y_1) \in E(\mathbb{F})$ and $P_2(x_2, y_2) \in E(\mathbb{F})$, where $P_1 \neq \pm P_2$. Then $P_1 + P_2 = (x_3, y_3)$, where

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \quad \text{and} \quad y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1. \tag{7.3}$$

Additionally, it is possible to do scalar multiplication with points (Washington, 2008, p. 18). Let k be a positive integer, and let P be a point on an elliptic curve. The following procedure computes kP

1. Start with $a = k, B = \infty, C = P$.
2. If a is even, let $a = a/2$, and let $B = B, C = 2C$.
3. If a is odd, let $a = a - 1$, and let $B = B + C, C = C$.
4. If $a \neq 0$, go to step 2.
5. Output B.

The output B then equals kP. When working with elliptic curves over a large finite field, figuring out the value of k from given points P and kP is computationally hard. This is known as the discrete logarithm problem for elliptic curves and is the basis for using elliptic curves in cryptography.

7.2 Exponential ElGamal over elliptic curves

7.2.1 Key generation

In order to use elliptic curves for encryption (Washington, 2008, p. 175), the sender S and the receiver R must agree on an elliptic curve E over a finite field F_p. This curve is picked such that the discrete log problem is hard. Once this is in place, R will pick a point P on the elliptic curve E. Then R will construct secret key s and point B, which later will be used for encryption

$$B = sP. \tag{7.4}$$

Here F_q, E, P, and B serves as the public key for R.

7.2.2 Encryption & decryption

For the sender S wanting to send an encrypted message m, the message m is encoded as a point $M \in E(F_p)$. This is trivial in the case of integer encryption, where the integer is multiplied by the base point P. Furthermore, an integer nonce k is chosen in order to introduce randomness. The encryption tuple is then defined as (Washington, 2008, p. 175)

$$(M_1, M_2) = (kP, M + kB). \tag{7.5}$$

The tuple is sent to the intended receiver R, which can be decrypted using the private key s

$$M = M_2 - sM_1. \tag{7.6}$$

Which holds since

$$M_2 - sM_1 = (M + kB) - s(kP) = M + k(sP) - skP = M.$$

The receiver R has not recovered m but instead mP and now has to solve the elliptic curve discrete log problem. However, as with exponential ElGamal over the integers, when the set of options for m is small, it is easy to recover m using linear search or a pre-computed table.

The homomorphic additive property also holds for exponential ElGamal over elliptic curves. Since the group the points lie within is defined under addition, the homomorphic property now states that for two messages m and m', it should hold that the encryptions $E(m) + E(m') = E(m + m')$ hold. This holds due to the following

$$\begin{aligned} E(m) + E(m') &= (M_1, M_2) + (M_1', M_2') = (kP + k'P, M + kB + M' + k'B) \\ &= ((k + k')P, (M + M') + (k + k')B) = E(m + m'). \end{aligned}$$

This means that exponential ElGamal over elliptic curves can be used in ElectionGuard in the same fashion as exponential ElGamal over the integers.

7.3 Comparison to exponential ElGamal over the integers

For benchmarking, we are interested in both the running time and the size of the ballots since both of these influence the performance of ElectionGuard.

When comparing the storage usage, we went through an ElectionGuard ballot, counting every encryption. This allows us to estimate how much storage exponential ElGamal over the integers uses compared to exponential ElGamal over elliptic curves. To benchmark the running time of both schemes, we implemented them in Go, where the code can be found on GitHub (Benchmarking). In order to implement exponential ElGamal over elliptic curves, we relied on the elliptic package[1], which allowed for easy key generation and point arithmetic as described in Section 7.1.2. We choose to use the P256 elliptic curve, which is defined over a 256-bit field with

[1] https://pkg.go.dev/crypto/elliptic#P256

a 256-bit order. This provides the same 128-bit security strength as the exponential ElGamal over the integers used by ElectionGuard (BlueKrypt, 2020).

7.3.1 Benchmarking of encryption & decryption speed

Our benchmarking consisted of encrypting 1000 *zeroes* and *ones* for both exponential ElGamal over the integers and elliptic curves because only *zeros* and *ones* are encrypted in ElectionGuard. It is important to note that we do not have a full decryption since we do not solve the discrete log problems $M = mP$ and $M = g^m \bmod p$. However, due to a small message space, this can be solved using a pre-computed table. As a result, it can be computed in constant time in practice. The benchmark results are shown in the figure below.

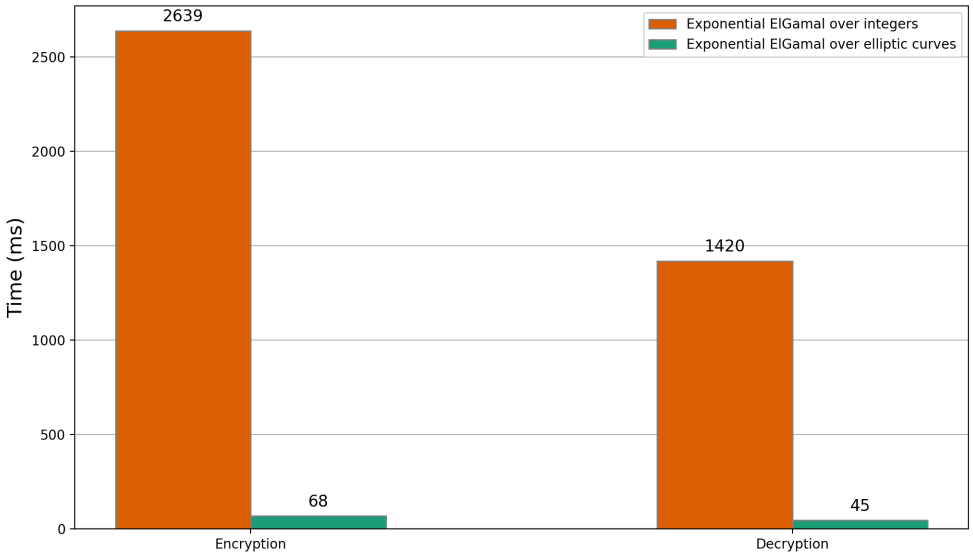

Figure 7.1: Comparison of exponential ElGamal over both the integers and elliptic curves

In Figure 7.1, it is evident that the exponential ElGamal over elliptic curves is way faster; approximately 39 times faster for encryption and 31 times faster for decryption.

It is not only the running time but also storage usage that is of importance when deciding which encryption scheme to use.

7.3.2 Storage usage

In the case of having an election with many voters and options, then the size of the ciphertexts has a huge impact on the total size of the election record. Therefore we would like to express the total size of the ciphertexts of each ballot. In ElectionGuard, each entry in an ElGamal ciphertext is of

size 4096 bits due to the size of the modulus p. For each contest in the ballots, we have:

$$\begin{aligned} 2 \cdot 4096 \text{ bits} &= 8192 \text{ bits} && \text{(Aggregation of ciphertext)} \\ 2 \cdot 4096 \text{ bits} &= 8192 \text{ bits} && \text{(Selection limit proof)} \\ 2 \cdot 4096 \text{ bits} &= 8192 \text{ bits}. && \text{(Contest data encryption)} \end{aligned}$$

For each selection (including the placeholder selections) in each contest, we have:

$$\begin{aligned} 2 \cdot 4096 \text{ bits} &= 8192 \text{ bits} && \text{(Ciphertext)} \\ 4 \cdot 4096 \text{ bits} &= 16384 \text{ bits}. && \text{(Cramer-Damgaard proof)} \end{aligned}$$

As an example, we use the Danish municipal election in Aarhus to get a sense of how much disk space is used to store the ballots. Data is taken from KMD 2021 (KMDValg, 2021), which can be found in Appendix F. In the following, we do not include spoiled ballots.

For this election, we have a selection limit of 1 with 199 selections (including placeholder options), with 196.274 cast ballots. Using the general observations above, we first make the calculations for exponential ElGamal over the integers.

$$(3 \cdot 8192 \text{ bits} + 6 \cdot 4096 \text{ bits} \cdot 199) \cdot 196.274 \text{ bits} \approx 120 \text{ GB}.$$

Using the P256 elliptic curve gives 512 bits for each point in the tuple since a point consists of two 256-bit coordinates. This means we will have $512 \text{ bits} \cdot 2 = 1024$ bits for each ciphertext, which reduces the ciphertext size by a factor of 4.

$$\frac{120 \text{ GB}}{4} = 30 \text{ GB}.$$

For elliptic curves, it is also possible to compress the ciphertext space by removing the y-coordinate and representing it by a single bit, which will cut the ciphertext size in half. This is possible since there are two possible y-coordinates for each x-coordinate. For point compression, there is a tradeoff between storage usage and encryption/decryption speed since each point will have to be decompressed again. Nevertheless, it is clear that it makes a significant difference in terms of storage exponential ElGamal over the integers is replaced with exponential ElGamal over elliptic curves.

7.3.3 Summary

We can conclude from our testing that exponential ElGamal over elliptic curves is faster by a factor of about 39 compared to exponential ElGamal over the integers. Furthermore, we would expect the storage to be reduced by about a factor of 4 using exponential ElGamal over the integers. At first sight, it seems unreasonable not to use elliptic curves; however, as ElectionGuard states, the use of elliptic curves introduces additional complexity and dependencies to the verifier, which is undesirable.

Chapter 8

Conclusion & future work

8.1 Conclusion

In conclusion, this thesis has delved into the potential of electronic voting and analyzed ElectionGuard. Our main contribution was to implement a verifier for ElectionGuard, which would allow for easier and more accurate implementations of future independent verifiers.

Throughout the thesis, a detailed analysis of ElectionGuard was conducted, exploring its technical aspects. The analysis covered various key elements of ElectionGuard, starting with an introduction to the general terminology, structure, and protocols. The theoretical foundations of exponential ElGamal over the integers and zero-knowledge proofs were also explained, which provides the ability to prove the integrity of a ballot while preserving the confidentiality of the voter. Subsequent chapters discussed the key generation protocols, ballot correctness proofs, ballot aggregation, and verifiable decryption processes employed by ElectionGuard. Where it is clear that using the joint public key for ballot encryption provides the voters with privacy because no single guardian is able to decrypt individual ballots. Furthermore, ElectionGuard uses multiple Chaum-Pedersen proofs to establish the validity of each selection, verifying that it is either a *zero* or a *one* and ensuring that the ballot does not exceed the selection limit. This approach guarantees the integrity of the election. Following this, ElectionGuard can safely aggregate the ballots by taking advantage of the homomorphic additive property of exponential ElGamal.

The thesis then transitioned into providing practical insights by presenting an ElectionGuard verifier implementation in Go, which can be used as a reference point for future verifiers. An overview of the necessary verification steps was provided, followed by a discussion of our implementation. Our verifier was shown to outperform MITRE's Julia verifier in a comparison between the two by a factor of approximately 12 on an i9-10850K. The importance of running time in a verifier's execution was emphasized, as a slow verifier could significantly delay election results, leading to frustration and mistrust among voters. Additionally, *analyzability* was also emphasized, as a verifier, that no one can analyze provides no assurance of its correctness.

Furthermore, the thesis explored potential improvements to ElectionGuard's performance. The fundamental theory of elliptic curves was introduced and presented as an alternative to the integers for exponential ElGamal, which was explained and experimentally evaluated. The results showed that exponential ElGamal over elliptic curves offered significant improvements in terms of

speed and reduced ciphertext size while maintaining the same level of security. More concretely, the speed was increased by a factor of 39, and the ciphertext size was reduced by a factor of 4. The use of elliptic curves appears to hold great potential. However, it introduces additional complexity and dependencies to the verifier, which is undesirable.

In conclusion, this thesis has provided a comprehensive analysis of ElectionGuard, covering its technical foundations, implementation considerations, and potential enhancements. The implementation of our verifier and the insights presented in this study contribute to the advancement of ElectionGuard, with the ultimate goal of ensuring fair and trustworthy elections in democratic societies.

8.2 Future work

We would like to cover ideas for future work in order to make both our verifier and ElectionGuard more efficient. First, we have already covered the idea of substituting exponential ElGamal over the integers with exponential ElGamal over elliptic curves. However, it is not only elliptic curves that can improve the efficiency of ElectionGuard. In the following, we will propose alternatives to improve the efficiency of the ElectionGuard.

In ElectionGuard, the election record is published after the election, meaning that the verifier can only start after the election is finished. In an election, the integrity of the election must be verified as soon as possible; therefore, it is desirable to publish data *during* the election. For instance, the spoiled and cast ballots can be verified at the point at which they are submitted. Furthermore, this would also enable us to detect irregularities at an earlier point rather than after the election is held. The data size, as covered in 7.3.2, could become a problem as the size of the election grows. Publishing ballots along the way will mitigate the problem of downloading the data rather than having everything downloaded at once at the end of the election.

Another approach to reducing the running time of the verifier is to distribute the (sub)steps among multiple machines. The bottleneck of step 4, as covered in Section 6.3.2, could be reduced by assigning more computational power. In case it is not realistic to utilize more power on a single machine, this could be distributed among several machines. However, this approach would decrease the *analyzability* of the verifier and may not be a suitable tradeoff between *analyzability* and *running time*.

Lastly, it is important to mention that ElectionGuard continuously improves and changes its specification. These changes propagate to verifiers, which means that it will be necessary to update our verifier to remain relevant. As for now, ElectionGuard 2.0 promises to improve encryption speed, reduce the complexity of the verifier, introduce range proofs, and support voting by email[1].

[1] ElectionGuard Roadmap 2023

Acknowledgments

We would like to express our sincere gratitude to the individuals who have contributed to the successful completion of this bachelor's thesis. First and foremost, we would like to thank our advisor, Diego F. Aranha, for his valuable guidance, support, and expertise throughout the research process. His feedback and input have greatly influenced the direction of this work. We would also like to extend our sincere appreciation to the ElectionGuard team for their invaluable contributions. In particular, we are grateful for their time and effort in engaging with us through discussions and for listening to our experiences building a verifier along with our suggestions for improving this process. Additionally, we would like to acknowledge ElectionGuard for their documentation, which has been an essential resource in understanding the system. Their comprehensive materials have contributed to the depth and accuracy of this thesis.

Bibliography

Christensen, H. (2020). *Flexible, reliable, distributed software 2nd edition: Still using patterns and agile development*. LeanPub.com.

Benaloh, J., & Naehrig, M. (2018). *Electionguard specification v1.1*. Retrieved February 1, 2023, from http://www.electionguard.vote/spec/

C.P., S. (1990). Efficient identification and signatures for smart cards. *435*. https://link.springer.com/content/pdf/10.1007%5C%2F0-387-34805-0_22.pdf

Chaum, D., & Pedersen, T. P. (1993). Wallet databases with observers. In E. F. Brickell (Ed.), *Advances in cryptology — crypto' 92* (pp. 89–105). Springer Berlin Heidelberg.

Cramer, R., Damgård, I., & Schoenmakers, B. (1994). Proofs of partial knowledge and simplified design of witness hiding protocols. In Y. G. Desmedt (Ed.), *Advances in cryptology — crypto '94* (pp. 174–187). Springer Berlin Heidelberg.

Fiat, A., & Shamir, A. (1986). How to prove yourself: Practical solutions to identification and signature problems. In A. M. Odlyzko (Ed.), *Advances in cryptology - CRYPTO '86, santa barbara, california, usa, 1986, proceedings* (pp. 186–194). Springer. https://doi.org/10.1007/3-540-47721-7_12

ElGamal, T. (1984). A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley & D. Chaum (Eds.), *Advances in cryptology, proceedings of CRYPTO '84, santa barbara, california, usa, august 19-22, 1984, proceedings* (pp. 10–18). Springer. https://doi.org/10.1007/3-540-39568-7_2

Damgård, I. (2022). *Introduction to cryptography*. Ivan Damgård.

Wikipedia. (2023). *Schnorr group*. Retrieved March 20, 2023, from https://en.wikipedia.org/wiki/Schnorr_group

Shanks, D. (1969). Class number, a theory of factorization, and genera. In *Number theory institute, 1969* (pp. 415–440).

Kogan, D. (2019). *Lecture 5: Proofs of knowledge, schnorr's protocol, nizk*. Retrieved March 3, 2023, from https://crypto.stanford.edu/cs355/19sp/lec5.pdf

Shamir, A. (1979). How to share a secret. *Commun. ACM*, *22*(11), 612–613. https://doi.org/10.1145/359168.359176

Farmer, J. (2018). *Lagrange's interpolation formula*. Retrieved February 17, 2023, from https://files.eric.ed.gov/fulltext/EJ1231189.pdf

Brewer, C. A., Hatchard, G. W., & Harrower, M. A. (2003). Colorbrewer in print: A catalog of color schemes for maps. *Cartography and Geographic Information Science*, *30*(1), 5–32. https://doi.org/10.1559/152304003100010929

Martin, R. (2009). *Clean code: A handbook of agile software craftsmanship*. Prentice Hall. https://books.google.dk/books?id=dwSfGQAACAAJ

Hankerson, D., Menezes, A. J., & Vanstone, S. (2003). *Guide to elliptic curve cryptography*. Springer-Verlag.

Washington, L. C. (2008). *Elliptic curves: Number theory and cryptography, second edition* (2nd ed.). Chapman & Hall/CRC.

BlueKrypt. (2020). *Cryptographic key length recommendation*. Retrieved May 2, 2023, from https://www.keylength.com/en/4/

KMDValg. (2021). *Kommunalvalg den 16. november 2021*. Retrieved April 20, 2023, from https://kmdvalg.dk/KV/2021/K84713751.htm

Weisstein, E. (2023). Discrete logarithm. https://mathworld.wolfram.com/DiscreteLogarithm.html

Appendix A

Discrete logarithms

To explain the definition of discrete logarithms, some basic concepts need to be explained first (Weisstein, 2023).

Relative prime: Two positive integers are relative prime to each other if their greatest common divisor is 1.
Modular congruence: Two integers a and b are congruent modulo n, if n is a divisor of their difference, meaning there exists an integer k such that $a - b = kn$. This is denoted as

$$a \equiv b \pmod{n}.$$

Primitive root: If n is some positive integer, then g is a primitive root mod n if for every number a that is relative prime to n, there exists some integer z such that the following relation holds

$$a \equiv g^z \pmod{n}.$$

Totient function: The totient function is a function denoted as $\phi(n)$, that denotes the count of integers from $0, ..., n$ that are relatively prime with n itself.
Discrete logarithm: Through the above definitions, a description of discrete logarithms can be given .
If a is some arbitrary integer that is relative prime to n which g is a primitive root of, then there exist exists amongst the numbers $0, 1, 2, ..., \phi(n) - 1$, one integer μ such that

$$a \equiv g^{\mu} \pmod{n}.$$

The number μ is then called the discrete logarithm of a with respect to the base g modulo n.

Appendix B

Encryption and decryption of key shares

When the shares $P_i(\ell)$ from G_i are sent out to each of the other guardians G_ℓ for $i \neq \ell$, then they must be encrypted to avoid revealing these shares to the other guardians. This is done by generating a random nonce $R_{i,\ell} \in \mathbb{Z}_q$ and using the public key K_ℓ for the receiving guarding G_ℓ in the following way (Benaloh & Naehrig, 2018, p. 13-14)

$$(\alpha_{i,\ell}, \beta_{i,\ell}) = \left(g^{R_{i,\ell}} \bmod p, K_\ell^{R_{i,\ell}} \bmod p\right). \tag{B.1}$$

In ElectionGuard, this is used to generate a MAC-key and an encryption key for the share being sent

$$k_{i,\ell} = H\left(\alpha_{i,\ell}, \beta_{i,\ell}\right). \tag{B.2}$$

Using this value, we can generate a MAC key k_0 and an encryption key k_1 (Benaloh & Naehrig, 2018, p. 13-14)

$$k_0 = \text{HMAC}\left(k_{i,\ell}, [0]_{32} \mid [256]_{32}\right) \tag{B.3}$$
$$k_1 = \text{HMAC}\left(k_{i,\ell}, [1]_{32} \mid [256]_{32}\right). \tag{B.4}$$

Then the encryption (Benaloh & Naehrig, 2018, p. 14) of $P_i(\ell)$ is

$$E_\ell\left(P_i(\ell)\right) = \left(C_{i,\ell,0}, C_{i,\ell,1}, C_{i,\ell,2}\right), \tag{B.5}$$

where

$$\begin{aligned} C_{i,\ell,0} &= g^{R_{i,\ell}} \bmod p \\ C_{i,\ell,1} &= [P_i(\ell)]_{256} \oplus k_1 \\ C_{i,\ell,2} &= \text{HMAC}\left(k_0, C_{i,\ell,0} \mid C_{i,\ell,1}\right). \end{aligned}$$

Notice that the encryption is based on a random nonce such that the ciphertext would appear completely random, and the same plaintext will not be encrypted to the same ciphertext.

When the triplet from equation B.5 has been calculated, it can be sent to the respective guardian.

The receiving guardian has the corresponding secret key meaning the respective guardian can compute $\beta_{i,\ell}$ from equation B.1

$$\beta_{i,\ell} = (g^{R_{i,\ell}} \bmod p)^{s_i} \bmod p = (g^{R_{i,\ell}})^{s_i} \bmod p = (g^{s_i})^{R_{i,\ell}} \bmod p = K_\ell^{R_{i,\ell}} \bmod p.$$

Furthermore, the guardian already received $\alpha_{i,\ell}$ since

$$\alpha_{i,\ell} = g^{R_{i,\ell}} \bmod p = C_{i,\ell,0},$$

meaning that now the receiving guardian can hash and compute k_0 and k_1 from the equations B.2, B.3 and B.4, respectively. At this point, the guardian can use $C_{i,\ell,2}$ to verify the MAC. After this, it can decrypt and get hold of the value P by computing

$$[P_i(\ell)]_{256} = C_{i,\ell,1} \oplus k_1, \tag{B.6}$$

and know the guardian G_ℓ is able to verify this share using the commitments, which are described in Section 3.2.2.

Appendix C

Chaum-Pedersen proofs for ElGamal encryption pair

This chapter briefly covers the general Chaum-Pedersen proof, which can be used to prove that an encryption is of a particular value. This can be combined using the Cramer-Damgård-Schoenmaker technique to prove that an encryption is of either *zero* or *one*. This is covered in Section 4.1.

C.1 Encryption of zero

For an encryption of *zero*, we have $m = 0$

$$(\alpha, \beta) = \left(g^R \bmod p, g^m \cdot K^R \bmod p\right) = \left(g^R \bmod p, K^R \bmod p\right).$$

Given the knowledge of the random nonce R, the voter can construct a NIZK proof that proves that a vote is *zero*. First, the prover picks a random $u \in \mathbb{Z}_q$. Then the prover can construct the following commitment

$$(a, b) = (g^u \bmod p, K^u \bmod p). \tag{C.1}$$

It is important to mention that we pick this random nonce u in order to avoid revealing m in case the protocol is run twice with the same u. This can be seen in the equation below.

$$R = \frac{u_1 - u_2}{c_1 - c_2}$$

which will reveal the random nonce R, and this will lead to the decryption of the message m. This will not be covered here but can be found in the lecture *Proofs of Knowledge, Schnorr's protocol, NIZK* (Kogan, 2019).

A challenge can then be computed using the Fiat-Shamir heuristic (Fiat & Shamir, 1986)

$$c = H(\overline{Q}, \alpha, \beta, a, b).$$

Where $\overline{Q}$ is the extended base hash and α, β, a, and b are defined in the equations above. The base hash Q is simply a hash of the parameters in the manifest. This includes p, q, g, n, k, and the date among other things (Benaloh & Naehrig, 2018, p. 10). The extended base hash $\overline{Q}$ is simply the hash of Q and the joint public key of the election K.
The prover will then construct

$$v = (u + cR) \bmod q,$$

to prove knowledge of R. Then the verifying party has to check that the two following equations are true

$$g^v \equiv_p a \cdot \alpha^c \tag{C.2}$$
$$K^v \equiv_p b \cdot \beta^c. \tag{C.3}$$

Equation C.2 holds since

$$g^v \equiv_p g^{u+cR} \equiv_p g^u \cdot g^{Rc} \equiv_p g^u \cdot (g^R)^c \equiv_p a \cdot \alpha^c,$$

and the equation can only be satisfied given the knowledge of the random nonce R. Equation C.3 holds since

$$K^v \equiv_p K^{u+cR} \equiv_p K^u \cdot K^{Rc} \equiv_p K^u \cdot (K^R)^c \equiv_p b \cdot \beta^c,$$

which is only satisfied when $m = 0$.

It is important to note that this can only be proven given the knowledge of the encryption nonce R that was used to do the encryption in the first place. However, this is not revealed in the zero-knowledge proof.

C.2 Encryption of one

The proof of having an encryption of *one* is close to the same proof as for an encryption of *zero*. When the vote is an encryption of *one*, it has the following format:

$$(\alpha, \beta) = \left(g^R \bmod p, g^1 \cdot K^R \bmod p\right) = \left(g^R \bmod p, g \cdot K^R \bmod p\right).$$

In the proof, ElectionGuard substitutes $\frac{\beta}{g}$ for β such that the same proof as for zero can be carried out

$$(\alpha, \frac{\beta}{g}) = \left(g^R \bmod p, K^R \bmod p\right).$$

Since the value of c is publicly known, the verifier can verify the following (Benaloh & Naehrig, 2018, p. 17):

$$g^c \cdot K^v \equiv_p b \cdot \beta^c. \tag{C.4}$$

If this equality holds, the verifier can confirm that this is an encryption of one given that the prover knows that value of R.

Appendix D

Verification steps

In this section, we will cover an overview of the steps that need to be verified. This is described based on the ElectionGuard official specification (Benaloh & Naehrig, 2018, p. 52-71).

1) Parameter validation
An election verifier must verify that it uses the standard baseline parameters, which may be hardcoded.
(1.A) The large prime p is as defined in
(1.B) q is equal to $q = 2^{256} - 189$
(1.C) $r = \frac{p-1}{q}$
(1.D) The generator g is as defined

The parameters need to be validated to ensure that safe-primes have been used and that g is a generator of q order subgroup of $\mathbb{Z}_p^*$.

2) Guardian public key validation
An election verifier must confirm the following for each guardian G_i and for each $j \in \mathbb{Z}_k$.
(2.A) The challenge $c_{i,j}$ is correctly computed as $c_{i,j} = H(K_{i,j}, h_{i,j}) \bmod q$.
(2.B) The equation $g^{u_{i,j}} \bmod p = h_{i,j} K_{i,j}^{c_{i,j}} \bmod p$ is satisfied.

These two checks ensure that the two values $c_{i,j}$ and $g^{u_{i,j}}$ have been computed correctly in the Schnorr protocol. This is covered in Section 3.2.1.

3) Election public key validation
An election verifier must verify the correct computation of the joint election public key and extended base hash.
(3.A) $K = \prod_{i=1}^{n} K_i \bmod p$
(3.B) $\bar{Q} = H(Q, K)$.

The election public key needs to be verified to make sure it's computed as described in Section 3.1.

4) Correctness of selection encryptions
An election verifier must confirm the following for each possible selection on each cast ballot
(4.A) The given values $\alpha, \beta, a_0, b_0, a_1$, and b_1 are all in the set $\mathbb{Z}_p^r$. (A value x is in $\mathbb{Z}_p^r$ if and only if x is an integer such that $0 \leq x < p$ and $x^q \bmod p = 1$ is satisfied.)
(4.B) The challenge c is correctly computed as $c = H\left(\bar{Q}, \alpha, \beta, a_0, b_0, a_1, b_1\right)$.
(4.C) The given values c_0, c_1, v_0, and v_1 are each in the set $\mathbb{Z}_q$. (A value x is in $\mathbb{Z}_q$ if and only if x is an integer such that $0 \leq x < q$.)
(4.D) The equation $c = (c_0 + c_1) \bmod q$ is satisfied.
(4.E) The equation $g^{v_0} \bmod p = a_0 \alpha^{c_0} \bmod p$ is satisfied.
(4.F) The equation $g^{v_1} \bmod p = a_1 \alpha^{c_1} \bmod p$ is satisfied.
(4.G) The equation $K^{v_0} \bmod p = b_0 \beta^{c_0} \bmod p$ is satisfied.
(4.H) The equation $g^{c_1} K^{v_1} \bmod p = b_1 \beta^{c_1} \bmod p$ is satisfied.

The correctness of selection encryptions must be verified to confirm that the individual selections are an encryption of either zero or one. This is covered in 4.1.

5) Adherence to vote limits
An election verifier must confirm the following for each contest on each cast ballot
(5.A) The number of placeholder positions matches the contest's selection limit L.
(5.B) The contest total $(\bar{\alpha}, \bar{\beta})$ satisfies $\bar{\alpha} = \prod_i \alpha_i \bmod p$ and $\bar{\beta} = \prod_i \beta_i \bmod p$ where the (α_i, β_i) represent all possible selections (including placeholder selections) for the contest.
(5.C) The given value V is in $\mathbb{Z}_q$.
(5.D) The given values a and b are each in $\mathbb{Z}_p^r$.
(5.E) The challenge value C is correctly computed as $C = H(\bar{Q}, \bar{\alpha}, \bar{\beta}, a, b)$.
(5.F) The equation $g^V \bmod p = \left(a \bar{\alpha}^C\right) \bmod p$ is satisfied.
(5.G) The equation $\left(g^{LC} K^V\right) \bmod p = \left(b \bar{\beta}^C\right) \bmod p$ is satisfied.

The adherence to voting limits must be verified to ensure that a voter is not voting for more options than specified by the selection limit L. This is covered in Section 4.2.

6) Validation of confirmation codes
An election verifier must confirm the following for each confirmation code H_i.
(6.A) $H_i = H\left(\bar{Q}, \ldots, B_i\right)$, where ... stands for any intermediate parameters defined by the election manifest.
An election verifier must also verify the following.
(6.B) There are no duplicate confirmation codes, i.e., among the set of submitted (cast and spoiled) ballots, no two have the same confirmation code.

The confirmation codes for each ballot must be verified. Namely, no two codes are the same for any submitted ballot, and the confirmation code is computed correctly. As of spec 1.1 for ElectionGuard, the details for computing the confirmation codes are not detailed enough, meaning

that we did not successfully implement this in the Go verifier.

7) Correctness of ballot aggregation
An election verifier must confirm for each (non-placeholder) option in each contest in the election manifest that the aggregate encryption (A, B) satisfies
(7.A) $A = \left(\prod_j \alpha_j\right) \bmod p$,
(7.B) $B = \left(\prod_j \beta_j\right) \bmod p$,
where the (α_j, β_j) are the corresponding encryptions on all cast ballots in the election record.

The verifier must verify that the ballots have been aggregated correctly by multiplying all of the α_j and all of the β_j. This is covered in Section 5.1.

8) Correctness of partial decryptions
An election verifier must then confirm for each (non-placeholder) option in each contest in the election manifest the following for each decrypting guardian G_i.
(8.A) The given value v_i is in the set $\mathbb{Z}_q$.
(8.B) The given values a_i and b_i are both in the set $\mathbb{Z}_p^r$.
(8.C) The challenge value c_i satisfies $c_i = H\left(\bar{Q}, A, B, a_i, b_i, M_i\right)$.
(8.D) The equation $g^{v_i} \bmod p = (a_i K_i^{c_i}) \bmod p$ is satisfied.
(8.E) The equation $A^{v_i} \bmod p = (b_i M_i^{c_i}) \bmod p$ is satisfied.

The verifier must verify that the partial decryptions have been decrypted correctly. This is covered in Section 5.2.1.

9) Correctness of substitute data for missing guardians
An election verifier must confirm for each (non-placeholder) option in each contest in the election manifest the following for each missing guardian G_i and for each surrogate guardian G_ℓ.
(9.A) The given value $v_{i,\ell}$ is in the set $\mathbb{Z}_q$.
(9.B) The given values $a_{i,\ell}$ and $b_{i,\ell}$ are both in the set $\mathbb{Z}_p^r$.
(9.C) The challenge value $c_{i,\ell}$ satisfies $c_{i,\ell} = H\left(\bar{Q}, A, B, a_{i,\ell}, b_{i,\ell}, M_{i,\ell}\right)$.
(9.D) The equation $g^{v_{i,\ell}} \bmod p = \left(a_{i,\ell} \cdot \left(\prod_{j=0}^{k-1} K_{i,j}^{\ell}\right)^{c_{i,\ell}}\right) \bmod p$ is satisfied.
(9.E) The equation $A^{v_{i,\ell}} \bmod p = \left(b_{i,\ell} M_{i,\ell}^{c_{i,\ell}}\right) \bmod p$ is satisfied.

The verifier must verify the NIZK proof of guardian G_ℓ knowing $P_i(\ell)$ for the missing guardian G_i. This has to be done for each guardian G_ℓ participating in the partial decryption. This is covered in Section 5.3.

10) Correctness of construction of replacement partial decryptions
An election verifier must confirm that for each guardian G_ℓ serving to help compute a missing share of a tally, the Lagrange coefficient w_ℓ is correctly computed by confirming the equation
(10.A) $\left(\Pi_{j\in(U-\{\ell\})} j\right) \bmod q = \left(w_\ell \cdot \prod_{j\in(U-\{\ell\})}(j-\ell)\right) \bmod q$
An election verifier must then confirm the correct missing tally share for each (non-placeholder) option in each contest in the election manifest for each missing guardian G_i as
(10.B) $M_i = \prod_{\ell\in U} (M_{i,\ell})^{w\ell} \bmod p$.

The verifier must verify that the Lagrange coefficients are calculated correctly and that the shares $M_{i,\ell}$ have been correctly multiplied together, forming M_i. This is covered in Section 5.3. Combining verification step 9 and step 10, the verifier can confirm that the shares are computed correctly and that they form the partial decryption M_i.

11) Validation of correct decryption of tallies
An election verifier must confirm the following equations for each (non-placeholder) option in each contest in the election manifest.
(11.A) $B = (M \cdot (\prod_{i=1}^{n} M_i)) \bmod p$.
(11.B) $M = g^t \bmod p$.
An election verifier must also confirm that the text labels listed in the election record tallies match the corresponding text labels in the election manifest. For each contest in a decrypted tally, an election verifier must confirm the following.
(11.C) The contest text label occurs as a contest label in the list of contests in the election manifest.
(11.D) For each (non-placeholder) option in the contest, the option text label occurs as an option label for the contest in the election manifest.
(11.E) For each option text label listed for this contest in the election manifest, the option label occurs for a (non-placeholder) option in the decrypted tally contest.
An election verifier must also confirm the following.
(11.F) For each contest text label that occurs in at least one submitted ballot, that contest text label occurs in the list of contests in the corresponding tally.

An election verifier must verify the tallies; this ensures that B, from the aggregate encryption, has been computed correctly. From equation 5.4, it is clear that $\prod_{i=1}^{n} M_i = K^R$, which means (11.A) follows directly from the definition of B. When ElectionGuard has decrypted, (11.B) proves that the decrypted value t corresponds to the M used in verification step (11.A). These equations are covered in Section 5.2.

12) Correctness of partial decryptions for spoiled ballots
An election verifier must then confirm for each spoiled ballot and each (non-placeholder) option in each contest on the spoiled ballot the following for each decrypting guardian G_i.
(12.A) The given value v_i is in the set $\mathbb{Z}_q$.
(12.B) The given values a_i and b_i are both in the set $\mathbb{Z}_p^r$.
(12.C) The challenge value c_i satisfies $c_i = H\left(\bar{Q}, \alpha, \beta, a_i, b_i, M_i\right)$.
(12.D) The equation $g^{v_i} \bmod p = (a_i K_i^{c_i}) \bmod p$ is satisfied.
(12.E) The equation $\alpha^{v_i} \bmod p = (b_i M_i^{c_i}) \bmod p$ is satisfied.

The same argument as in step 8.

13) Correctness of substitute data for spoiled ballots
An election verifier must confirm for each spoiled ballot and each (non-placeholder) option in each contest on the spoiled ballot the following for each missing guardian G_i and for each surrogate guardian G_ℓ.
(13.A) The given value $v_{i,\ell}$ is in the set $\mathbb{Z}_q$.
(13.B) The given values $a_{i,\ell}$ and $b_{i,\ell}$ are both in the set $\mathbb{Z}_p^r$.
(13.C) The challenge value $c_{i,\ell}$ satisfies $c_{i,\ell} = H\left(\bar{Q}, \alpha, \beta, a_{i,\ell}, b_{i,\ell}, M_{i,\ell}\right)$.
(13.D) The equation $g^{v_{i,\ell} \bmod p} = \left(a_{i,\ell} \cdot \left(\prod_{j=0}^{k-1} K_{i,j}^{\ell}\right)^{c_{i,\ell}}\right) \bmod p$ is satisfied.
(13.E) The equation $\alpha^{vi,\ell} \bmod p = \left(b_{i,\ell} M_{i,\ell}^{c_{i,\ell}}\right) \bmod p$ is satisfied.

The same argument as in step 9.

14) Correctness of replacement partial decryptions for spoiled ballots
An election verifier must confirm that for each guardian G_ℓ serving to help compute a missing share of a tally, that its Lagrange coefficient w_ℓ is correctly computed by confirming the equation
(14.A) $\left(\Pi_{j \in (U - \{\ell\})} j\right) \bmod q = \left(w_\ell \cdot \prod_{j \in (U-\{\ell\}} (j - \ell)\right) \bmod q$.
An election verifier must then confirm the correct missing decryption share for each (nonplaceholder) option in each contest on the spoiled ballot for each missing guardian G_i as
(14.B) $M_i = \prod_{\ell \in U} \left(M_{i,\ell}\right)^{w_\ell} \bmod p$

The same argument as in step 10.

15) Validation of correct decryption of spoiled ballots
An election verifier must confirm the correct decryption of each selection V in each contest.
(15.A) $\beta = (M \cdot \prod_{i=1}^{n} M_i) \bmod p$.
(15.B) $M = g^V \bmod p$.

The same argument as in step 11.

16) Validation of correctness of spoiled ballots
An election verifier must also confirm that the spoiled ballot is well-formed, i.e., for each contest on the spoiled ballot, it must confirm the following.
(16.A) For each option in the contest, the selection V is either a 0 or a 1.
(16.B) The sum of all selections in the contest is at most the selection limit L for that contest.
An election verifier must also confirm that for each decrypted spoiled ballot, the selections listed in text match the corresponding text in the election manifest.
(16.C) The contest text label occurs as a contest label in the list of contests in the election manifest.
(16.D) For each (non-placeholder) option in the contest, the option text label occurs as an option label for the contest in the election manifest.
(16.E) For each option text label listed for this contest in the election manifest, the option label occurs for a (non-placeholder) option in the decrypted spoiled ballot.

The verifier must ensure that all decrypted spoiled ballots are well-formed as described in Section 4. However, instead of checking the NIZK proofs, the verifier verifies the plaintext.

17) Correctness of contest data partial decryptions for spoiled ballots
An election verifier must confirm the correct decryption of the contest data field for each contest by verifying the conditions analogous to Verification 8 for the corresponding NIZK proof with (A,B) replaced by (C_0,C_1,C_2) and M_i by m_i as follows.
(17.A) The given value v_i is in the set $\mathbb{Z}_q$.
(17.B) The given values a_i and b_i are both in the set $\mathbb{Z}_p^r$.
(17.C) The challenge value c_i satisfies $c_i = H\left(\bar{Q}, C_0, C_1, C_2, a_i, b_i, m_i\right)$.
(17.D) The equation $g^{v_i} \bmod p = (a_i K_i^{c_i}) \bmod p$ is satisfied.
(17.E) The equation $C_0^{v_i} \bmod p = (b_i m_i^{c_i}) \bmod p$ is satisfied.

Same as step 8 but now verifying correct partial decryption of contest data. To understand the decryption of contest data, see ElectionGuard specification (Benaloh & Naehrig, 2018, p. 27-28).

18) Correctness of substitute contest data for spoiled ballots
An election verifier must confirm for each contest on the spoiled ballot the following for each missing guardian G_i and each surrogate guardian G_ℓ.
(18.A) The given value $v_{i,\ell}$ is in the set $\mathbb{Z}_q$.
(18.B) The given values $a_{i,\ell}$ and $b_{i,\ell}$ are both in the set $\mathbb{Z}_p^r$.
(18.C) The challenge value $c_{i,\ell}$ satisfies $c_{i,\ell} = H\left(\bar{Q}, C_0, C_1, C_2, a_{i,\ell}, b_{i,\ell}, m_{i,\ell}\right)$.
(18.D) The equation $g^{v_{i,\ell}} \bmod p = \left(a_{i,\ell} \cdot \left(\prod_{j=0}^{k-1} K_{i,j}^{\ell}\right)^{c_{i,\ell}}\right) \bmod p$ is satisfied.
(18.E) The equation $C_0^{v_{i,\ell}} \bmod p = \left(b_{i,\ell} m_{i,\ell}^{c_{i,\ell}}\right) \bmod p$ is satisfied.

The verifier has to verify that guardian G_i successfully proves procession of $P_i(\ell)$ where $m_{i,\ell} = C_0^{P_i(\ell)} \bmod p$ and $g^{P_i(\ell)} \bmod p = \prod_{j=0}^{k-1} K_{i,j}^{\ell^j} \bmod p$ hold. This ensures that the partial computation

of β is computed correctly. This proof is covered in the ElectionGuard specification (Benaloh & Naehrig, 2018, p. 29).

19) Correctness of contest replacement decryptions for spoiled ballots
An election verifier must confirm the correct missing contest data share for each contest on the spoiled ballot for each missing guardian G_i as
(19.A) $m_i = \prod_{\ell \in U} (m_{i,\ell})^{w_\ell} \bmod p$

The verifier must verify that the partial computation of β is done correctly if a guardian is missing.

Appendix E

Benchmarking individual steps

We ran the Go verifier on the dataset with 200 ballots and made a plot of the single-threaded and multi-threaded running time for each step.

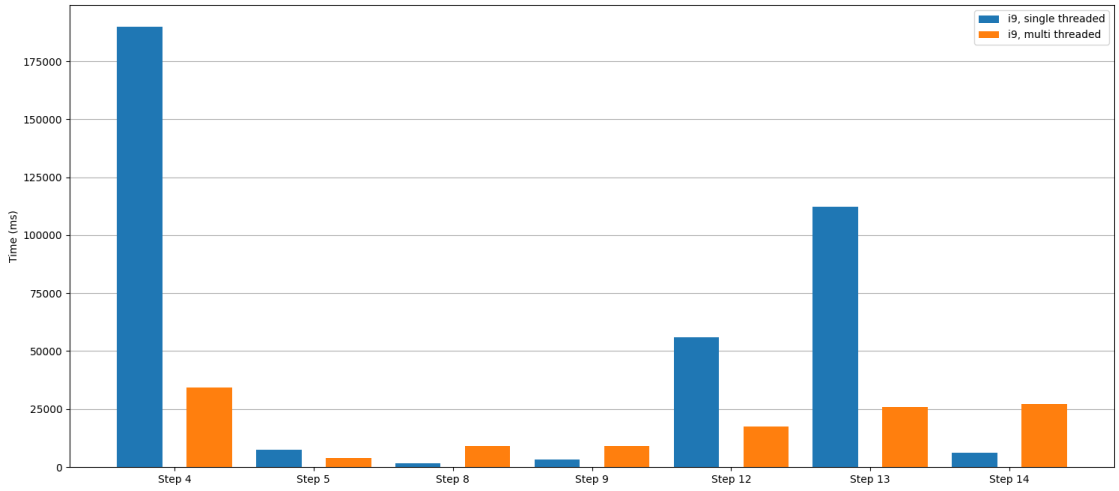

Figure E.1: Comparison of total running time for Go verifier

It is important to mention that the multi-threaded appears to take longer for some of the steps. This is due to the nature of multi-threaded programs, where some threads might sleep, meaning that the running time of the single-threaded and multi-threaded can not be directly compared. However, when the multi-threaded verifier is faster, we can be sure that the verifier had a performance increase from introducing threads.

Appendix F

Data from Aarhus municipal election

Using the data from KMD (KMDValg, 2021), we have counted the number of people running for each party, which can be seen in the below table.

Party	**Number of selections**
Socialdemokratiet	27
Radikal Venstre	19
Det Konservative Folkeparti	23
Nye Borgerlige	19
Socialistisk Folkeparti	29
Veganerpartiet	3
Trivsel og Reel Borgerinddragelse	4
Liberal Alliance	13
De Grønne	1
Kristendemokraterne	5
Dansk Folkeparti	8
Partiet Samfundssind	2
Kommunisterne	3
Bedre Aarhus	1
Venstre	14
Frihedslisten	1
Egalitært Folkeparti	1
Alternativet	7

Table F.1: Number of selections per party

Using the values from table F.1, we can conclude that an ElectionGuard election would need 199

selections due to

$$179 \text{ people} + 19 \text{ parties} + 1 \text{ placeholder} = 199 \text{ selections.} \tag{F.1}$$

The total amount of votes for this election was 196.274.