Using ElectionGuard for secure remote voting on untrusted devices

Bachelor's Thesis

Adrian Zanga

zangaa@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Darya Melnyk, Tejaswi Nadahalli
Prof. Dr. Roger Wattenhofer

September 13, 2020

Acknowledgements

I was fortunate that Darya Melnyk and Tejaswi Nadahalli gave me lots of constructive advice as well as helpful ideas and took time for lengthy discussions throughout the work on this thesis. Thank you!

Abstract

One of the most promising systems for electronic voting is currently being developed by Microsoft under the name ElectionGuard *(Benaloh et al., 2020)*. It is intended for use in dedicated voting booths and therefore has some functional limitations compared to e-voting systems specifically designed for use in a remote environment.
A recently proposed protocol, Artemis *(Boss et al., 2019)*, provides security and privacy against malicious voting devices. This paper describes ideas, slight modifications, and implementation for this protocol on ElectionGuard.

Contents

Chapter 1

Introduction

The popularity of electronic voting is growing rapidly as an increasing number of nations and organizations seek to improve their voting mechanisms. There are many reasons for electronically casting and counting ballots, such as speed, cost-efficiency, and voter turnout. On the other hand, such systems open up new vectors of attack for tampering in an election. Therefore, solutions must be found to ensure the integrity of such systems. A relatively new system for casting and accumulating ballots is ElectionGuard [1], which is currently being developed by Microsoft. It is intended to run on dedicated devices in the voting booths. This thesis will focus on how ElectionGuard can be adapted to enable remote voting and to deal with the new risks of such a system. Two main challenges of remote voting are voter authentication and guaranteeing privacy. The latter is the main focus of this thesis. Artemis [2] is a proposed protocol for the Helios e-voting system [3] that achieves the correctness and confidentiality of a vote as well as receipt-freeness using multiple devices, assuming that at least one of the devices is not corrupt. In this thesis, I will demonstrate how Artemis can be implemented on top of ElectionGuard and discuss some necessary modifications to both systems required to achieve this.
Chapter 2 gives an overview of ElectionGuard and Artemis. In Chapter 3, I will describe the necessary changes to the Artemis protocol such that it can be implemented on top of ElectionGuard. I will explain the details of my implementation in Chapter 4. Finally, in Chapter 5, I will briefly discuss some ideas for possible future directions.

Chapter 2

Background

2.1 Related Work

Two different proof systems are mentioned in this thesis. One of them is the Groth-Sahai [4] system which is used by Helios to prove that encrypted ballots are valid. The second proof system follows the Chaum-Pedersen protocol [5]. It is implemented in ElectionGuard and its adaption is described in detail in [6].
The PrivApollo paper [7] lies a foundation for the Artemis protocol, outlining the idea of using a second device together with a candidate-color mapping to guarantee the privacy of a voter.
Helios [3] is one of the first voting systems to enable cryptographic auditing. Although Helios is intended for use in remote voting, it is quite similar to ElectionGuard. Some differences are mentioned in Chapter 3 of this paper.

2.2 ElectionGuard

ElectionGuard is a system for holding end-to-end verifiable elections and consists of multiple open-source components developed by Microsoft. It is still in its early stages but a first pilot has been successfully carried out earlier this year in an election in Fulton, Wisconsin [8]. It is important to note that voter authentication is not part of ElectionGuard. Rather, it is the task of the election staff to grant access to the voting booths in which ElectionGuard is running only to those voters who are eligible to vote.

2.2.1 Protocol

Setup

ElectionGuard enables complex elections that span several different districts with different ballot styles, each consisting of a subset of all competitions. For example, a nationwide presidential election could be made possible in which each voter could additionally vote for representatives in his or her state. Different types of competitions are possible, e.g. *one-of-n* or *n-of-m*. Voting for unlisted candidates can also be allowed. All this data must be configured in the election manifest, which is provided in the form of a JSON file.

Key Ceremony

The key ceremony is the process by which the election trustees share encryption keys for an election. The trustees are usually election workers, party members, government officials, or media representatives. After the key ceremony, each trustee has a private key. The joint public key is published. The trustees are responsible for holding those private keys that are needed to decrypt the election results. A quorum of trustees can also be specified to compensate trustees who may be missing at the time of decryption.

Ballot Encryption

The ballots are encrypted on a uniquely identified device intended for use in a voting booth. For each selection, a non-interactive zero-knowledge proof is generated that the encryption is either an encryption of zero or one. In the same way, a proof is generated for each contest that the sum of all encrypted choices is equal to the selection limit of the contest (usually one). Once the user has submitted the ballot to the voting system, a tracking ID is derived from the encrypted ballot.

Cast or Spoil Ballot

The voter is allowed to audit, or "spoil", a ballot. If a ballot is spoiled, it cannot be included in the final tally, and the voting system must be able to decrypt the ballot. Unlike many other e-voting systems, including Helios, spoiled ballots are put away for later decryption. If the voter decides to "cast" his ballot, it is marked accordingly and included in the final tally.

Tally

Homomorphic tallying as described in the next section is used to add together properly formed ballots. This tally can then be decrypted by the trustees to determine the final election result. Proofs that this tally is correctly formed are published. Also, all cast ballots are published in their encrypted form, along with the proofs that they are well-formed. Spoiled ballots are decrypted and published along with their encrypted version and the proofs.

2.2.2 Cryptographic Primitives

An exponential form [6] of the ElGamal cryptosystem [9] is used for encryption. In an initial step, primes p and q are publicly fixed. A generator g of the order q subgroup $\mathbb{Z}_p^r$ is also fixed where $\mathbb{Z}_p^r$ is the set of r^{th}-residues in $\mathbb{Z}_p^*$.
The n trustees of an election are denoted by T_1, T_2, ..., T_n. Each trustee T_i generates an ElGamal key pair consisting of a random secret $s_i \in \mathbb{Z}_q$ and the corresponding public key $K_i = g^{s_i} \mod p$. The joint election public key is

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

There are some additional steps to allow compensation of missing trustees but they exceed the scope of this thesis.
To encrypt a message $M \in \mathbb{Z}_p^r$ a random nonce $R \in \mathbb{Z}_q$ is selected. The ciphertext is then constructed as the pair $(\alpha, \beta) = (g^R \mod p, g^M \cdot K^R \mod p)$. If the secret s is known, (α, β) can be decrypted as

$$\frac{\beta}{\alpha^s} \mod p = \frac{g^M \cdot K^R}{(g^R)^s} \mod p = \frac{g^M \cdot (g^s)^R}{(g^R)^s} \mod p = g^M \mod p$$

Because only two possible messages are encrypted by ElectionGuard, either zero or one, M can be computed from $g^M \mod p$.
For the tallying step, the homomorphic property of this form of ElGamal encryption is utilized. Given two ciphertexts $(\alpha_1, \beta_1) = (g^{R_1} \mod p, g^{M_1} \cdot K^{R_1} \mod p)$ and $(\alpha_2, \beta_2) = (g^{R_2} \mod p, g^{M_2} \cdot K^{R_2} \mod p)$ an encryption of the sum $M_1 + M_2$ can be computed by componenet-wise multiplication:

$$(\alpha, \beta) = (\alpha_1 \alpha_2 \mod p, \beta_1 \beta_2 \mod p) = (g^{R_1+R_2} \mod p, g^{M_1+M_2} \cdot K^{R_1+R_2} \mod p$$

This property is also used in the re-encryption step by adding an encryption of zero to every ciphertext.

2.2.3 End-To-End Verifiability

End to End Verifiability (E2E-V) of a voting system follows from three main principles: *Cast As Intended*, *Recorded As Cast* and *Tallied As Recorded* [10]. E2E-V election techniques enable individual voters to check election results without requiring voters to trust election software, hardware, or procedures. ElectionGuard is designed to give guarantees about these principles:

1. *Cast As Intended*: The voter can challenge a ballot after the voting terminal has committed to its encryption. Even though a voter will never know the contents of a cast encrypted ballot, they can verify the security of the voting terminal by spoiling as many ballots as they wish. Given that enough voters spoil a ballot, a malicious system will be detected with overwhelming probability.

2. *Recorded As Cast*: A *tracking ID* is generated for each encrypted ballot. Those *tracking IDs* are published alongside the election results enabling voters to verify that their ballot was included in the final tally.

3. *Tallied As Recorded*: Alongside the election results, every encrypted ballot including its zero-knowledge proofs is published. This allows full verifiability by third parties.

2.3 Artemis

Artemis [2] is a proposed protocol to secure against malicious voting devices for the Helios e-voting system. It is inspired by the PrivApollo [7] paper, which describes a concept for using additional devices in the vote creation step. The main goal of this thesis is to implement those ideas on the code base of ElectionGuard. Since the concepts of Helios and ElectionGuard are relatively similar it is reasonable to build on Artemis.

2.3.1 Protocol

In the original Helios protocol, the user enters his desired vote directly into the voting terminal *VT*. The *VT* is therefore a single point that an attacker would have to control to learn about a voter's choice. Because Helios allows remote voting, this scenario is not so far-fetched, since any user with any kind of computer that provides a web browser can participate in an election. Artemis introduces an active voting assistant *AVA* and an arbitrary number of passive voting assistants *VA*. The idea behind this is that an attacker cannot learn the choices made by the voter unless all of these devices are malicious.
VT generates a mapping between the list of encrypted ballots and colors and sends it to the AVA. VT then displays the unencrypted mapping, while AVA only displays the colors. The user can choose the color on the AVA that corresponds to the desired candidate as displayed on VT. AVA re-encrypts the already encrypted candidate corresponding to the selected color and publishes it. Therefore neither VT nor AVA will learn the selected candidate unless they collude. The other VA's are used to verify that the encryption steps have been performed correctly. While Artemis describes two types of vote accumulation, namely mix-net tallying and homomorphic tallying, this work focuses only on the latter. The protocol makes the tallying step more difficult, as the ballots are re-encrypted so that normal zero-knowledge proofs become invalid. The use of Groth-Sahai proofs [4] solves this problem. Although they are computationally expensive, they can also be re-randomized and are therefore valid after re-encryption. The rest of the tallying step remains unchanged.

Chapter 3

Adaption

3.1 Vote creation

One difference between Helios and ElectionGuard is the proof method that guarantees that the ballots are formed correctly. Helios uses Groth-Sahai proofs for this. Encrypted ballots with proofs can be pre-generated to avoid creating those proofs directly on a voter's device. ElectionGuard uses Chaum-Pedersen Zero-Knowledge proofs [5], which have the disadvantage that they are invalid after re-encryption.
To minimize changes to the existing ElectionGuard code, I chose a simpler solution instead to fix this problem. Before tallying, each ballot is verified by first accumulating all encrypted votes in a contest. In a second step, the tallying authority decrypts this new ciphertext, which should yield a zero or a one if the ballot has been filled out correctly. If this is the case for each competition, the ballot is included in the final tally. This solution is neither elegant nor efficient, but I chose it due to time constraints. A better solution would be to implement Groth-Sahai proofs for ballot validation.
Using this method, the communication between the voting terminal and voting assistant can be slightly simplified. Initial encryption of the ballot based on the list of choices is not necessary. Instead, the VA can create the first encryptions of the choices according to the voter's decision and return them to VT. VT maps those choices back to the candidates and re-encrypts them.

1. VT generates a random mapping between colors and candidates. VT displays this mapping to the voter.

2. VT sends only the colors to VA. VA displays these colors to the voter.

3. The voter checks the mapping and clicks on the color on VA that corresponds to his or her desired candidate.

4. VA encrypts a one for this color and a zero for all other colors with the election public key. It sends the color-encryption pairs back to VT.

5. VT re-encrypts the ciphertext and assigns it to the candidate using its initially generated mapping. Since the VA has created the ciphertexts and therefore knows the corresponding plaintext, this step is necessary. It prevents VA from learning the voter's choice from the resulting ciphertext.

6. Steps 1 to 5 are repeated for each contest in the election. Then VT sends the resulting encrypted ballot to the server running ElectionGuard.

I adapted the re-encryption step for the exponential ElGamal system. Instead of multiplying the ciphertext by an encryption of one, which would work for textbook ElGamal [9], it is multiplied by an encryption of zero. Due to the exponential ElGamal construction described above, this will have the effect of adding a zero to the ciphertext, which will ultimately re-encrypt the same plaintext.

3.2 Authentication

Helios requires each user to authenticate with a username and password before casting a ballot. ElectionGuard on the other hand delegates voter authentication to election workers, which have to make sure only eligible voters can use the ElectionGuard system.
For authentication, I chose a simpler approach compared to Helios. Before the election, each voter receives a random token. Upon casting a ballot, this token can be sent to the registrar together with the encrypted ballot. If the token is valid and has not yet been used, the registrar creates a signature of the encrypted ballot and publishes it. No signature is required to spoil a ballot. This allows non-eligible voters to check the security of the system as well, which is especially necessary if they are potentially affected by the election results. At the end of the election, anyone could verify that each encrypted ballot contained in the final tally has a corresponding valid signature from the registrar. The alternative solution has the advantage that it does not require secure storage of usernames and passwords and their transmission during the authentication step. It could potentially be further optimized by deriving the token from a secret together with the voter details. This would minimize the required secure storage to a single secret.

Chapter 4

Implementation

4.1 ElectionGuard Web API

In its early stages, ElectionGuard was implemented in `C`. It includes all core functions such as the generation of election keys, the encryption of a plaintext ballot, the casting of ballots, and the creation of a tally. It is not necessarily intended for direct access by the end-user. Microsoft has released a web API written in `C♯` that allows developers to program their own voting devices, such as a voting terminal or ballot box, and access ElectionGuard functionality through HTTP. However, while I was working on this thesis Microsoft released a completely rewritten version of ElectionGuard in `Python 3.8` [11]. The revised version is much better documented and the code is simple to understand. The main problem was that a web API was missing. Part of this project was therefore to write a web API to allow all devices involved in an election to communicate with a central ElectionGuard instance. Microsoft has already opened an issue on GitHub and described that they want to implement a web API using the *Flask* [12] framework. Therefore, it was an easy decision to use *Flask* for the web API with the intention to possibly open a pull request at a later date. The resulting *Flask* API provides an interface for:

- Creating an Election.
- Creating a *CiphertextBallot* object from a ballot provided in `JSON` format. This endpoint supports ballots in the format described in the Artemis protocol.
- Casting or Spoiling a ballot.
- Receiving the election public key which is used by the voting devices to encrypt candidate choices.
- Receiving the election manifest, which is a `JSON` object containing all contests and their possible candidates for different ballot styles.

The Web API supports multiple concurrent elections on a single instance. The state of each election is stored in files on the server using the *pickle* package [13]. *Pickle* is used for serialization and de-serialization of Python object structures. For security reasons it is very important to ensure that only ElectionGuard writes and modifies files containing these pickle objects.
A fork of the official ElectionGuard Python repository was created to add my code [14]. Most of this code is split into three files in the *src/electionguardFlaskApi* folder. The file *flask_ app.py* is the one that should be executed to start the program, as it provides the endpoints for the users of the API. The task of handling the loading and saving of election-specific data is separate and is always called by the API endpoints. The most interesting part of fulfilling the actual consumer request is done in a third file. It contains a lot of function calls to the existing ElectionGuard code. The only change made to the original ElectionGuard code itself was to disable ballot validation, which is not possible with the intended Chaum-Pedersen proofs due to the re-encryption step. The ballot validation was therefore replaced by the simpler method described above.

4.2 ElectionGuard Admin Device

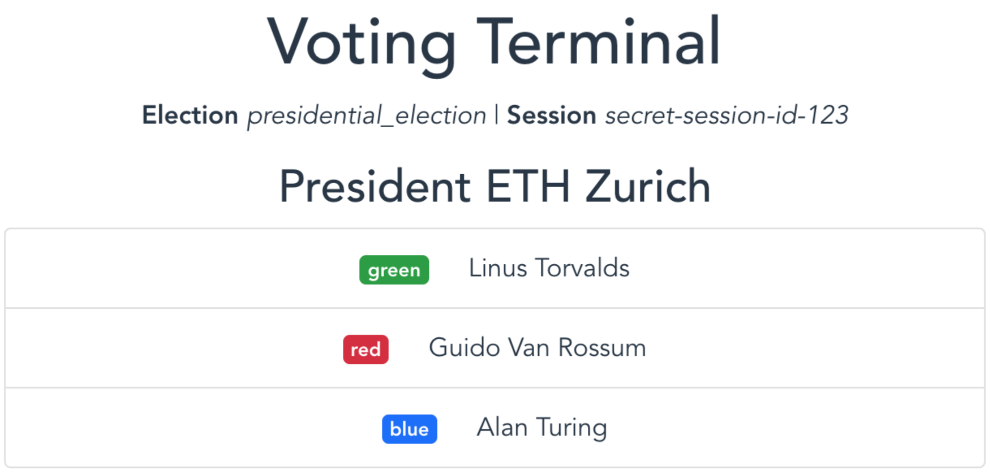
This project includes a simple user interface for creating an election, calling the tally endpoint, and clearly displaying the election results. It was implemented with *VueJS* [15].

4.3 ElectionGuard Voting Terminal and Assistant

Since most of the changes to the original ElectionGuard protocol take place during the vote creation phase, the Voting Terminal (VT) and the Voting Assistant (VA) were the focus of this work. Both the VT and the VA are bundled in a single project. JavaScript using the *VueJS* framework is used to create a reactive web user interface.

Setup

A user opens the same website on two separate devices to start the voting process. He then enters a random secret on both devices to link them together. A future solution using QR codes would make this step a little easier for the end-user. The communication between VT and VA is done over sockets using the library *socket.io* [16]. *Socket.io* enables efficient real-time communication, eliminating the need for each client to constantly poll for updates, resulting in a smooth user experience. To achieve this, in addition to the process that serves the site, a second process handles this communication by receiving messages and sending



Voting Terminal

Election *presidential_election* | **Session** *secret-session-id-123*

President ETH Zurich

Figure 4.1: Interface of the Voting Terminal.

them to all other clients who have registered with the same secret. It is therefore important that the user enters a long, random secret.

Candidate-color mapping

After the clients have established a connection, the next step is to present the candidates to the user. VT receives the public candidate list for the first contest and pairs each candidate with a random color from a predefined color set. It displays this pairing (Figure 4.1) and sends the list of used colors to VA.

Encryption

The user decides which candidate to choose and selects the matching color on the VA (Figure 4.2). Since the VA does not know the candidate-color mapping, it cannot learn the choice as long as the VT is not malicious. The VA then encrypts a *1* for the selected color and a *0* for all other colors with the public key of the election. It sends the resulting ciphertexts back to the VT in combination with the corresponding colors. Using the previously generated mapping, VT assigns each candidate to the received ciphertext and re-encrypts it by multiplying it with the encryption of *0*. This process is repeated for each election contest. Finally, VT sends the generated ballot, consisting of each candidate associated with an encrypted *1* or *0*, to the ElectionGuard Web API.
On the voting devices the open-source *elgamal.js* [17] library is used for encryption. Because ElectionGuard changed the encryption step as described in the

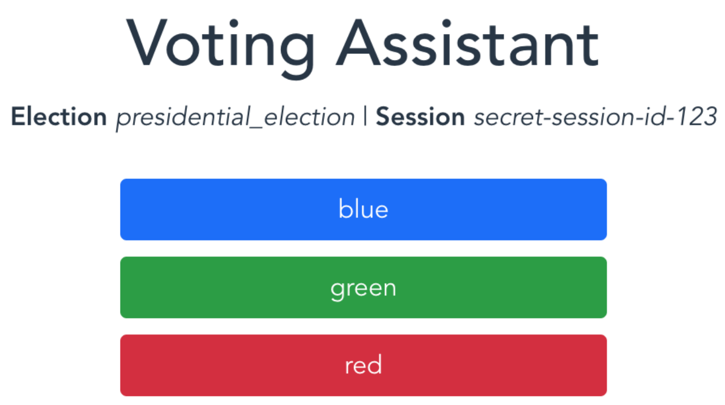

Voting Assistant
Election presidential_election | Session secret-session-id-123
blue
green
red

Figure 4.2: Interface of the Voting Assistant, giving the voter the option to select a color.

Cryptography section above, this change had to be incorporated into the *elgamal.js* library as well. The constants p and q used for encryption are hard-coded into the project, similar to ElectionGuard.

Cast & Spoil decision

At the end of the encryption step, the generated ballot was sent to the Web API in JSON format. To use further functionality provided by ElectionGuard, we need to convert this ballot into a Python *CiphertextBallot* object. Internally, such an object consists of all candidates in plaintext with either an encrypted *0* or *1* assigned to each candidate. Since this object is almost identical to our previously generated JSON object, the conversion is quite simple. Only the encryption steps provided by ElectionGuard can be omitted since we have already done this on the voter's devices. The last step a user has to take is to decide if he wants to either cast or spoil the ballot. ElectionGuard already provides the functionality to mark a *CiphertextBallot* accordingly. So it is sufficient to send this decision to the web API which utilizes this functionality.

4.4 Registrar

In the initial phase of this project, I implemented a simple registrar. It can be used to generate PINs to be distributed to the eligible voters for a specific election. The registrar then provides a web API that enables consumers to:

- Enter a ballot and PIN. If the PIN is valid, the API uses the RSA algorithm to produce a signed version of the ballot.
- Provide a signature and let the API check if the signature is valid.

At the end of an election, the registrar is intended to publish all the signatures along with the encrypted ballots. There is an additional verification script that can be executed by anyone who wishes to verify those signatures.

4.5 Additional Code

4.5.1 Controller

The controller script was used to run an election using the old `C♯` version of the ElectionGuard Web API. Since Microsoft has deprecated this version, the script is now obsolete.

4.5.2 Python Ballot Box

Similar to the controller script, the ballot box implementation in `Python` was intended for use with the now outdated web API. Microsoft delegated the task of keeping track of the IDs of the cast and spoiled ballots to the user, which led to the creation of this script. It provided another web API to do just that. Fortunately, this functionality was directly integrated into the revised version of ElectionGuard.

Chapter 5

Conclusions

In the past, many different systems have been proposed for electronic voting. Many of them lack a secure foundation, while others have made the leap into practice. Some did both. What makes ElectionGuard stand out in this mix is that it is developed by one of the largest technology companies in the world. Microsoft has very talented researchers with numerous publications in the field of cryptography and voting. They also get the necessary attention from researchers who take a second look at the system to possibly find design problems or errors in the code. It remains to be seen how and if ElectionGuard will grow in the future and possibly even be used in the US presidential election in 2020.
This thesis tries to present some ideas on how ElectionGuard can be used for electronic remote voting, what the challenges are, and an approach for a possible implementation. This implementation has not been subjected to a security analysis, neither has Artemis at the time of writing this thesis. It should therefore not be used in real elections.

5.1 Further Ideas

The time constraints for this work allowed me only so much to do. I want to end this thesis with a few improvements that could be considered in the future:

- **Registrar:** The current registrar is quite simple and not tightly integrated into the system. Instead, the registrar could be split into multiple trustees. This would aid in making sure that tokens don't get linked to individual voters and an accurate list of eligible voters is used.

- **Web API:** The web API currently does not support a real key ceremony, but only one trustee. Furthermore, in the future, an SQL database could be used instead of storing the state of an election in files. Solving both tasks correctly is not trivial, therefore a simpler approach was chosen for this project.

- **Ballot validation:** A clean solution for validating ballots needs to be considered. Using Groth-Sahai proofs similar to Helios would be a possible approach.

Bibliography

[1] "Building an end-to-end verifiable election using electionguard." [Online]. Available: https://github.com/microsoft/electionguard/wiki/Building-an-End-to-end-Verifiable-Election-with-Electionguard

[2] M. Boss, "Artemis: Solving the secure platform problem for the helios e-voting system," 2020.

[3] B. Adida, "Helios: Web-based open-audit voting," *USENIX security symposium*, 2008.

[4] J. Groth and A. Sahai, "Efficient non-interactive proof systems for bilinear groups," pp. 415–432, 2008.

[5] D. Chaum and T. P. Pedersen, "Wallet databases with observers," pp. 89–105, 1992.

[6] J. Benaloh, "Electionguard preliminary specification v0.85." [Online]. Available: https://github.com/microsoft/electionguard/wiki/Informal/ElectionGuardSpecificationV0.85.pdf

[7] P. L. V. Hua Wu and F. Zagórski, "Privapollo – secret ballot e2e-v internet voting," 2019.

[8] S. Fleming, "The inside story of microsoft's electionguard pilot in wisconsin," *Microsoft News*, 2020. [Online]. Available: https://news.microsoft.com/on-the-issues/2020/05/13/microsoft-electionguard-pilot-wisconsin/

[9] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE transactions on information theory*, vol. 31, no. 4, pp. 469–472, 1985.

[10] J. Benaloh, "End-to-end verifiability," 2016. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/11/e2e-primer.pdf

[11] "Electionguard-python." [Online]. Available: https://github.com/microsoft/electionguard-python

[12] "Flask." [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/

[13] "pickle." [Online]. Available: https://docs.python.org/3/library/pickle.html

[14] "Modified version of electionguard-python." [Online]. Available: https://github.com/Fredilein/electionguard-python

[15] "Vuejs." [Online]. Available: https://vuejs.org

[16] "Socket.io." [Online]. Available: https://socket.io

[17] "elgamal.js." [Online]. Available: https://github.com/kripod/elgamal.js