Chair for Embedded Systems
Universität Augsburg

Master's Thesis

Implementation of an IoT based Electronic Voting Machine

Gabriel Cmiel

Gutachter/Examiner:	Prof. Dr. Sebastian Altmeyer
Zweitgutachter/Second examiner:	Prof. Dr. Bernhard Bauer
Betreuer/Supervisor:	Prof. Dr. Sebastian Altmeyer
Date:	2nd March 2025

written at
Chair for Embedded Systems
Prof. Dr. Sebastian Altmeyer
Institute of Computer Science
University of Augsburg
D–86135 Augsburg, Germany
https://www.Informatik.uni-augsburg.de

Contents

Abstract

Eine kurze Zusammenfassung der Ausarbeitung.

List of Abbreviations

BSI Bundesamt für Sicherheit in der Informationstechnik

E2E End-to-end

VVSG Voluntary Voting System Guidelines

ZK Zero-Knowledge

PKE Public-Key Encryption

BLE Bluetooth Low Energy

AES Advanced Encryption Standard

SHA Secure Hash Algorithm

RSA Rivest-Shamir-Adleman

RNG Random Number Generator

TRNG True Random Number Generator

DRBG Deterministic Random Bit Generator

HMAC Keyed-Hash Message Authentication Code

KDF Key Derivation Function

ECC Elliptic Curve Cryptography

NAN Neighbor Awareness Networking

IoT Internet of Things

QoS Quality of Service

MTU Maximum Transmission Unit

AP Access Point

PPDA Privacy-Preserving Data Aggregation

MPC Multi-Party Computation

RTT Round Trip Time

1. Introduction

The Internet of Things (IoT) refers to a network of interconnected devices, objects, and systems embedded with sensors, software, and other technologies to collect and exchange data. These devices, ranging from smart appliances to industrial machines, enable communication and automation across various sectors [Roy+25, p. 1]. However, the sharing of sensitive data in IoT systems raises critical concerns about confidentiality and integrity, particularly due to limited computational resources, diverse standards, and network vulnerabilities [Hat24, p. 1].

Data aggregation-the process of gathering and summarizing information from multiple sources-is crucial for IoT data analysis but introduces privacy risks. For instance, in smart metering systems, Privacy-Preserving Data Aggregation (PPDA) is a leading solution for securing consumer data by securely aggregating meter readings at the gateways, preventing attackers from identifying individual user profiles [Wil22, p. 2]. While various security techniques have been developed, PPDA is considered more convenient. Many data aggregation schemes use cryptographic techniques, such as homomorphic encryption, to encrypt energy consumption data, their computationally intensive often renders them impractical for resource-constrained IoT devices [KQM21, pp. 113–114]. PPDA techniques also apply to other domains like electronic voting. Verifiable voting systems, for example, use homomorphic encryption to tally ballots while preserving voter anonymity. This method breaks the link between individual voters and their votes, keeping them secret [Mos+24, p. 53]. However, centralised decryption by a single tallier risk exposing individual votes. The tallier who owns the decryption key can decrypt all individual votes and learn how each voter has voted. To mitigate this, threshold cryptographic-a subfield of Multi-Party Computation (MPC)-distributes decryption keys among multiple talliers, requiring collaboration among the parties in order to decrypt the results [Mos+24, p. 40]. While threshold schemes enhance security, their reliance on synchronized interactions introduce communication bottlenecks [Mos+24, p. 45]. Verifiability and accountability are equilly critical in PPDA systems. Verifiability ensures the correctness of aggregated results (e.g., election outcomes) [Mos+24, p. 4], while accountability enables precise identification of error sources [Mos+24, pp. 10, 27]. Zero-Knowledge (ZK) proofs-a foundational cryptographic tool- allow parties to validate computational steps without revealing sensitive data [Mos+24, p. 13], though they incur additional overhead.

1.1. Research questions

The cryptographic building blocks of End-to-end (E2E) voting systems, such as homomorphic encryption and ZK proofs, threshold schemes enhance security but introduce communication and computation overhead. This thesis evaluates the implementation challenges and performance characteristics of E2E verifiable voting systems in resource-constrained IoT environments, focusing on the ElectionGuard 1.0 specification deployed on ESP32 microcontrollers:

1. **Implementability:** Can the ElectionGuard 1.0 specification be adapted to operate reliably on IoT devices with:
   - 520 KB RAM (ESP32-WROOM-32 constraints)
   - Limited cryptographic acceleration
   - Intermittent network connectivity
2. **Protocol Suitability:** Which communication protocol optimizes the trade-off between:
   - Feasibility
   - Latency
   - Functionality
   - Bandwidth

   in multi-device workflows?

1.2. Outline

The thesis is structured as follows:

- **Chapter 2: Background** - Presents cryptographic building blocks, ElectionGuard, and the ESP32 microcontroller.
- **Chapter 3: IoT Voting System Implementation** - Details the ESP32-based experimental setup, including:

  - Hardware/software Design
  - Computation
  - Communication

  Empirical results for network latency and computation performance are analyzed.

- **Chapter 4: Evaluation & Discussion** - Compares the results and discusses:
  - Recommendations

2. Background

2.1. Cryptography

Cryptography is the science of securing information through encryption. Encryption, also referred to as ciphering, involves transforming a message into an incomprehensible format [Ert07, p. 18]. The security of all cryptographic methods fundamentally relies on the difficulty of guessing a secret key or obtaining it through unauthorized means. While it is possible to guess a key, the likelihood diminishes as the lenght of the key increases. It should be noted that there is no absolute security in cryptography [Ert07, p. 25].

Practically all cryptographic methods aim to achieve one or more of the following properties [Ert07, p. 18]:

- **Confidentiality:** The aim of confidentiality is to make it impossible or difficult for unathorized persons to read a message [Ert07, p. 18].
- **Authenticity:** This property ensures that the recipient can verify the identity of the sender, ensuring that the message is not from an unauthorized sender [Ert07, p. 18].
- **Integrity:** This denotes that the message has not been altered during transmission [Ert07, p. 18].
- **Non-repudiation:** This means that the sender cannot later deny having sent a message [Ert07, p. 18].

Cryptographic algorithms are mathematical functions used for encryption and decryption [Ert07, p. 19]. A given cryptographic algorithm can be applied in various ways across different applications. To ensure that an application operates consistently and correctly, cryptographic protocols are defined. Cryptographic protocols are procedures that govern the flow of transactions within specific applications [Ert07, p. 22].

2.2. Cryptography in Voting Systems

The integration of cryptographic methods into voting systems has been a topic of discussion for several decades [Mos+24, p. 6]. In 1981, David Chaum introduced a cryptographic technique based on public key cryptography that effectively conceals both the identity of participants and the content of their communication. The untracable mail system requires messages to pass through a cascade of mixes (also known as a decryption mix nets) [Cha81, p. 86] [Mos+24, p. 84]. Chaum proposes that these cryptographic techniques can be adapted for use in elections. In this model, individual voters communicate with a record-keeping organization or an authorized party under a unique pseudonym, provided that this pseudonym appears in a roster of approved clients. This allows the organisation to verify that the message was sent by a registered voter while ensuring that the message was not altered during transmission [Cha81, p. 84].

The application of Chaum's method adheres to the four fundamental properties of cryptographic security:

- **Confidentiality:** Ensures that the voter's communication remains private from unauthorized entities.
- **Authenticity:** Confirms that the message indeed originated from a registered voter.
- **Integrity:** Guarantees that the message content was not modified during transmission.
- **Non-repudiation:** Provides assurance that the sender cannot deny having sent the message.

2.3. Building Public Trust in Electronic Elections

For a voting process to be deemed trustworthy, it is essential to provide voters and observers with compelling evidence that the election has been conducted properly while maintaining confidentiality (e.g., ballot secrecy). This aspect of public trust is further complicated by the necessity of trusting not only election officials but also the software and hardware utilized in the election process. Fortunately, modern cryptographic techniques provide viable solutions for ensuring both verifiability and confidentiality [Mos+24, p. 6].The objective of employing such methods is to minimize reliance on individual components of the voting system. Independent auditors

should be able to verify the correctness of the final election results without compromising the confidentiality of individual votes. Essentially, the goal is to reveal no more information about the votes than what can be inferred from the final tally [Mos+24, pp. 6, 10].

2.4. End-to-End Verifiability

A study conducted by the German Federal Office for Information Security Bundesamt für Sicherheit in der Informationstechnik (BSI) highlighted that **end-to-end E2E verifiability** is regarded as the gold standard for achieving the aforementioned goal in electronic voting systems [Mos+24, p. 10]. Furthermore, the Voluntary Voting System Guidelines Voluntary Voting System Guidelines (VVSG) 2.0, adopted by the U.S. Election Assistance Commission, mandates that voting systems must be software-independent. These guidelines are designed for designers and manufacturers developing voting systems [Com17]. The VVSG 2.0 outlines two primary methods for achieving software independence - the use of independent voter-verifiable paper records and E2E verifiable voting systems [Com21b, p. 181].

2.4.1. Key Components of End-to-End Verifiability

E2E verifiability encompasses two principal components [Ben+14, p. 2]:

- **Cast As Intended:** Voters can verify that their selections - whether indicated electronically, on paper, or by other means - are recorded correctly [Ben+14, p. 2].
- **Tallied As Cast:** Any member of the public is able to verify that every recorded vote is included correctly in the final tally [Ben+14, p. 2].

All E2E Verifiable voting systems incorporate cryptographic building blocks at their core. The most important and recurring cryptographic building blocks include [Mos+24, p. 13]:

- **Public-Key Encryption Public-Key Encryption (PKE):** Most verifiable voting systems use PKE to encrypt sensitive data, such as votes, using a public key.This ensures that authorized parties possessing the corresponding secret key can decrypt the data [Mos+24, p. 13].
- **Commitments:** Similar to PKE, commitments also serve to protect sensitive

data. However, in this case, the data cannot be decrypted using a secret key, but only with specific information generated during the individual commitment process, which is then distributed to selected parties [Mos+24, p. 13].

- **Digital Signatures:** These are commonly used in voting systems to allow different parties to confirm that the messages they are receiving originate from the indicated party [Mos+24, p. 13].

- **Zero-Knowledge ZK Proofs:** This technique permits a party to demonstrate that it has correctly performed a certain computational operation without revealing any additional information, such as the secret key involved in the computation [Mos+24, p. 13].

- **Threshold Secret Sharing:** This method is utilized to distribute information about a secret (e.g., a secret key) among multiple parties. A predetermined threshold of those parties must cooperate to reconstruct the secret from their individual shares [Mos+24, p. 13].

2.5. End-to-End Verifiable Software Libraries

Implementing an E2E verifiable voting system is a multifaceted challenge that requires specialized knowledge in cryptography alongside a foundation in software engineering. Successful implementation requires a comprehensive understanding of the specific algorithms and their correct implementation. Fortunately, several high-quality, well-maintained and tested software libraries are available, designed to simplify the implementability of E2E verifiable voting systems through robust cryptographic building blocks. Notable libraries include **CHVote**, **ElectionGuard**, **Verificatum**, **Belenios**, and **Swiss Post** [Mos+24, pp. 11, 26]. All of the aforementioned libraries use ElGamal's malleable PKE scheme, which is the most prevalent implementation in today's systems. The original ElGamal scheme is multiplicaticely homomorphic, frequently an exponential variant of ElGamal is employed, making it additively homomorphic [Mos+24, p. 40].

2.6. ElectionGuard Overview

Microsoft's **ElectionGuard** is a toolkit designed to provide E2E verifiable elections by seperating cryptographic functions from the core mechanisms and user interfaces of voting systems. This seperation empowers ElectionGuard to offer simple inter-

faces that can be used without requiring cryptographic expertise. Consequently, existing voting systems can function alongside ElectionGuard without necessitating their replacement, while still producing independently-verifiable tallies [Ben+24, pp. 1–2].

The cryptographic design of ElectionGuard is largely inspired by the cryptographic voting protocol developed by Cohen (now Benaloh) and Fischer in 1985, as well as the voting protocol by Cramer, Gennaro, and Schoenmakers in 1997 [Ben+24, p. 5]. One of the first pilots employing ElectionGuard was conducted in Preston, Idaho, on November 8, 2022, using the Verity scanner from Hart InterCivic, integrated with ElectionGuard. This pilot provided one of the first opportunities to see how an E2E verifiable election operates within a real election context. [23a, p. 4].

In all applications, the election process utilizing ElectionGuard begins with a key-generation ceremony, during which an election administrator collaborates with guardians to create an election key. The administrator will then work again with the guardians to produce verifiable tallies at the conclusion of the election. What transpires in between these stages can differ [Ben+24, p. 20]. The flexibility of ElectionGuard is a novel feature and one of its primary benefits [Ben+24, p. 22].

2.7. Phases of ElectionGuard

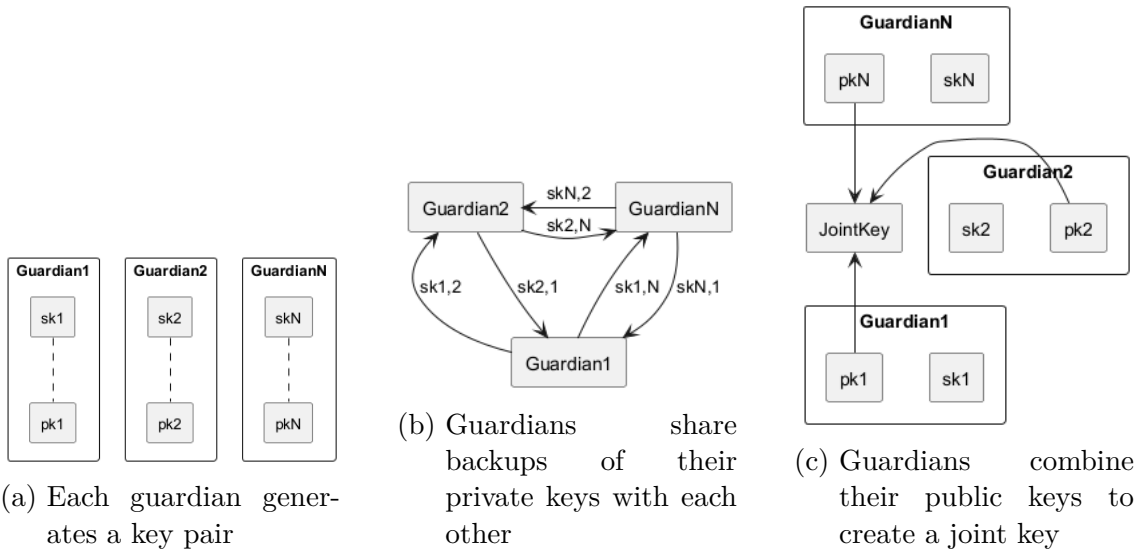
The election process in ElectionGuard can be divided into three main phases: Pre-election, Intra-election, and Post-election. Each phases involves distinc activities. Below, we will explore each phase in detail.

2.7.1. Pre-election Key Generation

.

The pre-election phase encompasses the administrative tasks necessary to configure the election and includes the key generation ceremony.

The **election manifest** defines the parameters and structure of the election. It ensures that the ElectionGuard software can correctly record ballots. The manifest defines common elements when conducting an election, such as locations, candidates, parties, contests, and ballot styles. Its structure is largely based on the NIST SP-1500-100 Election Results Common Data Format Specification and the Civics Common Standard Data Specification [].



(a) Each guardian generates a key pair

(b) Guardians share backups of their private keys with each other

(c) Guardians combine their public keys to create a joint key

Figure 2.1.: Key Ceremony. Adapted from []

In addition to defining the election, specific **cryptographic parameters** must be defined. A significant aspect is selecting mathematical constants used in cryptographic operations. The ElectionGuard specification provides both standard and reduced values for these constants [BN22, pp. 21, 36–38]. Furthermore, the pre-election phase defines the number of participating guardians and the minimum quorum of guardians required for the post-election phase. These parameters play a significant role in the key generation ceremony [Ben+24, pp. 8–9].

The key generation ceremony involves trustworthy individuals, known as **guardians**, who collaborate to create a joint election key through the multiplication of individual public keys. This joint key is essential for encrypting data, as it requires all guardians to apply their private keys for decryption. This process minimize the risk associated with a single party being responsible for the property of ballot confidentiality. If some guardians are unavailable in the post-election phase the quorum count allows a specified number of guardians to reconstruct missing private keys by sharing "backup" copies among themselves during the pre-election phase [Ben+24, p. 8] []. It is crucial to recognize that at least some level of trust is necessary for any component of the system. To ensure the distribution of trust is effective, it is essential that the guardians are genuinely independent of each other [Mos+24, p. 92].

The last step in the pre-election phase involves loading the election manifest, cryptographic parameters, and the joint key into an encryption device responsible for encrypting ballots throughout the intra-election phase. [Ben+24, p. 8].

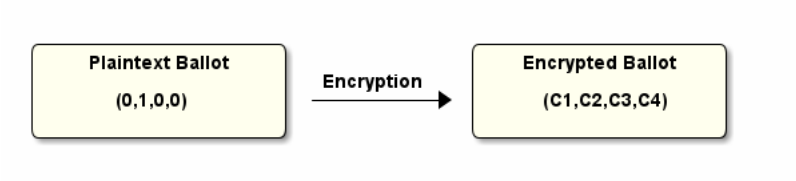


Figure 2.2.: Representation of plain and encrypted ballots

2.7.2. Intra-election Ballot Encryption

.

During the election phase, the ballots are encrypted and consist entirely of exponential ElGamal encryptions of binary values: a "1" signifies support for a particular option, while a "0" indicates a lack of support [Ben+24, p. 11] [BN22, p. 12]. In scenarios where a voter has multiple options, such as four choices in a single contest, the encrypted ballot will contain four corresponding encrypted bits. The exponential form of ElGamal encryption possesses an additive homomorphic property; therefore, the product of the encrypted reflects the count of selected options [BN22, p. 5].

A simple representation of this scenario is shown in Figure 2.2. The plaintext ballot consists of four options in which the second option has been selected. When all encrypted values are combined homomorphically by multiplying them, the result encrypts the count of selections made. This technique ensures that the ballot does not contain excessive votes [BN22, p. 5].

While encryption itself is a straightforward process, a significant portion of ElectionGuard's effort is dedicated to creating externally verifiable artifacts to confirm that each encryption is well-formed [BN22, p. 3]. ZK proofs are employed to validate that the encryptions correspond to the values of 1 and 0 [Ben+24, p. 11]. A Chaum-Pedersen proof verifies whether an encryption repreents a specific value. By utilizing the Cramer-Damgård-Schoenmakers technique, it can be demonstrated that an encryption belongs to a specific set of values, such as 0 or 1. The proofs are made non-interactive through the Fiat-Shamir heuristic [BN22, pp. 6, 13].

Once the encryption of a ballot is finalized, a confirmation code is generated for the voter [BN22, p. 17]. This confirmation code is a cryptographic hash derived entirely from the encrypted ballot [Ben+24, p. 14]. With the confirmation code, a voter can either cast the associated ballot or choose to spoilt it and restart the ballot preperation process. The two choices are mutually exclusive, since challenging reveals the selections on the ballot. A challenged ballot can never be cast, while a cast ballot cannot be challenged anymore. This casting and spoiling mechanism acts as an interactive proof to assure voters that their selections have been correctly

encrypted [BN22, p. 17] [].

2.7.3. Post-election Decryption of Tallies

At the end of the voting period, all encrypted ballots submitted for tallying are homomorphically combined to produce an encrypted tally [BN22, pp. 5, 18] [Ben+24, p. 15]. Each available guardian utilizes their private key to generate a decryption share, which represents a partial decryption for the encrypted tally or spoiled ballots. Decrypting spoiled ballots is unnecessary for determining the election outcome but may be performed to support cast-as-intended verifiability [Ben+24, pp. 15, 17] [BN22, p. 18]. To verify the correctness of these shares, guardians also publish a Chaum-Pedersen proof of each share [BN22, p. 18]. The full decryption is achieved through the ordinary multiplication of these partial decryptions. If any guardians are unavailable during decryption, the remaining guardians can utilize the backups to reconstruct the missing shares [].

To conclude the election, the final step involves the publication of the election record, which is vital for the integrity of a verifiable election. This record contains a complete account of all election artifacts, including the election manifest, cryptographic parameters, and the decrypted tally [BN22, p. 24]. Independent verification software can be leveraged at any time after completion of an election to confirm the election's integrity [Ben+24, p. 6]. The true value of a verifiable election is fully realized only when the election is actively verified by voters, election obsevers, or news organisations [BN22, p. 17].

2.8. ESP32-WROOM-32 Overview

The **ESP32-WROOM-32** is a powerful microcontroller module that integrates Wi-Fi, Bluetooth, and Bluetooth Low Energy Bluetooth Low Energy (BLE) technologies, along with 4MB of integrated SPI flash memory. At the core of this module is the **ESP32-D0WDQ6** chip, which belongs to the ESP32 series of chips [23b, p. 6]. In the context of this thesis, the term **ESP32** broadly refers to the family of chips within the series rather than a specific variant.

It's important to note that the ESP32-WROOM-32 module is marked as not recommended for new designs. Instead, designers are advised to use the ESP32-WROOM-32E, which is built around either the ESP32-D0WD-V3 or the ESP32-D0WDR2-V3 chips. These later revisions rectify some hardware issues present in previous versions [24b, p. 1], [25c, p. 11], [24a, pp. 3–4].The ESP32-D0WDQ6 chip is susceptible to

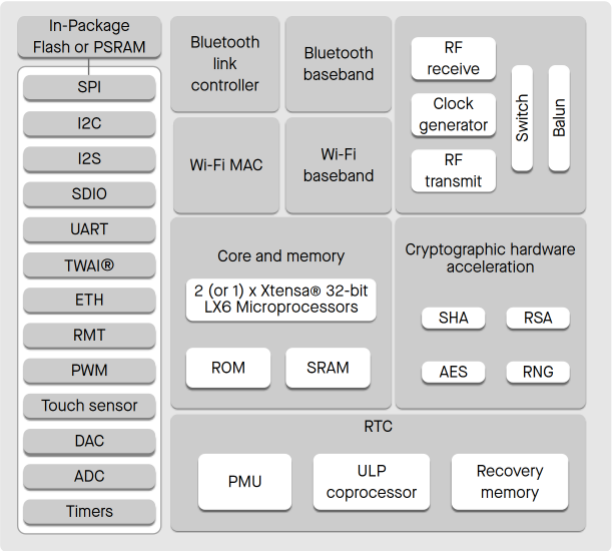


Figure 2.3.: ESP32 Functional Block Diagram

fault injection attacks. Sucessfully carrying out a fault injection attack could enable an attacker to recover the Flash Encryption key. This key allows unauthorized access to the device's flash contents, including firmware and data stored in flash. Such an attack necessitates physical access to the device [19].

ESP32 is designed for robustness, versatility, and reliability across a wide range of applications and power scenarios [25c, p. 2]. Due to its low power consumption, ESP32 is an ideal choice for IoT applications spanning smart homes, industrial automation, consumer electronics, healthcare, and battery-powered electronics [25c, p. 5] [23b, p. 6].

ESP32s come with either a single or dual-core Xtensa® 32-bit LX6 microprocessor, operating at frequencies of up to 240 MHz. They come equipped with:

- **448 KB of ROM** for booting and core functions [25c, pp. 4–5]
- **520 KB of SRAM** for data and instructions [25c, pp. 4–5]
- **34 programmable GPIOs** [25c, pp. 4–5]
- **Cryptographic hardware acceleration capabilities** [25c, pp. 4–5]

The supported cryptographic hardware acceleration capabilities include Advanced Encryption Standard (AES), SHA, RSA, and RNG [25c, pp. 4–5]. A functional block diagram illustrating the components and subsystems within ESP32 is presented in Figure 2.3.

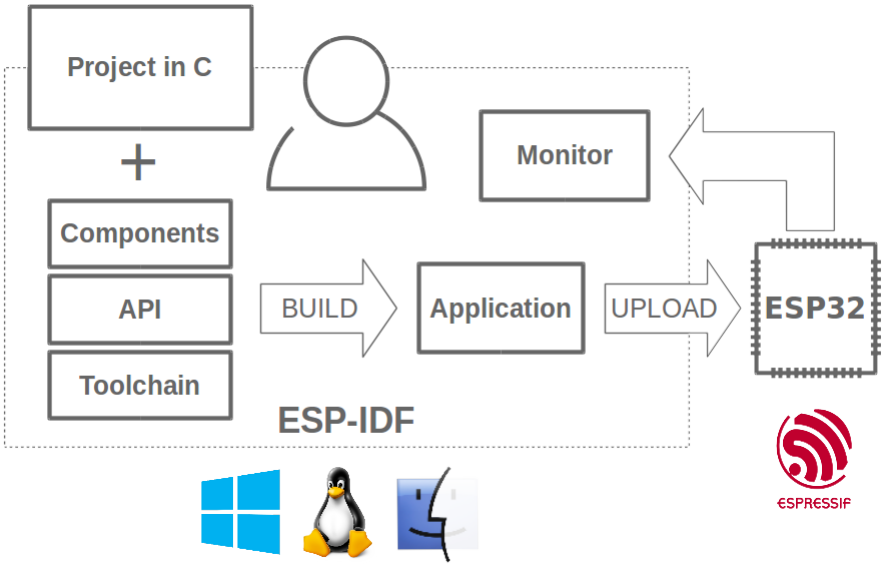


Figure 2.4.: ESP32 Functional Block Diagram

2.8.1. Development Environment

For application development, Espressif, the company behind the ESP32, offers the ESP-IDF (Espressif IoT Development Framework). ESP-IDF encompasses a comprehensive toolchain, API components, and defined workflows tailored for ESP32 application development **esp32-prog**. The development workflow for ESP-IDF is depicted in Figure **??**.

ESP-IDF consists of components written specifically for ESP-IDF as well as third-party libraries.[25b] For example, the real-time operating system kernel used for ESP-IDF applications is the third-party component called FreeRTOS [25b] [23b, p. 6]. ESP-IDF projects use the same componment based approach and can be seen as an amalgamation of a number of components [25b]. Components are modular pieces of standalone code that are compiled into static libraries which are subsequently linked into a project. These components can originate from various sources, including external git repositories or Espressif's Component Manager. Each component may also come with a Kconfig file, allowing users to configure the component through a text-based menu system [25b].

3. IoT Voting System Implementation

3.1. Hardware Setup

The experimental setup comprises the following components:

- **Tallier Node:** Sony Vaio E Series laptop (Intel Core i5-2450M, 8 GB DDR3 RAM) running Antix Linux 23.1, serving three critical roles:
  - Wi-Fi Access Point (using hostapd v2.9)
  - MQTT Broker (using Eclipse Mosquitto v2.0.20)
  - Election Tallier
- **Guardian Nodes:** 2× NodeMCU ESP32 boards (ESP32-WROOM-32 modules) with:
  - 240 MHz dual-core Xtensa LX6 CPU
  - 520 KB SRAM, 4 MB flash
  - Integrated 802.11 b/g/n Wi-Fi

3.2. Election Setup

The Guardian nodes are responsible for distributing trust in the encryption and decryption processess managed by the Tallier. Their responsibilities are divided into two main phases: the key generation ceremony, and the decryption process.

3.2.1. Cryptographic constants

Before implementing the ElectionGuard specification, it is essential to establish the mathematical constants for the cryptographic operations. ElectionGuard specifies standard values for the primes (p) and (q) and a generator (g) [BN22, p. 21]. The standard baseline parameters include:

- A 4096-bit prime (p) [BN22, p. 22]
- A 4096-bit generator (g) [BN22, p. 23]
- A 256-bit prime (q) [BN22, p. 21]

For this experiment, we will use reduced parameters that offer better performance at a lower security level [BN22, p. 23]. The reduced parameters are defined as

- A 3072-bit prime (p) [BN22, p. 36]
- A 3072-bit generator (g) [BN22, pp. 36–37]
- A 256-bit prime (q) [BN22, p. 36]

3.2.2. Pre-Election Key Generation

The pre-election phase involves administrative tasks necessary to configure the election, including the key generation ceremony.

The tallier node defines the election manifest, sets the cryptographic constants according to the reduced baseline parameters (as discussed in 3.2.1), and sets the quorum size. Given that only two Guardian nodes are available for this experiment the quorum size is set to two. Figure 3.1 shows a visualisation of the election manifest used in this election. The election is limited to a simple YES/NO/ABSTAIN referendum. The Tallier collaborates with the Guardians to generate a joint key used to encrypt the ballots in the next election phase.

Figure 3.2 illustrates the communication sequence involved in this phase. The Tallier sends the ceremony details required for the key generation ceremony to the Guardians. The quorum size influences how each Guardian generates their ElectionKeyPair. After generating their ElectionKeyPair and stripping the secret key for transmission, the Guardians exchange their ElectionKeyPairs. Each Guardian generates a designatted backup for each ElectionKeyPair received, which is then sent

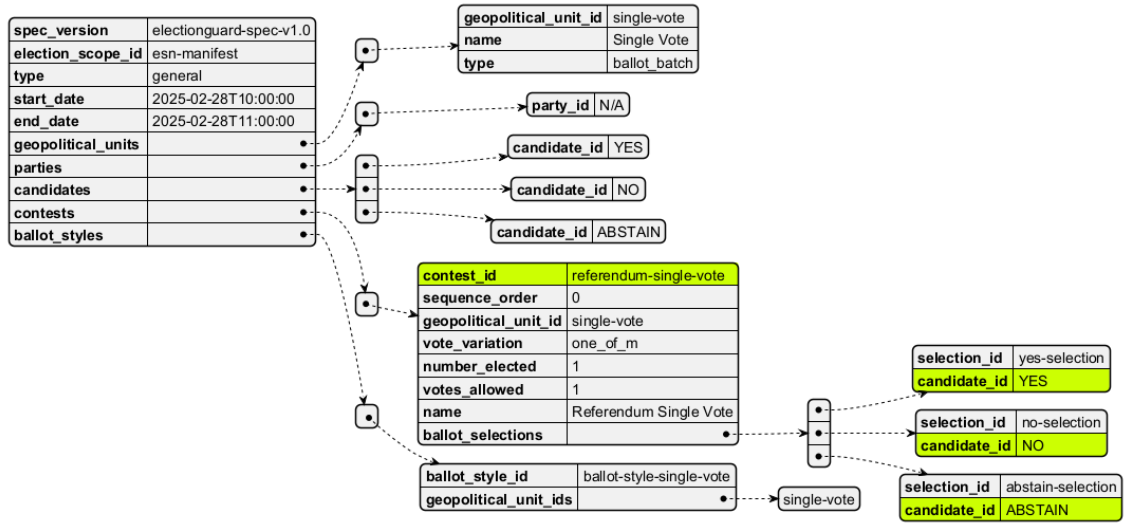


Figure 3.1.: Visualisation of the JSON Election Manifest

back to the sending Guardian. Each backup contains a proof that must be verified by the receiver. The challenge mechanism and eviction mechanism for failed verifications are not implemented in this experiment. This verification process for a challenge or a backup is identical. Once all backups successfully verify, the ElectionKeyPairs are combined into a joint key.

3.2.3. Intra-Election Ballot Encryption

During this election, the Tallier simulates the voting process by generating random ballots and encrypting them using the joint key established in the previous phase. The ballots are cast and processed as detailed in Appendix A.1.1.

3.2.4. Post-Election Decryption of Tallies

In the post-election phase, all encrypted ballots from the previous phase are homomorphically aggregated to produce an encrypted tally. This encrypted tally is send to the Guardians for decryption. Each guardian uses their private key to generate a decryption share, which is essentialy a partial decryption. The Tallier then combines all decryption shares into a decrypted tally. Figure 3.3 illustrates the communication involved in this phase. The decryption of spoiled ballots is optional and not implemented in this experiment. The decryption process of the spoiled ballots mirrors that of the encrypted tallly. The case of missing Guardians is also not addressed in this experiment. Compensating for missing decryptions functions similarly to the

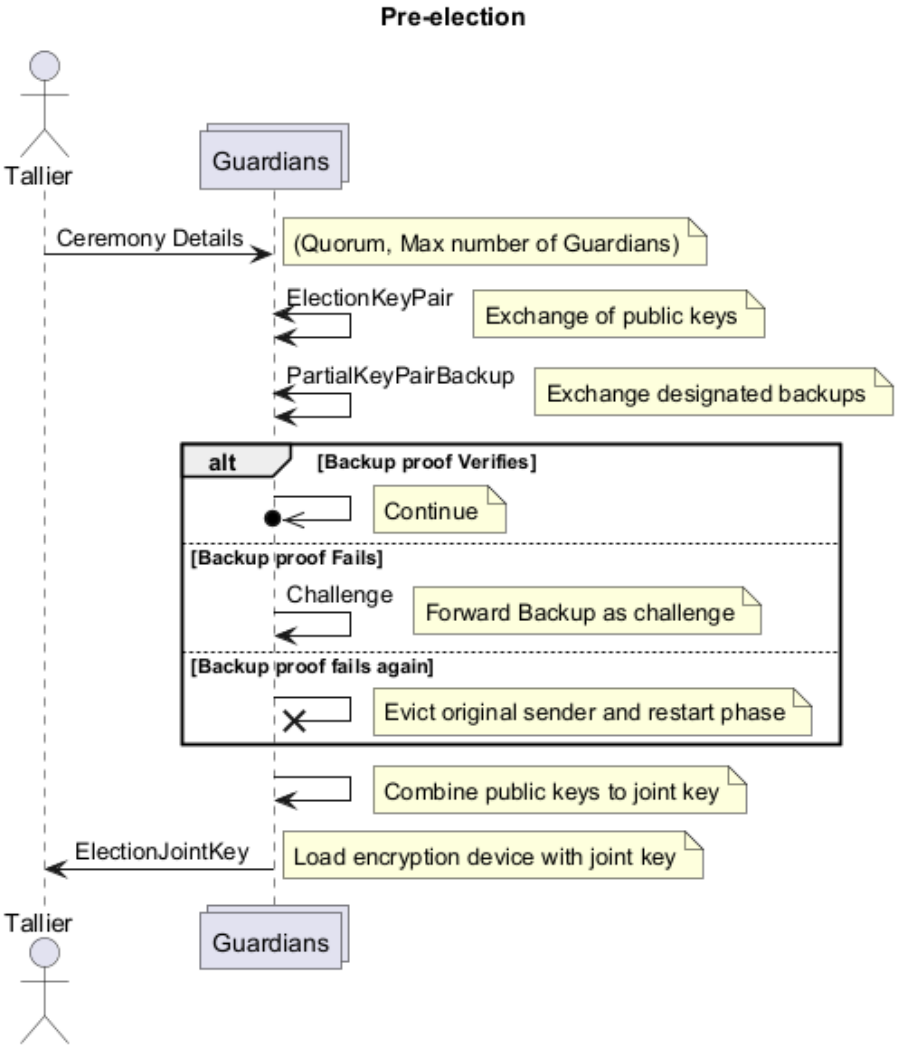


Figure 3.2.: Communication Sequence in the Pre-election phase

generation of the decryption share. The publication of the election artifacts is also not implemented. Publishing the election artifacts is not implemented, the interaction between the Guardians and the Tallier requires inherent verification due to the proofs involved.

3.3. Comparison of ElectionGuard Implementations

Due to the open-source nature of ElectionGuard, various community ports exist, such as a Java port []. However, this discussion focuses solely on the official ports from Microsoft. There are 2 ElectionGuard libraries implementing the ElectionGuard 1.0 specification: a Python reference implementation and a C++ reference implementation. The Python implementation encompasses the entire suite of functionality and processes necessary to implement E2E verifiable election as part of a voting system [Ini]. It is designed to be universal and portable, albeit less performant []. In contrast, the C++ reference implementation focuses solely on the encryption components and is optimized for execution on low-powered devices **cpp-reference**.

3.3.1. Python Reference

Running the Python reference implementation on the ESP32 may be feasible through **MicroPython**, an implementation of Python 3.x targeted for microcontrollers and embedded systems. MicroPython adapts standard Python library functionalities to accommodate the limitations inherent of microcontrollers, such as restricted memory and processing speed [Mic] [IE20, p. 234]. There are drawbacks to this approach, as essential modules, functions, and classes may be absent in MicroPython [Mic]. Additionally, applications developed in MicroPython are prone to memory fragmentation and may experience issues with objects expending in size. [IE20, p. 234]. A comparative study of software-based SHA-256 computation for the ESP32 revealed that the performance of MicroPython was inferior to that of a C implementation [IE20, p. 237]. For instance, the C implementation outperformed the MicroPython implementation by 45% [IE20, p. 237].

The Electionguard Python is not designed to be performant, and compatibility issues may arise when using MicroPython, along with potential memory fragmentation problems. Due to these limitations, the Python reference implementation may not be the most suitable choice for the ESP32. However, it remains a valuable resource for understanding the ElectionGuard specification and can serve as a reference for developing a possible port. Our Tallier node, which has more computational resource

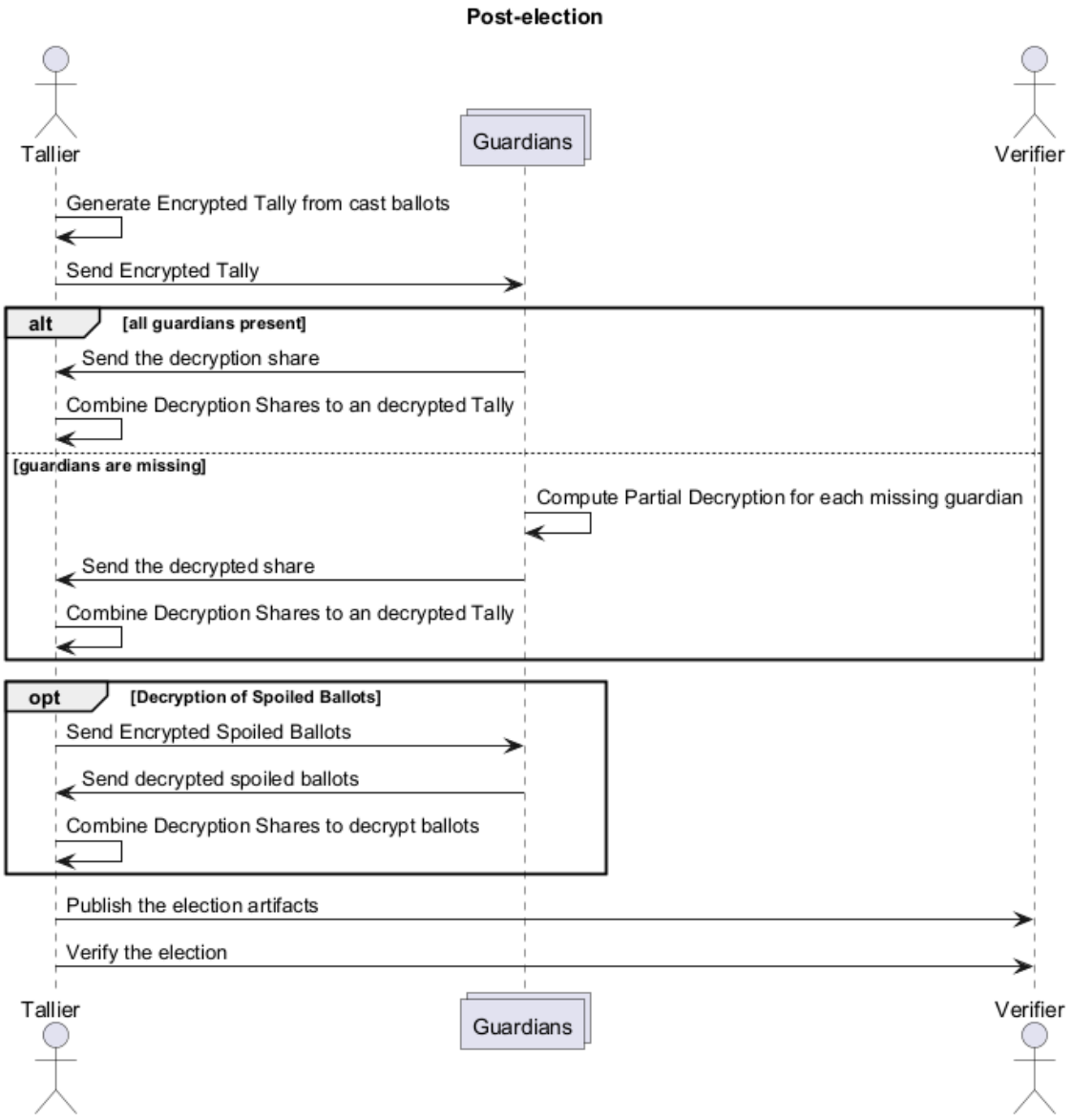


Figure 3.3.: Communication Sequence in the Post-election phase

compared to our Guardians, will run the Python reference implementation as a Python package for convenience.

3.3.2. C++ Reference

The ElectionGuard C++ reference implements focuses solely on the encryption library, omitting abstractions such as Guardians. This implementation is designed for execution on low-powered devices with Intel Atom-level processor performance in mind **cpp-reference** []. Porting the C++ library to an ESP32 should be feasible, as ESP-IDF supports C++ application development. Certain C++ features, such as exception handling, must be enabled in the project configuration beforehand. The ESP32 also supports C++ threads, which are implemented as wrappers around C pthreads, which in turn wrap around FreeRTOS tasks [25b].

Compared to the Python implementation, the C++ implementation incorporates optimisations to accelerate computations of certain modular exponentiations **cpp-reference**. These optimisations come at a memory cost. The optimisation for modular exponentiation uses pre-computed tables to speed up calculations for certain exponentiations **cpp-reference**. A modular exponentiation computes $X^Y P$. This optimization is possible because many of the exponentiations in ElectionGuard are performed with a fixed base, such as the generator (g). The pre-computed table contains certain powers of these bases, allowing the computation G^Y to be turned into a series of table lookups [Ben+24, pp. 22–23].

To store the table, we must consider the capabilities of the ESP32. The ESP32 has 520 KB of volatile memory (SRAM) and 4 MB of non-volatile memory (in-package flash). The SRAM is divided into 320 KB of DRAM and 200 KB of IRAM. IRAM is used for instruction memory, while DRAM is used for data memory. The maximum statically allocated DRAM usage is 160 KB, the remaining 160 KB can only be allocated as heap memory [25d].

Listing 3.1: FixedBaseTable Definition

```
typedef std::array<std::array<uint64_t[MAX_P_LEN], OrderBits>,
    TableLength> FixedBaseTable;
```

The lookup table (FixedBaseTable) is implemented and tuned to the following values. Any modifications to these parameters could affect the function's internal operations.

- b= 256 (OrderBits)

- k = 8 (WindowSize)
- m = 32 (TableLength)

To estimate the memory requirements of the **FixedBaseTable**, we can calculate the total size in bytes as follows:

$$\text{FixedBaseTableSize} = \text{sizeof(uint64\_t)} \times \text{MAX\_P\_LEN} \times \text{OrderBits} \times \text{TableLength} \tag{3.1}$$

Given that MAX_P_LEN is defined as 64 (4096-bit), we will calulate with 48 (3072-bit) since we are using the reduced baseline parameters. Substituting the values into the equation gives:

$$\text{FixedBaseTableSize} = 8 \times 48 \times 256 \times 32 = 3145728\text{bytes} = 3\text{MB} \tag{3.2}$$

Consequently, the **FixedBaseTable** exceeds the volatile memory capacity of the ESP32. Although storing the table in the non-volatile memory might be feasible, it may require further optimizations because it must accommodate the bootloader and the application binary. Reducing the **WindowSize** would result in smaller tables and reduced memory usage, but it simultaneously increases the number of multiplications [BN22, p. 22]. Given these memory requirements, it is evident that the C++ implementation would require further optimizations to be feasible on the ESP32. Therefore, we opt to implement a native C implementation based on the Python reference implementation.

3.3.3. Implementation Strategy for ESP32

ElectionGuard uses integer ElGamal cryptography within its specific cryptographic operations. The system performs four key operations on very large integer values: **modular exponentiation**, **modular multiplication**, **modular addition**, and **SHA-256** hash computation.

To handle the large integer values involved in these operations, specialized libraries for large integers may be employed, or the operations can be developed from scratch [BN22, pp. 21, 25–26]. When developing these modular operations from the ground up, it is common for intermediate values to become excessively large. Techniques such as modular reduction are often necessary to ensure that values remain manageable [BN22, pp. 21, 25–26]. Consequently, employing fast libraries for modular arithmetic becomes crucial for achieving good performance [Ben+24, p. 22]. The Python implementation uses the C-coded GnuMP library for large integer arithmetic, the C++ implementation uses HACL* a performant C implementation of a wide variety of cryptographic primitives [Ben+24, p. 22] **cpp-reference**. In ESP32

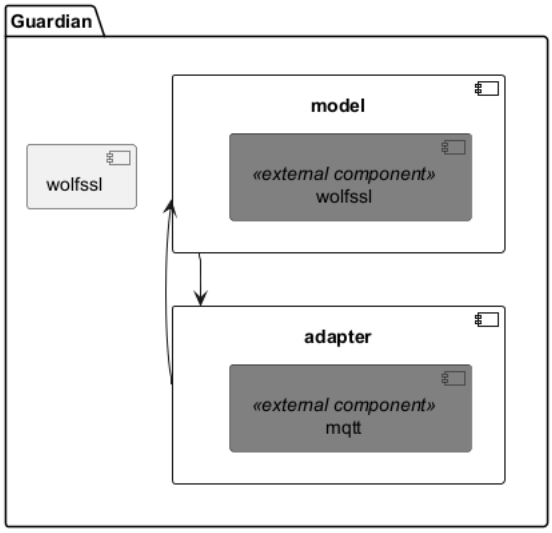


Figure 3.4.: Software components for the Guardian Prototype

cryptographic primitives are implemented through a fork of the mbedTLS library. Optionally, a port of the WolfSSL library is also available [25d]. The benefit of these two libraries over the other mentioned libraries is that these libraries include patches related to hardware routines for on-board cryptographic hardware acceleration [25d] [Inc25, p. 114]. The ESP32 supports supports several cryptographic hardware acceleration capabilities including AES, SHA, RSA, and RNG as illustrated in the functional block diagram 2.3. These hardware accelerators significantly enhance operational speed and reduce software complexity for the aforementioned cryptographic primitives [25c, p. 32]. More details on the hardware acceleration capabilities of the ESP32 are provided in Section **??**.

Figure 3.4 detauls the ESP32 software stack

Figure 3.4 depicts the software components used in the Guardian ESP32 port. The model component includes the business logic for the guardians operations. The adapter component implements the communication aspects. The adapter component calls the model component to perform guardian specific operations. The model component uses the wolfSSL library for cryptographic primitives. The wolfSSL component is overwritten using c pre-processor macros to reduce the binary size by removing unused features. Furthermore, the library is configured to run on the memory-restricted ESP32 by configuring the library to use less memory, albait at a performance cost [Cmi25, pp. 56–58]. The adapter component implements a MQTT client. It is used in order to communicate with the Tallier and other Guardians through an intermediary the MQTT broker.The adapter component implements an event handler which response to events such as establishing a connection to the MQTT broker or receiving a message via the MQTT broker. Further information on the implementation of the adapter component and the communication can be

found in 3.5.

3.4. Hardware Acceleration

Fundamentally, ElectionGuard encryption is a CPU-bound operation [Ben+24, p. 24]. The ESP32 microcontroller features hardware acceleration for several cryptographic primitives, including AES, SHA, RSA, and RNG [25d]. These hardware accelerators significantly enhance the performance of cryptographic operations compared to software implementations [25c, p. 32]. The hardware accelerators are designed to offload the CPU from cryptographic operations, thereby reducing the computational load on the CPU and improving overall system performance [25c, p. 32]. The hardware accelerators are particularly beneficial for computationally intensive operations, such as modular exponentiation and modular multiplication, which are central to the ElectionGuard encryption process [Ben+24, p. 24]. In the following sections we will look into how each of the hardware accelerators can be utilized in the ElectionGuard implementation.

3.4.1. RNG

In the context of ElectionGuard, generating random values is crucial for various operations, including the generation of private keys and the use of nonces in various proofs [BN22, pp. 9, 13]. The ESP32 features a True Random Number Generator True Random Number Generator (TRNG) that can produce 32-bit random numbers that are suitable for cryptographic purposes. Unlike Deterministic Random Bit Generators Deterministic Random Bit Generator (DRBG), which rely on algorithms to produce random numbers, the ESP32's TRNG generates randomness from physical processes. This includes leveraging thermal noise and asynchronous clock mismatches, ensuring a high level of unpredictability, essential for cryptographic operations [25d, p. 604].

To utilize the TRNG effectively, it is necessary to enable a source for thermal noise; otherwise, the TRNG will return pseudo-random number [25d, p. 609]. The High-speed ADC is enabled automatically when the Wi-Fi or Bluetooth module is enabled, which is the case in our design [25d, p. 610]. When the noise is sourced from the high-speed ADC, it is advisable to read the **RNG_DATA_REG** register at a maximum rate of 5 MHz [25d, p. 609]. However, the values from the high-speed ADC can be saturated in extreme cases, leading to lower entropy. It is advisable to enable the SAR_ADC as a secondary noise source. The **RNG_DATA_REG**

register should then be read at a maximum rate of 500 kHz to obtain the maximum entropy [25d, p. 609].

In our ESP32 implementation , the rand_q function generates a random number below the constant 256 bit constant (q). To achieve this objective, we fill a buffer with 256-bits of randomness utilizing the system API function esp_fill_random(). To ensure that the generated 256-bit number does not exceed (q), we perform a modulo operation with the value of (q). To obtain a 256-bit number, esp_fill_random() reads the 32-bit RNG_DATA_REG register eight times. The function will busy-wait if the reading frequency exceeds acceptable limits [25d]. This limitation arises because the function must ensure that sufficient external entropy has been introduced into the hardware RNG state. For our use case the function should not delay. If the function delays, we should consider using a strong software DRBG such as mbedTLS CTR-DRBG, mbedTLS HMAC-DRBG, or WolfSSL DRBG, which can be initialized with the TRNG values as a seed [25d] [Inc25, p. 588]. The RNG accelerater does not give a performance benefit for our use case, it is used to ensure that the random numbers are cryptographically secure.

3.4.2. SHA Accelerator

ElectionGuard encrypts non-vote data, such as cryptographic shares of a guardian's private key (e.g. backups), using hashed ElGamal encryption. This method employs a key derivation function Key Derivation Function (KDF) to generate a key stream that is XORed with the data. The Keyed-Hash Message Authentication Code Keyed-Hash Message Authentication Code (HMAC) is used for message authentication and is integral to the implementation of the KDF. In ElectionGuard, HMAC is instantiated as HMAC-SHA-256, which uses the SHA-256 function [BN22, p. 7].

The SHA-256 hash function is frequently applied in various cryptographic operations within ElectionGuard, with implementation details provided in Appendix **??**. The ESP32 microcontroller is equipeed with a SHA Accelerator that significantly enhances the performance ofSHA operations compared to purely software implementations [25d, p. 589]. Notably, this accelerator supports the SHA-256 algorithm used in ElectionGuard. However, it processes only one message block at a time and does not handle padding operations. Therefore, software must manage the division of longer messages into 512-bit blocks, along with any required padding [25c, p. 2].

In multi-core environments, libraries like mbedTLS and WolfSSL implement fallback mechanisms to software implementations when multiple concurrent hashing operations are initiated. As a result, simultaneous computations revert to software calculations [Espb] [Incb]. Benchmarks uzilizing mbedTLS at processor speeds of

240 MHz reveal that hardware acceleration achieves performance nearly three times faster than software couterparts [Jin22, pp. 41–42]. When using the WolfSSL library with the fastmath library, benchmarks indicate that the SHA Accelerator operates more than eight times faster than its software counterpart. Thus, the SHA Accelerator is an effective solution for speeding up SHA-256 hashing operations on the ESP32.

The data format of hashes is ambiuous

The 1.0 specification under-spoecifies on how inputs the the cryptographic hash function should be serialized. This leads to compability issues between different implementations [Ben+24, pp. 23–24]. The Python implementation uses utf-8 hexadecimal encoding. Thus in order to ensure compability with the Python implementation, the C implementation should also use hexadecimal encoding. All big integer values need to be converted to hexadecimal strings before being hashed. This leads to a less efficient implementation.

3.4.3. RSA Accelerator

ElectionGuard's decision to use integer ElGamal instead of elliptic-curve ElGamal was driven by its conceptual simplicity and lower implementation barrier [Ben+24, p. 7]. While elliptic-curve cryptographic Elliptic Curve Cryptography (ECC) techniques offer computational advantages, such as reduced computing requirements and smaller key sizes for the same security level [Sag12, pp. 1, 6], the integer ElGamal approach aligns well with the ESP32 hardware. This is because the RSA algorithm, like integer ElGamal, relies on large integer arithmetic. Specifically, the ESP32 chip supports independent arithmetic operations, including large-number multiplication, large-number modular multiplication, and large-number modular exponentiation [25c, p. 32] [25d, p. 603]. Consequently, the RSA Accelerator can accelerate two key operations: modular multiplication and the computationally intensive modular exponentiation. However, modular addition cannot be accelerated using dedicated hardware and must rely on a software implementation.

The RSA Accelerator supports eight operand lengths for modular exponentiation and modular multiplication, including the reduced 3072-bit and even the 4096-bit baseline parameters used in our implementation [25d, p. 598]. The large-number modular exponentiation operation computes $Z = X^Y \bmod M$, while the large-number modular multiplication operation computes $Z = X \times Y \bmod M$. Both operations are based on Montgomery multiplication. In addition to the input arguments X, Y, and M, two additional arguments are required: the Montgomery Inverse $\overline{r}$ and the inverse of M'. These additional arguments are precomputed by software [25d, pp. 598–599].

The wolfSSL library defaults to a software implementation for smaller operands, whereas the mbedTLS library lacks such a fallback mechanism. Interestingly, using hardware acceleration for small operands can be less efficient than a software implementation. For example, in one test, the mbedTLS modular exponentiation function with hardware acceleration was 1.44 times slower for small operands but 12.84 times faster for large operands [Jin22, p. 51]. This inefficiency for small operands likely stems from the initialization overhead of the hardware accelerator, which outweighs the benefits for smaller values. However, the mbedTLS library provides functions that allow caching of the $\overline{r}$-inverse and M'-inverse values, which can significantly speed up operations [Jin22, p. 51]. Since calculating the $\overline{r}$-inverse is computationally expensive, precomputing and caching these values can enhance performance [Jin22, p. 51].

In our implementation, the modular exponentiation function switches to a software implementation for small values, as shown in Appendix **??**. This switch occurs during polynomial calculations, where the number of polynomials (starting from 0 and incrementing) is used as the exponent in the modular exponentiation operation. The polynomial function is detailed in Appendix **??**.

In summary, the RSA Accelerator effectively accelerates modular exponentiation and modular multiplication in the ElectionGuard implementation. The mbedTLS library offers efficiency gains through caching of the $\overline{r}$-inverse and M'-inverse values. However, its lack of a fallback mechanism for small operands necessitates an additional software implementation to avoid inefficiencies when using the RSA Accelerator with small values.

3.4.4. Performance Analysis

Quorum	Type	Average Time (s)	Standard Deviation (ms)
Quorum 2	HW	1.02	0.24
	SW	4.41	13.24
Quorum 3	HW	1.53	0.23
	SW	6.62	15.87
Quorum 4	HW	2.03	0.24
	SW	8.81	15.68
Quorum 5	HW	2.54	0.26
	SW	11.02	19.15
Quorum 6	HW	3.05	0.31
	SW	13.22	22.04

Table 3.1.: Comparison accelerated and non accelerated Key Generation with different Quorums, 30 measurements

Function	Type	Average Time (s)	Standard Deviation (ms)
Backup	HW	0.53	0.15
	SW	2.21	8.52
Verification	HW	1.06	0.1
	SW	2.58	0.04
Decryption	HW	2.29	1.61
	SW	9.95	19.4

Table 3.2.: Comparison of operations with and without hardware acceleration, 30 measurements

Table **??** compares the performance of key generation with and without hardware acceleration across different quorums . HW acceleration is more then 4 times faster than the SW implementation across all quorum sizes (e.g., 1.02s vs. 4.41s for Quorum 2). HW shows remarkably low standard deviations (0.24-0.31ms) compared to SW (13.24-22.04ms). This indicates that HW accelerated indicating a more stalbe, deterministic performance. The average time increase by 0.5s per additional quorum member (1.02s → 3.05s from Q2→Q6). The SW implementation shows a similar trend, but with a steeper increase by 2.2s per additional member (4.41s → 13.22s from Q2→Q6). This suggest O(n) complexity for both implementations.

Large quorums provide better security but at the cost of increased computational time. HW acceleration mitigates this penality. HW acceleration brings key generation times down to 1-3 seconds even for Q6.

Table 3.2 indicates that HW acceleration backup operation (1.06s vs. 2.58s) and the decryption (2.29s vs. 9.95s) is more than 4 times faster. The Verification is more than two times faster compared to the SW only implementation. The HW operations have much lower variability (Backup (HW): 0.15 ms vs. SW: 8.52 ms) (Decryption (HW): 1.61 ms vs. SW: 19.4 ms). The SW implementation show a higher standard deviation, indicating a more variable performance. The HW acceleration is again more stable and deterministic. In this test the speed up of the verification is less pronounced due to initial computation steps using lower values and thus executing in SW.

The results indicate that hardware-accelerated key generation is significantly faster than non-accelerated key generation. For instance, with a quorum of 3 guardians, the average time for key generation with hardware acceleration is 1.53 seconds, compared to 6.62 seconds without acceleration. The standard deviation for hardware-accelerated key generation is also lower than for non-accelerated key generation, suggesting greater consistency in performance. As the quorum size increases, the time required for key generation also increases, with hardware-accelerated key generation consistently outperforming non-accelerated key generation.

The decryption is performend with 1 contest and 3 selections (3 Proofs) and the performance results are as follows:

Table **??** summarizes the performance of hardware-accelerated versus software-based cryptographic operations across three tasks: key generation, verification, and backup. 30 Measurements were taken for each operation. Results include mean execution times and standard deviations (SD) for a quorum of 3 guardians. Hardware-accelerated operations employed dedicated RSA and SHA accelerators, whereas non-accelerated operations relied on the wolfSSL software library. Across all operations, hardware-accelerated tasks exhibited significantly lower mean execution times than software-based implementations. Key generation, for instance, required 1.53 seconds with hardware acceleration compared to 10.8 seconds without—a 7× improvement. Similarly, backup operations completed in 0.51 seconds (accelerated) versus 3.59 seconds (non-accelerated), also reflecting a 7× speed increase. Verification saw a 3× enhancement (0.83 s vs. 2.43 s). Notably, hardware acceleration also improved temporal consistency: standard deviations for key generation and backup were orders of magnitude lower in accelerated runs, suggesting reduced performance variability. These results underscore the efficacy of hardware accelerators in optimizing both the speed and reliability of cryptographic workflows. Table **??** compares the performance of accelerated key generation across different quorums. As the quorum size increased from 3 to 5 guardians, mean execution times also increases significantly. The larger the quorum, the longer the operation takes.

3.5. Communication

The laptop is configured as an Access Point, enabling its wireless interface to create a local Wi-Fi network. This communication range is limited by the Wi-Fi signal strength of the laptop's access point. However, the entire system portable as long as the laptop and the connected IoT devices are powered. Our IoT devices-The NodeMCU ESP32 development boards-are connected to the Access Point. To facilitate Device-to-Device communication the laptop is configured as an MQTT broker using Eclipse Mosquitto version 2.0.20. It is important to note that this setup is not purely Device-to-Device communication, as it relies on the MQTT broker as an intermediary. Chapter 3.5 describes the communication aspects in more detail.

IoT systems rely primarily on using messaging protocols for exchanging IoT data and there exists several protocols or frameworks that support distinct types of messaging patterns Given that IoT devices typically have limited computational resources and processing power, choosing a lightweight, reliable, scalable, interoperable, extensible and secure messaging protocol becomes a very challenging task. [Al-+20, p. 1].

3.5.1. Data Link Layer Protocols

When selecting an appropriate messaging protocol for IoT devices, it is essential to consider the hardware characteristics of these devices and the types of data link layer protocols they support. The data link layer is responsible for facilitating data transfers between network entities [Al-+20, pp. 1–3]. For instance, the ESP32 microcontroller supports both Wi-Fi and Bluetooth data link layer protocols [25b]. The Bluetooth system on the ESP32 can be further divided into Classic Bluetooth and BLE [25b] [25a]. Both Wi-Fi and Bluetooth can operate simultaneously, but this requires time-sharing control [25a, p. 77].

The throughput of IoT devices can vary significantly based on the bandwidth they support. Since there is no universal radio technology for IoT devices, the physical data rates they can achieve depend heavily on their size and hardware components cite[1-2]protocols. Additionaly, throughout can be influenced by various factors, including environmental interference, connection intervals, and the size of the Maximum Transmission Unit (MTU) [25a]. The maximum BLE throughput achievable on the ESP32 is about 90 KB/s, for Classic Bluetooth is about 200 KB/s, and for Wi-Fi it is about 20 MBit/s TCP and 30 MBits/ UDP [25a, pp. 38, 58, 71] [25b, p. 2666].

Wi-Fi (802.11n) generally has a higher transmission range of up to approximately 1 km compared to BLE, which has a range of up to approximately 100 m [Al-+20, p. 3].

Beyond the data link layer protocols, there are networking protocols that operate on top of the BLE and Wi-Fi stacks. Both Wi-Fi and BLE support mesh networking, which facilitates many-to-many device communication and is optimized for creating large-scale device networks. The Wi-Fi stack also includes the Neighbor Awareness Networking (NAN) protocol, which allows direct device-to-device communication among NAN devices without requireing an Access Point (AP) connection. However, it is important to note that NAN Datapath security is not supported, meaning that data packets cannot be encrypted, making it less suitable for transmitting sensitive information [25b, p. 2694].

Understanding protocols at the data link layer is not sufficient for build IoT applications. It is essential to also consider the protocols that exist at the application level, which complement those at the data link layer. Choosing a protocol that is closer to the application layer while taking into account crucial system requirements- such as Quality of Service (QoS), bandwidth, interoperability, and security- becomes inevitable [Al-+20, p. 2]

3.5.2. Application Layer Protocols

When developing IoT systems, choosing the most appropriate messaging protocols becomes a challenging task. While all messaging protocols facilitate data communication between entities via a transmission medium, their characteristics vary. Understanding how these protocols operate and addressing potential challenges is essential for identifying a suitable protocol. A well-suited messaging protocol can help reduce network traffic and latency, thereby enhancing the reliability of an IoT application. For instance, application layer protocols that capture data faster than the actual physical data rates can lead to increased latency. Therefore, it is advisable to consider messaging protocols that can accommodate physical data rates at the data link layer [Al-+20, pp. 2, 15].

The application layer serves as an abstraction layer [Al-+20, p. 3]. Within the ESP32 microcontroller, several application layer protocols address a wide range of application requirements. Some of the notable application layer protocols available as firmware components of the ESP32 include:

- HyperText Transfer Protocol (HTTP) [25b]
- Message Queueing Telemetry Transport (MQTT) [25b]
- Modbus (primarily used in industrial IoT environments)[Al-+20, p. 3] [25b]
- ESP-NOW (a proprietary protocol for ESP32 devices) [25b][25b]

Each protocol offers various features that differ in terms of reliability, quality of service, performance, functionality, and scalability, among other factors [Al-+20, p. 3].

3.5.3. Payload Size

One important aspect that narrow down the choice of messaging protocols is the maximum payload size.

The proof sizes play a significant role in the choice of messaging protocols in our IoT application. At the post-election phase each guardian produces a Chaum-Pedersen proof of correct decryption [Ben+24, p. 15]. A Chaum-Pedersen proof contains the following values:

- commitment(pad,data): 1024 bytes (standard parameters) or 768 bytes (reduced parameters)
- challenge: 32 bytes
- response: 32 bytes

A Chaum-Pedersen proof to proof a decryption share generated by a guardian is thus 1088 bytes (standard parameters) or 960 bytes (reduced parameters) in size.

During pre-election to ensure robustness and handle missing guardians at the post-election phase, ElectionGuard uses a key generation process that involves sharing private keys among guardians. This allows a Quorum of guardians to decrypt the election results without needing to reconstruct the private keys of missing guardians. These shares are accompanied by Schnorr proofs too ensure the receiving guardians can confirm the shares they receive are meaningful [BN22, p. 9]. A Schnorr proof contains the following values:

- commitment: 512 bytes (standard parameters) or 384 bytes (reduced parameters)
- challenge: 32 bytes
- response: 32 bytes

A Schnorr proof is thus 576 bytes (standard parameters) or 448 bytes (reduced parameters) in size. If the Quorum is 3 guardians, each guardian would need to generate 3 Schnorr proofs for each guardian. The total size of Schnorr proofs for a Quorum of 3 guardians is 1728 bytes (standard parameters) or 1344 bytes (reduced parameters).

The maximum packet size for the mesh networking technologies is 384 bytes for BLE and 1456 bytes for Wi-Fi [25a, pp. 35, 54]. A BLE network is not suitable for the proposed voting system as the maximum payload size for a Chaum-pedersen proof using reduced parameters is 960 bytes. A Wi-Fi mesh would have trouble with a Quorum of 3 guardians as the maximum payload size for the Schnorr proofs using reduced parameters is already 1344 bytes and this does not include any additional data that needs to be transmitted. The NAN. The maximum packet sizes for the Application protocols ESP-NOW is 250 bytes, MQTT is 265 MB and HTTP does not have a limit on the message size [25a, p. 47] [Al-+20, p. 16].

3.5.4. Message Reliability

IoT systems are driven by IoT devices that are typically resource-constrained having limited power, networking and processing capabilities. Messaging protocols need to be optimized such that they require minimal resources (e.g. processing power, memory, storage, network bandwidth) which are often needed by IoT devices when communicating data. To this extent, it is imperative that the messaging protocols employed in IoT systems maintain high-levels of quality for data transmission. [Al-+20, p. 15]. An IoT system may require that messages be delivered in a reliable manner where all clients acknowledge the receipt of these messages [Al-+20, p. 11]. MQTT uses three levels of message transmission reliability, each representing a different level of QoS [PC20, p. 12]:

- **QoS 0 (most once):** Messages arrives at the receiver either once or not at all [PC20, p. 11]
- **QoS 1 (least once):** Ensures that a message arrives at the receiver at least once [PC20, p. 11]
- **QoS 2 (exactly once):** Ensures that a message arrives at the receiver exactly once without duplication [PC20, p. 11]

As the QoS level increases, the reliability of messages' delivery also increases. However, this also increases the overhead associated with ensuring that all clients receive the intended messages. The more clients are subscribed to receive a message with QoS 2, for example, will increase the overhead on the message broker while ensuring the delivery of the message without duplication or retransmission. [Al-+20, p. 11].

3.5.5. MQTT

MQTT is designed for constrained environments with low bandwidth. MQTT overs several benefits over HTTP such as asynchronous messaging, lower power consumption, and Quality of Service support [Al-+20, pp. 23, 27].

MQTT uses a publish/subscribe model and is composed of a broker and clients. In this model, clients (publisher) publish messages to a broker via a specific topic. Then, the broker filters these incoming messages and distributes them to to clients (subscriber) who are interested in receiving these messages. To this extent, a client that is interested in receiving these messages must first subscribe to this specific topic. In short, a publisher can send messages to a number of subscribers with one single publish operation to the broker. The broker handles the "broadcasting" or

sending messages to all subscribers subscribed to topic of the message. [Al-+20, p. 10] [PC20, p. 12].

Figure **??** shows the communication sequence in the pre-election phase using MQTT.Figure **??** shows the communication sequence in the intra-election phase using MQTT. Figure **??** shows the communication sequence in the post-election phase using MQTT. All figures shows a high-level overview of the MQTT brokering model that shows all of the entities involved in this architecture including: (a) centralized broker, (b) publishers and (c) subscribers.

The further publishers and subscribers are from the broker, the longer the travel time of the MQTT messages and the higher the latency [Al-+20, p. 20]. Subscribers can receive published messages at different times. Some studies found that the throughput in MQTT drops significantly as the number of clients' subscriptions increases. As more clients subscribe to topics the number of messages increases [Al-+20, pp. 19, 21, 22].

3.5.6. Data serialization

Data serialization is the process of structuring data into a streamlined format before storing or transmitting it. Broadly speaking, there are two approaches to serialization: text-based and binary. In text-based serialization, data is typically structured into key-value pairs in a readable text format. In binary serialization, key-value pairs are stored in a binary format, which typically reduces space requirements [PC20, p. 11]. The design specification of ElectionGuard does not specify serialization methods or data structures. However, every implementation of ElectionGuard should be compatible with other implementations [Ben+24, p. 23]. The Python implementation expects data to be serialized into the text-based JSON format.

Exchanging data in different formats across IoT devices raises syntactic interoperability issues that need to be addressed [Al-+20, p. 17]. However, if we want to transmit data through the network faster, smaller data sizes are preferable. Additionally, the data does not need to be human-readable during transmission like with text-based formats [Cur22, p. 225]. Binary formats are typically preferred as they provide smaller message sizes compared to text-based formats like JSON [PC20, p. 11]. For instance, in a test using ESP32, the encoding size was, on average, smaller for Protocol Buffers (a binary format) compared to the text-based JSON format [Lui+21, p. 15]. Thus we could use a binary format for sending data over the network to reduce the message size however we would need to convert the data into a JSON format for the Python implementation.

Another benefit of more efficient formats is improved serialization and deserializ-

ation speeds. This indicates that fewer CPU cycles are used for data processing, leading to lower power consumption. In one test on the ESP32, the serialization and deserialization speed was almost halved when using Protocol Buffers compared to JSON [Lui+21, pp. 11–12].

In our case, choosing a binary serialization approach could be beneficial. The in-memory data representation of our data structures in the ESP32 implementation uses structs, as seen in Appendix **??**. These structures contain a custom data type, sp_int, which is a large integer representation. To parse the large integer into a hexadecimal JSON string, we would need to convert each byte of the large integer into a hex representation. In contrast, parsing into a binary format involves simply copying the bytes directly into the output array, which is a more efficient operation.

Our implementation, therefore, chooses Protocol Buffers as the serialization format. A Protocol Buffer implementation is already included in the ESP32 as a component. A significant advantage of Protocol Buffers is that we only need to define the structure for the data to be transferred once and can then exchange it over a wide variety of channels. The programming language is secondary since Protocol Buffers are language-neutral [Cur22, p. 224]. Thus, with our .proto files, as seen in Appendix **??**, we can generate code for both the ESP32 and the Python client, as seen in Appendices **??** and **??**. An example of serialising ElectionPartialKeyPair which is the backup shared with other guardians is seen in Appendix **??** an example of deserialisation is seen in Appendix **??**.

3.5.7. Networ Traffic Analysis

Network traffice between the Laptop and the ESP32 was captured using Wireshark (4.4.3) by listening to the access point interface of the laptop. The traffic was narrowed down by applying a display filter to filter out all non mqtt traffic. The precision of the timestamps is in milliseconds. The capture focused on the communication related to the key ceremony and the decryption phase. Round Trip Time (RTT) was calculated based on the time difference between request and acknowlegement packets for specific message exchanges. A display filter

3.6. Security

Furthermore, IoT devices are generally used by humans which makes them vulnerable to intruders that attempt to gain unsolicited access or collect confidential

personal data in a malicious manner. [Al-+20, p. 23]

An IoT communication protocol needs to ensure that only authorized users regardless if they are publishers or subscribers Furthermore, such vulnerabilities may occur when offering QoS level 2 which may explain why many IoT cloud providers not to provide support at this level as presented in Table VII. Although each protocol provides different levels of security measure[Al-+20, p. 23]

The MQTT handling of disallowed Unicode code points provides a client or server the option to decide on the validation of these code points (e.g. UTF-8 encoded strings). As a result, an endpoint does not necessarily need to validate UTF-8 encoded strings (e.g. topic name or property). As such, a client could potentially use this as a vulnerability and causes a subscribed client to close the network connection using a topic that contains an invalid Unicode code point. A malicious client can then use this as a security exploit for possibly causing a Denial of Service (DoS) attack. Therefore, enabling UTF-8 encoded strings, for example, can allow these security exploits to occur in cases they are used as control characters or in control packets (see the first [Al-+20, p. 11]

lack of encryption; can use TLS/SSL for security and encryption, however, extra connection overhead [Al-+20, p. 27]

The case can become worse if the broker's resources are maximized and hence a broker can be a Single Point of Failure (SPoF) [Al-+20, p. 19].

4. Conclusions

Im Schlusskapitel wird die Arbeit und ihre Ergebnisse zusammengefasst sowie ein Ausblick gegeben.

List of Figures

List of Tables

Bibliography

[] *ElectionGuard Docs*. Version 2.1. Election Tech Initiative. URL: https://github.com/Election-Tech-Initiative/electionguard/tree/0f99229669139d3cae6cfddc0a3d554319a84a2e/docs.

[19] 'Security Advisory concerning fault injection and eFuse protections (CVE-2019-17391)'. In: (Nov. 2019). URL: https://www.espressif.com/en/news/Security_Advisory_Concerning_Fault_Injection_and_eFuse_Protections.

[23a] *End-to-End Verifiability in Real-World Elections*. Tech. rep. Microsoft, Jan. 2023.

[23b] *ESP32-WROOM-32 Datasheet*. 3.4. Not Recommended For New Designs (NRND). Espressif Systems (Shanghai) Co., Ltd. Feb. 2023. URL: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf.

[24a] *ESP32 Series SoC Errata*. 2.8. Espressif Systems (Shanghai) Co., Ltd. July 2024. URL: https://docs.espressif.com/projects/esp-chip-errata/en/latest/esp32/esp-chip-errata-en-master-esp32.pdf.

[24b] *ESP32-WROOM-32E Datasheet*. 1.7. Espressif Systems (Shanghai) Co., Ltd. Sept. 2024. URL: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf.

[25a] *ESP-FAQ Handbook*. Espressif Systems (Shanghai) Co., Ltd. Feb. 2025. URL: https://docs.espressif.com/projects/esp-faq/en/latest/esp-faq-en-master.pdf.

[25b] *ESP-IDF Programming Guide*. Version 5.1. Espressif Systems (Shanghai) Co., Ltd. Feb. 2025. URL: https://docs.espressif.com/projects/esp-faq/en/latest/esp-faq-en-master.pdf.

[25c] *ESP32 Series Datasheet*. 4.8. Espressif Systems (Shanghai) Co., Ltd. Jan. 2025. URL: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[25d] *ESP32 Technical Reference Manual*. 5.3. Espressif Systems (Shanghai) Co., Ltd. Jan. 2025. URL: https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf.

[Al-+20] Eyhab Al-Masri et al. 'Investigating Messaging Protocols for the Internet of Things (IoT)'. In: *IEEE Access* 8 (2020), pp. 94880–94911. DOI: 10.1109/ACCESS.2020.2993363.

[Ben+14] Josh Benaloh et al. *End-to-end verifiablity*. Feb. 2014. URL: https://www.microsoft.com/en-us/research/publication/end-end-verifiablity/.

[Ben+24] Josh Benaloh et al. 'ElectionGuard: a Cryptographic Toolkit to Enable Verifiable Elections'. In: *USENIX Security*. Aug. 2024. URL: https://www.microsoft.com/en-us/research/publication/electionguard-a-cryptographic-toolkit-to-enable-verifiable-elections/.

[BN22] Josh Benaloh and Michael Naehrig. *ElectionGuard Specification*. GitHub repository. Accessed on 2025-01-21. Microsoft Research, Jan. 2022. URL: https://github.com/Election-Tech-Initiative/electionguard/releases/tag/v1.0.

[Cha81] David L. Chaum. 'Untraceable electronic mail, return addresses, and digital pseudonyms'. In: *Commun. ACM* 24.2 (Feb. 1981), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/358549.358563. URL: https://doi.org/10.1145/358549.358563.

[Cmi25] Gabriel Cmiel. *ESP32 Guardian Component*. Generated by Doxygen. 2025.

[Com17] Technical Guidelines Development Committee. *VVSG Introduction*. Accessed on 2024-01-10. May 2017. URL: https://www.nist.gov/itl/voting/vvsg-introduction.

[Com21a] United States Election Assistance Commission. *U.S. Election Assistance Commission Adopts New Voluntary Voting System Guidelines 2.0*. Accessed on 2024-01-10. Feb. 2021. URL: https://www.eac.gov/news/2021/02/10/us-election-assistance-commission-adopts-new-voluntary-voting-system-guidelines-20.

[Com21b] Technical Guidelines Development Committee. *Voluntary Voting System Guidelines VVSG 2.0*. Feb. 2021. URL: https://www.eac.gov/sites/default/files/TestingCertification/Voluntary_Voting_System_Guidelines_Version_2_0.pdf.

[Cur22] Chris Currier. 'Protocol Buffers'. In: *Mobile Forensics – The File Format Handbook: Common File Formats and File Systems Used in Mobile Devices*. Ed. by Christian Hummert and Dirk Pawlaszczyk. Cham: Springer International Publishing, 2022, pp. 223–260. ISBN: 978-3-030-98467-0. DOI: 10.1007/978-3-030-98467-0_9. URL: https://doi.org/10.1007/978-3-030-98467-0_9.

[Ert07] Wolfgang Ertel. *Angewandte Kryptographie*. 3., aktualisierte Auflage. München: Carl Hanser Verlag, 2007. ISBN: 978-3-446-41195-1.

[Espa] Ltd. Espressif Systems (Shanghai) Co. *esp_romComponent*. GitHub repository. Version 5.4. URL: https://github.com/espressif/esp-idf/tree/473771bc14b7f76f9f7721e71b7ee16a37713f26/components/esp_rom/esp32.

[Espb] Ltd. Espressif Systems (Shanghai) Co. *mbedTLS Component*. GitHub repository. Version 3.6.2. URL: https://github.com/espressif/esp-idf/tree/473771bc14b7f76f9f7721e71b7ee16a37713f26/components/mbedtls.

[eta19] Andrew Banks et.al. *MQTT Version 5.0*. Rev1.0. LCDWiki. Mar. 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[Gmb20] SIMAC Electronics GmbH. *SBC-Button 2 Datasheet*. SIMAC Electronics GmbH. Pascalstr. 8, 47506 Neukirchen-Vluyn, Oct. 2020.

[GS22] Himanshu Goyal and Sudipta Saha. ‘ Multi-Party Computation in IoT for Privacy-Preservation ’. In: *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2022, pp. 1280–1281. DOI: 10.1109/ICDCS54860.2022.00133. URL: https://doi.ieeecomputersociety.org/10.1109/ICDCS54860.2022.00133.

[Hat24] Rik van de Haterd. *Enhancing Privacy and Security in IoT Environments through Secure Multiparty Computation*. Feb. 2024. URL: http://essay.utwente.nl/98341/.

[IE20] Valeriu Manuel Ionescu and Florentina Magda Enescu. ‘Investigating the performance of MicroPython and C on ESP32 and STM32 microcontrollers’. In: *2020 IEEE 26th International Symposium for Design and Technology in Electronic Packaging (SIITME)*. 2020, pp. 234–237. DOI: 10.1109/SIITME50350.2020.9292199.

[Inca] wolfSSL Inc. *ESP32 Hardware Cryptography*. Accessed on 2024-01-10. URL: https://www.wolfssl.com/docs/espressif/.

[Incb] wolfSSL Inc. *wolfSSL ESP-IDF Port*. GitHub repository. Version 5.7.4. URL: https://github.com/wolfSSL/wolfssl/tree/eb15a1213cdc2cd43f155cedd06b257wolfcrypt/src/port/Espressif.

[Inc25] wolfSSL Inc. *wolfSSL Documentation*. Accessed on 2025-01-21. Feb. 2025. URL: https://www.wolfssl.com/documentation/manuals/wolfssl/wolfSSL-Manual.pdf.

[Ini] Election Tech Initiative. *ElectionGuard Python*. GitHub repository. Version 1.4.0. URL: https://github.com/Election-Tech-Initiative/electionguard-python.

[Jin22] Qiao Jin. 'Performance Evaluation of Cryptographic Algorithms on ESP32 with Cryptographic Hardware Acceleration Feature (Dissertation)'. Retrieved from https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-309833. MA thesis. KTH, School of Electrical Engineering and Computer Science (EECS), Feb. 2022.

[Joy18] Joy-it. *NodeMCU ESP32 Datasheet*. Joy-it. Sept. 2018.

[KQM21] Farzana Kabir, Amna Qureshi and David Megias. 'A Study on Privacy-Preserving Data Aggregation Techniques for Secure Smart Metering System'. In: Apr. 2021.

[LCD] LCDWiki. *1.8inch Arduino SPI Module MAR1801 User Manual*. Rev1.0. LCDWiki. URL: http://www.lcdwiki.com/res/MAR1801/1.8inch_Arduino_SPI_Module_MAR1801_User_Manual_EN.pdf.

[LL] Light and Versatile Embedded Graphics Library. *Display and touchpad drivers for ESP32 using LVGL*. GitHub repository. Version 0.0.2. URL: https://github.com/lvgl/lvgl_esp32_drivers.

[Lui+21] Álvaro Luis et al. 'PSON: A Serialization Format for IoT Sensor Networks'. In: *Sensors* 21.13 (2021). ISSN: 1424-8220. DOI: 10.3390/s21134559. URL: https://www.mdpi.com/1424-8220/21/13/4559.

[Mic] MicroPython. *MicroPython documentation*. GitHub repository. Version 1.24.0. Accessed on 2025-01-21. URL: https://github.com/micropython/micropython.

[Mos+24] Florian Moser et al. *A Study of Mechanisms for End-to-End Verifiable Online Voting*. Tech. rep. Bundesamt für Sicherheit in der Informationstechnik, Aug. 2024. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/Cryptography/End-to-End-Verifiable_Online-Voting.pdf.

[PC20] Daniel Persson Proos and Niklas Carlsson. 'Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV'. In: *2020 IFIP Networking Conference (Networking)*. 2020, pp. 10–18.

[Roy+25] Anandarup Roy et al. *A Combinatorial Approach to IoT Data Security*. Cryptology ePrint Archive, Paper 2025/010. 2025. URL: https://eprint.iacr.org/2025/010.

[Sag12] Ali Makki Sagheer. 'Elliptic curves cryptographic techniques'. In: *2012 6th International Conference on Signal Processing and Communication Systems*. 2012, pp. 1–7. DOI: 10.1109/ICSPCS.2012.6507952.

[Wil22] Newton Carlos Will. 'A Privacy-Preserving Data Aggregation Scheme for Fog/Cloud-Enhanced IoT Applications Using a Trusted Execution Environment'. In: *2022 IEEE International Systems Conference (SysCon)*. 2022, pp. 1–5. DOI: 10.1109/SysCon53536.2022.9773838.

A. Appendix

A.1. Election Administrator

A.1.1. Intra-election Phase

```
print("Generating Random Ballots")
plaintext_ballots = BallotFactory().
    generate_fake_plaintext_ballots_for_election(
    internal_manifest, 10, None, False, False)
for plain in plaintext_ballots:
        for contest in plain.contests:
                for selection in contest.ballot_selections:
                        if selection.vote == 1:
                                print(f"Ballot ID: {plain.
                                    object_id}, Selected: {
                                    selection.object_id}")

expected_result = accumulate_plaintext_ballots(
    plaintext_ballots)
print("Expected Result: ", expected_result)

for plain in plaintext_ballots:
        encrypted_ballot = encrypter.encrypt(plain)
        ciphertext_ballots.append(get_optional(
            encrypted_ballot))

ballot_store = DataStore()
ballot_box = BallotBox(internal_manifest, context,
    ballot_store)

for ballot in ciphertext_ballots:
        ballot_box.cast(ballot)

ciphertext_tally = get_optional(tally_ballots(ballot_store,
    internal_manifest, context))
```

A.2. Guardian Code

A.2.1. Adapter

```
    case MQTT_EVENT_DATA:
        ESP_LOGI(TAG, "MQTT_EVENT_DATA");
        char topic[20];
        snprintf(topic, event->topic_len + 1, "%.*s", event->
            topic_len, event->topic);
        ESP_LOGI(TAG, "Topic: %s", topic);
        if(strncmp(topic, "ceremony_details", event->topic_len) ==
            0)
        {
            if(sscanf(event->data, "%d,%d", &quorum, &max_guardians
                ) == 2) {
                ESP_LOGI(TAG, "Received Ceremony Details");
                ESP_LOGI(TAG, "Quorum: %d, Max Guardians: %d",
                    quorum, max_guardians);
                // Exclude self from guardian count
                max_guardians--;
                ESP_LOGI(TAG, "Max Guardians: %d", max_guardians);
                pubkey_map = (ElectionKeyPair*)malloc(max_guardians
                    * sizeof(ElectionKeyPair));
                backup_map = (ElectionPartialKeyPairBackup*)malloc(
                    max_guardians * sizeof(
                    ElectionPartialKeyPairBackup));
                esp_mqtt_client_unsubscribe(client, "
                    ceremony_details");
                esp_mqtt_client_subscribe(client, "pub_keys", 1);
                esp_mqtt_client_subscribe(client, "backups", 1);
                esp_mqtt_client_subscribe(client, "challenge", 1);
                publish_public_key(client, event->data, event->
                    data_len);
            }
        }
        else if(strncmp(topic, "pub_keys", event->topic_len) == 0)
        {
            ESP_LOGI(TAG, "Received Public Key");
            handle_pubkeys(client, event->data, event->data_len);
        }
        else if(strncmp(topic, "backups", event->topic_len) == 0)
        {
            ESP_LOGI(TAG, "Received Backup");
            handle_backups(client, event->data, event->data_len);
        }
        else if(strncmp(topic, "challenge", event->topic_len) == 0)
        {
            ESP_LOGI(TAG, "Received Challenge");
            handle_challenge(client, event->data, event->data_len);
        }
        else if (strncmp(topic, "ciphertally", event->topic_len) ==
```

```
            0)
        {
            ESP_LOGI(TAG, "Received ciphertext tally");
            handle_ciphertext_tally(client, event->data, event->
                data_len);
        }
        else {
            ESP_LOGI(TAG, "Unknown topic");
        }
        break;
```