Chair for Embedded Systems
Universität Augsburg

Master's Thesis

Implementation of an IoT based Electronic Voting Machine

Gabriel Cmiel

Examiner:	Prof. Dr. Sebastian Altmeyer
Second examiner:	Prof. Dr. Bernhard Bauer
Supervisor:	Prof. Dr. Sebastian Altmeyer
Date:	6th 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

The rapid expansion of the IoT introduces critical challenges in maintaining data confidentiality, integrity, and verifiability in resource-constrained environments.This thesis investigates the implementation of E2E verifiable voting systems on IoT devices, focusing on the ElectionGuard 1.0 specification deployed on ESP32 microcontrollers. This work underscores the viability of ElectionGuard in constrained environments while highlighting critical areas for performance refinement. Key research questions explore computational efficiency (e.g., hardware accelerators for cryptographic operations, memory constraints, and processing bottlenecks) and communication efficiency (protocol selection, bandwidth optimization, and network latency mitigation).

A custom C port of ElectionGuard was developed to leverage ESP32's hardware features, including RSA and SHA accelerators, achieving a 4x speed improvement in cryptographic operations compared to software-only implementations. The system leverages MQTT for communication, balancing payload size with reliable transmission via QoS levels. Protocol Buffers were used for efficient binary serialization, ensuring interoperability between the Python-based Tallier and C-based Guardian nodes. Network traffic analysis reveals consistent Round-Trip Time (RTT) under load, although initial bursts suggest the need for congestion management in larger deployments. Homomorphic aggregation on the Tallier (implemented in Python) emerged as the primary bottleneck, limiting scalability due to its CPU-bound nature, requiring 251.9 seconds for 1000 ballots.

List of Abbreviations

ADC Analog-to-Digital Converter

AES Advanced Encryption Standard

AP Access Point

BLE Bluetooth Low Energy

DRBG Deterministic Random Bit Generator

E2E End-to-end

ECC Elliptic Curve Cryptography

ESP-IDF Espressif IoT Integrated Development Framework

HMAC Keyed-Hash Message Authentication Code

HTTP Hypertext Transfer Protocol

IoT Internet of Things

KDF Key Derivation Function

MQTT Message Queueing Telemetry Transport

MPC Multi-Party Computation

MTU Maximum Transmission Unit

PKE Public-Key Encryption

PPDA Privacy-Preserving Data Aggregation

QoS Quality of Service

RNG Random Number Generator

RSA Rivest-Shamir-Adleman

RTT Round-Trip Time

SHA Secure Hash Algorithm

TRNG True Random Number Generator

VVSG Voluntary Voting System Guidelines

ZK Zero-Knowledge

1. Introduction

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 summarising 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. Nevertheless, the computational intensity of these techniques often renders them impractical for resource-constrained IoT devices [KQM21, pp. 113–114]. PPDA techniques also apply to other domains, such as 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 cryptography—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 synchronised interactions introduce communication bottlenecks [Mos+24, p. 45]. Verifiability and accountability are equally critical in PPDA systems. Verifiability ensures the correctness of aggregated results (e.g., election outcomes) [Mos+24, p. 4], while accountability-a stronger notion of verifiability-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

This thesis evaluates the implementation challenges and performance characteristics of E2E verifiable voting systems in resource-constrained IoT environments, focusing on a subset of the ElectionGuard 1.0 specification deployed on ESP32 microcontrollers. The following research questions guide this evaluation:

Computational Efficiency: How does the choice of hardware components (e.g., Tallier Node, Guardian Nodes) impact the performance of an IoT-based voting system running ElectionGuard? How feasible is it to run the existing ElectionGuard implementations on ESP32 microcontrollers, and what are the potential limitations in terms of memory usage and processing power? How do hardware accelerators on the ESP32 improve the performance of cryptographic operations in ElectionGuard? How does the computational load on the Tallier Node impact the overall performance of the voting system?

Communication Efficiency: What communication protocol is suitable for reliably handling large data transfers on the ESP32 microcontroller? What strategies can be employed to reduce the bandwidth requirements? What are the challenges in implementing communication protocols on resource-constrained IoT devices like ESP32, and how can these be addressed? How does the network latency between the Tallier Node and Guardian Nodes impact the performance of the voting system?

1.2. Structure of the Thesis

The thesis is structured as follows. Chapter 2 establishes foundational knowledge required for understanding the cryptographic voting system, covering core cryptographic principles, verifiability concepts, and the technical components of the ElectionGuard 1.0 specification and ESP32 hardware. Chapter 3 details the implementation of an IoT voting system's architecture, focusing on the integration of hardware, cryptographic processes, and communication protocols to support the election process. Chapter 4 presents the conclusions of the thesis, summarising the findings related to the performance of ElectionGuard on ESP32 microcontrollers, and suggesting future work directions.

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. 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 length of the key increases. It should be noted that there is no absolute security in cryptography [Ert07, pp. 18, 25].

Practically all cryptographic methods aim to achieve one or more of the following properties:

- **Confidentiality:** The aim of confidentiality is to make it impossible or difficult for unathorised 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 unauthorised 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. 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, pp. 19, 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 satisfies the four fundamental properties of cryptographic security:

- **Confidentiality:** Ensures that the voter's communication remains private from unauthorised 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. 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. E2E Verifiability

A study conducted by the German Federal Office for Information Security (BSI) highlighted that **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 (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 [Com21, p. 181].

2.4.1. Key Components of E2E Verifiability

E2E verifiability encompasses two principal components:

- **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 [Mos+24, p. 13]. The most important and recurring cryptographic building blocks include:

- **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].

- **ZK Proofs:** This technique permits a party to demonstrate that it has correctly performed a certain computation without revealing any additional information, such as the secret key involved in the said 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].

2.4.2. E2E 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 multiplicatively homomorphic, frequently an exponential variant of ElGamal is employed, making it additively homomorphic [Mos+24, p. 40].

2.5. ElectionGuard Overview

Microsoft's **ElectionGuard** is a toolkit designed to provide E2E verifiable elections by separating cryptographic functions from the core mechanisms and user interfaces of voting systems. This separation empowers ElectionGuard to offer simple interfaces that can be used without requiring cryptographic expertise. Consequently, existing voting systems can function alongside ElectionGuard without necessitating

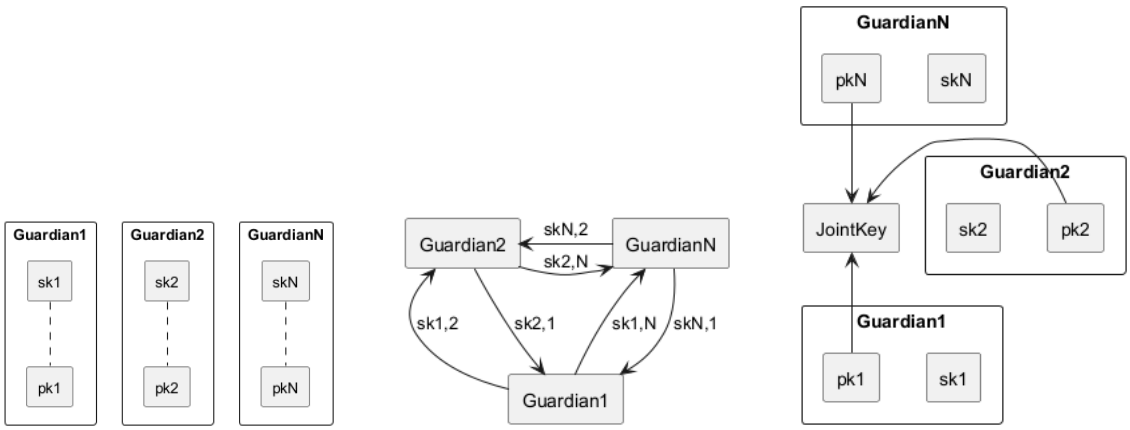
their replacement, while still producing independently-verifiable tallies [Ben+24, pp. 1–2]. The cryptographic design of ElectionGuard is largely inspired by the cryptographic voting protocol developed by Cohen (now Benaloh) and Fischer in 1985, as well as the voting protocol by Cramer, Gennaro, and 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 a joint 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].

2.6. Phases of ElectionGuard

The election process in ElectionGuard can be divided into three main phases: Key-Generation Ceremony, Ballot Encryption, and Decryption of Tallies. Each phases involves distinct activities. Below, we will explore each phase in detail.

2.6.1. Pre-Election Key-Generation Ceremony

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 [24a]. 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 (e.g., threshold) of Guardians required for the threshold secret sharing mechanism. 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



(a) Key generation (b) Backup sharing (c) Joint key generation

Figure 2.1.: Illustration of the Key Ceremony Process. Adapted from [24a]

through the multiplication of individual public keys [24a]. The steps involved in the key-generation ceremony are illustrated in Figure 2.1. The 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 [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]. 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] [24a]. 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].

2.6.2. Intra-Election Ballot Encryption

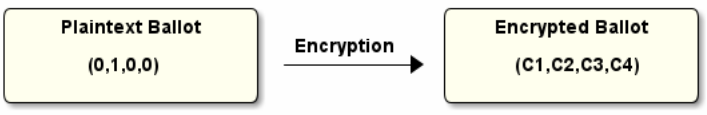


Figure 2.2.: Representation of Plaintext and Encrypted Ballots Adapted from [24a]

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 encryption 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 or 0 [Ben+24, p. 11]. A Chaum-Pedersen proof verifies whether an encryption represents 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 preparation process. The two choices are mutually exclusive, since challenging reveals the ballot selections during the decryption phase. 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] [24a].

2.6.3. 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 proofs for 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 [24a].

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].

2.7. ESP32-WROOM-32 Overview

The **ESP32-WROOM-32** is a powerful microcontroller module that integrates Wi-Fi, Bluetooth, and 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 [24c, p. 1]. These later revisions rectify some hardware issues [24b, pp. 3–4] present in previous versions. The ESP32-D0WDQ6 chip is susceptible to fault injection attacks. Successfully 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]. ESP32 devices 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]

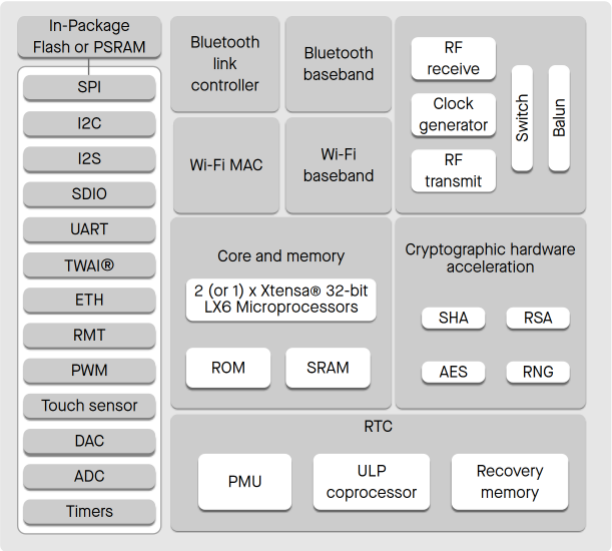


Figure 2.3.: ESP32 Functional Block Diagram

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.

2.7.1. Development Environment

For application development, Espressif, the company behind the ESP32, offers the ESP-IDF. ESP-IDF encompasses a comprehensive toolchain, API components, and defined workflows tailored for ESP32 application development [25b, p. 14]. The development workflow for ESP-IDF is depicted in Figure 2.4. ESP-IDF consists of components written specifically for ESP-IDF as well as third-party libraries [25b, p. 2801]. For example, the real-time operating system kernel used for ESP-IDF applications is the third-party component called FreeRTOS [25b, p. 2412] [23b, p. 6]. ESP-IDF projects use the same component based approach and can be seen as an amalgamation of a number of components. 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, pp. 2412–2413]

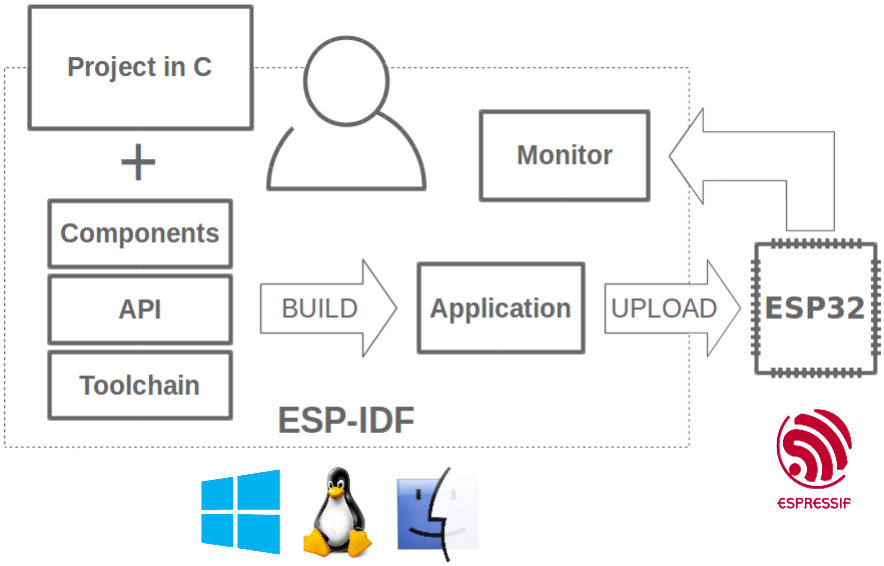


Figure 2.4.: Espressif IoT Integrated Development Framework (ESP-IDF) [25b, p. 2805]

3. IoT Voting System Implementation

3.1. Hardware Setup

The IoT voting system 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 (AP) (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

The Guardian nodes are responsible for distributing trust in the encryption and decryption processes managed by the Tallier.

3.2. Election Setup

The Guardian nodes responsibilities are divided into two main phases: the key-generation ceremony, and the decryption process.

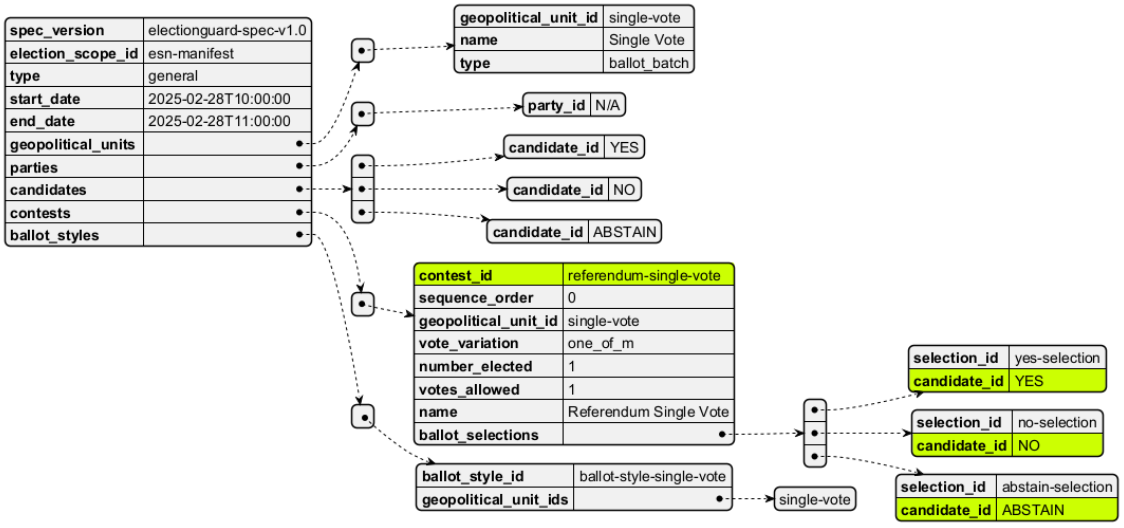


Figure 3.1.: Visualisation of the JSON Election Manifest

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]

3.2.1. Pre-Election Key-generation

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

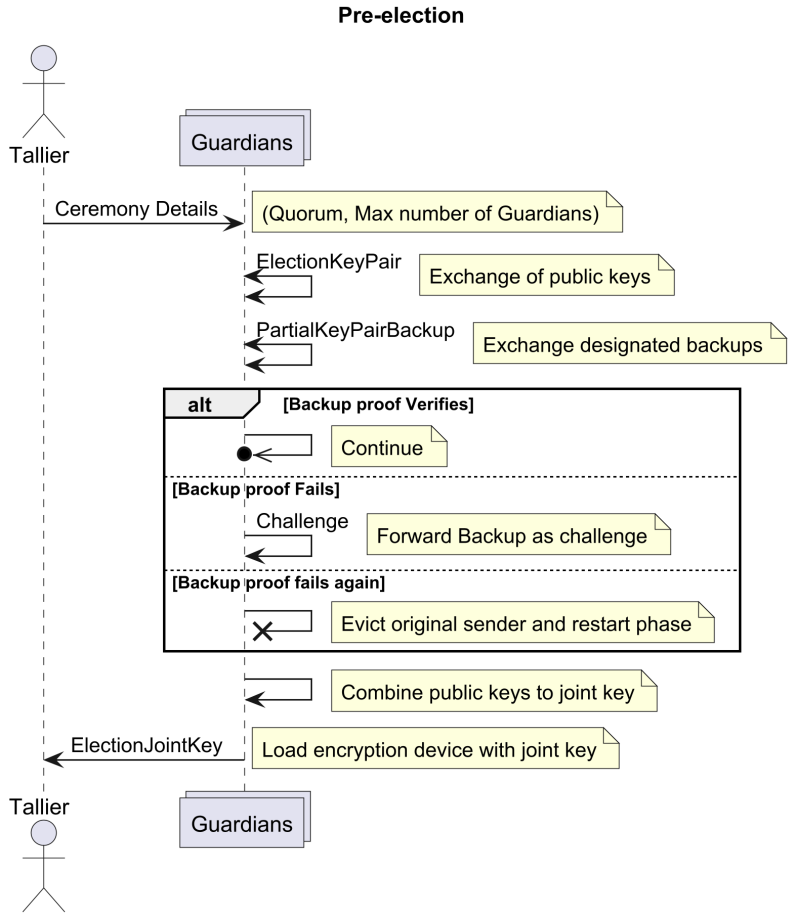


Figure 3.2.: Communication Sequence in the Pre-election phase

The tallier node defines the election manifest, sets the cryptographic constants according to the reduced baseline parameters (as discussed in 3.2), 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 designated backup for each ElectionKeyPair received, which is then sent back to the sending Guardian. Each backup contains a proof that must be verified by the receiver. Once all backups successfully verify, the ElectionKeyPairs are com-

bined into a joint key. The challenge mechanism and eviction mechanism for failed verifications are not implemented in this experiment. ElectionGuard assumes the key-generation runs between publicly identified parties, and there is little benefit for a malicious participant in introducing errors. Therefore, it is assumed that Guardians abort the key-generation as soon as one of them detects an error, restarting the protocol from scratch, possibly replacing Guardians that are identified as acting maliciously [Ben+24, p. 9].

3.2.2. 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. In the example in the Appendix A.4.1 the Tallier generates 10 ballots and encrypts them. We test the configuration with 10, 100 and 1000 ballots.

3.2.3. 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 sent 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 tally. The case of missing Guardians is also not addressed in this experiment. Compensating for missing decryptions functions similarly to the generation of the decryption share. The publication of the election artifacts is also not implemented, the interaction between the Guardian and the Tallier requires inherent verification due to the proofs involved as seen in Appendix A.1.

3.3. Comparison of ElectionGuard Implementations

Due to the open-source nature of ElectionGuard, various community ports exist, such as a Java port [24a]. 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

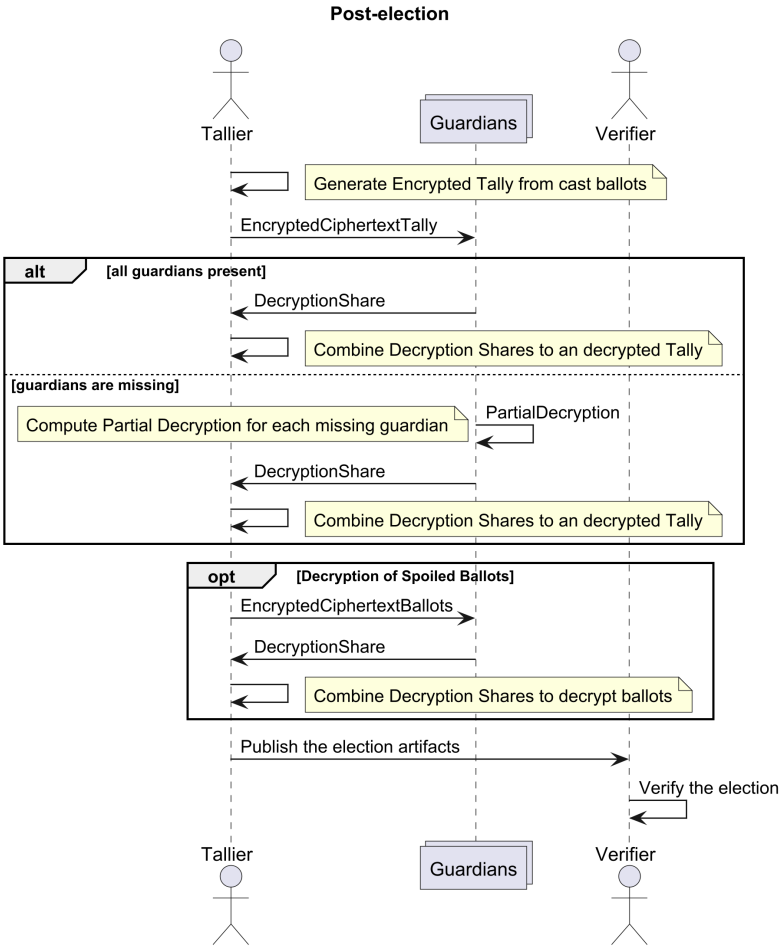


Figure 3.3.: Communication Sequence during the Pre-Election Key-Generation Ceremony

functionality and processes necessary to implement E2E verifiable election as part of a voting system [Inib]. It is designed to be universal and portable, albeit less performant [24a]. In contrast, the C++ reference implementation focuses solely on the encryption components and is optimised for execution on low-powered devices [Inia].

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 reference 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 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 [Inia] [24a]. 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 [Inia]. These optimisations come at a memory cost. The optimisation for modular exponentiation uses pre-computed tables to speed up calculations for certain exponentiations [Inia]. A modular exponentiation computes $X^Y \bmod P$. This optimisation 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 from [Inia]

```
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 [Inia].

- b= 256 (OrderBits) [Inia].
- k = 8 (WindowSize) [Inia].
- m = 32 (TableLength) [Inia].

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 calculate 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 optimisations 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 optimisations 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 [BN22, p. 25].

To handle the large integer values involved in these operations, specialised 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 [Inib] for large integer arithmetic, the C++ implementation uses HACL* a performant C implementation of a wide variety of cryptographic primitives [Ben+24, p. 22] [Inia]. In ESP32 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 3.4.

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

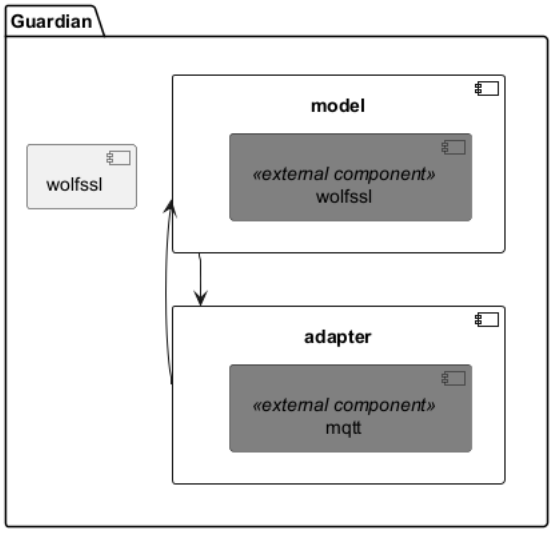


Figure 3.4.: Software components for the Guardian Prototype

by removing unused features. Furthermore, the library is configured to run on the memory-restricted ESP32 by configuring the library to use less memory, albeit 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 section 3.5.

3.4. Hardware Acceleration

ElectionGuard encryption is fundamentally 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]. By offloading cryptographic operations from the CPU, these accelerators reduce computational load and improve overall system performance [25c, p. 32]. They 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]. The following sections explore how each hardware accelerator can be utilised in the ElectionGuard implementation.

3.4.1. RNG

In ElectionGuard, generating random values is crucial for various operations, including the key-generation and nonce usage in various proofs [BN22, pp. 9, 13]. The ESP32 features a True Random Number Generator (TRNG) that produce 32-bit random numbers that are suitable for cryptographic purposes. Unlike Deterministic Random Bit Generator (DRBG), which rely on algorithms to produce random numbers, the ESP32's TRNG generates randomness from physical processes, such as thermal noise and asynchronous clock mismatches. This ensures a high level of unpredictability essential for cryptographic operations [25d, p. 604]. To effectively utilise the TRNG, a source of thermal noise must be enabled; otherwise, the TRNG will return pseudo-random numbers [25d, p. 609]. The high-speed Analog-to-Digital Converter (ADC) is automatically enabled when the Wi-Fi or Bluetooth module is active. When sourcing noise 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]. Extreme conditions can lower the resulting entropy. To mitigate this, enabling the **SAR_ADC** as a secondary noise source is recommended. 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 the ESP32 Guardian implementation, the **rand_q** function generates a random number below the constant 256-bit constant (q). A 256-bit buffer is filled with randomness from the TRNG using the ESP32 system API. To ensure the generated 256-bit number does not exceed (q) a modulo operation is performed with the value of (q). The 32-bit **RNG_DATA_REG** is read eight times to fill the 256-bit buffer. The system API function will delay if the reading frequency exceeds the maximum reading rate. This delay is necessary to ensure that sufficient external entropy has been introduced into the hardware TRNG state [25d]. Alternatively, using a strong software DRBG can be initialised with the TRNG values as a seed [25d] [Inc25, p. 588]. In this implementation the the risks of delays are neglibile since the reading frequency is low.

3.4.2. SHA Accelerator

In ElectionGuard, hashes are computed using the SHA-256 hash function [BN22, p. 26]. The implementation requires caution because all inputs, whether textual or numeric, are represented as utf-8 encoded strings [BN22, p. 26]. In our case, this is less efficient than hashing the raw bytes of the large integer values due to the conversion overhead to convert the large integer values to utf-8 encoded strings before hashing. This is necessary to get hash value consistent with the Python implementation. The SHA-256 hash function is frequently applied in various cryp-

tographic operations within ElectionGuard. For example, non-vote data such as a Guardians backup share don't need to be homomorphically encrypted. Instead, they are encrypted using hashed ElGamal encryption [BN22, p. 7]. The function hashed ElGamal encryption uses a Key Derivation Function (KDF) based on Keyed-Hash Message Authentication Code (HMAC) that is instantiated with SHA-256 (HMAC-SHA-256) [BN22, p. 7].

The SHA Accelerator significantly enhances the performance of SHA operations compared to purely software implementations [25d, p. 589]. This accelerator support the SHA-256 algorithm used in the ElectionGuard specification. It processes one 512-bit message block at a time. Therefore, software must manage the division of longer messages, along with any required padding [25c, p. 2]. In multi-core environments, libraries like mbedTLS and WolfSSL implement fallback mechanisms to software when concurrent hashing operations are initiated. As a result, computations revert to software calculations when the hardware accelerator is busy [Espb] [Inc]. Benchmarks utilising mbedTLS at processor speeds of 240 MHz reveal that hardware acceleration achieves performance nearly three times faster than the software counterpart [Jin22, pp. 41–42]. Thus, the SHA Accelerator is an effective solution for speeding up SHA-256 hashing operations on the ESP32.

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 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 ElectionGuard operations: modular multiplication and the computationally intensive modular exponentiation. The RSA Accelerator supports operand lenghts of up to 4096 bits [25d, p. 603], which is sufficient for the reduced and standard baseline parameters used in ElectionGuard. A modular exponentation computes $Z = X^Y \bmod M$, while modular multiplication 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 M'. These additional arguments are precomputed by software [25d, pp. 598–599].

Benchmarks comparing the modular exponentation using the mbedTLS library re-

veal that hardware acceleration is more than 12.84 times faster than software implementations [Jin22, p. 51]. However, for small operands the hardware acceleration was 1.44 times slower than the software implementation. This is likely due to the initialisation overhead of the hardware accelerator outweighing the benefits for smaller values [Jin22, p. 51]. Initialisation with small values occurs during the polynomial calculation used in the key-generation. The number of polynomials (starting from 0 and incrementing) is used as the exponent in the operation. Therfore, this penalty could effect us. However, the wolfSSL library used in the ESP32 implementation fallbacks to the software implementation for small operands. Sadly, the mbedTLS has one benefit over the wolfSSL library in that it allows caching of the $\overline{r}$ and M' values, which can significantly speed up operations [Jin22, p. 51]. Due to the fact that all M values are either (q) or (p) and thus constant throughout the computations, caching the M' is likely to be beneficial due to the avoiding of recalculating the inverse of M each time. The $\overline{r}$ can also be cached. $\overline{r}$ is calculated as R^2 mod M where R is calculate as $2^{n \cdot 32 \cdot 2}$. (n) is the number of 32-bit words in M.

In summary, the RSA Accelerator effectively accelerates modular exponentiation and modular multiplication in the ElectionGuard implementation. The mbedTLS library might be a better choice due to the caching of the $\overline{r}$ and M' values. But a fallback mechanism for small operands is necessary to avoid inefficiencies when using the RSA Accelerator with small values.

3.4.4. Performance Analysis

ElectionGuard perform essentialy four key operations on very large integer values: **modular exponentiation**, **modular multiplication**, **modular addition**, and **SHA-256** hash computation [BN22, p. 25]. Using the previously discussed hardware accelerators, the performance of the ElectionGuard operations can be significantly improved. The RSA Accelerator can accelerate the computationally intensive modular exponentiation and modular multiplication operations. The SHA Accelerator can speed up the SHA-256 hash computation. The modular addition cannot be accelerated using dedicated hardware and must rely on a software implementation. The TRNG is only used as a entropy source for generating random numbers and does not accelerate any cryptographic operations.

Test software was written to measure the performance of the the pre-election key-generation ceremony operations and the post-election decryption operations. The key-generation ceremony operations tests the key-generation with different quorums, the backup generation, and the verification of the backups. The post-election decryption operations tests the decryption of the encrypted tally. The decryption is performed with 1 contest and 3 selections (e.g., YES/NO/ABSTAIN). The test measures only the time taken to complete the operation and does not include any

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 (HW) and non accelerated (SW) 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 (HW) and without (SW) hardware acceleration, 30 measurements

prior parameter initialisation. The test performs each operation 30 times to obtain an average time. The standard deviation was also calculated to measure the dispersion of the measurements relative to the mean. High standard deviation values indicate a more variable performance. Each operation was performed after a cold boot of the ESP32 to ensure that the results were not influenced by any prior operations. The software was compiled with the -O2 GCC optimisation flags to optimise compilation for speed. For the software-only runs the hardware acceleration was disabled by setting specific wolfSSL C-preprocessor macros. Table 3.1 compares the performance of accelerated key-generation (HW) and non-accelerated key-generation (SW) across different quorums. Table 3.2 summarises the performance of hardware-accelerated operations (HW) versus without (SW) across three tasks: decryption, verification, and backup.

The results of 3.1 indicate that the accelerated operations (HW) are more then four times faster than purely software operations (SW) across all quorum sizes (e.g., 1.02s vs. 4.41s for Quorum 2). HW shows remarkably low standard deviations (0.23-0.31ms) compared to SW (13.24-22.04ms). This indicates that accelerated operations have a more stable, consistent performance. Large quorums provide better

security but at the cost of increased computational time. The average time for the accelerated oporation 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. Table 3.2 indicates that the accelerated backup operation (0.53s vs. 2.21s) and the accelerated decryption operation (2.29s vs. 9.95s) is more than four times faster. The accelerated verification operation is more than twice as fast as the SW implementation. Accelerated operations show again a more stable performance with lower standard deviations compared to non-accelerated operations. The speedup of the verification is likely less pronounced due to the initial computation steps using small values and thus executing with the software implementation. These results underscore the efficacy of hardware accelerators in optimising both the speed and reliability of cryptographic workflows.

3.5. Communication

The laptop is configured as an AP, 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 AP. 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 AP. 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]. There are additional networking protocols built on top of Wi-Fi and BLE. Both Wi-Fi and BLE support mesh networking,

which facilitates many-to-many device communication and is optimised for creating large-scale device networks. The Wi-Fi stack also supports the proprietary ESP-NOW protocol, which allows direct device-to-device communication with ESP32 devices without requiring an AP connection [25b].

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 [Al-+20, pp. 1–2]. 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]. 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. 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 [Al-+20, pp. 2, 15].

Within the ESP32 microcontroller, several application layer protocols address a wide range of application requirements. Modbus, for example, is a protocol primarily used in industrial IoT environments [Al-+20, p. 3] [25b]. ESP also supports the HyperText Transfer Protocol (HTTP) and the MQTT protocol [25b].

3.6. Communication Factors

In the following sections we will look into communication factors that influence the choice of messaging protocol for the proposed voting system.

3.6.1. Payload Size

One important aspect that narrow down the choice of messaging protocols is the maximum payload size. For each payload we send in order to demonstrate that the payload was computed correctly we add a ZK proof that allows the receiving party to verify the correctness of the payload without revealing any additional information (e.g., secret key) [Mos+24, p. 13]. Proofs consist of commitments, challenge

and response values. Challenge and response values take up 32 bytes in size each [Ben+24, p. 23].

At the post-election decryption phase Guardians send the decryption share to the Tallier. The decryption share is accompanied by Chaum-Pedersen proofs for each selection in a contest. In our experiment we have one contest with 3 selections YES/NO/ABSTAIN. Thus, the decrpytion share contains 3 Chaum-Pedersen proofs. The commitment of the Chaum-Pedersen proof is 1024 bytes (standard parameters) or 768 bytes (reduced parameters) in size. The minimum size of the Chaum-Pedersen proof is thus 1088 bytes (standard parameters) or 832 bytes (reduced parameters) in size. A contest with 3 selection would thus require 3264 bytes (standard parameters) or 2496 bytes (reduced parameters) in size.

During the pre-election key-generation the Guardians share private key shares among each other. 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]. The quorum size of this experiment is 2 Guardians, thus each Guardian would need to generate 2 Schnorr proofs, one for each Guardian. The commitment of a SchnorrProof is 512 bytes (standard parameters) or 384 bytes (reduced parameters) in size. A Schnorr proof is thus 576 bytes (standard parameters) or 448 bytes (reduced parameters) in size. The total size of a Guardians Schnorr proofs for a Quorum of 2 Guardians is 1152 bytes (standard parameters) or 896 bytes (reduced parameters).

Maximum packet size at the application layer:

- BLE Mesh Network: 384 bytes [25a, p. 35]
- Wi-Fi Mesh Network: 1456 bytes [25a, p. 54]
- ESP-NOW: 250 bytes [25a, p. 47]
- MQTT: 265 MB [Al-+20, p. 16]
- HTTP: No limit [Al-+20, p. 16]

At the application layer the BLE mesh network and the ESP-NOW protocol have the smallest maximum packet size. We would need to fragment the data into smaller packets to transmit the data and would need some form of reassembly at the receiving end. The Wi-Fi mesh network has a larger maximum packet size, but it is still not large enough to transmit the Decryption share which contains 3 Chaum-Pedersen proofs. However, we could simply transmit each Chaum-Pedersen proof

seperately. For the MQTT and HTTP protocols the maximum packet size is large enough to transmit the decryption share in one packet.

3.6.2. Payload Format

Data serialisation is the process of structuring data into a streamlined payload format before storing or transmitting it. Broadly speaking, there are two approaches to serialisation: text-based and binary. In text-based serialisation, data is typically structured into key-value pairs in a readable text format. In binary serialisation, 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 serialisation 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 serialised 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 serialise and deserialise the data into a compatible format for the Python implementation. Another benefit of more efficient formats is improved serialisation and deserialisation speeds. This indicates that fewer CPU cycles are used for data processing, leading to lower power consumption. In one test on the ESP32, the serialisation and deserialisation speed was almost halved when using Protocol Buffers compared to JSON [Lui+21, pp. 11–12].

In our case, choosing a binary serialisation approach could be beneficial. The in-memory data representation of our data in the ESP32 implementation uses structs. 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 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 serialisation format. A Protocol Buffer library 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, by defining .proto files, we can generate C code for our ESP32 client and Python code for the Python client.

3.6.3. 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 optimised 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]. Wifi-Mesh supports ACK mechanism but lacks a built-in timeout/retranmission mechanism [25a, p. 51]. MQTT uses three levels of message transmission reliability, each representing a different level of Quality of Service (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]

All messages in our experiment are send with at least QoS 1. As the QoS level increases, the reliability of message delivery also increases. However, this also increases the overhead associated with ensuring that all clients receive the intended messages [Al-+20, p. 11].

3.7. MQTT Implementation

MQTT is designed for constrained environments with low bandwidth. MQTT was chosen due to several advantages over Hypertext Transfer Protocol (HTTP), such as asynchronous messaging, lower power consumption, QoS support [Al-+20, pp. 23, 27]. MQTT uses a publish/subscribe model and consists of 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 clients (subscriber) who are interested in receiving them. To this extent, a client must first subscribe to the specific topic. A client can send messages to multiple clients with a single publish operation to the broker. The broker manages the broadcasting to all subscribers of the message topic [Al-+20, p. 10] [PC20, p. 12]. Clients can receive published messages at different times [Al-+20, pp. 19, 21, 22].

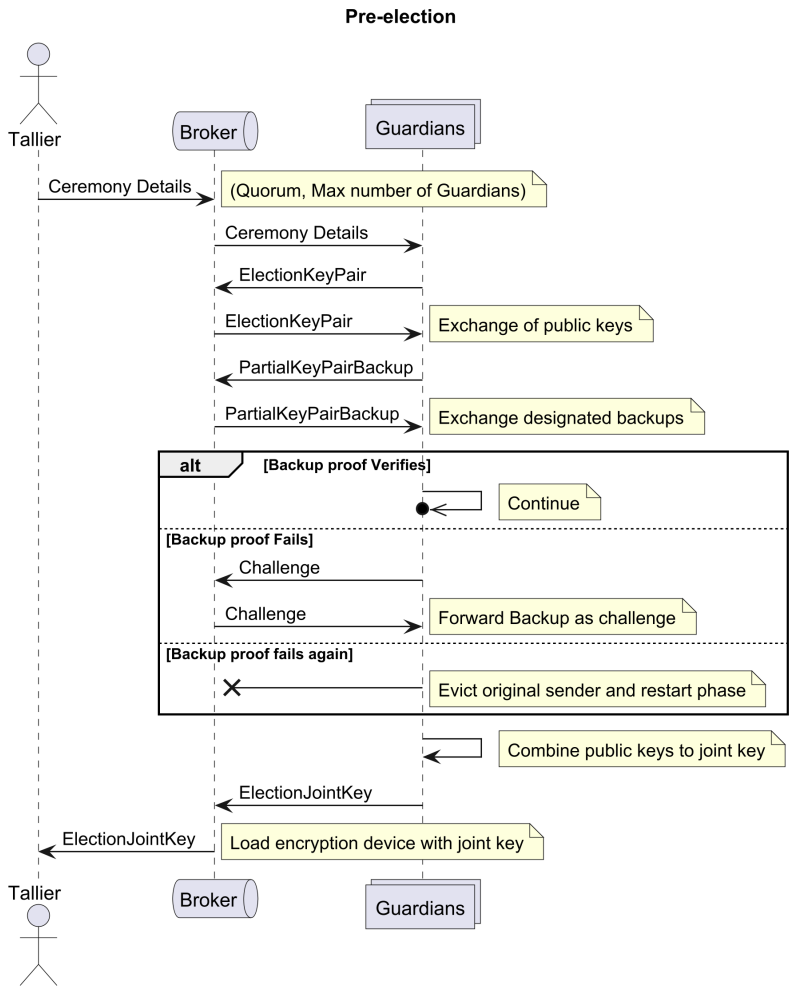


Figure 3.5.: Communication Sequence in the Pre-Election Phase using MQTT

Figure 3.5 shows a sequence diagram of the MQTT brokering model, illustrating all entities involved in the key-generation ceremony, including the Tallier, Guardians, and the MQTT broker. The MQTT broker is running on the Tallier. Both Tallier and Guardians act as publishers and subscribers. During key-generation ceremony, the Guardians programmatically ignore their own public key. For backups, the Guardians ignore all backups that are not meant for them. The MQTT specification contains a **"No Local"** [eta19, p. 73] subscription option that allows a client to

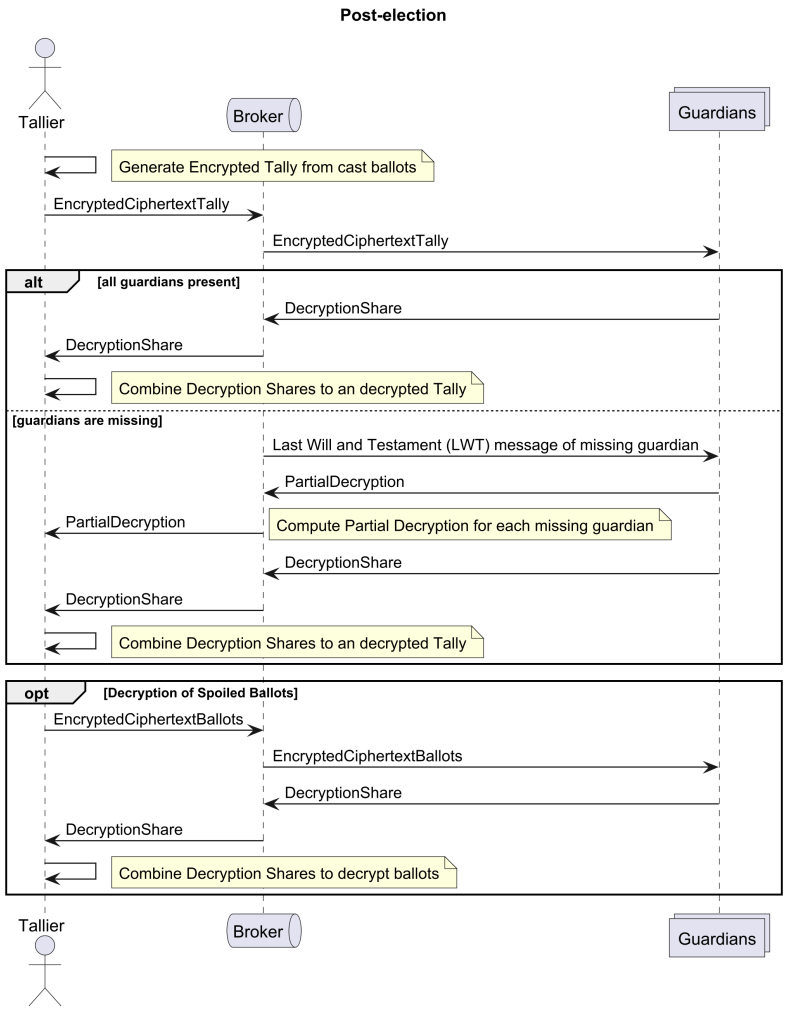


Figure 3.6.: Communication Sequence in the Pre-Election Phase using MQTT

avoid receiving messages it itself has published. However, the ESP32 MQTT library does not implement this option [25b, p. 53]. It is also possible to create one-to-one communication. Guardians could send messages to specific topics that only a specific client is subscribed too. The broker would need to implement some form of access control. In our implementation clients publish to the topics "public_key", "backups", "ciphertally", "decryption_share", "ceremony_details" without any access control. At the end of the key-generation ceremony, the Guardians generate a joint key; the Tallier will simply ignore all other keys sent to it after receiving the first key.

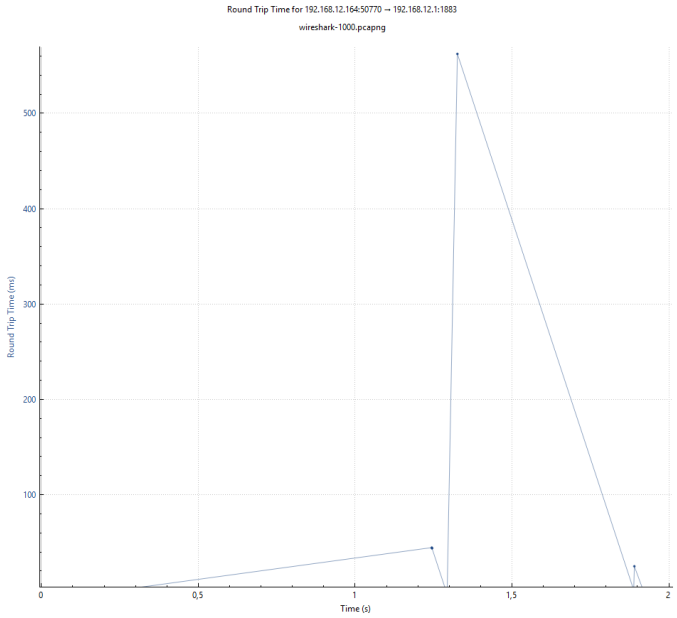
Figure 3.6 shows a sequence diagram of the MQTT brokering model, illustrating all entities involved in the decryption phase, including the Tallier, Guardians, and the MQTT broker. The case where a Guardian is missing is not implemented in this scenario. This could be addressed through Last Will and Testament message, which

allow clients to notify other clients on unexpected disconnects [25b, p. 53]. The Guardian could then act accordingly. Decrypting spoiled ballots is optional and is not implemented in the current system. The decryption of spoiled ballots is similar to the decryption of the encrypted tally.

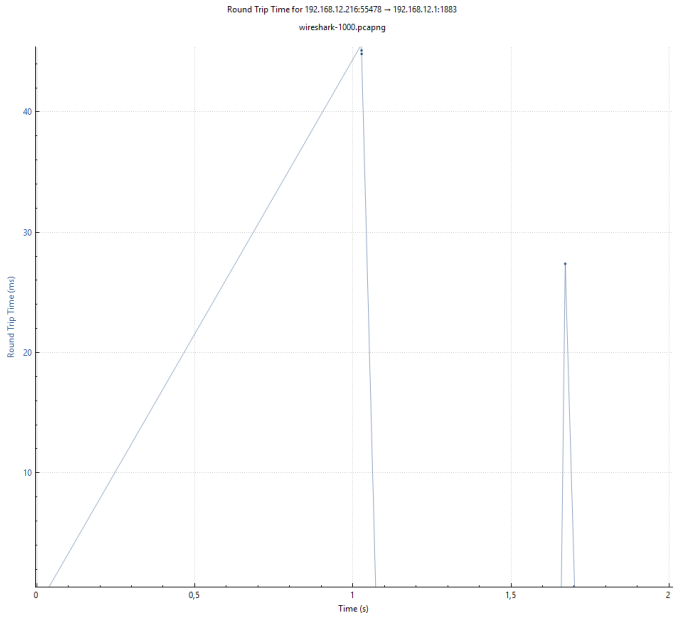
To facilitate the sending and receiving of large packets, we set the input and output buffer size of the ESP32 MQTT implementation to 4253 bytes. If this is not set, large packets are sent sequentially by the client, leading to additional overhead.

3.8. Network Traffic Analysis

Network traffic between Tallier and Guardians was captured using Wireshark (4.4.3) by listening to the AP interface of the laptop. The MTU size of the AP is 1500 Bytes. Both Guardians where positioned in close proximity to the laptop. The further clients are from the broker, the longer the travel time of the MQTT messages and the higher the latency [Al-+20, p. 20]. The traffic was narrowed down by applying a display filter to exclude all non-MQTT traffic. The capture focused on the unencrypted communication described in 3.7. The workflow is automated, requiring no human interaction. Three captures were taken: one with the number of cast ballots set to 10, another with it set to 100, and the third with it set to 1000 votes. The captures will be refered to as Capture 10, Capture 100, and Capture 1000 respectively. In all capture scenarions, no packet loss occured. Large payloads (e.g., keys/backups) exceeding the MTU size are split into multiple TCP segments and reassembled at the receiving end. RTT is the time it takes for a packet to travel from the sender to the receiver and back. RTT is a key-metric for measuring network performance. The RTT of both Guardians is consisten at around 45ms throughout all captures. However, the RTT peaks consistently to over 500ms for Guardian 1 (192.168.12.164) near the beginning of each capture, as seen in Figure 3.7. This could be due to network congestion at the start of the capture. To identify potential bottlenecks, we analyse the most active topics, as shown in Figure 3.8. The figure excludes packets related to QoS. The peak at the beginning of both captures 10 and 100 coincides with peak in RTT. The most active topics are the "backups" and "public_keys" topic. We send a significant number of packets here because clients that send a payload to these topics will receive their own packets again. The throughput could drop significantly as the number of subscriptions increases. As more clients subscribe to topic the number of messages increases [Al-+20, pp. 19, 21, 22]. The payload of these packets is also large, measuring 2940 bytes for "public_keys" and 474 bytes for "backups". Furthermore, payloads send to the topics "public_keys" and "backups" are transmitted with QoS 2. The more clients subscribed to receive a message with QoS 2, the greater the overhead on the message broker [Al-+20, p. 11]. The payloads sent to the topics "decryption_share" and



(a) RTT of Guardian 1 (192.168.12.164) at the start of Capture 1000



(b) RTT of Guardian 2 (192.168.12.216) at the start of Capture 1000

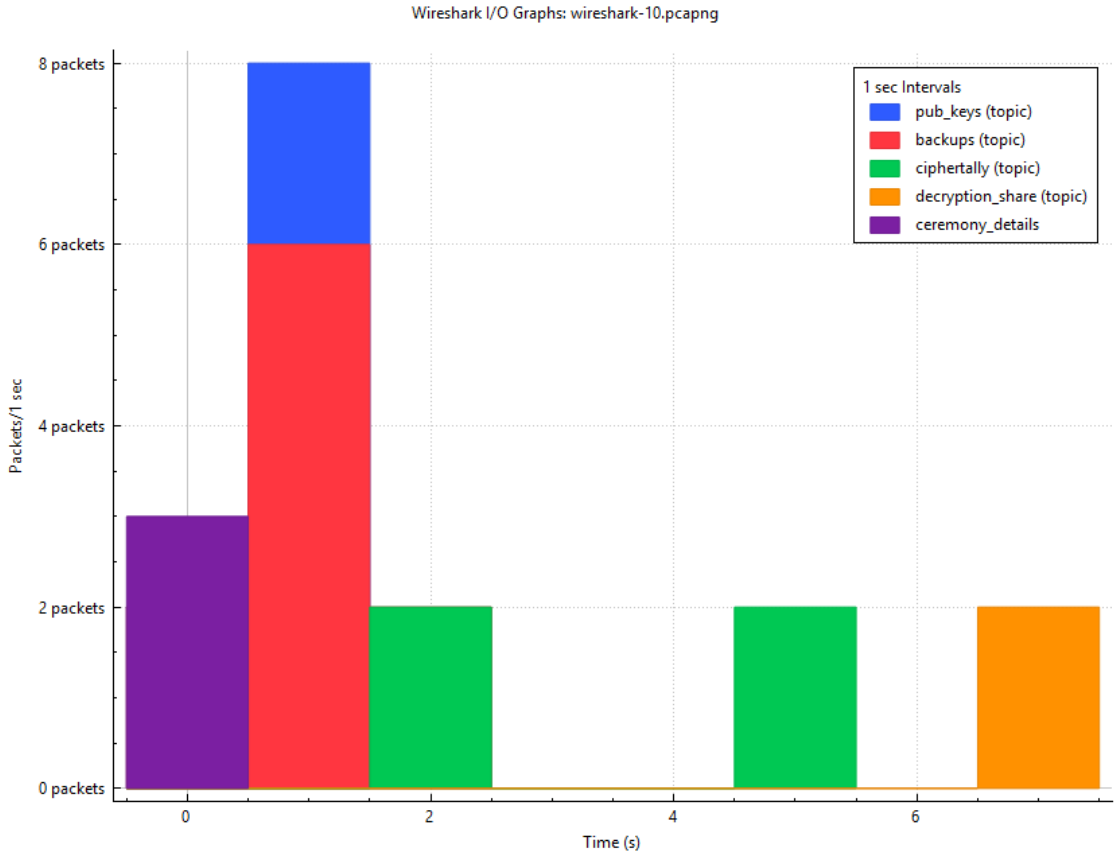
Figure 3.7.: Comparison of RTT of Guardians at the start of Capture 1000

"ciphertally" towards the end of the capture are also quite large, but are published with QoS 1. The payload size send to the "ciphertally" topic is 2498 bytes, while the payload to the "decryption_share" topic is 4253 bytes. The RTT remained under 45ms for both clients towards the end of the capture. Wiresharks conversation

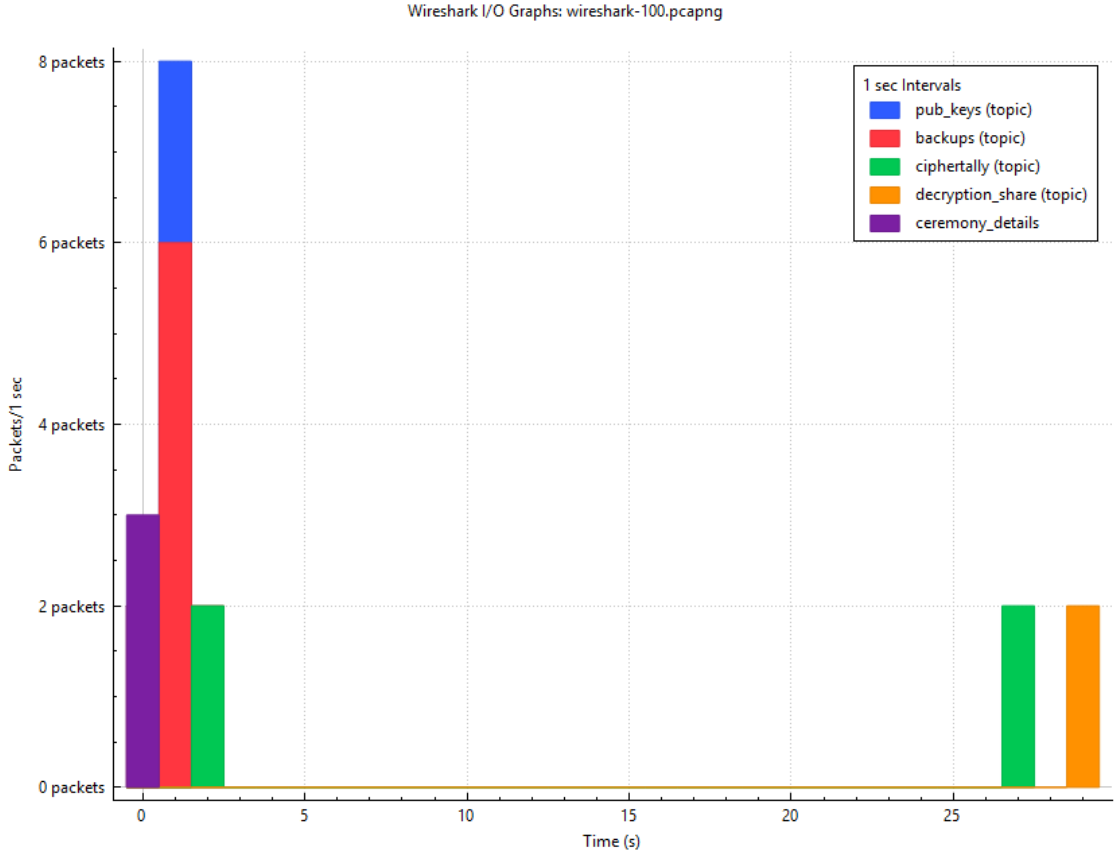
Capture	Duration (s)	G1->Tallier (Bits/s)	G2->Tallier (Bits/s)
10	7.54	10086	10146
100	29.57	2572	2587
1000	251.90	305	305

Table 3.3.: Throughput and Duration of Captures, G1 and G2 are the Guardians

statistic shows us the throughput of the capture, as detailed in Table 3.3. The data volume per capture is consistent at approximately 20 KB, but the transmission frequency varies. Capture 10 exhibits a high-frequency burst with a duration of 7.54s. Capture 100 has a duration of 29.57 seconds, while Capture 1000 is a slow transmission with a duration of 251.90 seconds. Each capture transmits roughly the same data volume but spreads it out over longer periods. Comparing this to the packet rate on specific topics between different captures, it becomes evident that the most time spent is after key-generation and before decryption, during which the Guardians are idling. The increase in duration correlates with the increase in votes. This is due to the processing time required by the Tallier, which needs to homomorphically encrypt the votes before sending the ciphertally. The Tallier's CPU and memory utilisation is plotted in Figure 3.9. Memory usage peaks at 50 MB for Capture 10. Slightly increases in Capture 100, and increases to over 120 MB in Capture 1000. The computation time increases significantly with the number of votes, indicating that the homomorphic encryption performed by the Tallier is the major bottleneck of the system.

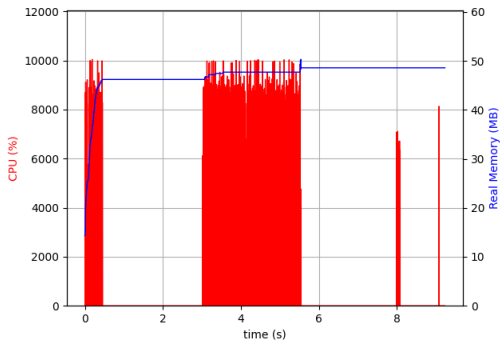


(a) Package rate on specific topics in Capture 10

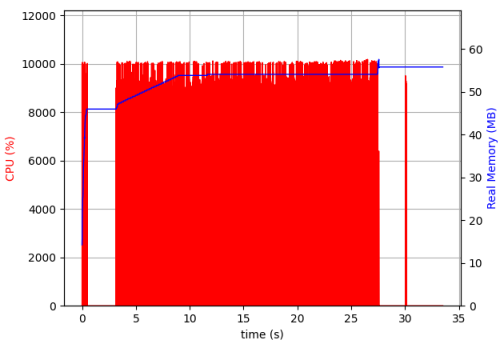


(b) Package rate on specific topics in Capture 100

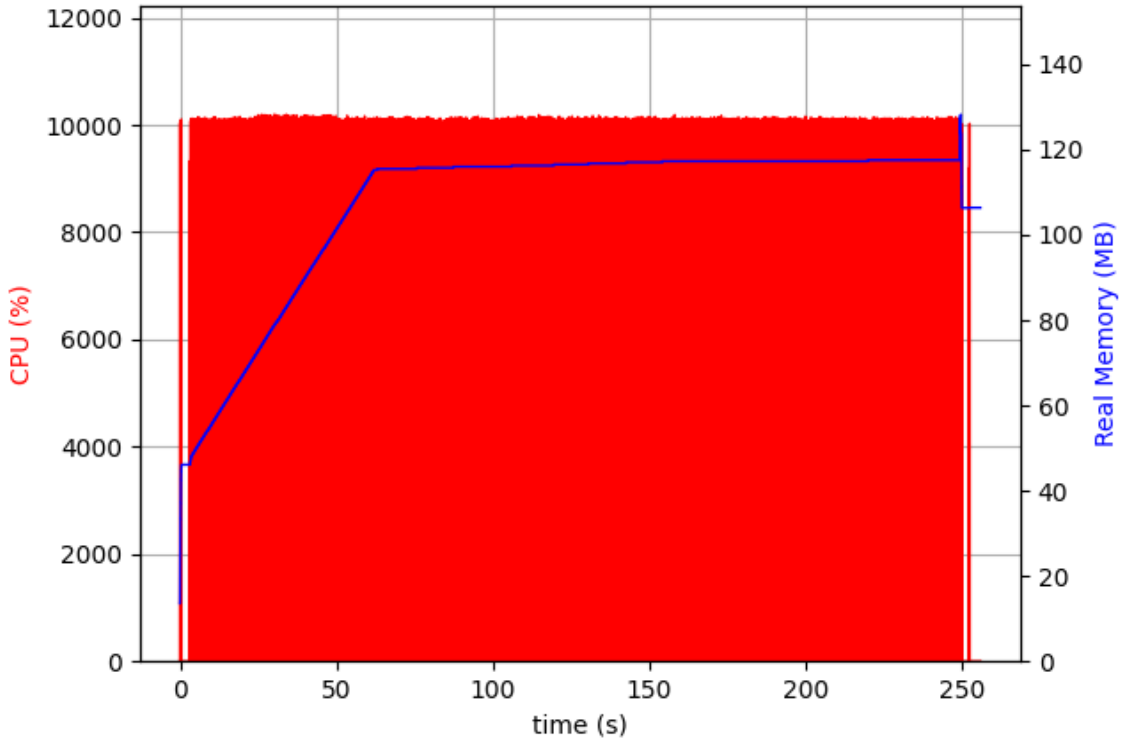
Figure 3.8.: Communication Sequences using MQTT in Different Phases



(a) Capture 10



(b) Capture 100



(c) Capture 1000

Figure 3.9.: Tallier CPU and Memory utilisation in Capture 10, 100, and 1000, plotted with the python package psrecord

4. Conclusions

The implementation of the IoT voting system demonstrates the feasibility of deploying secure, E2E verifiable elections on resource-constrained hardware. Reduced cryptographic parameters provided a pragmatic balance between security and performance, enabling functional execution while adhering to resouce limits. The ESP32s memory constrains necessitated a custom C port of ElectionGuard. However, by leveraging the ESP32s hardware accelerators (RSA, SHA) most cryptographic operations speeds improved by 4x compared to software-only implementations (Table 3.2). This validates the use of ESP32s for guardian roles in the voting system. The homomorphic aggregation on the Tallier (Python) dominated system latency, requiring an election 251.9s for 1000 ballots (Figure 3.9). This CPU-bound process limits scalability, suggesting a need for optimized or distributed computation.The choice of MQTT as the communication protocol proved effective for IoT constraints, balancing payload size limitations (up to 4.2 KB for decryption shares) with reliable transmissions through QoS levels. Network traffic analysis revealed consistend RTT (45 ms) under load, though initial bursts highlight the need for congestion management in larger-scale deployments. The use of Protocol Buffers for binary serialisation minimized overhead, ensuring interoperability between the Python based Tallier and the C-based Guardian nodes.

4.0.1. Future Work

Building upon the success of this implementation, future research should on the following key areas:

Decentralised Communication: While threshold cryptography eliminates single points of failure, our broker-dependent architecture remains vulnerable to central infrastructure failures. A decentralised device-to-device communication model using Wi-Fi mesh networks could enhance robustness through inherent redundancy and self-healing capabilities. This approach would require developing retransmission protocols to maintain delivery guarantees without centralized coordination. **Leveraging ElectionGuard 2.0:** The updated ElectionGuard 2.0 specification and updated C++ core SDK offer promising opportunities for significant performance gains. The new SDK aims to deliver production-level performance for encryp-

tion. Specifically, faster proof computations and more compact proof structures could directly translate to reduced latency and improved resource utilization [24a]. A thorough evaluation and integration of the new SDK, with its focus on production-level encryption performance, is a critical next step.

By addressing these challenges, future research can pave the way for the widespread adoption of secure and verifiable voting systems, fostering greater trust and transparency in democratic processes.

List of Figures

List of Tables

Bibliography

[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] *ElectionGuard Docs*. Version 2.1. Election Tech Initiative. Aug. 2024. URL: https://github.com/Election-Tech-Initiative/electionguard/tree/0f99229669139d3cae6cfddc0a3d554319a84a2e/docs.

[24b] *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.

[24c] *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.

[Com21] 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.

[Inc] wolfSSL Inc. *wolfSSL ESP-IDF Port*. GitHub repository. Version 5.7.4. URL: https://github.com/wolfSSL/wolfssl/tree/v5.7.4-stable/IDE/Espressif.

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

[Inia] Election Tech Initiative. *ElectionGuard C++*. GitHub repository. Version 1.0.1. URL: https://github.com/Election-Tech-Initiative/electionguard-cpp.

[Inib] 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.

[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. Tallier Implementation

A.2. ESP32 Guardian Implementation

A.3. ESP32 Guardian Documentation

A.4. Error Message

Listing A.1: Tallier receives Decryption with invalid Chaum-Pedersen proof

```
[5546:2025-02-24 10:38:36,167]:WARNING:decryption_share.py.is_valid
   :#L129: CiphertextDecryptionSelection is_valid failed for
   guardian: 083af2b6253c selection: yes-selection with invalid
   proof
[5546:2025-02-24 10:38:36,178]:WARNING:decrypt_with_shares.py.
   decrypt_selection_with_decryption_shares:#L182: share: yes-
   selection has invalid proof or recovered parts
[5546:2025-02-24 10:38:36,188]:WARNING:decrypt_with_shares.py.
   decrypt_contest_with_decryption_shares:#L146: could not decrypt
   contest referendum-single-vote with selection yes-selection
[5546:2025-02-24 10:38:36,200]:WARNING:decrypt_with_shares.py.
   decrypt_tally:#L66: contest: referendum-single-vote failed to
   decrypt with shares
```

A.5. Additional Figures

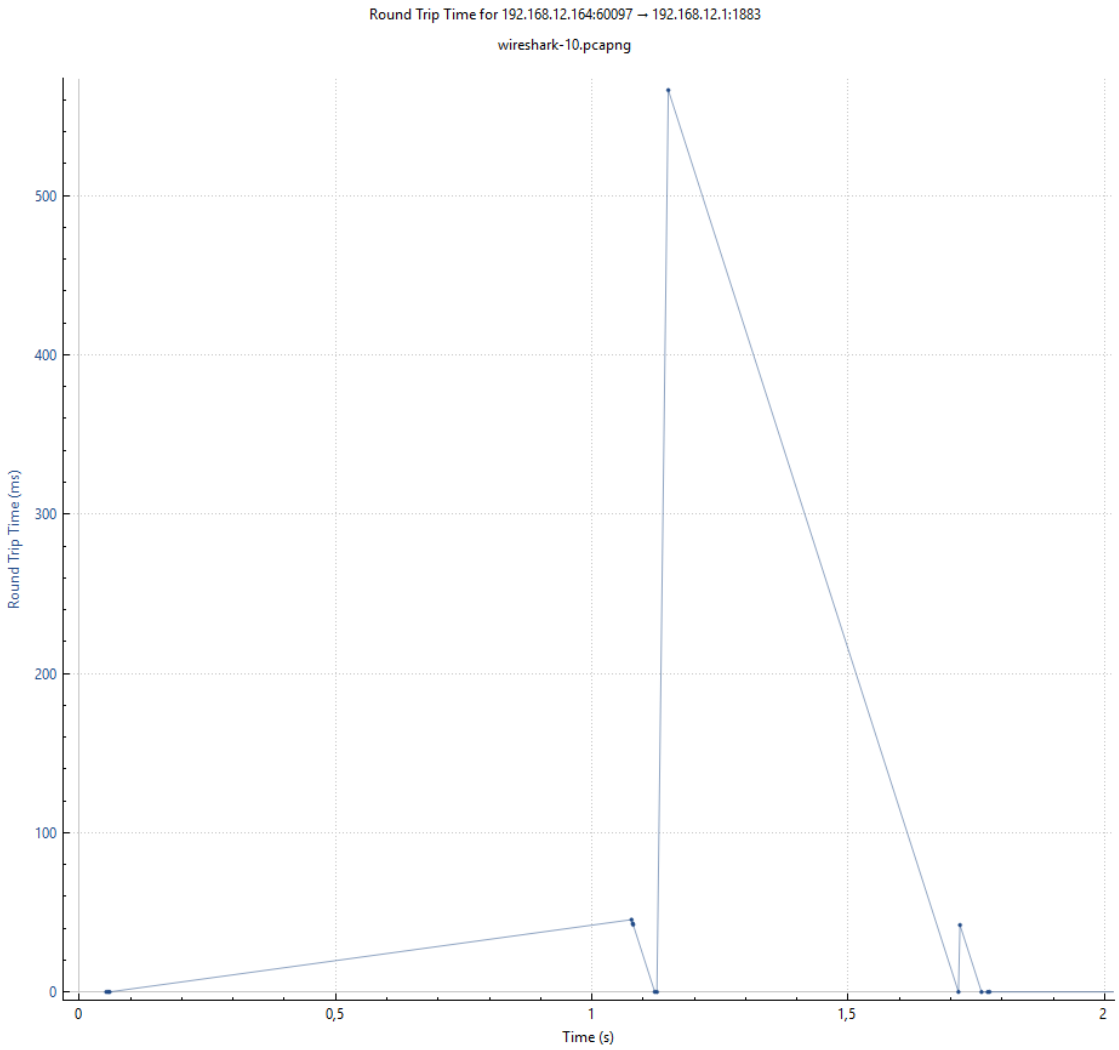


Figure A.1.: RTT of Guardian 1 (192.168.12.164) at the start of Capture 10

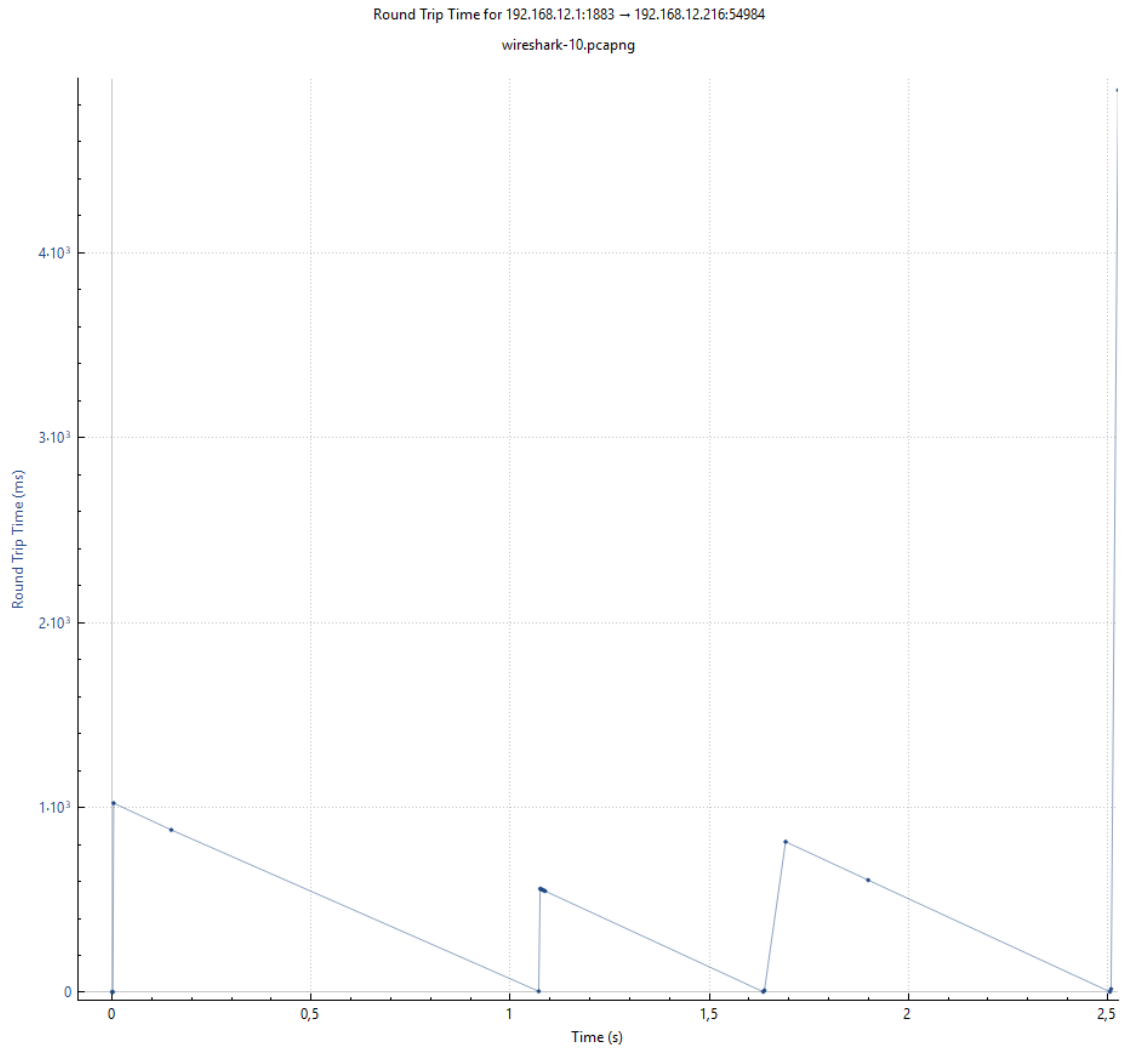


Figure A.2.: RTT of Guardian 2 (192.168.12.164) at the start of Capture 10

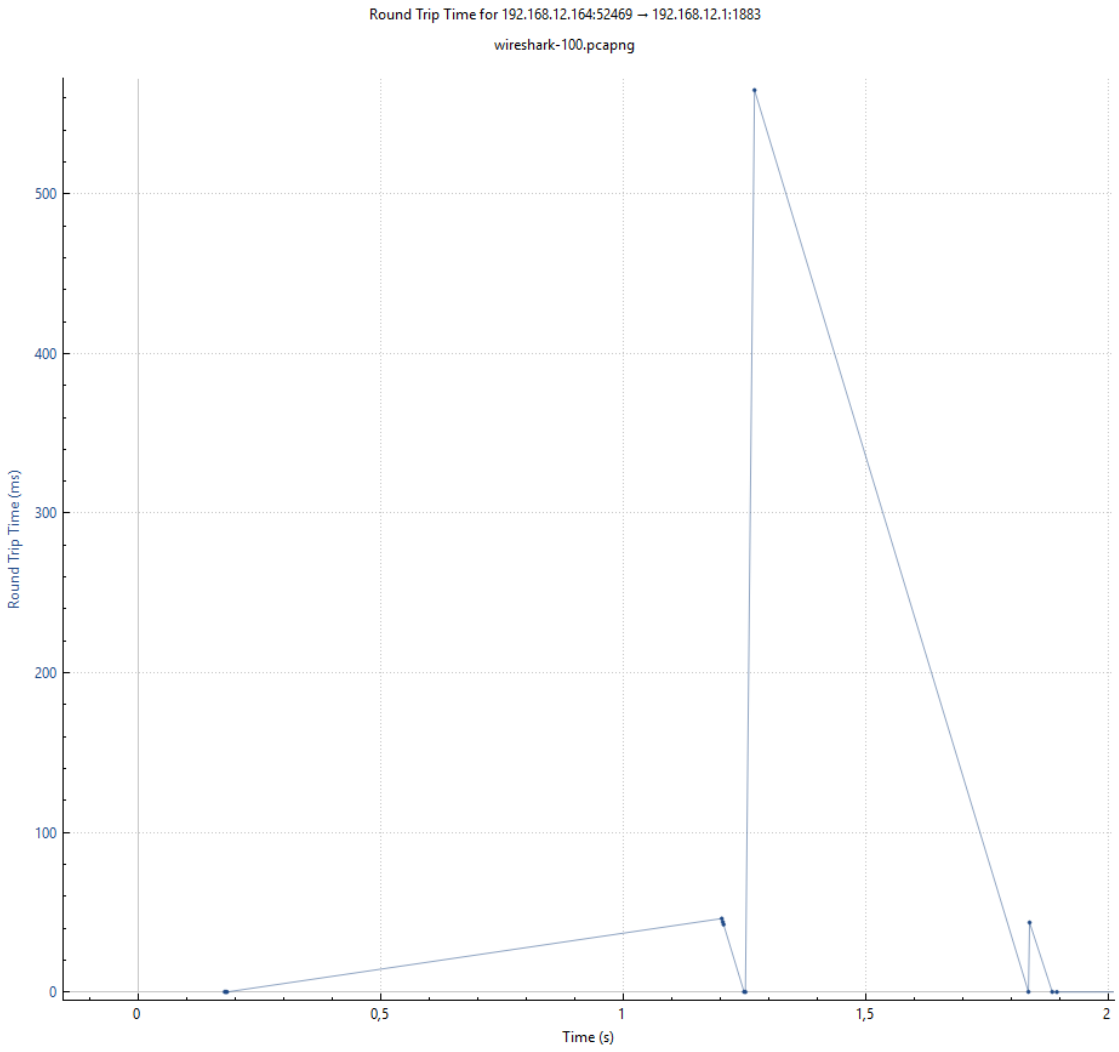


Figure A.3.: RTT of Guardian 1 (192.168.12.164) at the start of Capture 100

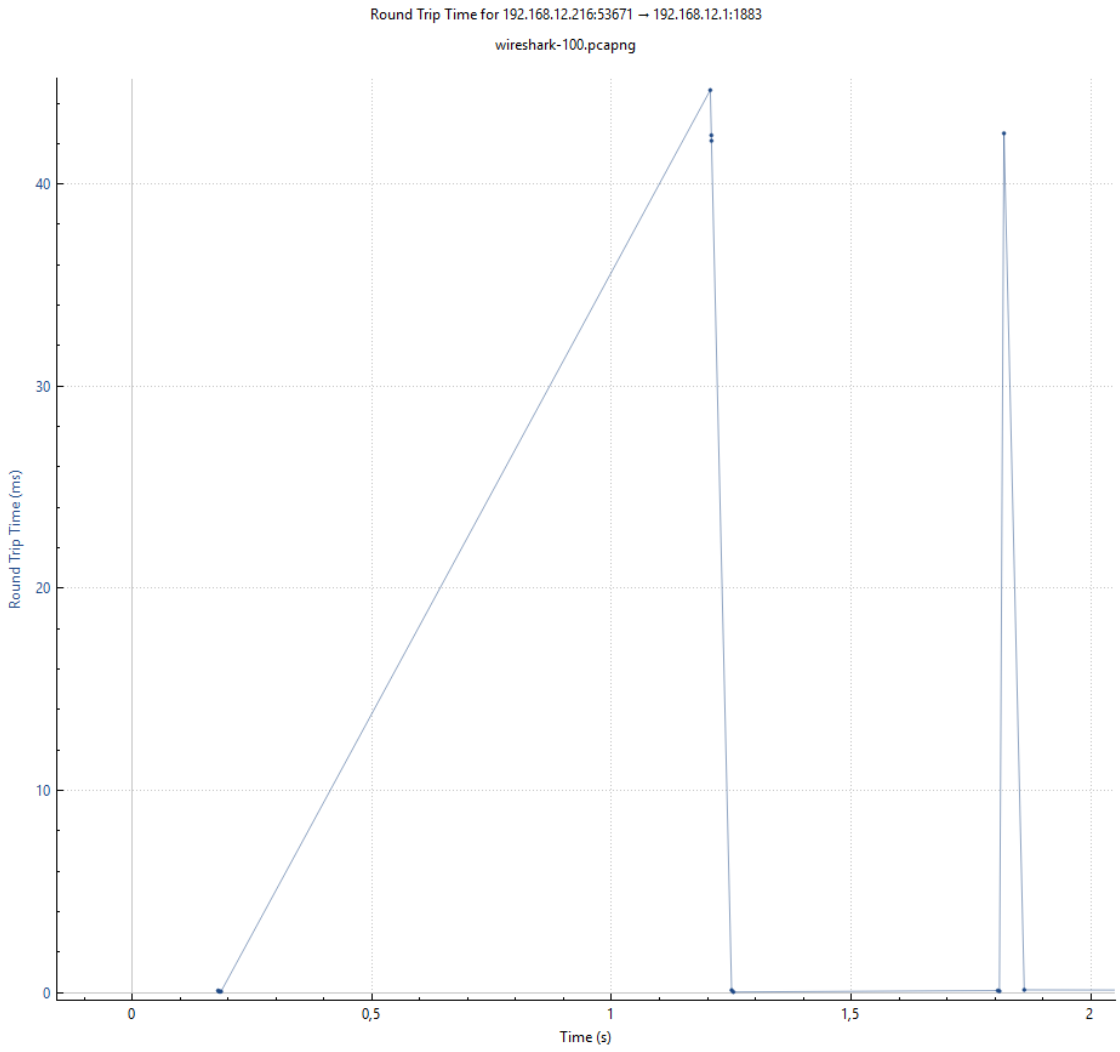


Figure A.4.: RTT of Guardian 1 (192.168.12.164) at the start of Capture 100