Chair for Embedded Systems
Universität Augsburg

Master's Thesis

Implementation of an IoT based Electronic Voting Machine

Gabriel Cmiel

Gutachter/Examiner:	Prof. Dr. Vorname Nachname
Zweitgutachter/Second examiner:	Prof. Dr. Vorname Nachname
Betreuer/Supervisor:	Prof. Dr. Sebastian Altmeyer
Date:	19th February 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

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 **combinatorial**. However, the sharing of sensitive data raises significant concerns about the confidentiality and integrity of information in IoT security. Additionally, the IoT environment is characterized by limited resources, diverse standards, and network vulnerabilities, posing challenges for privacy during data aggregation and transport encryption **smpc**.

Data aggregation involves gathering data from multiple sources and presenting it in a summarized format. This process is crucial, for example, during data analysis but raises privacy concerns, as many systems handle sensitive data. Ensuring the privacy of aggregated data is therefore a primary concern **ppda-fog**. In smart metering systems, Privacy-Preserving Data Aggregation (PPDA) is a leading solution for securing consumer data by aggregating smart meter data at the gateway, preventing attackers from identifying individual users information. While various security techniques have been developed, PPDA is considered more convenient. Many data aggregation schemes use cryptographic techniques, such as homomorphic encryption and digital signatures, to encrypt consumers energy consumption data and send the ciphertext through a trusted gateway to the central distribution system **smart-meter**. Most PPDA solutions rely on computationally intensive homomorphic encryption, making them unsuitable for resource constrained IoT systems.

PPDA techniques can also be found in voting systems. For example, many verifiable voting sytems use homomorphic encryption to tally ballots in a privacy-preserving manner. This method breaks the link between individual voters and their votes, keeping them secret **stuve-study**. However, the tallier who owns the secret key can decrypt all individual votes and thus learns how each individual voter has voted. To prevent, this Threshold secret sharing can be used to distribute the decryption key among multiple talliers, requiring a certain number of them to collaborate to decrypt a ciphertext **stuve-study**. Threshold cryptographic is an area of Multi-Party Computation (MPC). MPC enables a group of parties who do not necessarily trust each other to perform operations on their private data without revealing any information about their data **mpc-ing**. With threshold secret sharing the talliers setup and tallying phase includes more involved interactions and these interactions include blocking steps impacting the communication efficiency **stuve-study**.

In order to ensure the voters that the final election result is correct after the PPDA. The election result must be verifiable **stuve-study**. A stronger form of verifiability is accountability, which enables one to identify the responsible component (or at least limit the possible sources of error) in the event of a detected error **stuve-study**. A important cryptographic building block are Zero-knowledge proofs which allow a party to prove that it performed a certain computational step correctly, without having to reveal any further information **stuve-study**. Zero-knowlege proofs add additional computation and communication overhead to the system.

1.1. Research questions

This research aims to contribute to the literature by gaining insight into the practical application of End-to-end (E2E) verifiable voting system using threshold secret sharing within the IoT environment.

1. What communication protocols are most suitable for inter-guardian communication in the ElectionGuard implementation on ESP32, considering factors like range, bandwidth, and security?

2. Can communication efficiency be improved by optimizing message formats?

1.2. Überblick

Der Autor führt einen potentiellen Leser durch die Arbeit und beschreibt kurz, was den Leser in den folgenden Kapiteln erwartet.

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 **crypto**. 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 **crypto**.

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

- **Confidentiality:** The aim of confidentiality is to make it impossible or difficult for unathorized persons to read a message **crypto**.

- **Authenticity:** This property ensures that the recipient can verify the identity of the sender, ensuring that the message is not from an unauthorized sender **crypto**.

- **Integrity:** This denotes that the message has not been altered during transmission **crypto**.

- **Non-repudiation:** This means that the sender cannot later deny having sent a message **crypto**.

Cryptographic algorithms are mathematical functions used for encryption and decryption **crypto**. 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 **crypto**.

2.2. Cryptography in Voting Systems

The integration of cryptographic methods into voting systems has been a topic of discussion for several decades **stuve-study**. 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) **chaum stuve-study**. 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 **chaum**.

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 **stuve-study**.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 **stuve-study**.

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 **stuve-study**. 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 **vvsg-intro**. 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 **vvsg**.

2.4.1. Key Components of End-to-End Verifiability

E2E verifiability encompasses two principal components **e2e-primer**:

- **Cast As Intended:** Voters can verify that their selections - whether indicated electronically, on paper, or by other means - are recorded correctly **e2e-primer**.
- **Tallied As Cast:** Any member of the public is able to verify that every recorded vote is included correctly in the final tally **e2e-primer**.

All E2E Verifiable voting systems incorporate cryptographic building blocks at their core. The most important and recurring cryptographic building blocks include **stuve-study**:

- **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 **stuve-study**.
- **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 **stuve-study**.

- **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 **stuve-study**.

- **Zero-Knowledge 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 **stuve-study**.

- **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 **stuve-study**.

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 stuve-study**. 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 **stuve-study**.

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 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 **eg-paper**.

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 **eg-paper**. 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. **e2e-pilot**.

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 **eg-paper**. The flexibility of ElectionGuard is a novel feature and one of its primary benefits **eg-paper**.

2.7. Phases of ElectionGuard

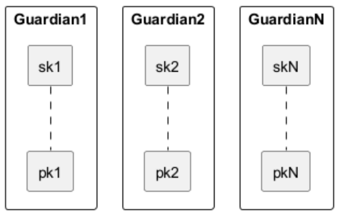
The election process in ElectionGuard can be divided into three main phases: Pre-election, Intra-election, and Post-election. Each phases involves distinc activities. Below, we will explore each phase in detail.

2.7.1. Pre-election

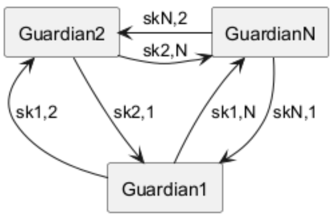
.

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

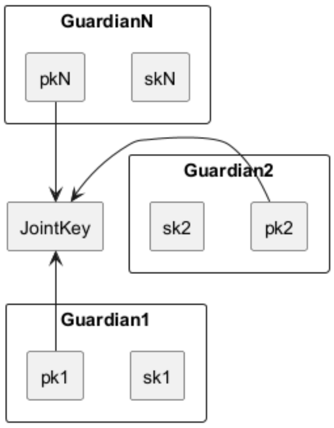
The **election manifest** defines the parameters and structure of the election. It ensures that the ElectionGuard software can correctly record ballots. The manifest defines common elements when conducting an election, such as locations, candidates, parties, contests, and ballot styles. Its structure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



(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 **eg-docs**

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 **eg-spec**. 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 **eg-paper**.

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 **eg-paper eg-docs**. 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 **stuve-study**.

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. **eg-paper**.

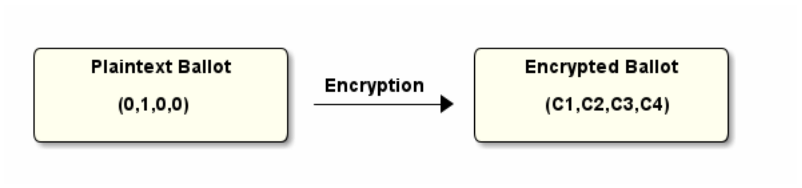


Figure 2.2.: Representation of plain and encrypted ballots

2.7.2. Intra-election

.

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 **eg-paper eg-spec**. 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 **eg-spec**.

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 **eg-spec**.

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 **eg-spec**. ZK proofs are employed to validate that the encryptions correspond to the values of 1 and 0 **eg-paper**. 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 **eg-spec**.

Once the encryption of a ballot is finalized, a confirmation code is generated for the voter **eg-spec**. This confirmation code is a cryptographic hash derived entirely from the encrypted ballot **eg-paper**. 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 **eg-spec**

eg-docs.

2.7.3. Post-election

At the end of the voting period, all encrypted ballots submitted for tallying are homomorphically combined to produce an encrypted tally **eg-spec eg-paper**. 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 **eg-paper eg-spec**. To verify the correctness of these shares, guardians also publish a Chaum-Pedersen proof of each share **eg-spec**. 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 **eg-docs**.

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 **eg-spec**. Independent verification software can be leveraged at any time after completion of an election to confirm the election's integrity **eg-paper**. The true value of a verifiable election is fully realized only when the election is actively verified by voters, election obsevers, or news organisations **eg-spec**.

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 **esp32-module**. 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 **esp32-module-new**, **esp32-series**, **esp32-errata**.The ESP32-D0WDQ6

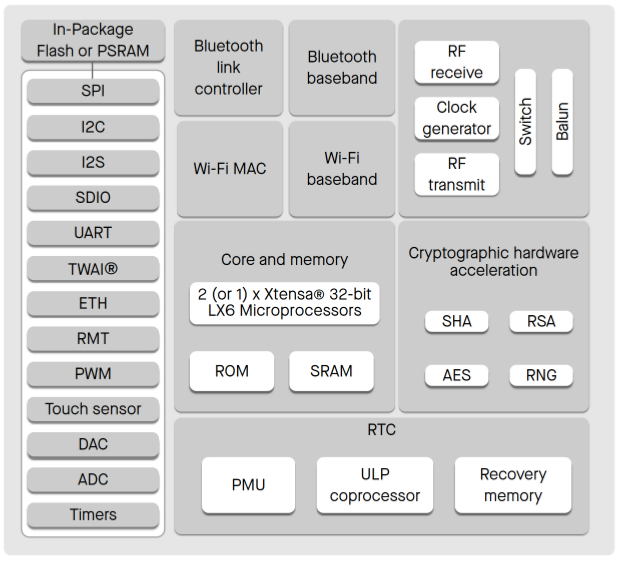


Figure 2.3.: ESP32 Functional Block Diagram

chip is susceptible to 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 **chip-revision**.

ESP32 is designed for robustness, versatility, and reliability across a wide range of applications and power scenarios **esp32-series**. 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 **esp32-series esp32-module**.

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 **esp32-series**
- **520 KB of SRAM** for data and instructions **esp32-series**
- **34 programmable GPIOs esp32-series**
- **Cryptographic hardware acceleration capabilities esp32-series**

The supported cryptographic hardware acceleration capabilities include Advanced Encryption Standard (AES), SHA, RSA, and RNG **esp32-series**. A functional block diagram illustrating the components and subsystems within ESP32 is presen-

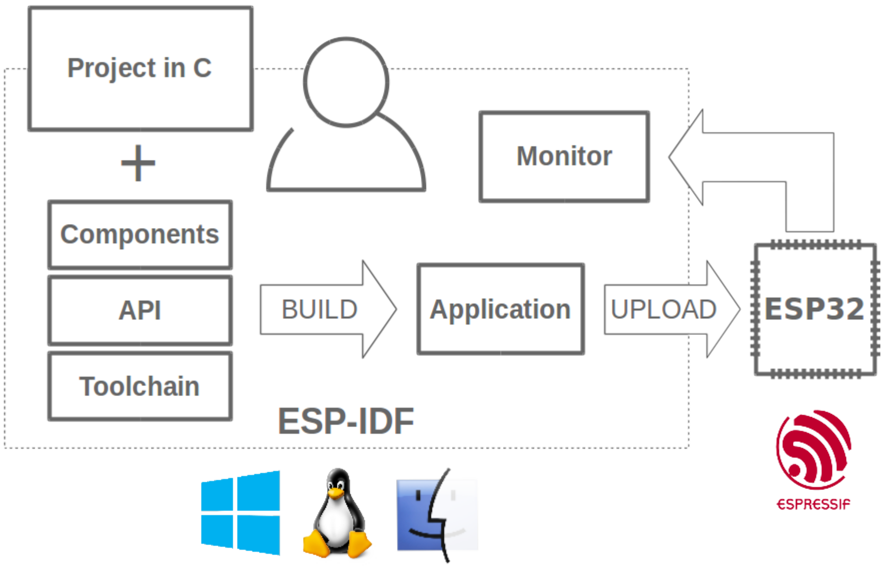


Figure 2.4.: ESP32 Functional Block Diagram

ted in Figure 2.3.

2.8.1. Development Environment

The freeRTOS operating system powers the ESP32 **esp32-module**. 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 **??**.

An ESP-IDF project can be seen as an amalgamation of numerous components coded in C. 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 **esp-prog**.

3. Implementation of an IoT based Electronic Voting Machine

3.1. Hardware and software setup

Device	**Number**	**Notes**
ESP32	2	NodeMCU Dev. Board
Laptop	1	Sony Vaio VPCEH3G1E

Table 3.1.: Hardware devices

The hardware configuration is presented in Table 3.1, providing a detailed account of the devices involved in the experimental setup. The laptop and the ESP32 devices are connected via wi-fi

The NodeMCU ESP32 development board serves as the main controller, while the LCD display module is used to present information to the user. The button module allows user interaction, and the jumper wires are used to connect the components. The breadboard is used to build the circuits.

Figure 3.1 depicts the software components used in the Guardian Prototype. The software components model contains the business logic, the adapter component includes the communication logic, and the view component is responsible for the user interface.

The model component implements the ElectionKeyPair and ElectionPartialKeyBackup classes from the python reference implementation as seen in figure as c structs 3.2 the c structs can be found in Appendix **??**

. These classes are based from the python classes as seen in Figure 3.2.

The hardware configuration is presented in Table **??**, providing a detailed account of the devices involved in the experimental setup. Table **??** contains the specific software tools and applications pivotal to this research.

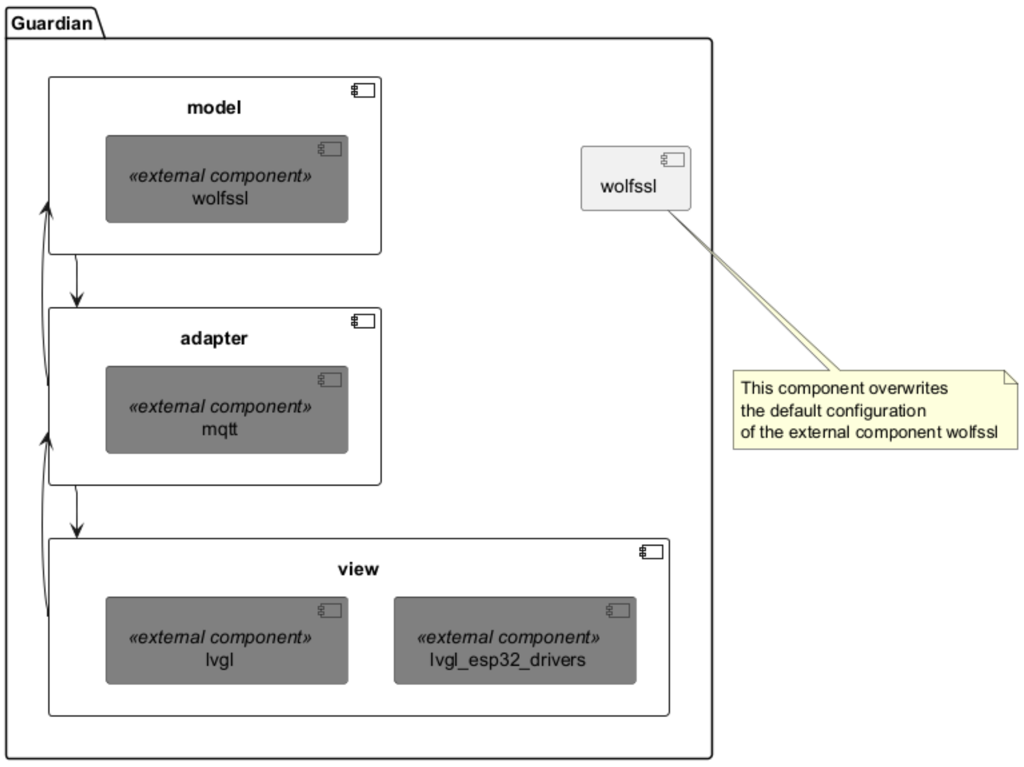

Guardian
model
«external component» wolfssl
adapter
«external component» mqtt
view
«external component» lvgl
«external component» lvgl_esp32_drivers
wolfssl
This component overwrites the default configuration of the external component wolfssl

Figure 3.1.: Software components for the Guardian Prototype

3.1.1. Algorithms

3.2. Computation

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 **eg-spec**. 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 **eg-spec**.Consequently, employing optimized libraries for modular arithmetic becomes crucial for achieving good performance **eg-paper**.

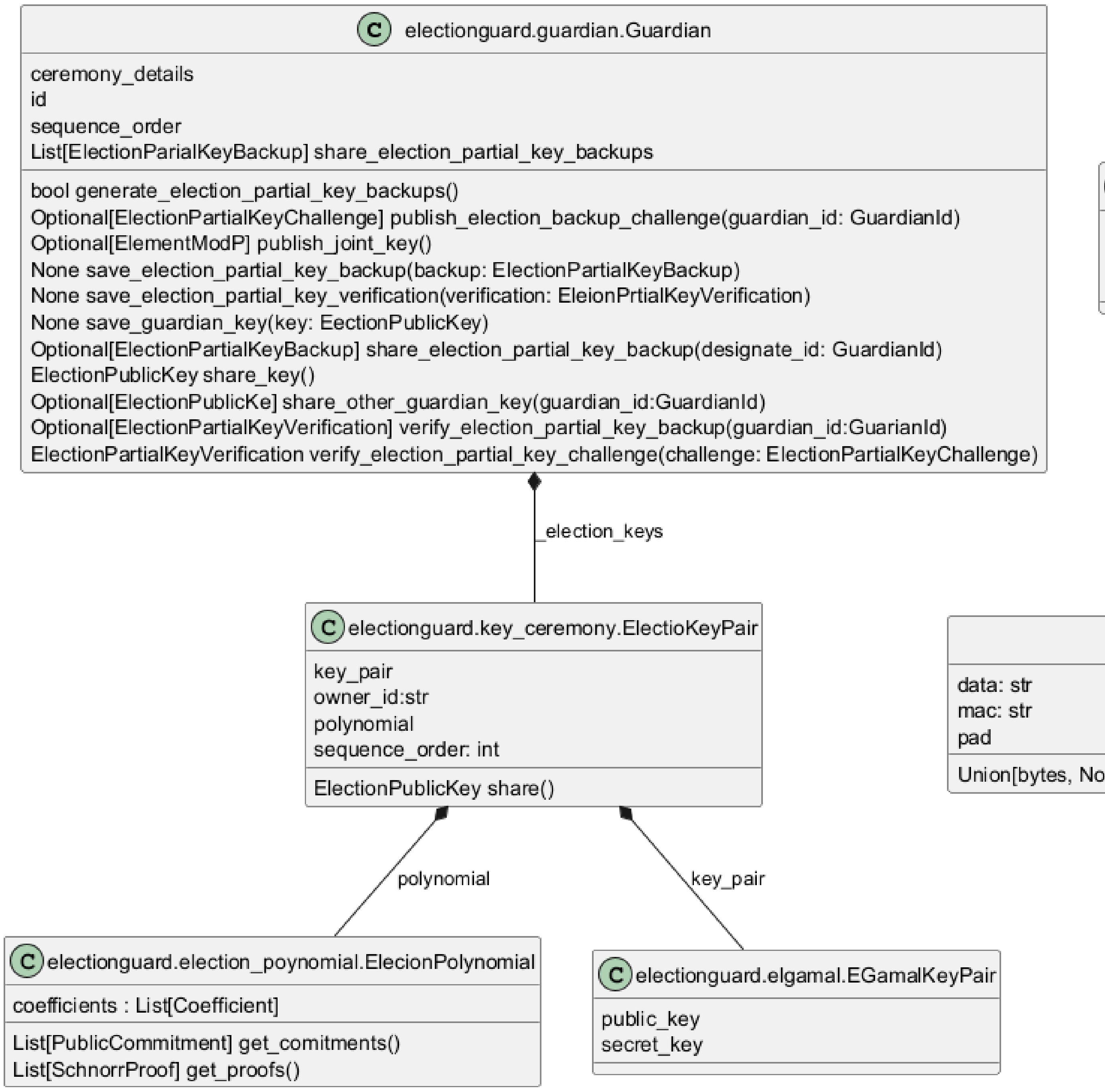

electionguard.guardian.Guardian
ceremony_details
id
sequence_order
List[ElectionParialKeyBackup] share_election_partial_key_backups
bool generate_election_partial_key_backups()
Optional[ElectionPartialKeyChallenge] publish_election_backup_challenge(guardian_id: GuardianId)
Optional[ElementModP] publish_joint_key()
None save_election_partial_key_backup(backup: ElectionPartialKeyBackup)
None save_election_partial_key_verification(verification: EleionPrtialKeyVerification)
None save_guardian_key(key: EectionPublicKey)
Optional[ElectionPartialKeyBackup] share_election_partial_key_backup(designate_id: GuardianId)
ElectionPublicKey share_key()
Optional[ElectionPublicKe] share_other_guardian_key(guardian_id:GuardianId)
Optional[ElectionPartialKeyVerification] verify_election_partial_key_backup(guardian_id:GuarianId)
ElectionPartialKeyVerification verify_election_partial_key_challenge(challenge: ElectionPartialKeyChallenge)
_election_keys
electionguard.key_ceremony.ElectioKeyPair
key_pair
owner_id:str
polynomial
sequence_order: int
ElectionPublicKey share()
data: str
mac: str
pad
Union[bytes, No
polynomial
key_pair
electionguard.election_poynomial.ElecionPolynomial
coefficients : List[Coefficient]
List[PublicCommitment] get_comitments()
List[SchnorrProof] get_proofs()
electionguard.elgamal.EGamalKeyPair
public_key
secret_key

Figure 3.2.: Python Classes

3.2.1. Cryptographic constants

Among all computations in ElectionGuard, modular exponentiation presents the highest computational cost, serving as the primary limiting factor in performance analysis **eg-spec**. Within ElectionGuard, most modular exponentiation operations utilize a fixed base, which may be either the generator (g) or a public key **eg-paper**. The generator G is a mathematical constant specified in the ElectionGuard specification.

The standard baseline parameters include:

- A 4096-bit prime (p) **eg-spec**
- A 4096-bit generator (g) **eg-spec**
- A 256-bit prime (q) **eg-spec**

Alternatively, the system can also utilize reduced baseline parameters, which consist of:

- A 3072-bit prime (p) **eg-spec**
- A 3072-bit generator (g) **eg-spec**
- A 256-bit prime (q) **eg-spec**

For this application, we opt for the reduced parameters, as they offer enhanced performance, albeit at the expense of lower security **eg-spec**. The cryptographic constants used in the implementation can be found in Appendix A.2.1.

3.2.2. Comparison of ElectionGuard Implementations

The Python reference implementation of ElectionGuard utilizes the C-coded Gmpy2 library for large integer arithmetic. In contrast the C++ and Kotlin implementations rely on the HACL* C library for similar purposes **eg-docs**. Interestingly, both implementation use C-coded libraries for handling large integer arithmetic. Moreover, the C++ implementation incorporates pre-computed tables to speed up certain modular exponentiations **eg-docs**. This optimization is possible because many of the exponentiations are performed with a fixed base, either the constant generator (g) or a public key. The pre-computed tables contain specific powers of these bases **eg-docs**.

Feasability of the Python reference on ESP32

The Python reference implementation encompasses the entire ElectionGuard specification **python-reference**. Running the Python reference implementation on the ESP32 may be achievable through **MicroPython**, an implementation of Python 3.x targeted for microcontrollers and embedded systems. MicroPython mirrors and adapts the functionalities of the standard Python library to accommodate the limitations inherent to to microcontrollers, such as restricted memory and processing speed **micropython micropython-performance**. There are drawbacks to this approach, as certain essential modules, functions, and classes may be absent in MicroPython **micropython**. Additionally, applications developed in MicroPython are prone to memory fragmentation and may experience issues with objective expending in size. **micropython-performance**. Depending on the complexity of the task and memory allocation, the performance of MicroPython might be inferior to that of a C implementation **micropython-performance**. For instance, in a comparative study of software-based SHA-256 computation for the ESP32, the C implementation outperformed the MicroPython implementation by 45% **micropython-performance**.

Feasability of the C++ reference on ESP32

The ElectionGuard C++ reference implements only a subset of the ElectionGuard specification, focusing on the encryption library and thus addressing only intra-election functionalities citecpp-reference. ESP-IDF supports the development of applications in C++. Certain C++ features, such as exception handling, must be enabled in the project configuration. This results in a slight increase in the application binary size **esp-prog**. The Runtime Type Information (RTTI) feature can remain disabled since dynamic cast conversions and the typeid operator are not utilized in the ElectionGuard C++ implementation. C++ Threads are supported, these are implemented as wrappers around C pthreads, which in turn wrap around FreeRTOS tasks **esp-prog**.

Upon porting the C++ implementation to an ESP32 component, we conducted tests using the provided C Unit Tests, particularly focusing on the ElGamal encryption test. However, the test failed upon invoking the **pow_mod_p()** function, which is intended to optimize modular exponentiation by utilizing pre-computed tables.

Listing 3.1: FixedBaseTable Definition

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

The **FixedBaseTable** is defined as a 2D array. Here, **MAX_P_LEN** denotes the lenght of each **uint64_t** array, **OrderBits** indicates the number of **uint64_t** arrays in each inner array, and **TableLength** represents the number of inner arrays in the outer array.

The **pow_mod_p()** function is optimized for specific parameters: (order_bits = 256), (window_size = 8), and (table_length = 32). Any modifications to these parameters could affect the function's internal operations. The window_size determines into how many groups of (k) bits the exponent is parsed, while the order_bits reflect the number of possible values within each group. With an 8-bit window_size, the order_bits would amount to 256 2^8 **eg-spec**.

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

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

Given that **MAX_P_LEN** is defined as 64, substituting into the equation yields a total size of 4MB:

$$\text{FixedBaseTableSize} = 8 \times 64 \times 256 \times 32 = 4194304\text{bytes} = 4\text{MB} \tag{3.2}$$

If we calculate the size of the **FixedBaseTable** with our reduced baseline parameters, we find:

$$\text{FixedBaseTableSize} = 8 \times 48 \times 256 \times 32 = 3145728\text{bytes} = 3\text{MB} \tag{3.3}$$

The ESP32 features (320 KB) of DRAM. Due to a technical limitation, a maximum of (160 KB) is available for dynamic allocation **esp32-ref**. Consequently, the **FixedBaseTable** exceeds the memory capacity of the ESP32. While reducing the **window_size** results in smaller tables and reduced memory usage, it simultaneously increases the number of multiplications **eg-spec**. Considering these resource limitations and the fact that the ElectionGuard C++ implementation is designed with Intel Atom-level processor performance in mind, alongside the memory requirements of the **FixedBaseTable** optimization, it becomes evident that the C++ implementation is not feasible on the ESP32 **eg-docs**.

3.2.3. Implementation Strategy for ESP32

As previously discussed, both the Python and C++ reference implementations prove impractical for porting to the ESP32. While the Python implementation may be feasible with MicroPython, the performance may be suboptimal due to possible memory fragmentation. The C++ implementation, on the other hand, is not viable due to memory and processor constraints. Ultimatelty, a pure C implementation stands out as the most feasible approach for the ESP32. To achieve optimal performance on the ESP32, we must consider the dedicated hardware accelerators available. The ESP32 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 **esp32-series**.

In ESP32 the cryptographic primitives are implemented in a fork of the mbedTLS library. The fork includes patches related to hardware routines for on-board cryptographic hardware acceleration **esp32-ref**. Optionally, a port of the WolfSSL library is also available, which also includes ESP32-specific hardware routines **wolfSSL-manual**. To sucessfully run WolfSSL on the memory-restricted ESP32, it is essential to configure the component to use less memory, albeit at a potential performance cost. An excerpt from the Component configurations for WolfSSL can be found in Appendix A.1.1

3.2.4. 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 **eg-spec**. 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 **esp32-ref**.

To utilize the TRNG effectively, it is necessary to enable a source for thermal noise; otherwise, the TRNG will return pseudo-random number **esp32-ref**. The High-speed ADC is enabled automatically when the Wi-Fi or Bluetooth module is enabled, which is the case in our design **esp32-ref**. 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 **esp32-ref**. 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 **esp32-ref**.

In our ESP32 implementation, as outlined in Appendix A.2.2, we aim to generate random values below a specific mathematical constant (q), which is a 256-bit prime number. 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 **esp32-ref**.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 **esp32-ref wolfSSL-manual**.

3.2.5. SHA Accelerator

ElectionGuard encrypts non-vote data, such as cryptographic shares of a guardian's private key, 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 **eg-spec**. An implementation of the HMAC-SHA-256 function can be found in Appendix **??**.

The SHA-256 hash function is frequently applied in various cryptographic operations within ElectionGuard, with implementation details provided in Appendix A.2.3. The ESP32 microcontroller is equipeed with a SHA Accelerator that significantly enhances the performance ofSHA operations compared to purely software implementations **esp32-ref**. 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 **esp32-series**.

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 **mbedTLS-fork wolfSSL-port**. Benchmarks uzilizing mbedTLS at processor speeds of 240 MHz reveal that hardware acceleration achieves performance nearly three times faster than software couterparts **eval-crypto**. 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.

3.2.6. RSA Accelerator

ElectionGuard's decision to use integer ElGamal instead of elliptic-curve ElGamal was driven by its conceptual simplicity and lower implementation barrier **eg-paper**. 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 **ecc-eval**, 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 **esp32-series esp32-ref**. 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 **esp32-ref**. 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 **esp32-ref**.

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 **eval-crypto**. 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 **eval-crypto**. Since calculating the $\overline{r}$-inverse is computationally expensive, precomputing and caching these values can enhance performance **eval-crypto**.

In our implementation, the modular exponentiation function switches to a software implementation for small values, as shown in Appendix A.2.4. 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 A.2.4.

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.

3.2.7. Performance Analysis

Function	**Metric**	**Accelerated**	**Non Accelerated**
Key generation	Average Time (μs)	1531095.75	10800822.00
	Standard Deviation (μs)	221.07	23598.24
Verification	Average Time (μs)	832673.31	2437181.75
	Standard Deviation (μs)	125.65	171.50
Backup	Average Time (μs)	513578.91	3599221.75
	Standard Deviation (μs)	125.65	22529.10

Table 3.2.: Comparison accelerated and non accelerated Key Ceremony operations

Function	**Quorum 3**	**Quorum 4**	**Quorum 5**
Average Time (μs)	1531095.75	2041469.88	2551797.75
Standard Deviation (μs)	221.07	252.68	291.58

Table 3.3.: Comparison accelerated Key generation with different quorums

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 3.3 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.

3.3. Communication

A sequence diagram of the pre-election steps are seen in 3.3. In it we see that the Election administrator sends ceremony details to the Guardians and receives a joint key in return. The joint key is then loaded into the encryption devices. In order to generate a joint key each guardian send their public key to a receiving guardians. The receiving guardian generates a designated backup for each guardian. If a guardian receives a designated backup the included proofs are verified. If a proof failed the guardian send the unverified backup to other guardians as a challenge. If the proof fails again this guardian that generated the backup is evicted and the ceremony has to be restarted.

A sequence diagram of the intra-election steps is seen in 3.4. The election administrator would need an empty ballot to the voter. A voter then sends their filled ballot to the encryption device. The encryption devices sends back a verification code. The voter can then decide to either cast or spoil the ballot associated with the verification code. If the voter decides to spoil the ballot the encryption devices needs to mark the encrypted ballot as spoiled and the process starts a new. The loop is broken if the voter decides to cast a ballot.

In the post-election steps the encryption device generates a decrypted tally from all cast encrypted ballots and send the encrypted tally to the guardians. If all guardians are present each guardian generates a decryption share which is combined to a decrypted tally. This decrypted tally is then send back to the administrator. If guardians are missing the guardians in addition to generating their decryption share have to compensate for the missing guardians. Missing decryption shares are reconstructed from the backups generated in the pre-election phase. Finally all

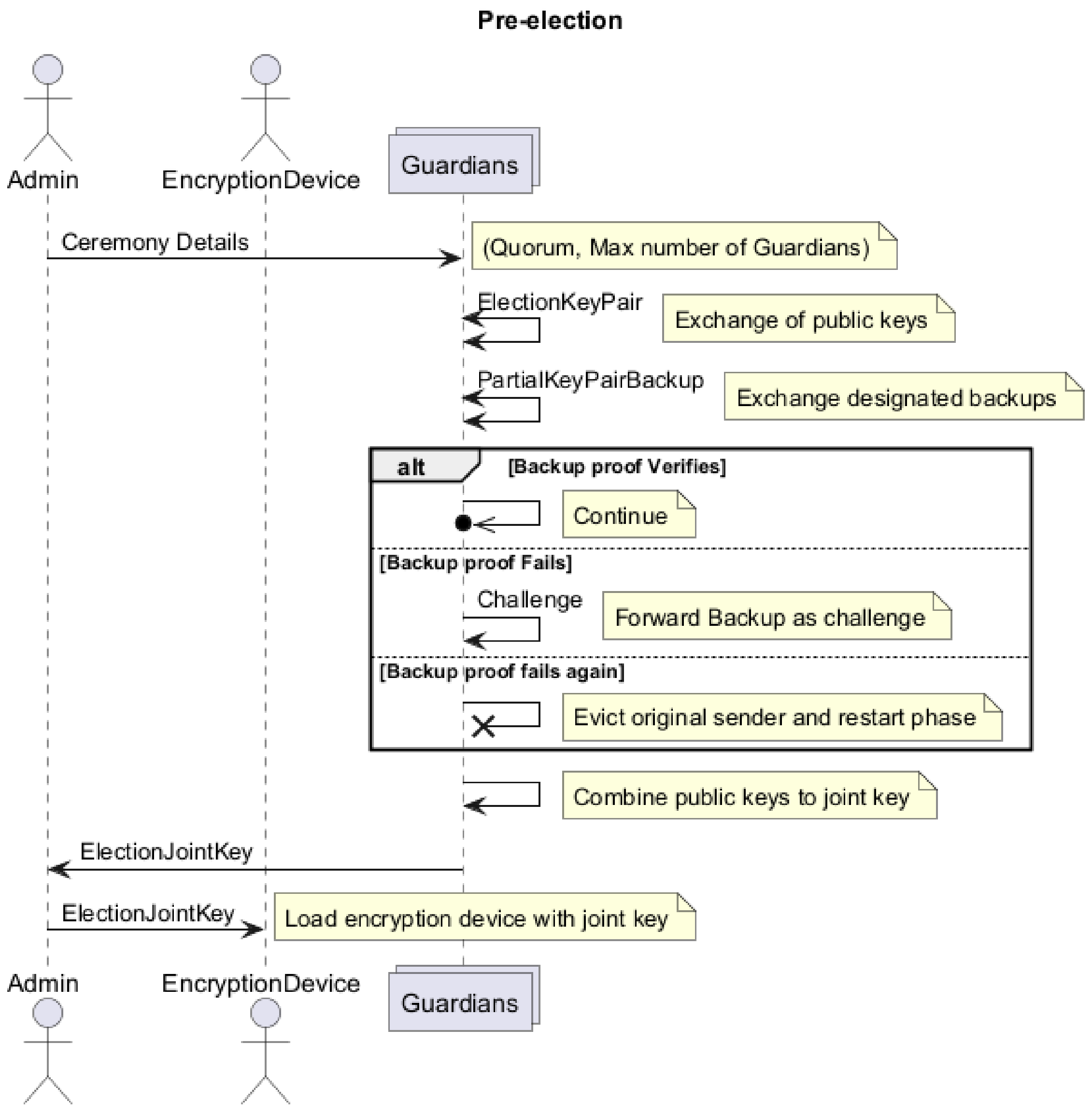


Figure 3.3.: Communication Sequence in the Pre-election phase

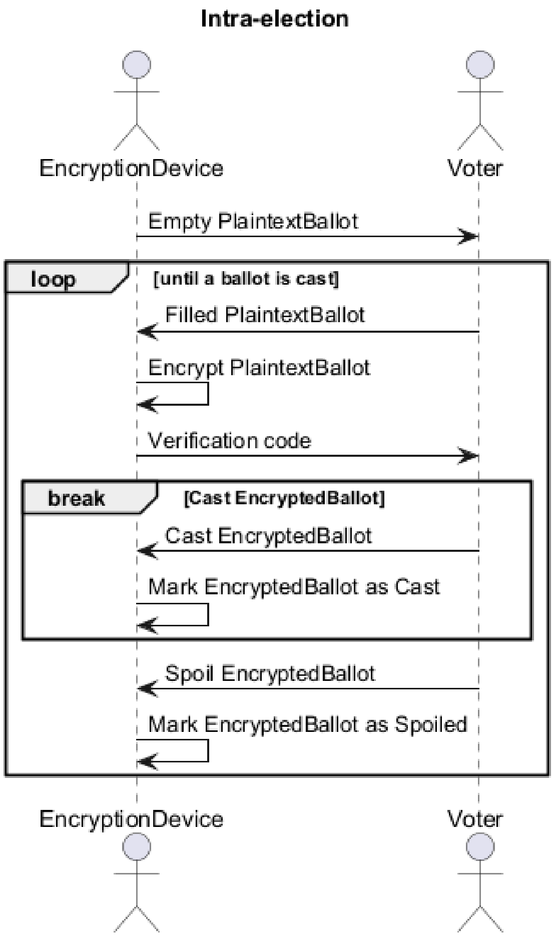


Figure 3.4.: Communication Sequence in the Intra-election phase

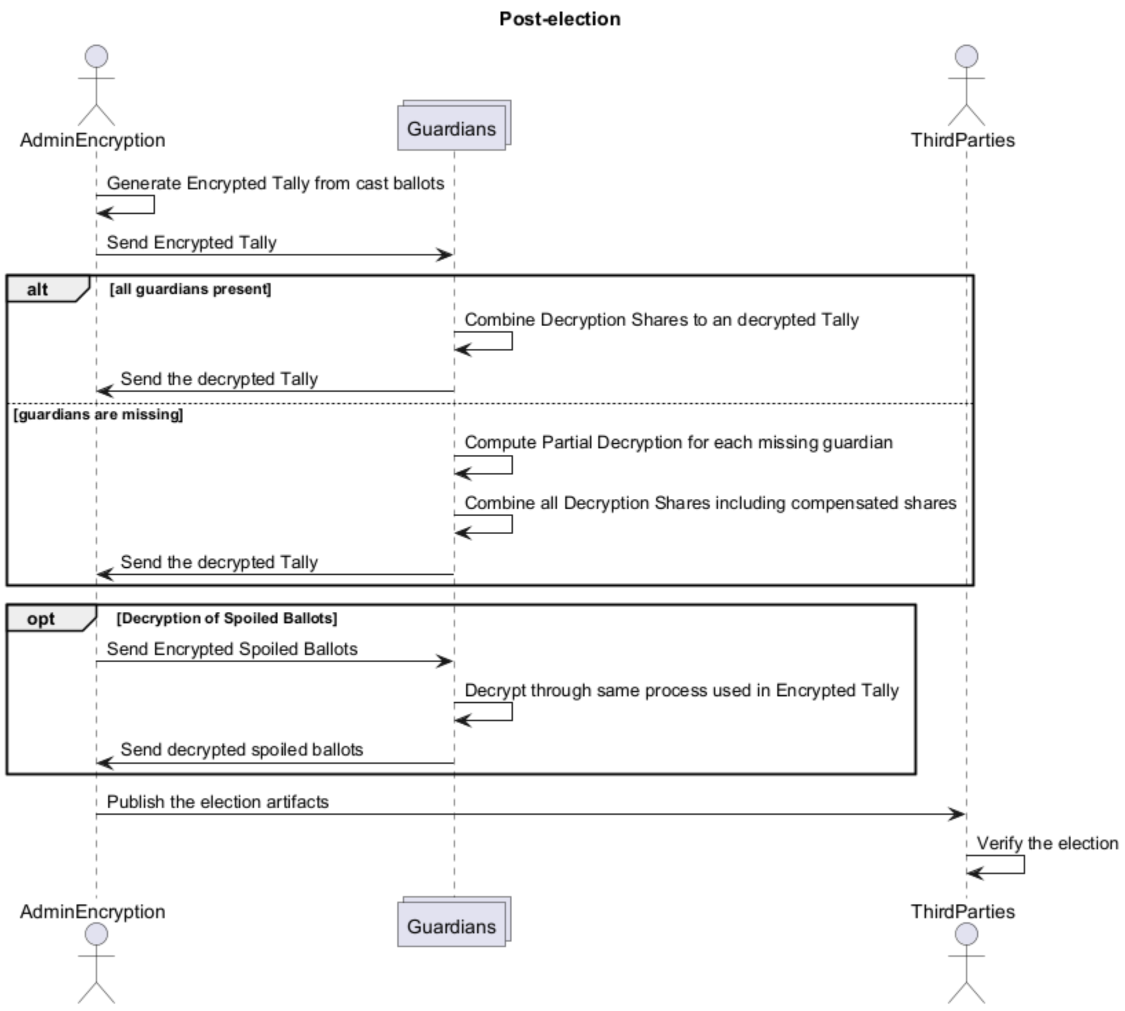

Post-election
AdminEncryption
Guardians
ThirdParties
Generate Encrypted Tally from cast ballots
Send Encrypted Tally
alt
[all guardians present]
Combine Decryption Shares to an decrypted Tally
Send the decrypted Tally
[guardians are missing]
Compute Partial Decryption for each missing guardian
Combine all Decryption Shares including compensated shares
Send the decrypted Tally
opt
[Decryption of Spoiled Ballots]
Send Encrypted Spoiled Ballots
Decrypt through same process used in Encrypted Tally
Send decrypted spoiled ballots
Publish the election artifacts
Verify the election
AdminEncryption
Guardians
ThirdParties

Figure 3.5.: Communication Sequence in the Post-election phase

decryption shares including the reconstructed shares are combined into a decrypted tally. The decrypted tally is then send back to the administrator. Optionally the spoiled ballots can be decrypted through the same mechanisms as the encrypted tally. At the end of the election the administratir publishes all election artifacts to the public for scrutiny.

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. **protocols**.

3.3.1. 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 **protocols**. For instance, the ESP32 microcontroller supports both Wi-Fi and Bluetooth data link layer protocols **esp-prog**. The Bluetooth system on the ESP32 can be further divided into Classic Bluetooth and BLE **esp-prog esp-faq**. Both Wi-Fi and Bluetooth can operate simultaneously, but this requires time-sharing control **esp-faq**.

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) **esp-faq**. 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 **esp-faq esp-prog**.

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 **protocols**.

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 **esp-prog**.

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 **protocols**

3.3.2. 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 **protocols**.

The application layer serves as an abstraction layer **protocols**. 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) **esp-prog**
- Message Queueing Telemetry Transport (MQTT) **esp-prog**
- Modbus (primarily used in industrial IoT environments)**protocols esp-prog**
- ESP-NOW (a proprietary protocol for ESP32 devices) **esp-progesp-prog**

Each protocol offers various features that differ in terms of reliability, quality of service, performance, functionality, and scalability, among other factors **protocols**.

3.3.3. 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 **eg-paper**. 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 **eg-spec**. 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 **esp-faq**. 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 **esp-faq protocols**.

3.3.4. 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. **protocols**. An IoT system may require that messages be delivered in a reliable manner where all clients acknowledge the receipt of these messages **protocols**. MQTT uses three levels of message transmission reliability, each representing a different level of QoS **serialisation**:

- **QoS 0 (most once):** Messages arrives at the receiver either once or not at all **serialisation**
- **QoS 1 (least once):** Ensures that a message arrives at the receiver at least once **serialisation**
- **QoS 2 (exactly once):** Ensures that a message arrives at the receiver exactly once without duplication **serialisation**

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. **protocols**.

3.3.5. 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 **protocols**.

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. **protocols serialisation**.

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 **protocols**. 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 **protocols**.

3.3.6. 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 **serialisation**. The design specification of ElectionGuard does not specify serialization methods or data structures. However, every implementation of ElectionGuard should be compatible with other implementations **eg-paper**. 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 **protocols**. 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 **protobuffer**. Binary formats are typically preferred as they provide smaller message sizes compared to text-based formats like JSON **serialisation**. 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 **serialisation-comparison**. 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 deserialization 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 **serialisation-comparison**.

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 A.2.5. 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 **protobuffer**. 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 A.3.4 and A.3.4. An example of serialising ElectionPartialKeyPair which is the backup shared with other guardians is seen in Appendix A.3.1 an example of deserialisation is seen in Appendix A.3.2.

3.4. 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. **protocols**

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**protocols**

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 **protocols**

lack of encryption; can use TLS/SSL for security and encryption, however, extra connection overhead **protocols**

The case can become worse if the broker's resources are maximized and hence a broker can be a Single Point of Failure (SPoF) **protocols**.

4. Conclusions

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

List of Figures

List of Tables

A. Appendix

A.1. wolfSSL

A.1.1. User Settings

```
#define NO_SESSION_CACHE

/* Small Stack uses more heap. */
#define WOLFSSL_SMALL_STACK
```

```
#define RSA_LOW_MEM

// * FP_MAX_BITS defaults to 4096 bits. This allows only
    multiplication of up to 2048x2048 bits.
#define FP_MAX_BITS 6144
```

```
/* adjust wait-timeout count if you see timeout in RSA HW
    acceleration */
#define ESP_RSA_TIMEOUT_CNT    0x349F00

/* hash limit for test.c */
#define HASH_SIZE_LIMIT

/* USE_FAST_MATH is default */
#define USE_FAST_MATH

/*****      Use SP_MATH      *****/
#undef USE_FAST_MATH            //+-
#define SP_MATH                 //+-
#define WOLFSSL_SP_MATH_ALL   //+-

#define WOLFSSL_SP_SMALL //++
#define WOLFSSL_SP_LOW_MEM //++
#define WOLFSSL_HAVE_SP_RSA //++
#define WOLFSSL_SP_4096 //++
/* #define WOLFSSL_SP_RISCV32     */

/***** Use Integer Heap Math *****/
/* #undef USE_FAST_MATH          */
/* #define USE_INTEGER_HEAP_MATH */
```

```

#define WOLFSSL_SMALL_STACK
```

A.2. Model Component

A.2.1. Cryptographic Constants

```
#include "constants.h"

const unsigned char p_3072[] = {
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0x93, 0xC4, 0x67, 0xE3, 0x7D, 0xB0, 0xC7, 0xA4,
    0xD1, 0xBE, 0x3F, 0x81, 0x01, 0x52, 0xCB, 0x56,
    0xA1, 0xCE, 0xCC, 0x3A, 0xF6, 0x5C, 0xC0, 0x19,
    0x0C, 0x03, 0xDF, 0x34, 0x70, 0x9A, 0xFF, 0xBD,
    0x8E, 0x4B, 0x59, 0xFA, 0x03, 0xA9, 0xF0, 0xEE,
    0xD0, 0x64, 0x9C, 0xCB, 0x62, 0x10, 0x57, 0xD1,
    0x10, 0x56, 0xAE, 0x91, 0x32, 0x13, 0x5A, 0x08,
    0xE4, 0x3B, 0x46, 0x73, 0xD7, 0x4B, 0xAF, 0xEA,
    0x58, 0xDE, 0xB8, 0x78, 0xCC, 0x86, 0xD7, 0x33,
    0xDB, 0xE7, 0xBF, 0x38, 0x15, 0x4B, 0x36, 0xCF,
    0x8A, 0x96, 0xD1, 0x56, 0x78, 0x99, 0xAA, 0xAE,
    0x0C, 0x09, 0xD4, 0xC8, 0xB6, 0xB7, 0xB8, 0x6F,
    0xD2, 0xA1, 0xEA, 0x1D, 0xE6, 0x2F, 0xF8, 0x64,
    0x3E, 0xC7, 0xC2, 0x71, 0x82, 0x79, 0x77, 0x22,
    0x5E, 0x6A, 0xC2, 0xF0, 0xBD, 0x61, 0xC7, 0x46,
    0x96, 0x15, 0x42, 0xA3, 0xCE, 0x3B, 0xEA, 0x5D,
    0xB5, 0x4F, 0xE7, 0x0E, 0x63, 0xE6, 0xD0, 0x9F,
    0x8F, 0xC2, 0x86, 0x58, 0xE8, 0x05, 0x67, 0xA4,
    0x7C, 0xFD, 0xE6, 0x0E, 0xE7, 0x41, 0xE5, 0xD8,
    0x5A, 0x7B, 0xD4, 0x69, 0x31, 0xCE, 0xD8, 0x22,
    0x03, 0x65, 0x59, 0x49, 0x64, 0xB8, 0x39, 0x89,
    0x6F, 0xCA, 0xAB, 0xCC, 0xC9, 0xB3, 0x19, 0x59,
    0xC0, 0x83, 0xF2, 0x2A, 0xD3, 0xEE, 0x59, 0x1C,
    0x32, 0xFA, 0xB2, 0xC7, 0x44, 0x8F, 0x2A, 0x05,
    0x7D, 0xB2, 0xDB, 0x49, 0xEE, 0x52, 0xE0, 0x18,
    0x27, 0x41, 0xE5, 0x38, 0x65, 0xF0, 0x04, 0xCC,
    0x8E, 0x70, 0x4B, 0x7C, 0x5C, 0x40, 0xBF, 0x30,
    0x4C, 0x4D, 0x8C, 0x4F, 0x13, 0xED, 0xF6, 0x04,
    0x7C, 0x55, 0x53, 0x02, 0xD2, 0x23, 0x8D, 0x8C,
    0xE1, 0x1D, 0xF2, 0x42, 0x4F, 0x1B, 0x66, 0xC2,
    0xC5, 0xD2, 0x38, 0xD0, 0x74, 0x4D, 0xB6, 0x79,
    0xAF, 0x28, 0x90, 0x48, 0x70, 0x31, 0xF9, 0xC0,
    0xAE, 0xA1, 0xC4, 0xBB, 0x6F, 0xE9, 0x55, 0x4E,
    0xE5, 0x28, 0xFD, 0xF1, 0xB0, 0x5E, 0x5B, 0x25,
    0x62, 0x23, 0xB2, 0xF0, 0x92, 0x15, 0xF3, 0x71,
    0x9F, 0x9C, 0x7C, 0xCC, 0x69, 0xDE, 0xD4, 0xE5,
    0x30, 0xA6, 0xEC, 0x94, 0x0C, 0x45, 0x31, 0x4D,
    0x16, 0xD3, 0xD8, 0x64, 0xB4, 0xA8, 0x93, 0x4F,
    0x8B, 0x87, 0xC5, 0x2A, 0xFA, 0x09, 0x61, 0xA0,
    0xA6, 0xC5, 0xEE, 0x4A, 0x35, 0x37, 0x77, 0x73,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
};


const unsigned char g_3072[] = {
    0xAF, 0x8D, 0xC2, 0x05, 0x79, 0x63, 0xC6, 0xC3,
    0x64, 0x11, 0x9C, 0x01, 0x4A, 0x27, 0x68, 0x6B,
    0xA7, 0x80, 0x57, 0x67, 0x48, 0xB7, 0x2F, 0x67,
    0x0C, 0x4A, 0x5D, 0x4C, 0x3F, 0xAC, 0x1E, 0x22,
    0x8B, 0x84, 0xFB, 0xA8, 0x8C, 0x4E, 0xAF, 0x94,
    0xDF, 0x98, 0x75, 0x5C, 0x6C, 0x73, 0x61, 0x1B,
    0xB5, 0x4A, 0x14, 0xA6, 0xE2, 0x32, 0xD2, 0x38,
    0xC9, 0x17, 0xDA, 0x76, 0xD8, 0xA6, 0x2B, 0x70,
    0x83, 0x7A, 0x15, 0xEE, 0xC1, 0x11, 0x0C, 0x11,
    0x25, 0x61, 0xAB, 0x0E, 0xAE, 0x9E, 0x11, 0xDD,
    0xCE, 0xC6, 0x1F, 0x2B, 0xBD, 0x54, 0xBB, 0x76,
    0x2F, 0xC9, 0x03, 0x49, 0x4E, 0xF2, 0x1F, 0x0F,
    0x33, 0x8F, 0xE2, 0x65, 0x82, 0x45, 0x3C, 0xE3,
    0xFF, 0x02, 0xC5, 0x3A, 0x77, 0x29, 0x61, 0x26,
    0xE5, 0x9E, 0x19, 0x80, 0xCD, 0x49, 0xA5, 0x67,
    0x26, 0xA4, 0x0C, 0xFD, 0xEF, 0x93, 0xA1, 0x81,
    0x41, 0xCF, 0x83, 0x44, 0x2D, 0x0F, 0xDC, 0xDF,
    0x9F, 0x13, 0x51, 0xB2, 0xD0, 0xCF, 0x81, 0x4C,
    0xE9, 0xC7, 0x96, 0x40, 0x2D, 0xC2, 0x21, 0x81,
    0x32, 0xD2, 0x83, 0x60, 0x5B, 0xDD, 0x15, 0x46,
    0x8E, 0xAB, 0xA4, 0xB6, 0xF7, 0x8E, 0x4D, 0xE3,
    0xDE, 0x07, 0x66, 0xFA, 0x99, 0x15, 0xED, 0x28,
    0xE0, 0x0D, 0x90, 0x75, 0x7F, 0x49, 0x49, 0x86,
    0x09, 0x24, 0x77, 0xC9, 0x0C, 0x5F, 0xC3, 0x05,
    0xA5, 0x68, 0x29, 0x08, 0x8D, 0x99, 0x6D, 0x22,
    0x7D, 0x2F, 0x01, 0x8C, 0x1A, 0x16, 0x37, 0x7B,
    0x00, 0x14, 0xA8, 0x18, 0x3F, 0x59, 0xCF, 0x88,
    0x71, 0xC4, 0x65, 0x91, 0x32, 0xBD, 0xDB, 0xA7,
    0x9E, 0x86, 0x9A, 0xE8, 0xF6, 0x5C, 0x93, 0x60,
    0x8D, 0x17, 0x9A, 0x07, 0xD7, 0xD9, 0x94, 0xE0,
    0x58, 0xE5, 0xF5, 0x1B, 0x47, 0xC7, 0x20, 0x9A,
    0x25, 0x86, 0x4D, 0xA9, 0xF1, 0x37, 0x7C, 0x16,
    0xB1, 0xC0, 0x9C, 0x85, 0xB6, 0x6C, 0xC3, 0xD5,
    0x27, 0xFA, 0xB3, 0xF6, 0xB2, 0xDF, 0x6D, 0x6B,
    0xEA, 0x15, 0x20, 0x62, 0x98, 0xBA, 0xC3, 0xE2,
    0x93, 0xF1, 0x0E, 0x2E, 0x9B, 0x78, 0x0E, 0xCE,
    0x03, 0x3A, 0x47, 0xCF, 0xC4, 0x51, 0x22, 0x15,
    0x22, 0xBB, 0x70, 0x9E, 0x1B, 0x94, 0xD8, 0xEA,
    0x74, 0x87, 0x24, 0x21, 0x85, 0xD8, 0xF8, 0xFB,
    0x01, 0x3E, 0x9E, 0x10, 0x73, 0x95, 0xD5, 0x3E,
    0x22, 0xC5, 0x55, 0x02, 0xFC, 0x1E, 0x4A, 0x91,
    0x57, 0x66, 0xF3, 0xC3, 0xB4, 0x63, 0xA3, 0xEE,
    0x4C, 0xB6, 0x82, 0x92, 0x6A, 0x0C, 0x4F, 0x87,
    0xCD, 0x86, 0x18, 0x1A, 0xBC, 0x6F, 0xB9, 0x02,
    0xBD, 0x83, 0x31, 0xDE, 0x18, 0xF5, 0x98, 0x20,
    0xC5, 0xD9, 0x67, 0xD7, 0x84, 0xB1, 0xC0, 0x6E,
    0x5A, 0x94, 0xF3, 0x1E, 0xF8, 0x61, 0x1B, 0x54,
    0x5D, 0x2F, 0x1E, 0x18, 0x4C, 0xEA, 0xC3, 0x12
};

const unsigned char q_256[] = {
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x43
};
```

A.2.2. RNG

```
/**
 * @brief Compute Large number Modular Exponetiation with known G (
    generator also known as base) and P (large prime also known as
    modulus). Y = (G ^ X) mod P
 * @param seckey: Exponent
 * @param pubkey: Result
 * @return 0 on success, -1 on failure
 */
static int g_pow_p(sp_int *seckey, sp_int *pubkey) {
    int ret;
    DECL_MP_INT_SIZE(large_prime, 3072);
    NEW_MP_INT_SIZE(large_prime, 3072, NULL, DYNAMIC_TYPE_BIGINT);
    INIT_MP_INT_SIZE(large_prime, 3072);
    sp_read_unsigned_bin(large_prime, p_3072, sizeof(p_3072));

    DECL_MP_INT_SIZE(generator, 3072);
    NEW_MP_INT_SIZE(generator, 3072, NULL, DYNAMIC_TYPE_BIGINT);
    INIT_MP_INT_SIZE(generator, 3072);
    sp_read_unsigned_bin(generator, g_3072, sizeof(g_3072));

    ret = exptmod(generator,seckey,large_prime,pubkey);
    if(ret != 0) {
        ESP_LOGE("G_POW_P", "Failed to compute g^x mod p");
        ESP_LOGE("G_POW_P", "Error code: %d", ret);
    }
    FREE_MP_INT_SIZE(large_prime, NULL, DYNAMIC_TYPE_BIGINT);
    FREE_MP_INT_SIZE(generator, NULL, DYNAMIC_TYPE_BIGINT);
    return 0;
}
```

```
/**
```

A.2.3. Hash Functions

```
 * @param result: The random number
 * @return 0 on success, -1 on failure
 */
int rand_q(sp_int *result) {
    DECL_MP_INT_SIZE(small_prime, 256);
    NEW_MP_INT_SIZE(small_prime, 256, NULL, DYNAMIC_TYPE_BIGINT);
    INIT_MP_INT_SIZE(small_prime, 256);
    sp_read_unsigned_bin(small_prime, q_256, sizeof(q_256));

    int sz = 32;
    unsigned char *block = (unsigned char *)malloc(sz * sizeof(
       unsigned char));
    if (block == NULL) {
        ESP_LOGE("RAND_Q", "Failed to allocate memory for block");
        return -1; // Memory allocation failed
    }

    esp_fill_random(block, sz);
    sp_read_unsigned_bin(result, block, sz);
    sp_mod(result, small_prime, result);

    // Clear
    memset(block, 0, sz);
    free(block);
    FREE_MP_INT_SIZE(small_prime, NULL, DYNAMIC_TYPE_BIGINT);
    return 0;
}


/**
 * @brief Given two mp_ints, calculate their cryptographic hash
    using SHA256.
 * Possible collision. In the python implementation a delimiter (|)
     is used
 * @param a: First element
 * @param b: Second element
 * @param result: The result of the hash
 * @return 0 on success, -1 on failure
 */
int hash(sp_int *a, sp_int *b, sp_int *result) {
    int ret;
    word32 a_size = sp_unsigned_bin_size(a);
    word32 b_size = sp_unsigned_bin_size(b);
    word32 tmp_size = a_size + b_size;

    NEW_MP_INT_SIZE(modifier, 64, NULL, DYNAMIC_TYPE_BIGINT);
    INIT_MP_INT_SIZE(modifier, 64);
    sp_read_unsigned_bin(modifier, exponent_modifier, sizeof(
        exponent_modifier));

    for(size_t i = 0; i < polynomial->num_coefficients; i++) {
        //Not accelerated Operator lenght to small
        sp_set_int(exponent_i, i);
        exptmod(modifier, exponent_i, large_prime, exponent);
        exptmod(polynomial->coefficients[i].commitment, exponent,
            large_prime, factor);
        mulmod(commitment_output, factor, large_prime,
            commitment_output);
    }
    g_pow_p(coordinate, value_output);
    if(sp_cmp(value_output, commitment_output) == MP_EQ) {
        return 1;
    } else
    {
        return 0;
    }
    FREE_MP_INT_SIZE(exponent, NULL, DYNAMIC_TYPE_BIGINT);
    FREE_MP_INT_SIZE(exponent_i, NULL, DYNAMIC_TYPE_BIGINT);
    FREE_MP_INT_SIZE(modifier, NULL, DYNAMIC_TYPE_BIGINT);
    FREE_MP_INT_SIZE(large_prime, NULL, DYNAMIC_TYPE_BIGINT);
    FREE_MP_INT_SIZE(factor, NULL, DYNAMIC_TYPE_BIGINT);
    FREE_MP_INT_SIZE(value_output, NULL, DYNAMIC_TYPE_BIGINT);
    FREE_MP_INT_SIZE(commitment_output, NULL, DYNAMIC_TYPE_BIGINT);
}
```

A.2.4. RSA Accelerator

```
/**
 * @brief Compute Large number Modular Exponetiation with hardware
    Y = (G ^ X) mod P. If the inputs are to small switches to
    unaccelerated version
 * @param g: Base
 * @param x: Exponent
 * @param p: Modulus
 * @param y: Result
 * @return 0 on success, -1 on failure
 */
static int exptmod(sp_int *g, sp_int *x, sp_int *p, sp_int *y) {
    int ret = sp_exptmod(g, x, p, y);
    /*

    ret = esp_mp_exptmod(g,x,p,y);
    if(ret == INPUT_CASE_ERROR) {
        ESP_LOGI("MODEXPT", "Input are too small switching to
            software exptmod");
        ret = sp_exptmod(g,x,p,y);
        if (ret != MP_OKAY) {
            ESP_LOGE("MULMOD", "Error code: %d", ret);
            return -1;
        }
    }
```

```
    ret = sp_to_unsigned_bin_at_pos(a_size,b, tmp);

    byte *result_byte = (byte *)malloc(WC_SHA256_DIGEST_SIZE);
    if (result_byte == NULL) {
        ESP_LOGE("HASH_ELEMS", "Failed to allocate memory for
            result_byte");
        return -1; // Return an error code
    }

    // Conveniencefunction. Handles Initialisation, Update and
       Finalisation
    if ((ret = wc_Sha256Hash(tmp, tmp_size, result_byte)) != 0) {
        WOLFSSL_MSG("Hashing Failed");
        return ret;
    }

    ret = sp_read_unsigned_bin(result, result_byte,
       WC_SHA256_DIGEST_SIZE);
    free(tmp);
    free(result_byte);
    return ret;
}


/**
 * @brief Computes a single coordinate value of the election
     polynomial used for sharing
 * @param exponent_modifier: Unique modifier (usually sequence
     order) for exponent [0, Q]
 * @param polynomial: Election polynomial
 * @param coordinate: The computed coordinate
 * @return 0 on success, -1 on failure
 */
int compute_polynomial_coordinate(uint8_t* exponent_modifier,
   ElectionPolynomial* polynomial, sp_int *coordinate) {
    DECL_MP_INT_SIZE(modifier, 48);
    NEW_MP_INT_SIZE(modifier, 48, NULL, DYNAMIC_TYPE_BIGINT);
    INIT_MP_INT_SIZE(modifier, 48);
    sp_read_unsigned_bin(modifier, exponent_modifier, sizeof(
       exponent_modifier));

    DECL_MP_INT_SIZE(exponent_i, 48);
    NEW_MP_INT_SIZE(exponent_i, 48, NULL, DYNAMIC_TYPE_BIGINT);
    INIT_MP_INT_SIZE(exponent_i, 48);
```

A.2.5. Data Structures

```
typedef struct {
    sp_int* pubkey;
    sp_int* commitment;
    sp_int* challenge;
    sp_int* response;
} SchnorrProof;

typedef struct {
    sp_int* value;
    sp_int* commitment;
    SchnorrProof proof;
} Coefficient;

typedef struct {
    int num_coefficients;
    Coefficient* coefficients;
} ElectionPolynomial;

// contains also private key. Be careful when sending!
typedef struct {
    uint8_t guardian_id[6];
    sp_int* public_key;
    sp_int* private_key;
    ElectionPolynomial polynomial;
} ElectionKeyPair;

 typedef struct {
    sp_int* pad;
    sp_int* data;
    sp_int* mac;
} HashedElGamalCiphertext;

 typedef struct {
    uint8_t sender[6];
    uint8_t receiver[6];
    HashedElGamalCiphertext encrypted_coordinate;
 } ElectionPartialKeyPairBackup;


 typedef struct {
    uint8_t sender[6];
    uint8_t receiver[6];
    uint8_t verifier[6];
    bool verified;
 } ElectionPartialKeyVerification;

 typedef struct {
    sp_int* joint_key;
    sp_int* commitment_hash;
 } ElectionJointKey;

typedef struct {
    char* object_id;
```

```
        sp_int* ciphertext_pad;
        sp_int* ciphertext_data;
} CiphertextTallySelection;

typedef struct {
    char* object_id;
    int sequence_order;
    sp_int* description_hash;
    int num_selections;
    CiphertextTallySelection* selections;
} CiphertextTallyContest;

typedef struct{
    char* object_id;
    sp_int* base_hash;
    int num_contest;
    CiphertextTallyContest* contests;
} CiphertextTally;

typedef struct {
    sp_int* pad;
    sp_int* data;
    sp_int* challenge;
    sp_int* response;
} ChaumPedersenProof;

typedef struct {
    char* object_id;
    uint8_t guardian_id[6];
    sp_int* decryption;
    ChaumPedersenProof proof;
} CiphertextDecryptionSelection;

typedef struct{
    char* object_id;
    uint8_t guardian_id[6];
    sp_int *public_key;
    sp_int* description_hash;
    int num_selections;
    CiphertextDecryptionSelection* selections;
} CiphertextDecryptionContest;

typedef struct {
    char* object_id;
    uint8_t guardian_id[6];
    sp_int* public_key;
    int num_contest;
    CiphertextDecryptionContest* contests;
} DecryptionShare;
```

A.3. Adapter Component

A.3.1. Serialization

```c
uint8_t* serialize_election_partial_key_backup(
   ElectionPartialKeyPairBackup* backup, unsigned* len) {
    ElectionPartialKeyPairBackupProto proto =
       ELECTION_PARTIAL_KEY_PAIR_BACKUP_PROTO__INIT;
    proto.sender.len = sizeof(backup->sender);
    proto.sender.data = backup->sender;
    proto.receiver.len = sizeof(backup->receiver);
    proto.receiver.data = backup->receiver;

    HashedElGamalCiphertextProto hash =
       HASHED_EL_GAMAL_CIPHERTEXT_PROTO__INIT;
    hash.pad.len = sp_unsigned_bin_size(backup->
       encrypted_coordinate.pad);
    hash.pad.data = (uint8_t*)malloc(hash.pad.len);
    sp_to_unsigned_bin(backup->encrypted_coordinate.pad, hash.pad.
       data);
    hash.data.len = sp_unsigned_bin_size(backup->
       encrypted_coordinate.data);
    hash.data.data = (uint8_t*)malloc(hash.data.len);
    sp_to_unsigned_bin(backup->encrypted_coordinate.data, hash.data
       .data);
    hash.mac.len = sp_unsigned_bin_size(backup->
       encrypted_coordinate.mac);
    hash.mac.data = (uint8_t*)malloc(hash.mac.len);
    sp_to_unsigned_bin(backup->encrypted_coordinate.mac, hash.mac.
       data);
    proto.encrypted_coordinate = &hash;

    *len = election_partial_key_pair_backup_proto__get_packed_size
       (&proto);
    uint8_t* buffer = (uint8_t *)calloc(*len, sizeof(char));
    election_partial_key_pair_backup_proto__pack(&proto, buffer);
    return buffer;
}
```

A.3.2. Deserialization

```c
int deserialize_ciphertext_tally(uint8_t *buffer, unsigned len,
   CiphertextTally* ciphertally) {
    CiphertextTallyProto* tally = ciphertext_tally_proto__unpack(
       NULL, len, buffer);
    if (tally == NULL) {
        fprintf(stderr, "Error unpacking CiphertextTallySelections\
           n");
        return -1;
    }
    ciphertally->object_id = strdup(tally->object_id);
    ciphertally->base_hash = NULL;
    ciphertally->base_hash = (sp_int*)XMALLOC(MP_INT_SIZEOF(
        MP_BITS_CNT(256)), NULL, DYNAMIC_TYPE_BIGINT);
    sp_read_unsigned_bin(ciphertally->base_hash, tally->base_hash.
        data, tally->base_hash.len);

    ciphertally->num_contest = tally->num_contest;
    ciphertally->contests = (CiphertextTallyContest*)malloc(sizeof(
        CiphertextTallyContest) * ciphertally->num_contest);
    for(int i = 0; i < ciphertally->num_contest; i++) {
        ciphertally->contests[i].object_id = strdup(tally->contests
            [i]->object_id);
        ciphertally->contests[i].sequence_order = tally->contests[i
            ]->sequence_order;
        ciphertally->contests[i].description_hash = NULL;
        ciphertally->contests[i].description_hash = (sp_int*)
            XMALLOC(MP_INT_SIZEOF(MP_BITS_CNT(256)), NULL,
            DYNAMIC_TYPE_BIGINT);
        sp_read_unsigned_bin(ciphertally->contests[i].
            description_hash, tally->contests[i]->description_hash.
            data, tally->contests[i]->description_hash.len);
        ciphertally->contests[i].num_selections = tally->contests[i
            ]->num_selections;

        ciphertally->contests[i].selections = (
            CiphertextTallySelection*)malloc(sizeof(
            CiphertextTallySelection) * ciphertally->contests[i].
            num_selections);
        for(int j = 0; j < ciphertally->contests[i].num_selections;
             j++) {
            ciphertally->contests[i].selections[j].object_id =
                strdup(tally->contests[i]->selections[j]->object_id)
                ;
            ciphertally->contests[i].selections[j].ciphertext_pad =
                 NULL;
            ciphertally->contests[i].selections[j].ciphertext_pad =
                 (sp_int*)XMALLOC(MP_INT_SIZEOF(MP_BITS_CNT(3072)),
                NULL, DYNAMIC_TYPE_BIGINT);
            sp_read_unsigned_bin(ciphertally->contests[i].
                selections[j].ciphertext_pad, tally->contests[i]->
                selections[j]->ciphertext_pad.data, tally->contests[
                i]->selections[j]->ciphertext_pad.len);
            ciphertally->contests[i].selections[j].ciphertext_data
                = NULL;
            ciphertally->contests[i].selections[j].ciphertext_data
                = (sp_int*)XMALLOC(MP_INT_SIZEOF(MP_BITS_CNT(3072)),
                 NULL, DYNAMIC_TYPE_BIGINT);
            sp_read_unsigned_bin(ciphertally->contests[i].
                selections[j].ciphertext_data, tally->contests[i]->
                selections[j]->ciphertext_data.data, tally->contests
                [i]->selections[j]->ciphertext_data.len);
        }

    }


    ciphertext_tally_proto__free_unpacked(tally, NULL);
    return 0;
}
```

A.3.3. Proto file

```
syntax = "proto2";

message SchnorrProofProto {
    required bytes pubkey = 1;
    required bytes commitment = 2;
    required bytes challenge = 3;
    required bytes response = 4;
}

message CoefficientProto {
    required bytes value = 1;
    required bytes commitment = 2;
    required SchnorrProofProto proof = 3;
}

message ElectionPolynomialProto {
    required int32 num_coefficients = 1;
    repeated CoefficientProto coefficients = 2;
}

message ElectionKeyPairProto {
    required bytes guardian_id = 1; // fixed length of 6 bytes
    required bytes public_key = 2;
    required bytes private_key = 3;
    required ElectionPolynomialProto polynomial = 4;
}

message HashedElGamalCiphertextProto {
    required bytes pad = 1;
    required bytes data = 2;
    required bytes mac = 3;
}

message ElectionPartialKeyPairBackupProto {
    required bytes sender = 1; // fixed length of 6 bytes
    required bytes receiver = 2; // fixed length of 6 bytes
    required HashedElGamalCiphertextProto encrypted_coordinate = 3;
}
```

```
message ElectionPartialKeyVerificationProto {
    required bytes sender = 1; // fixed length of 6 bytes
    required bytes receiver = 2; // fixed length of 6 bytes
    required bytes verifier = 3; // fixed length of 6 bytes
    required bool verified = 4;
}
```

```
syntax = "proto2";


message PlaintextBallotSelectionProto {
        required bool vote = 1;
        required bool is_placeholder_selection = 2;
}


message PlaintextBallotContestProto {
        required int32 num_selections = 1;
        repeated PlaintextBallotSelectionProto ballot_selections =
           2;
}

message PlaintextBallotProto {
        required string style_id = 1;
        required int32 num_contests = 2;
        repeated PlaintextBallotContestProto contests = 3;
}


message CiphertextBallotProto {
        required string style_id = 1;
        required bytes code_seed = 2;
        required bytes code = 3;
        required bytes crypto_hash = 4;
}
```

```
syntax = "proto2";


message CiphertextTallySelectionProto {
        required string object_id = 1;
        required bytes ciphertext_pad = 2;
        required bytes ciphertext_data = 3;
}

message CiphertextTallyContestProto {
        required string object_id = 1;
        required int32 sequence_order = 2;
        required bytes description_hash = 3;
        required int32 num_selections = 4;
        repeated CiphertextTallySelectionProto selections = 5;
}
```

```
message CiphertextTallyProto {
        required string object_id = 1;
        required bytes base_hash = 2;
        required int32 num_contest = 3;
        repeated CiphertextTallyContestProto contests = 4;
}


message CiphertextDecryptionSelectionProto {
        required string object_id = 1;
        required bytes guardian_id  = 2;
        required bytes decryption = 3;
        required bytes proof_pad = 4;
        required bytes proof_data = 5;
        required bytes proof_challenge = 6;
        required bytes proof_response = 7;
}

message CiphertextDecryptionContestProto {
        required string object_id = 1;
        required bytes guardian_id = 2;
        required bytes description_hash = 3;
        required int32 num_selections = 4;
        repeated CiphertextDecryptionSelectionProto selections = 5;
}

message DecryptionShareProto {
        required string object_id = 1;
        required bytes guardian_id = 2;
        required bytes public_key = 3;
        required int32 num_contests = 4;
        repeated CiphertextDecryptionContestProto contests = 5;
}
```

A.3.4. Generated Code

```
/* Generated by the protocol buffer compiler.  DO NOT EDIT! */
/* Generated from: buff.proto */

/* Do not generate deprecated warnings for self */
#ifndef PROTOBUF_C__NO_DEPRECATED
#define PROTOBUF_C__NO_DEPRECATED
#endif

#include "buff.pb-c.h"
void   schnorr_proof_proto__init
                     (SchnorrProofProto         *message)
{
  static const SchnorrProofProto init_value =
      SCHNORR_PROOF_PROTO__INIT;
  *message = init_value;
}
size_t schnorr_proof_proto__get_packed_size
                     (const SchnorrProofProto *message)
{
  assert(message->base.descriptor == &
     schnorr_proof_proto__descriptor);
  return protobuf_c_message_get_packed_size ((const
     ProtobufCMessage*)(message));
}
size_t schnorr_proof_proto__pack
                     (const SchnorrProofProto *message,
                      uint8_t       *out)
{
  assert(message->base.descriptor == &
     schnorr_proof_proto__descriptor);
  return protobuf_c_message_pack ((const ProtobufCMessage*)message,
      out);
}
size_t schnorr_proof_proto__pack_to_buffer
                     (const SchnorrProofProto *message,
                      ProtobufCBuffer *buffer)
{
  assert(message->base.descriptor == &
     schnorr_proof_proto__descriptor);
  return protobuf_c_message_pack_to_buffer ((const ProtobufCMessage
     *)message, buffer);
}
SchnorrProofProto *
       schnorr_proof_proto__unpack
                     (ProtobufCAllocator  *allocator,
                      size_t               len,
                      const uint8_t       *data)
{
  return (SchnorrProofProto *)
     protobuf_c_message_unpack (&schnorr_proof_proto__descriptor,
                                allocator, len, data);
}
void   schnorr_proof_proto__free_unpacked
                     (SchnorrProofProto *message,
                      ProtobufCAllocator *allocator)
{
  if(!message)
    return;
  assert(message->base.descriptor == &
     schnorr_proof_proto__descriptor);
  protobuf_c_message_free_unpacked ((ProtobufCMessage*)message,
     allocator);
}
```

```
# -*- coding: utf-8 -*-
# Generated by the protocol buffer compiler.  DO NOT EDIT!
# source: ballot.proto
"""Generated protocol buffer code."""
```

```
from google.protobuf.internal import builder as _builder
from google.protobuf import descriptor as _descriptor
from google.protobuf import descriptor_pool as _descriptor_pool
from google.protobuf import symbol_database as _symbol_database
# @@protoc_insertion_point(imports)

_sym_db = _symbol_database.Default()




DESCRIPTOR = _descriptor_pool.Default().AddSerializedFile(b'\n\x0c\
   x62\x61llot.proto\"O\n\x1dPlaintextBallotSelectionProto\x12\x0c\
   n\x04vote\x18\x01 \x02(\x08\x12 \n\x18is_placeholder_selection\
   x18\x02 \x02(\x08\"p\n\x1bPlaintextBallotContestProto\x12\x16\n\
   x0enum_selections\x18\x01 \x02(\x05\x12\x39\n\x11\x62\
   x61llot_selections\x18\x02 \x03(\x0b\x32\x1e.
   PlaintextBallotSelectionProto\"n\n\x14PlaintextBallotProto\x12\
   x10\n\x08style_id\x18\x01 \x02(\t\x12\x14\n\x0cnum_contests\x18\
   x02 \x02(\x05\x12.\n\x08\x63ontests\x18\x03 \x03(\x0b\x32\x1c.
   PlaintextBallotContestProto\"_\n\x15\x43iphertextBallotProto\x12
   \x10\n\x08style_id\x18\x01 \x02(\t\x12\x11\n\tcode_seed\x18\x02
   \x02(\x0c\x12\x0c\n\x04\x63ode\x18\x03 \x02(\x0c\x12\x13\n\x0b\
   x63rypto_hash\x18\x04 \x02(\x0c')

_builder.BuildMessageAndEnumDescriptors(DESCRIPTOR, globals())
_builder.BuildTopDescriptorsAndMessages(DESCRIPTOR, 'ballot_pb2',
   globals())
if _descriptor._USE_C_DESCRIPTORS == False:

  DESCRIPTOR._options = None
  _PLAINTEXTBALLOTSELECTIONPROTO._serialized_start=16
  _PLAINTEXTBALLOTSELECTIONPROTO._serialized_end=95
  _PLAINTEXTBALLOTCONTESTPROTO._serialized_start=97
  _PLAINTEXTBALLOTCONTESTPROTO._serialized_end=209
  _PLAINTEXTBALLOTPROTO._serialized_start=211
  _PLAINTEXTBALLOTPROTO._serialized_end=321
  _CIPHERTEXTBALLOTPROTO._serialized_start=323
  _CIPHERTEXTBALLOTPROTO._serialized_end=418
# @@protoc_insertion_point(module_scope)
```