ESP32 Guardian Component

1

Generated by Doxygen 1.10.0

1 Guardian - ElectionGuard ESP32 Client

1.1 Overview

This project provides the source code and configuration necessary to build an ElectionGuard guardian client for the ESP32 platform. The guardian client is responsible for securely participating in the key generation ceremony and the tallying phase required by the ElectionGuard protocol. This client communicates via MQTT to a central server (Laptop).

1.2 Project Structure

The project is organized into the following directories and files:

- `**main/**`: Contains the main application entry point and performance testing code.
  - `main.c`: The main application logic that initializes the MQTT client and handles communication with the central server.
  - `test_performance.c`: A performance testing module used to evaluate the efficiency of cryptographic operations used during the ElectionGuard key generation ceremony.
- `**/components**`: Contains reusable modules that encapsulate specific functionalities.
  - `**/wolfssl**`: This component overrides the default settings of the WolfSSL third-party library.
  - `**/adapter**`: Handles all communication aspects of the guardian client.
    - Sets up an MQTT client using the configured Wi-Fi connection.
    - Manages incoming and outgoing MQTT messages.
    - Serializes and deserializes data for transmission.
  - `**/model**`: Contains the core business logic and cryptographic operations required by the ElectionGuard protocol.
    - Implements the key ceremony and the distributed decryption
    - Provides an abstraction layer for the underlying cryptographic primitives.

1.3 Functionality

The `main.c` file initializes the ESP32, connects to the configured Wi-Fi network, and starts the MQTT client using the `adapter` component. The `adapter` component then handles communication with the central server, receiving instructions and sending responses. The `model` component implements the cryptographic logic required to perform the ElectionGuard operations.

The `test_performance.c` file provides a means to benchmark the performance of the cryptographic operations used in the `model` component. This is crucial for ensuring that the guardian client can perform its tasks within a reasonable timeframe on the resource-constrained ESP32 platform.

1.4 Building and Flashing

This project is designed to be built using the ESP-IDF framework.

1. **Install ESP-IDF version 5.2:**
2. **Clone the Repository:** Clone this repository to your local machine.
3. **Configure the Project:**
   - Navigate to the project directory in your terminal.
   - Run `idf.py menuconfig` to open the ESP-IDF configuration menu.
   - **Wi-Fi Configuration:** Configure the Wi-Fi SSID and password to connect to your Access Point. This is essential for the guardian to communicate with the MQTT broker.
   - **MQTT Broker Configuration:** Set the MQTT broker address (IP address or hostname) and port. This should match the configuration of the MQTT broker running on the central server.
   - **Other Settings:** Configure any other project-specific settings as needed.
4. **Build the Project:** Run `idf.py build` to compile the project.
5. **Flash the Firmware:** Run `idf.py flash monitor` to flash the firmware to your ESP32 and monitor the serial output.

1.5 Testing and Performance Evaluation

To evaluate the performance of the cryptographic operations, you can use the `test_performance.c` module.

1. **Enable Performance Testing:** Modify `main.c` to call the functions in `test_performance.c`. You need to comment out the MQTT client initialization code to isolate the performance tests.
2. **Build and Flash:** Build and flash the firmware as described above.
3. **Monitor Serial Output:** The serial output will display the results of the performance tests, including the execution time for various cryptographic operations.

2 File Index

2.1 File List

Here is a list of all files with brief descriptions:

3 File Documentation

3.1 D:/Dokuments/IoT-election/guardian/components/adapter/adapter.c File Reference

```
#include "adapter.h"
```

Functions

- static void verify_backups (esp_mqtt_client_handle_t client)
- static void generate_backups (esp_mqtt_client_handle_t client)
- static void publish_public_key (esp_mqtt_client_handle_t client, const char ∗data, int data_len)
- static void handle_pubkeys (esp_mqtt_client_handle_t client, const char ∗data, int data_len)
- static void handle_backups (esp_mqtt_client_handle_t client, const char ∗data, int data_len)
- static void handle_challenge (esp_mqtt_client_handle_t client, const char ∗data, int data_len)
- static void handle_ciphertext_tally (esp_mqtt_client_handle_t client, const char ∗data, int data_len)
- static void add_key_pair (ElectionKeyPair ∗key_pair)
- static ElectionKeyPair ∗ find_key_pair (uint8_t ∗guardian_id)
- static void delete_key_pair (uint8_t ∗guardian_id)
- static void add_backup (ElectionPartialKeyPairBackup ∗backup)
- static ElectionPartialKeyPairBackup ∗ find_backup (uint8_t ∗guardian_id)
- static void delete_backup (uint8_t ∗guardian_id)
- static void log_error_if_nonzero (const char ∗message, int error_code)
- void mqtt_event_handler (void ∗handler_args, esp_event_base_t base, int32_t event_id, void ∗event_data)
- void mqtt_app_start (void)

    Start the MQTT client.

Variables

- static const char ∗ TAG = "Adapter"
- uint8_t mac [6] = {0}
- int pubkey_count = 0
- int backup_count = 0
- int max_guardians = 0
- int quorum
- ElectionKeyPair guardian
- ElectionKeyPair ∗ pubkey_map
- ElectionPartialKeyPairBackup ∗ backup_map

3.1.1 Function Documentation

add_backup()

```
static void add_backup (
            ElectionPartialKeyPairBackup * backup )  [static]
```

add_key_pair()

```
static void add_key_pair (
            ElectionKeyPair * key_pair )  [static]
```

delete_backup()

```
static void delete_backup (
            uint8_t * guardian_id )  [static]
```

delete_key_pair()

```
static void delete_key_pair (
            uint8_t * guardian_id )  [static]
```

find_backup()

```
static ElectionPartialKeyPairBackup * find_backup (
            uint8_t * guardian_id )  [static]
```

find_key_pair()

```
static ElectionKeyPair * find_key_pair (
            uint8_t * guardian_id )  [static]
```

generate_backups()

```
static void generate_backups (
            esp_mqtt_client_handle_t client )  [static]
```

handle_backups()

```
static void handle_backups (
            esp_mqtt_client_handle_t client,
            const char * data,
            int data_len )  [static]
```

handle_challenge()

```
static void handle_challenge (
            esp_mqtt_client_handle_t client,
            const char * data,
            int data_len )  [static]
```

handle_ciphertext_tally()

```
static void handle_ciphertext_tally (
            esp_mqtt_client_handle_t client,
            const char * data,
            int data_len )  [static]
```

handle_pubkeys()

```
static void handle_pubkeys (
            esp_mqtt_client_handle_t client,
            const char * data,
            int data_len )  [static]
```

log_error_if_nonzero()

```
static void log_error_if_nonzero (
            const char * message,
            int error_code )  [static]
```

mqtt_app_start()

```
void mqtt_app_start (
            void  )
```

Start the MQTT client.

This function initializes the MQTT client, registers the event handler and starts the client.

mqtt_event_handler()

```
void mqtt_event_handler (
            void * handler_args,
            esp_event_base_t base,
            int32_t event_id,
            void * event_data )
```

publish_public_key()

```
static void publish_public_key (
            esp_mqtt_client_handle_t client,
            const char * data,
            int data_len )  [static]
```

verify_backups()

```
static void verify_backups (
            esp_mqtt_client_handle_t client )  [static]
```

3.1.2 Variable Documentation

backup_count

```
int backup_count = 0
```

backup_map

```
ElectionPartialKeyPairBackup* backup_map
```

guardian

```
ElectionKeyPair guardian
```

mac

```
uint8_t mac[6] = {0}
```

max_guardians

```
int max_guardians = 0
```

pubkey_count

```
int pubkey_count = 0
```

pubkey_map

```
ElectionKeyPair* pubkey_map
```

quorum

```
int quorum
```

TAG

```
const char* TAG = "Adapter"  [static]
```

3.2 D:/Dokuments/IoT-election/guardian/components/adapter/util/serialize.c File Reference

```
#include "serialize.h"
```

Functions

- uint8_t ∗ serialize_election_key_pair (ElectionKeyPair ∗key_pair, unsigned ∗len)

    Serialize an ElectionKeyPair struct into a byte array.
- static int deserialize_schnorr_proof (SchnorrProofProto ∗proto, SchnorrProof ∗proof)

    Deserializes a SchnorrProofProto object into a SchnorrProof struct.
- static int deserialize_election_polynomial (ElectionPolynomialProto ∗poly, ElectionPolynomial ∗polynomial)

    Deserializes an ElectionPolynomialProto object into an ElectionPolynomial struct.
- int deserialize_election_key_pair (uint8_t ∗buffer, unsigned len, ElectionKeyPair ∗key_pair)

    Deserializes a byte array into an ElectionKeyPair struct.
- uint8_t ∗ serialize_election_partial_key_verification (ElectionPartialKeyVerification ∗verification, unsigned ∗len)

    Serializes an ElectionPartialKeyVerification object into a byte array.
- int deserialize_election_partial_key_verification (uint8_t ∗buffer, unsigned len, ElectionPartialKeyVerification ∗verification)

    Deserializes a byte array into an ElectionPartialKeyVerification struct.
- uint8_t ∗ serialize_election_partial_key_backup (ElectionPartialKeyPairBackup ∗backup, unsigned ∗len)

    Serializes an ElectionPartialKeyPairBackup object into a byte array.
- int deserialize_election_partial_key_backup (uint8_t ∗buffer, unsigned len, ElectionPartialKeyPairBackup ∗backup)

    Deserializes a byte array into an ElectionPartialKeyPairBackup struct.
- int deserialize_ciphertext_tally (uint8_t ∗buffer, unsigned len, CiphertextTally ∗ciphertally)

    Deserializes a byte array into a CiphertextTally struct.
- uint8_t ∗ serialize_DecryptionShare (DecryptionShare ∗share, unsigned ∗len)

    Serializes a DecryptionShare object into a byte array.

3.2.1 Function Documentation

deserialize_ciphertext_tally()

```
int deserialize_ciphertext_tally (
            uint8_t * buffer,
            unsigned len,
            CiphertextTally * ciphertally )
```

Deserializes a byte array into a CiphertextTally struct.

This function takes a byte array and its length, and deserializes it into a CiphertextTally struct using Protocol Buffers. It unpacks the data and populates the tally object with contest and selection information.

Parameters

buffer	Pointer to the byte array containing the serialized CiphertextTally data.
len	The length of the byte array.
ciphertally	Pointer to the CiphertextTally struct to populate with the deserialized data.

Returns

0 on success, or -1 on failure.

deserialize_election_key_pair()

```
int deserialize_election_key_pair (
            uint8_t * buffer,
            unsigned len,
            ElectionKeyPair * key_pair )
```

Deserializes a byte array into an ElectionKeyPair struct.

This function takes a byte array and its length, and deserializes it into an ElectionKeyPair struct using Protocol Buffers. It unpacks the data, copies the guardian ID, deserializes the public key and the election polynomial.

Parameters

buffer	Pointer to the byte array containing the serialized ElectionKeyPair data.
len	The length of the byte array.
key_pair	Pointer to the ElectionKeyPair struct to populate with the deserialized data.

Returns

0 on success, or 1 on failure.

deserialize_election_partial_key_backup()

```
int deserialize_election_partial_key_backup (
            uint8_t * buffer,
            unsigned len,
            ElectionPartialKeyPairBackup * backup )
```

Deserializes a byte array into an ElectionPartialKeyPairBackup struct.

This function takes a byte array and its length, and deserializes it into an ElectionPartialKeyPairBackup struct using Protocol Buffers. It unpacks the data and copies the sender, receiver, and the encrypted coordinate (HashedEl↩ GamalCiphertext).

Parameters

buffer	Pointer to the byte array containing the serialized ElectionPartialKeyPairBackup data.
len	The length of the byte array.
backup	Pointer to the ElectionPartialKeyPairBackup struct to populate with the deserialized data.

Returns

0 on success, or 1 on failure.

deserialize_election_partial_key_verification()

```
int deserialize_election_partial_key_verification (
            uint8_t * buffer,
```

```
            unsigned len,
            ElectionPartialKeyVerification * verification )
```

Deserializes a byte array into an ElectionPartialKeyVerification struct.

This function takes a byte array and its length, and deserializes it into an ElectionPartialKeyVerification struct using Protocol Buffers. It unpacks the data and copies the sender, receiver, verifier, and the verification status.

Parameters

buffer	Pointer to the byte array containing the serialized ElectionPartialKeyVerification data.
len	The length of the byte array.
verification	Pointer to the ElectionPartialKeyVerification struct to populate with the deserialized data.

Returns

0 on success, or 1 on failure.

deserialize_election_polynomial()

```
static int deserialize_election_polynomial (
            ElectionPolynomialProto * poly,
            ElectionPolynomial * polynomial )  [static]
```

Deserializes an ElectionPolynomialProto object into an ElectionPolynomial struct.

This function takes an ElectionPolynomialProto object (protobuf representation) and populates an Election↩ Polynomial struct with the deserialized data. It allocates memory for the coefficients and their associated data, including Schnorr proofs.

Parameters

poly	Pointer to the ElectionPolynomialProto object to deserialize.
polynomial	Pointer to the ElectionPolynomial struct to populate with the deserialized data.

Returns

0 on success, or an error code on failure.

deserialize_schnorr_proof()

```
static int deserialize_schnorr_proof (
            SchnorrProofProto * proto,
            SchnorrProof * proof )  [static]
```

Deserializes a SchnorrProofProto object into a SchnorrProof struct.

This function takes a SchnorrProofProto object (protobuf representation) and populates a SchnorrProof struct with the deserialized data. It allocates memory for the big integer fields within the SchnorrProof struct.

Parameters

proto	Pointer to the SchnorrProofProto object to deserialize.
proof	Pointer to the SchnorrProof struct to populate with the deserialized data.

Returns

0 on success, or an error code on failure.

serialize_DecryptionShare()

```
uint8_t * serialize_DecryptionShare (
            DecryptionShare * share,
            unsigned * len )
```

Serializes a DecryptionShare object into a byte array.

This function serializes a DecryptionShare struct into a byte array using Protocol Buffers. It includes information about the guardian, contests, and selections, along with their corresponding decryption proofs.

Parameters

share	Pointer to the DecryptionShare object to serialize.
len	Pointer to an unsigned integer that will store the length of the serialized byte array.

Returns

A pointer to the serialized byte array. The caller is responsible for freeing the allocated memory.

serialize_election_key_pair()

```
uint8_t * serialize_election_key_pair (
            ElectionKeyPair * key_pair,
            unsigned * len )
```

Serialize an ElectionKeyPair struct into a byte array.

This function takes an ElectionKeyPair struct and serializes it into a byte array using Protocol Buffers. The serialized data includes the guardian ID, public key, and the election polynomial with its coefficients and Schnorr proofs.

Parameters

key_pair	Pointer to the ElectionKeyPair object to serialize.
len	Pointer to an unsigned integer that will store the length of the serialized byte array.

Returns

A pointer to the serialized byte array. The caller is responsible for freeing the allocated memory.

serialize_election_partial_key_backup()

```
uint8_t * serialize_election_partial_key_backup (
            ElectionPartialKeyPairBackup * backup,
            unsigned * len )
```

Serializes an ElectionPartialKeyPairBackup object into a byte array.

This function serializes an ElectionPartialKeyPairBackup struct into a byte array using Protocol Buffers. The serialized data includes the sender, receiver, and the encrypted coordinate (HashedElGamalCiphertext).

Parameters

backup	Pointer to the ElectionPartialKeyPairBackup object to serialize.
len	Pointer to an unsigned integer that will store the length of the serialized byte array.

Returns

A pointer to the serialized byte array. The caller is responsible for freeing the allocated memory.

serialize_election_partial_key_verification()

```
uint8_t * serialize_election_partial_key_verification (
            ElectionPartialKeyVerification * verification,
            unsigned * len )
```

Serializes an ElectionPartialKeyVerification object into a byte array.

This function serializes an ElectionPartialKeyVerification struct into a byte array using Protocol Buffers. The serialized data includes the sender, receiver, verifier, and the verification status.

Parameters

verification	Pointer to the ElectionPartialKeyVerification object to serialize.
len	Pointer to an unsigned integer that will store the length of the serialized byte array.

Returns

A pointer to the serialized byte array. The caller is responsible for freeing the allocated memory.

3.3 D:/Dokuments/IoT-election/guardian/components/adapter/util/test_decrypt.c File Reference

```
#include "test_decrypt.h"
```

Macros

- #define MEASUREMENT_RUNS 30

Functions

- static float calculate_std_dev (uint64_t *data, size_t count)

  Calculates the standard deviation of a given dataset.
- static void calculate_statistics (uint64_t *timings)

  Calculates and prints statistical metrics for a given dataset.
- void perform_measurement (ElectionKeyPair *guardian, CiphertextTally *tally)

  Performs timing measurements for decryption operations and calculates statistics.

3.3.1 Macro Definition Documentation

MEASUREMENT_RUNS

```
#define MEASUREMENT_RUNS 30
```

3.3.2 Function Documentation

calculate_statistics()

```
static void calculate_statistics (
            uint64_t * timings )  [static]
```

Calculates and prints statistical metrics for a given dataset.

Parameters

timings	Pointer to the array of timing data points.

calculate_std_dev()

```
static float calculate_std_dev (
            uint64_t * data,
            size_t count )  [static]
```

Calculates the standard deviation of a given dataset.

Parameters

data	Pointer to the array of data points.
count	Number of data points in the array.

Returns

float The standard deviation of the dataset.

perform_measurement()

```
void perform_measurement (
            ElectionKeyPair * guardian,
            CiphertextTally * tally )
```

Performs timing measurements for decryption operations and calculates statistics.

Parameters

guardian	Pointer to the ElectionKeyPair object used for decryption.
tally	Pointer to the CiphertextTally object to be decrypted.

3.4 D:/Dokuments/IoT-election/guardian/components/model/model.c File Reference

```
#include "model.h"
```

Functions

- int generate_election_key_pair (int quorum, ElectionKeyPair *key_pair)
  Generates an election key pair, including the public key, private key, polynomial coefficients, and commitments.
- int generate_election_partial_key_backup (ElectionKeyPair *sender, ElectionKeyPair *receiver, ElectionPartialKeyPairBackup *backup)
  Generates an election partial key backup for sharing between guardians.
- int verify_election_partial_key_backup (ElectionKeyPair *receiver, ElectionKeyPair *sender, ElectionPartialKeyPairBackup *backup, ElectionPartialKeyVerification *verification)
  Verifies an election partial key backup to confirm it contains a point on the owner's polynomial.
- int combine_election_public_keys (ElectionKeyPair *guardian, ElectionKeyPair *pubkey_map, size_t count, ElectionJointKey *joint_key)
  Combines individual election public keys into a joint public key.
- int compute_decryption_share (ElectionKeyPair *guardian, CiphertextTally *ciphertally, DecryptionShare *share)
  Computes a decryption share for a given ciphertext tally using a guardian's private key.

3.4.1 Function Documentation

combine_election_public_keys()

```
int combine_election_public_keys (
            ElectionKeyPair * guardian,
            ElectionKeyPair * pubkey_map,
            size_t count,
            ElectionJointKey * joint_key )
```

Combines individual election public keys into a joint public key.

This function aggregates the public keys of multiple guardians to form a joint public key, which happens at the end of the key ceremony. It also generates a commitment hash of the combined public key.

Parameters

guardian	An `ElectionKeyPair` representing the current guardian.
pubkey_map	An array of `ElectionKeyPair` structures, each containing a guardian's public key.
count	The number of guardians (and thus the number of public keys in `pubkey_map`).
joint_key	A pointer to an `ElectionJointKey` struct where the resulting joint public key and commitment hash will be stored. The caller must allocate memory for this struct before calling the function.

Returns

0 on success.

Note

The function computes both the joint public key and its corresponding commitment hash.

compute_decryption_share()

```
int compute_decryption_share (
            ElectionKeyPair * guardian,
            CiphertextTally * ciphertally,
            DecryptionShare * share )
```

Computes a decryption share for a given ciphertext tally using a guardian's private key.

Each guardian computes a decryption share for each contest in the ciphertext tally. These shares are later combined to decrypt the tally and reveal the election results.

Parameters

guardian	The `ElectionKeyPair` of the guardian computing the decryption share.
ciphertally	A pointer to the `CiphertextTally` struct containing the encrypted tallies for each contest.
share	A pointer to a `DecryptionShare` struct where the computed decryption share will be stored. The caller must allocate memory for this struct before calling the function.

Returns

0 on success.

Note

The function iterates through each contest in the ciphertext tally and computes a decryption share for it.

generate_election_key_pair()

```
int generate_election_key_pair (
            int quorum,
            ElectionKeyPair * key_pair )
```

Generates an election key pair, including the public key, private key, polynomial coefficients, and commitments.

This function generates an election key pair based on a specified quorum. It involves creating a polynomial with a degree determined by the quorum size. The coefficients of this polynomial form the basis for the private key, while commitments to these coefficients constitute the public key.

Parameters

quorum	The number of guardians required to decrypt the election. This determines the degree of the polynomial.
key_pair	A pointer to an `ElectionKeyPair` struct where the generated key pair, polynomial, and related data will be stored. The caller is responsible for allocating the `ElectionKeyPair` structure before calling this function.

Returns

0 on success.

-1 on failure, typically due to memory allocation issues.

Note

This function allocates memory for the public key, private key, and polynomial coefficients. It is crucial to free this memory after use to prevent memory leaks.

```
ElectionKeyPair key_pair;
int quorum = 10;
if (generate_election_key_pair(quorum, &key_pair) == 0) {
  // Use the generated key pair
  // ...
  // Free the allocated memory
  free_ElectionKeyPair(&key_pair);
} else {
  // Handle error
}
```

generate_election_partial_key_backup()

```
int generate_election_partial_key_backup (
            ElectionKeyPair * sender,
            ElectionKeyPair * receiver,
            ElectionPartialKeyPairBackup * backup )
```

Generates an election partial key backup for sharing between guardians.

This function creates a backup of a guardian's partial key, encrypts it using the recipient's public key, and prepares it for secure sharing. The backup includes the encrypted coordinate, sender and receiver identifiers.

Parameters

sender	The `ElectionKeyPair` of the guardian sending the backup.
receiver	The `ElectionKeyPair` of the guardian receiving the backup.
backup	A pointer to an `ElectionPartialKeyPairBackup` struct where the encrypted backup will be stored. The caller must allocate memory for this struct before calling the function.

Returns

0 on success.

-1 on failure.

Note

The function uses the receiver's public key to encrypt a coordinate derived from the sender's polynomial. It's essential to ensure the receiver's public key is valid and trusted.

verify_election_partial_key_backup()

```
int verify_election_partial_key_backup (
            ElectionKeyPair * receiver,
            ElectionKeyPair * sender,
            ElectionPartialKeyPairBackup * backup,
            ElectionPartialKeyVerification * verification )
```

Verifies an election partial key backup to confirm it contains a point on the owner's polynomial.

This function decrypts the encrypted coordinate from the backup using the receiver's private key and then verifies that the decrypted coordinate corresponds to a point on the sender's polynomial. This ensures the backup is valid and originates from the claimed sender.

Parameters

receiver	The `ElectionKeyPair` of the guardian receiving and verifying the backup.
sender	The `ElectionKeyPair` of the guardian who sent the backup.
backup	A pointer to the `ElectionPartialKeyPairBackup` struct containing the encrypted backup to be verified.
verification	A pointer to an `ElectionPartialKeyVerification` struct where the verification result will be stored. The caller must allocate memory for this struct before calling the function.

Returns

0 on success.

Note

The function updates the `verified` field in the `ElectionPartialKeyVerification` struct to indicate whether the backup is valid.

3.5 D:/Dokuments/IoT-election/guardian/components/model/util/constants.c File Reference

`#include "constants.h"`

Variables

- const unsigned char p_3072 [ ]
- const unsigned char g_3072 [ ]
- const unsigned char q_256 [ ]

3.5.1 Variable Documentation

g_3072

```
const unsigned char g_3072[]
```

p_3072

```
const unsigned char p_3072[]
```

q_256

```
const unsigned char q_256[]
```

Initial value:

```
= {
    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
}
```

3.6 D:/Dokuments/IoT-election/guardian/components/model/util/crypto_utils.c File Reference

```
#include "crypto_utils.h"
```

Functions

- void print_sp_int (sp_int ∗num)

    Prints an sp_int to the ESP_LOGI.
- static void int_to_bytes (int value, unsigned char ∗bytes)

    Converts an integer to a byte array (big-endian).
- void print_byte_array (const byte ∗array, int size)

    Prints a byte array to the ESP_LOGI.
- static int update_sha256_with_sp_int (Sha256 ∗sha256, sp_int ∗value)

    Updates a SHA256 context with an sp_int.
- static int get_hmac (unsigned char ∗key, unsigned char ∗in, unsigned char ∗out)

    Get a hash-based message authentication code(hmac) digest.
- static int kdf_xor (sp_int ∗key, sp_int ∗salt, sp_int ∗message, sp_int ∗encrypted_message)

    generates a Keystream using a KDF then XORs it with the message
- static int exptmod (sp_int ∗g, sp_int ∗x, sp_int ∗p, sp_int ∗y)

Modular Exponetiation. If the inputs are to small switches to unaccelerated version.
- static int mulmod (sp_int ∗a, sp_int ∗b, sp_int ∗c, sp_int ∗result)

  Modular Multiplication. If the inputs are to small switches to unaccelerated version.
- static int g_pow_p (sp_int ∗seckey, sp_int ∗pubkey)

  Compute Large number Modular Exponetiation with known G (generator also known as base) and P (large prime also known as modulus).
- int rand_q (sp_int ∗result)

  Generate a random number below constant Prime Q (small prime)
- static int finalise_hash (sp_int ∗result)

  Some hash operations require the intermediate results to be hashed again.
- int hash (sp_int ∗a, sp_int ∗b, sp_int ∗result)

  Given two mp_ints, calculate their cryptographic hash using SHA256.
- int compute_polynomial_coordinate (uint8_t ∗exponent_modifier, ElectionPolynomial ∗polynomial, sp_int ∗coordinate)

  Computes a single coordinate value of the election polynomial used for sharing.
- int verify_polynomial_coordinate (uint8_t ∗exponent_modifier, ElectionPolynomial ∗polynomial, sp_int ∗coordinate)

  Verifies that a coordinate lies on the election polynomial.
- int hashed_elgamal_encrypt (sp_int ∗message, sp_int ∗nonce, sp_int ∗public_key, sp_int ∗encryption_seed, HashedElGamalCiphertext ∗encrypted_message)

  Encrypts a message using Hashed ElGamal encryption.
- int hashed_elgamal_decrypt (HashedElGamalCiphertext ∗encrypted_message, sp_int ∗secret_key, sp_int ∗encryption_seed, sp_int ∗message)
- static int make_schnorr_proof (sp_int ∗seckey, sp_int ∗pubkey, sp_int ∗nonce, SchnorrProof ∗proof)

  Generates a Schnorr proof to demonstrate knowledge of a secret key without revealing it.
- int generate_polynomial (ElectionPolynomial ∗polynomial)

  Generates a polynomial for sharing election keys using Shamir's Secret Sharing.
- int elgamal_combine_public_keys (ElectionKeyPair ∗guardian, ElectionKeyPair ∗pubkey_map, size_t count, sp_int ∗key)

  Combines multiple ElGamal public keys into a single aggregate key.
- int hash_keys (ElectionKeyPair ∗guardian, ElectionKeyPair ∗pubkey_map, size_t count, sp_int ∗commitment)

  Computes a SHA256 hash of the commitments from multiple guardians.
- static int nonces (sp_int ∗seed, sp_int ∗nonce)

  Generates a deterministic nonces using SHA256 hashing.
- static int hash_challenge (sp_int ∗header, sp_int ∗alpha, sp_int ∗beta, sp_int ∗pad, sp_int ∗data, sp_int ∗m, sp_int ∗challenge)

  Computes a SHA256 hash to generate a challenge value for a Chaum-Pedersen proof.
- int make_chaum_pedersen (sp_int ∗alpha, sp_int ∗beta, sp_int ∗secret, sp_int ∗m, sp_int ∗seed, sp_int ∗hash_header, ChaumPedersenProof ∗proof)

  Generates a Chaum-Pedersen proof to demonstrate that a ciphertext corresponds to a specific plaintext.
- static int compute_decryption_share_for_selection (sp_int ∗privatekey, sp_int ∗pad, sp_int ∗data, sp_int ∗base_hash, CiphertextDecryptionSelection ∗dec_selection)

  Computes a partial decryption share of an ElGamal ciphertext for a specific selection.
- int verify_chaum_pedersen (sp_int ∗public_key, ChaumPedersenProof ∗proof, sp_int ∗alpha, sp_int ∗m)

  Verifies a Chaum-Pedersen proof.
- int compute_decryption_share_for_contest (ElectionKeyPair ∗guardian, CiphertextTallyContest ∗contest, sp_int ∗base_hash, CiphertextDecryptionContest ∗dec_contest)

  Computes a decryption share for a contest.

3.6.1 Function Documentation

compute_decryption_share_for_contest()

```
int compute_decryption_share_for_contest (
            ElectionKeyPair * guardian,
            CiphertextTallyContest * contest,
            sp_int * base_hash,
            CiphertextDecryptionContest * dec_contest )
```

Computes a decryption share for a contest.

This function computes a decryption share for a given contest based on the guardian's key pair and the contest data. It allocates memory for the decryption contest and its selections, copies relevant data, and computes the decryption share for each selection in the contest.

Parameters

guardian	A pointer to the `ElectionKeyPair` structure containing the guardian's key pair.
contest	A pointer to the `CiphertextTallyContest` structure representing the contest data.
base_hash	A pointer to the base hash (`sp_int`) used in the decryption process.
dec_contest	A pointer to the `CiphertextDecryptionContest` structure where the computed decryption share will be stored.

Returns

0 on success, -1 on memory allocation failure.

compute_decryption_share_for_selection()

```
static int compute_decryption_share_for_selection (
            sp_int * privatekey,
            sp_int * pad,
            sp_int * data,
            sp_int * base_hash,
            CiphertextDecryptionSelection * dec_selection )  [static]
```

Computes a partial decryption share of an ElGamal ciphertext for a specific selection.

This function computes a partial decryption share of an ElGamal ciphertext using a known ElGamal secret key. The partial decryption share is computed for a specific selection within a contest. The function also generates a Chaum-Pedersen proof to demonstrate that the decryption share is computed correctly.

Parameters

privatekey	A pointer to the secret key used to decrypt the ciphertext.
pad	A pointer to the pad of the ElGamal ciphertext.
data	A pointer to the data of the ElGamal ciphertext.
base_hash	A pointer to the base hash of the election.
dec_selection	A pointer to the `CiphertextDecryptionSelection` struct where the decryption share and proof will be stored. The caller is responsible for allocating memory for the `CiphertextDecryptionSelection` struct. The function will allocate memory for the `decryption` field within the `CiphertextDecryptionSelection` struct, as well as for the fields within the `proof` field.

Returns

0 on success.

-1 on failure (e.g., memory allocation error).

Note

The function allocates memory for the fields within the `dec_selection` structure. It is the caller's responsibility to free this memory when the decryption share is no longer needed to prevent memory leaks. See `CiphertextDecryptionSelection` and `ChaumPedersenProof` documentation for details.

compute_polynomial_coordinate()

```
int compute_polynomial_coordinate (
            uint8_t * exponent_modifier,
            ElectionPolynomial * polynomial,
            sp_int * coordinate )
```

Computes a single coordinate value of the election polynomial used for sharing.

Parameters

exponent_modifier	Unique modifier (guardian id) for exponent [0, Q]
polynomial	Election polynomial
coordinate	The computed coordinate

Returns

0 on success, -1 on failure

elgamal_combine_public_keys()

```
int elgamal_combine_public_keys (
            ElectionKeyPair * guardian,
            ElectionKeyPair * pubkey_map,
            size_t count,
            sp_int * key )
```

Combines multiple ElGamal public keys into a single aggregate key.

This function combines the public keys of a guardian and a set of other guardians into a single aggregate public key. The aggregate key is computed by multiplying all the individual public keys together modulo a large prime.

Parameters

guardian	A pointer to the `ElectionKeyPair` struct of the primary guardian. Its public key will be included in the combination.
pubkey_map	An array of `ElectionKeyPair` structs representing the other guardians whose public keys will be combined.
count	The number of elements in the `pubkey_map` array.
key	A pointer to an `sp_int` where the resulting combined public key will be stored. The caller must allocate memory for this `sp_int` before calling the function.

Returns

0 on success.

-1 on failure (e.g., if any of the multiplication operations fail).

Note

The function assumes that all public keys are valid ElGamal public keys with respect to the same large prime p.

exptmod()

```
static int exptmod (
            sp_int * g,
            sp_int * x,
            sp_int * p,
            sp_int * y )  [static]
```

Modular Exponetiation. If the inputs are to small switches to unaccelerated version.

Parameters

g	Base
x	Exponent
p	Modulus
y	Result

Returns

0 on success, -1 on failure

finalise_hash()

```
static int finalise_hash (
            sp_int * result )  [static]
```

Some hash operations require the intermediate results to be hashed again.

This function takes an sp_int and hashes it with delimiters before and after the value (e.g. H(|value|)). To ensure the result is below the small prime Q, the result is taken modulo Q.

Parameters

result	A pointer to the sp_int to be finalized.

Returns

0 on success, -1 on failure.

g_pow_p()

```
static int g_pow_p (
            sp_int * seckey,
            sp_int * pubkey )  [static]
```

Compute Large number Modular Exponetiation with known G (generator also known as base) and P (large prime also known as modulus).

Parameters

seckey	Exponent (X)
pubkey	Result

Returns

0 on success, -1 on failure

generate_polynomial()

```
int generate_polynomial (
            ElectionPolynomial * polynomial )
```

Generates a polynomial for sharing election keys using Shamir's Secret Sharing.

This function generates a polynomial of a specified degree, where each coefficient is a secret value. The polynomial is used to share an election key among multiple guardians using Shamir's Secret Sharing scheme. Each guardian receives a share of the secret key, which is an evaluation of the polynomial at a specific point. Any subset of guardians above a threshold can reconstruct the original secret key.

For each coefficient of the polynomial, the function generates a random value, computes a commitment to that value, and creates a Schnorr proof of knowledge for the coefficient.

Parameters

polynomial	A pointer to the `ElectionPolynomial` struct where the generated polynomial will be stored. The caller must allocate memory for the `ElectionPolynomial` struct and set the `num_coefficients` field before calling this function. This function will allocate memory for the `coefficients` array within the `ElectionPolynomial` struct, as well as for the `value`, `commitment`, and `proof` fields within each `PolynomialCoefficient` struct.

Returns

0 on success.

-1 on failure (e.g., memory allocation error).

Note

The function allocates memory for the fields within the `polynomial` structure. It is the caller's responsibility to free this memory when the polynomial is no longer needed to prevent memory leaks. See `Election↩ Polynomial` and `PolynomialCoefficient` documentation for details.

get_hmac()

```
static int get_hmac (
            unsigned char * key,
            unsigned char * in,
            unsigned char * out )  [static]
```

Get a hash-based message authentication code(hmac) digest.

Parameters

key	key (key) in bytes
in	input data in bytes
out	output hmac digest in bytes

Returns

0 on success, -1 on failure

hash()

```
int hash (
            sp_int * a,
            sp_int * b,
            sp_int * result )
```

Given two mp_ints, calculate their cryptographic hash using SHA256.

Each value is converted to a hexadecimal string and hashed with a delimiter in between (e.g. H(|a|b|)).

Parameters

a	First element
b	Second element
result	The result of the hash

Returns

0 on success, -1 on failure

hash_challenge()

```
static int hash_challenge (
            sp_int * header,
            sp_int * alpha,
            sp_int * beta,
            sp_int * pad,
            sp_int * data,
```

```
            sp_int * m,
            sp_int * challenge )  [static]
```

Computes a SHA256 hash to generate a challenge value for a Chaum-Pedersen proof.

This function computes a SHA256 hash of several input values to generate a challenge value for a Chaum-Pedersen proof. The input values include a header, two ElGamal ciphertext components (alpha and beta), two commitment values (pad and data), and the message being proven (m). The hash serves to bind all these values together in the challenge, ensuring that the proof is sound.

Parameters

header	A pointer to a header value (`sp_int`) that provides context for the hash. This is often the election extended base hash.
alpha	A pointer to the ElGamal ciphertext's alpha component (`sp_int`).
beta	A pointer to the ElGamal ciphertext's beta component (`sp_int`).
pad	A pointer to the commitment's pad value (`sp_int`).
data	A pointer to the commitment's data value (`sp_int`).
m	A pointer to the message being proven (`sp_int`).
challenge	A pointer to an `sp_int` where the resulting challenge value will be stored. The caller must allocate memory for this `sp_int` before calling the function.

Returns

0 on success.

-1 on failure (e.g., if SHA256 initialization or update fails, or memory allocation error).

Note

The function uses the WolfCrypt library for SHA256 hashing.

hash_keys()

```
int hash_keys (
            ElectionKeyPair * guardian,
            ElectionKeyPair * pubkey_map,
            size_t count,
            sp_int * commitment )
```

Computes a SHA256 hash of the commitments from multiple guardians.

This function calculates a SHA256 hash of the commitments made by a primary guardian and a set of other guardians. The hash serves as a binding value that commits all guardians to their respective commitments.

Parameters

guardian	A pointer to the `ElectionKeyPair` struct of the primary guardian. The commitments from its polynomial will be included in the hash.
pubkey_map	An array of `ElectionKeyPair` structs representing the other guardians whose commitments will be included in the hash.
count	The number of elements in the `pubkey_map` array.
commitment	A pointer to an `sp_int` where the resulting hash value will be stored. The caller must allocate memory for this `sp_int` before calling the function.

Returns

0 on success.

-1 on failure (e.g., if SHA256 initialization or update fails, or memory allocation error).

Note

The function concatenates the commitments of all guardians, separated by delimiters, before hashing the result.

hashed_elgamal_decrypt()

```
int hashed_elgamal_decrypt (
            HashedElGamalCiphertext * encrypted_message,
            sp_int * secret_key,
            sp_int * encryption_seed,
            sp_int * message )
```

Decrypt an ElGamal ciphertext using a known ElGamal secret key

Parameters

encrypted_message	struct containing pad, data, and mac
secret_key	The corresponding ElGamal secret key.
encryption_seed	Encryption seed (Q) for election.
message	Decrypted plaintext message.

Returns

0 on success, -1 on failure

hashed_elgamal_encrypt()

```
int hashed_elgamal_encrypt (
            sp_int * message,
            sp_int * nonce,
            sp_int * public_key,
            sp_int * encryption_seed,
            HashedElGamalCiphertext * encrypted_message )
```

Encrypts a message using Hashed ElGamal encryption.

This function encrypts a given message using the Hashed ElGamal encryption scheme. It generates a ciphertext consisting of a pad, data, and MAC (Message Authentication Code).

Parameters

message	The message (sp_int) to be encrypted.
nonce	A random nonce (sp_int) used for encryption.
public_key	The recipient's public key (sp_int).
encryption_seed	An encryption seed (sp_int) used in the key derivation function.
encrypted_message	A pointer to the HashedElGamalCiphertext structure where the resulting ciphertext will be stored.

Returns

0 on success, 1 on memory allocation failure.

int_to_bytes()

```
static void int_to_bytes (
            int value,
            unsigned char * bytes )  [static]
```

Converts an integer to a byte array (big-endian).

This helper function converts a 32-bit integer to a 4-byte array in big-endian order.

Parameters

value	The integer to convert.
bytes	A pointer to the byte array where the result will be stored.

kdf_xor()

```
static int kdf_xor (
            sp_int * key,
            sp_int * salt,
            sp_int * message,
            sp_int * encrypted_message )  [static]
```

generates a Keystream using a KDF then XORs it with the message

Parameters

key	Key used for the hashing
salt	Salt used for the hashing
message	Message to be hashed
encrypted_message	Write Encrypted message back to encrypted_message

Returns

0 on success, 1 on failure

make_chaum_pedersen()

```
int make_chaum_pedersen (
            sp_int * alpha,
            sp_int * beta,
            sp_int * secret,
            sp_int * m,
            sp_int * seed,
```

```
sp_int * hash_header,
ChaumPedersenProof * proof )
```

Generates a Chaum-Pedersen proof to demonstrate that a ciphertext corresponds to a specific plaintext.

This function generates a Chaum-Pedersen proof, which is a zero-knowledge proof that demonstrates that a given ElGamal ciphertext (alpha, beta) encrypts a specific plaintext value (m). The proof shows that the prover knows the secret (nonce) used to create the ciphertext, without revealing the secret itself.

Parameters

alpha	A pointer to the ElGamal ciphertext's alpha component ((g^r \mod p)), which is a large integer.
beta	A pointer to the ElGamal ciphertext's beta component ((m \cdot y^r \mod p)), which is a large integer.
secret	A pointer to the nonce ((r)) used to encrypt the message, which is a large integer.
m	A pointer to the plaintext message ((m)), which is a large integer.
seed	A pointer to a seed value used to generate random values within the proof.
hash_header	A pointer to a header value used when generating the challenge, typically the election extended base hash.
proof	A pointer to the `ChaumPedersenProof` struct where the generated proof will be stored. The caller is responsible for allocating memory for the `ChaumPedersenProof` struct. The function will allocate memory for the `pad`, `data`, `challenge`, and `response` fields within the `ChaumPedersenProof` struct.

Returns

0 on success.

-1 on failure (e.g., memory allocation error).

Note

The function allocates memory for the fields within the `proof` structure. It is the caller's responsibility to free this memory when the proof is no longer needed to prevent memory leaks. See `ChaumPedersenProof` documentation for details.

make_schnorr_proof()

```
static int make_schnorr_proof (
            sp_int * seckey,
            sp_int * pubkey,
            sp_int * nonce,
            SchnorrProof * proof ) [static]
```

Generates a Schnorr proof to demonstrate knowledge of a secret key without revealing it.

This function creates a Schnorr proof, which is a cryptographic proof-of-knowledge protocol. It proves that the prover (the entity possessing the secret key) knows the secret key corresponding to a given public key, without disclosing the secret key itself.

The Schnorr proof is generated using the provided secret key, public key, and a nonce (a random number). The proof consists of a commitment, a challenge, and a response.

Parameters

seckey	A pointer to the secret key, which is a large integer.
pubkey	A pointer to the public key, which is a large integer.
nonce	A pointer to the nonce, a random element in the range [0, Q].
proof	A pointer to the SchnorrProof struct where the generated proof will be stored. The caller is responsible for allocating memory for the SchnorrProof struct. The function will allocate memory for the pubkey, commitment, challenge, and response fields within the SchnorrProof struct.

Returns

0 on success.

-1 on failure (e.g., memory allocation error).

Note

The function allocates memory for the fields within the proof structure. It is the caller's responsibility to free this memory when the proof is no longer needed to prevent memory leaks. See `SchnorrProof` documentation for details.

mulmod()

```
static int mulmod (
            sp_int * a,
            sp_int * b,
            sp_int * c,
            sp_int * result )  [static]
```

Modular Multiplication. If the inputs are to small switches to unaccelerated version.

Parameters

a	First element
b	Second element
c	Modulus
result	The result of the multiplication

Returns

0 on success, -1 on failure

nonces()

```
static int nonces (
            sp_int * seed,
            sp_int * nonce )  [static]
```

Generates a deterministic nonces using SHA256 hashing.

This function generates a nonces in the range [0, Q), where Q is a small prime number. The nonce is generated deterministically from an initial seed value using SHA256 hashing. If the same seed is used, the same sequence of nonces will be generated.

The function uses a header string "constant-chaum-pedersen-proof|" to ensure that the generated nonces are specific to a particular context (e.g., a Chaum-Pedersen proof).

Parameters

seed	A pointer to the initial seed value, which is an `sp_int`.
nonce	A pointer to an `sp_int` where the generated nonce will be stored. The caller must allocate memory for this `sp_int` before calling the function.

Returns

0 on success.

-1 on failure (e.g., if SHA256 initialization or update fails, or memory allocation error).

Note

The function uses the WolfCrypt library for SHA256 hashing.

print_byte_array()

```
void print_byte_array (
            const byte * array,
            int size )
```

Prints a byte array to the ESP_LOGI.

This function converts a byte array to a hexadecimal string and prints it using ESP_LOGI.

Parameters

array	The byte array to print.
size	The size of the byte array.

print_sp_int()

```
void print_sp_int (
            sp_int * num )
```

Prints an sp_int to the ESP_LOGI.

This function converts an sp_int to a hexadecimal string and prints it using ESP_LOGI.

Parameters

num	The sp_int to print.

rand_q()

```
int rand_q (
            sp_int * result )
```

Generate a random number below constant Prime Q (small prime)

Parameters

result	The random number

Returns

0 on success, -1 on failure

update_sha256_with_sp_int()

```
static int update_sha256_with_sp_int (
            Sha256 * sha256,
            sp_int * value ) [static]
```

Updates a SHA256 context with an sp_int.

This function converts an sp_int to a hexadecimal string and updates a SHA256 context with it. It also adds a delimiter "|" after the hexadecimal string.

Parameters

sha256	A pointer to the SHA256 context.
value	A pointer to the sp_int to hash.

Returns

0 on success.

-1 on failure.

verify_chaum_pedersen()

```
int verify_chaum_pedersen (
            sp_int * public_key,
            ChaumPedersenProof * proof,
            sp_int * alpha,
            sp_int * m )
```

Verifies a Chaum-Pedersen proof.

This function verifies that a given Chaum-Pedersen proof is valid with respect to the provided public key

Parameters

public_key	The public key (`sp_int`) used in the Chaum-Pedersen proof.
proof	A pointer to the `ChaumPedersenProof` structure containing the proof data.
alpha	An `sp_int` representing the generator `g` of the group.
m	An `sp_int` representing the message.

Returns

0 if the proof is valid, otherwise logs an error message and returns a non-zero value.

verify_polynomial_coordinate()

```
int verify_polynomial_coordinate (
            uint8_t * exponent_modifier,
            ElectionPolynomial * polynomial,
            sp_int * coordinate )
```

Verifies that a coordinate lies on the election polynomial.

Given a guardian's identifier, a polynomial, and a coordinate, this function verifies that the coordinate corresponds to a point on the polynomial. This is used to validate partial key backups.

Parameters

exponent_modifier	The unique identifier of the guardian (usually sequence order).
polynomial	The election polynomial to verify against.
coordinate	The coordinate to verify.

Returns

1 if the coordinate is on the polynomial, 0 otherwise. Returns -1 on failure.

3.7 D:/Dokuments/IoT-election/guardian/components/model/util/utils.c File Reference

```
#include "utils.h"
#include <stdlib.h>
```

Functions

- void free_CiphertextTallySelection (CiphertextTallySelection ∗selection)

  Frees the memory allocated for a CiphertextTallySelection struct.
- void free_CiphertextTallyContest (CiphertextTallyContest ∗contest)

  Frees the memory allocated for a CiphertextTallyContest struct.
- void free_CiphertextTally (CiphertextTally ∗tally)

  Frees the memory allocated for a CiphertextTally struct.
- void free_ChaumPedersenProof (ChaumPedersenProof ∗proof)

Frees the memory allocated for a ChaumPedersenProof struct.

- void free_CiphertextDecryptionSelection (CiphertextDecryptionSelection ∗selection)

  Frees the memory allocated for a CiphertextDecryptionSelection struct.
- void free_CiphertextDecryptionContest (CiphertextDecryptionContest ∗contest)

  Frees the memory allocated for a CiphertextDecryptionContest struct.
- void free_DecryptionShare (DecryptionShare ∗share)

  Frees the memory allocated for a DecryptionShare struct.
- void free_ElectionPartialKeyPairBackup (ElectionPartialKeyPairBackup ∗backup)

  Frees the memory allocated for an ElectionPartialKeyPairBackup struct.
- void free_ElectionKeyPair (ElectionKeyPair ∗key_pair)

  Frees the memory allocated for an ElectionKeyPair struct.
- void free_ElectionPolynomial (ElectionPolynomial ∗polynomial)

  Frees the memory allocated for an ElectionPolynomial struct.
- void free_Coefficient (Coefficient ∗coefficient)

  Frees the memory allocated for a Coefficient struct.
- void free_SchnorrProof (SchnorrProof ∗proof)

  Frees the memory allocated for a SchnorrProof struct.

3.7.1 Function Documentation

free_ChaumPedersenProof()

```
void free_ChaumPedersenProof (
            ChaumPedersenProof * proof )
```

Frees the memory allocated for a ChaumPedersenProof struct.

This function releases the memory associated with the pad, data, challenge, and response members of the ChaumPedersenProof struct. It also sets these pointers to NULL to prevent double freeing.

Parameters

proof	A pointer to the ChaumPedersenProof struct to free. If proof is NULL, the function returns immediately.

free_CiphertextDecryptionContest()

```
void free_CiphertextDecryptionContest (
            CiphertextDecryptionContest * contest )
```

Frees the memory allocated for a CiphertextDecryptionContest struct.

This function releases the memory associated with the object_id, description_hash, and selections members of the CiphertextDecryptionContest struct. It iterates through the selections array and calls free_CiphertextDecryption←Selection for each element before freeing the array itself. It also sets pointers to NULL to prevent double freeing.

Parameters

contest	A pointer to the CiphertextDecryptionContest struct to free. If contest is NULL, the function returns immediately.

free_CiphertextDecryptionSelection()

```
void free_CiphertextDecryptionSelection (
            CiphertextDecryptionSelection * selection )
```

Frees the memory allocated for a CiphertextDecryptionSelection struct.

This function releases the memory associated with the object_id and decryption members of the Ciphertext↩ DecryptionSelection struct. It also calls free_ChaumPedersenProof to free the proof member. It sets pointers to NULL to prevent double freeing.

Parameters

selection	A pointer to the CiphertextDecryptionSelection struct to free. If selection is NULL, the function returns immediately.

free_CiphertextTally()

```
void free_CiphertextTally (
            CiphertextTally * tally )
```

Frees the memory allocated for a CiphertextTally struct.

This function releases the memory associated with the object_id, base_hash, and contests members of the CiphertextTally struct. It iterates through the contests array and calls free_CiphertextTallyContest for each element before freeing the array itself. It also sets pointers to NULL to prevent double freeing.

Parameters

tally	A pointer to the CiphertextTally struct to free. If tally is NULL, the function returns immediately.

free_CiphertextTallyContest()

```
void free_CiphertextTallyContest (
            CiphertextTallyContest * contest )
```

Frees the memory allocated for a CiphertextTallyContest struct.

This function releases the memory associated with the object_id, description_hash, and selections members of the CiphertextTallyContest struct. It iterates through the selections array and calls free_CiphertextTallySelection for each element before freeing the array itself. It also sets pointers to NULL to prevent double freeing.

Parameters

contest	A pointer to the CiphertextTallyContest struct to free. If contest is NULL, the function returns immediately.

free_CiphertextTallySelection()

```
void free_CiphertextTallySelection (
            CiphertextTallySelection * selection )
```

Frees the memory allocated for a CiphertextTallySelection struct.

This function releases the memory associated with the object_id, ciphertext_pad, and ciphertext_data members of the CiphertextTallySelection struct. It also sets these pointers to NULL to prevent double freeing.

Parameters

selection	A pointer to the CiphertextTallySelection object to free. If selection is NULL, the function returns immediately.

free_Coefficient()

```
void free_Coefficient (
            Coefficient * coefficient )
```

Frees the memory allocated for a Coefficient struct.

This function releases the memory associated with the commitment and value members of the Coefficient struct. It also calls free_SchnorrProof to free the proof member. It sets pointers to NULL to prevent double freeing.

Parameters

coefficient	A pointer to the Coefficient struct to free. If coefficient is NULL, the function returns immediately.

free_DecryptionShare()

```
void free_DecryptionShare (
            DecryptionShare * share )
```

Frees the memory allocated for a DecryptionShare struct.

This function releases the memory associated with the object_id, public_key, and contests members of the DecryptionShare struct. It iterates through the contests array and calls free_CiphertextDecryptionContest for each element before freeing the array itself. It also sets pointers to NULL to prevent double freeing.

Parameters

share	A pointer to the DecryptionShare struct to free. If share is NULL, the function returns immediately.

free_ElectionKeyPair()

```
void free_ElectionKeyPair (
            ElectionKeyPair * key_pair )
```

Frees the memory allocated for an ElectionKeyPair struct.

This function releases the memory associated with the public_key and private_key members of the ElectionKeyPair struct. It also calls free_ElectionPolynomial to free the polynomial member. It sets pointers to NULL to prevent double freeing. The private key is zeroed out before freeing.

Parameters

key_pair	A pointer to the ElectionKeyPair struct to free. If key_pair is NULL, the function returns immediately.

free_ElectionPartialKeyPairBackup()

```
void free_ElectionPartialKeyPairBackup (
            ElectionPartialKeyPairBackup * backup )
```

Frees the memory allocated for an ElectionPartialKeyPairBackup struct.

This function releases the memory associated with the encrypted_coordinate members of the ElectionPartialKey↩ PairBackup struct. It also sets pointers to NULL to prevent double freeing.

Parameters

backup	A pointer to the ElectionPartialKeyPairBackup struct to free. If backup is NULL, the function returns immediately.

free_ElectionPolynomial()

```
void free_ElectionPolynomial (
            ElectionPolynomial * polynomial )
```

Frees the memory allocated for an ElectionPolynomial struct.

This function releases the memory associated with the coefficients member of the ElectionPolynomial struct. It iterates through the coefficients array and calls free_Coefficient for each element before freeing the array itself. It also sets the pointer to NULL to prevent double freeing.

Parameters

polynomial	A pointer to the ElectionPolynomial struct to free. If polynomial is NULL, the function returns immediately.

free_SchnorrProof()

```
void free_SchnorrProof (
            SchnorrProof * proof )
```

Frees the memory allocated for a SchnorrProof struct.

This function releases the memory associated with the pubkey, commitment, challenge, and response members of the SchnorrProof struct. It also sets pointers to NULL to prevent double freeing.

Parameters

proof	A pointer to the SchnorrProof struct to free. If proof is NULL, the function returns immediately.

3.8 D:/Dokuments/IoT-election/guardian/main/main.c File Reference

```
#include "adapter.h"
#include "nvs_flash.h"
#include "test_performance.h"
#include "esp_task_wdt.h"
```

Functions

- static void run_test ()
- static void guardian_client ()
- void app_main (void)

3.8.1 Function Documentation

app_main()

```
void app_main (
            void )
```

guardian_client()

```
static void guardian_client ( )  [static]
```

run_test()

```
static void run_test ( )  [static]
```

3.9 D:/Dokuments/IoT-election/guardian/main/test_performance.c File Reference

```
#include "test_performance.h"
```

Macros

- #define MEASUREMENT_RUNS 30

Functions

- static float calculate_std_dev (uint64_t ∗data, size_t count)
  Calculates the standard deviation of a given dataset.
- static void calculate_statistics (uint64_t ∗timings)
  Calculates and prints statistical metrics for a given dataset.
- void perform_measurements_keygen (int quorum)
  Performs timing measurements for election key pair generation and calculates statistics.
- void perform_measurements_backup ()
  Performs timing measurements for election partial key backup generation and calculates statistics.
- void perform_measurements_verification ()
  Performs timing measurements for election partial key backup verification and calculates statistics.

3.9.1 Macro Definition Documentation

MEASUREMENT_RUNS

```
#define MEASUREMENT_RUNS 30
```

3.9.2 Function Documentation

calculate_statistics()

```
static void calculate_statistics (
            uint64_t * timings ) [static]
```

Calculates and prints statistical metrics for a given dataset.

Parameters

timings	Pointer to the array of timing data points.

calculate_std_dev()

```
static float calculate_std_dev (
            uint64_t * data,
            size_t count ) [static]
```

Calculates the standard deviation of a given dataset.

Parameters

data	Pointer to the array of data points.
count	Number of data points in the array.

Returns

float The standard deviation of the dataset.

perform_measurements_backup()

```
void perform_measurements_backup ( )
```

Performs timing measurements for election partial key backup generation and calculates statistics.

perform_measurements_keygen()

```
void perform_measurements_keygen (
            int quorum )
```

Performs timing measurements for election key pair generation and calculates statistics.

Parameters

quorum	The quorum size used for key generation.

perform_measurements_verification()

```
void perform_measurements_verification ( )
```

Performs timing measurements for election partial key backup verification and calculates statistics.

3.10 D:/Dokuments/IoT-election/guardian/README.md File Reference

Index