Politecnico di Torino

Cybersecurity for Embedded Systems (01UDNOV)

Asymmetric cryptography for SEcube

Final Project Report

Master Degree in Computer Engineering

Referents: Prof. Paolo Prinetto, Matteo Fornero, Nicoló Maunero, Gianluca Roascio

Brignone Giovanni (s274148)
Castagneri Dario (s277967)
Licastro Dario (s280121)

September 11, 2021

Contents

Abstract

The purpose of this project is to add support to RSA cryptography and X.509 certificates to SEcube devices, by adding these functionalities to the firmware and exposing them through a level 1 API on the host side.

The initial objective was to implement the RSA cipher in hardware, on the SEcube FPGA, but due to limitations of the HW at disposal, a full software implementation seemed the best choice, using `mbedtls` by ARM as underlying library for cryptography functionalities.

CHAPTER 1

Introduction

This manual covers the details and the features of the Asymmetric cryptography implementation of the SEcube™ Open Source Security Platform. To learn more about the SEcube™ and the capabilities of this Open Source Security Platform, please check out the SEcube™ SDK documentation. Asymmetric cryptography implementation concerns asymmetric key management, an RSA cipher, Digital signature and X.509 certificates.

1.1 Cryptography

Cryptography is the discipline in charge of developing techniques for securing communication in presence of opponents willing to read and alter messages. Encryption uses an algorithm and a key to transform an input (i.e., plaintext) into an encrypted output (i.e., ciphertext).

Algorithms are considered secure if an attacker cannot determine any properties of the plaintext or key, given the ciphertext. An attacker should not be able to determine anything about a key given a large number of plaintext/ciphertext combinations which used the key.

There are typically two main kind of cryptographic solutions:

- *Symmetric cryptography*: the actors involved in the communication must share a common secret to secure the communication. Performance is generally acceptable, therefore it is suitable for encrypting large amounts of data. Communication problem is not completely solved: the common secret still has to be confidentially shared.
- *Asymmetric cryptography*: the actors can securely communicate without any shared common secret. Communication problem is solved, but it is necessary to pay in terms of performance.

1.1.1 Asymmetric cryptography

Characteristics

In an asymmetric (or public-key) cryptosystem each actor owns a pair of keys: private, known to the owner only, and public, known to anyone. There is a single algorithm $F(key, msg)$ which can be applied to different keys, thanks to its reciprocal properties:

- *Encryption:* $ciphertext = F(key_{pub,recipient}, plaintext)$
- *Decryption:* $plaintext = F(key_{priv,recipient}, ciphertext)$

In order to counteract brute force attacks, the keys must be very long (i.e. minimum 2048 bits for an acceptable security level, with today algorithms and technology).

Applications

- **Symmetric key distribution** To solve the secure communication problem between actors A and B:

  1. A generates a symmetric cryptosystem secret.
  2. A encrypts the secret using B's public key.
  3. A sends the encrypted secret over an untrusted channel to B.
  4. B decrypts the secret using its own private key.
  5. A and B now share the common secret and can start using symmetric cryptography.

  This solution guarantees confidentiality.

- **Digital signature**

  – *Sign:* the signer computes the digest (e.g. SHA256) of the message to be signed, then applies the asymmetric cryptography algorithm using its private key.

  – *Verify:* the verifier computes the digest of the signed message and applies the asymmetric cryptography algorithm using the signer public key: if the two digests match, the signature is verified.

  This solution guarantees authentication and integrity, moreover, if keys are taken from certificates by Certification Authorities, non-repudiation is guaranteed too.

1.1.2 RSA cryptography

The RSA algorithm is the basis of a public-key cryptosystem, a suite of cryptographic algorithms used for security purposes. RSA was first described publicly in 1977 by Ron Rivest, Adi Shamir and Leonard Adleman of the Massachusetts Institute of Technology, although the 1973 creation of a public key algorithm by British mathematician Clifford Cocks has been kept secret by the UK's GCHQ until 1997. PKCS (Public Key Cryptography Standards) are specifications used in computer cryptography. PKCS#1 [1], in particular, identifies the RSA cryptosystem standard.

Algorithm

- **Key generation**:

  1. select P and Q such that they are prime, big, random and secret.
  2. let $N = P \cdot Q$ and $\varphi = (P-1) \cdot (Q-1)$
  3. select E such that $1 < E < \varphi$ and E and φ are coprime
  4. let $D = E^{-1} \mod \varphi$
  5. public key is composed of $\{E, N\}$ and can be shared to anyone; private key is composed of $\{D, N\}$ and must be kept secret

- **Modular exponential**:
  the keys are chosen in such a way that: $(P^E)^D \mod N \equiv (P^D)^E \mod N$, therefore the primitives provided by PKCS#1 are:

  – *RSA Encryption Primitive*:
  anyone (since E and N are public) can encrypt a message by applying:

$$ciphertext = (plaintext)^E \mod N$$

- *RSA Decryption Primitive*:
  the owner of the key only (since D is private) can decrypt the ciphertext by applying:

$$plaintext = (ciphertext)^D \mod N$$

- *RSA Signature Primitive*:
  the owner of the key only (since D is private) can sign a message by applying:

$$signature = (hash(message))^D \mod N$$

- *RSA Verification Primitive*:
  anyone (since E and N are public) can verify a signature by:
  - retrieving the signed has by applying:

$$hash = (signature)^E \mod N$$

  - computing the *hash* of the *message* again
  - comparing the two hashes

Performance and security

- **Key length**:
  The factorization complexity is $O(e^n)$, therefore, in order to make it unfeasible to break RSA by means of brute force, it is necessary to use very long keys (1024 bits for a bare minimum level of security, but 2048 bits or more are recommended).

  The exponentiation complexity is instead $O(\log^3 n)$, therefore RSA performance are low and it is discouraged to encrypt long messages with such an algorithm, but it is suitable for secure key exchange and digital signature.

- **Public key optimizations**:
  Execution time of exponential highly depends on the number of `1`s in the exponent (in binary format), therefore E is usually chosen between 3, 17 or 65537, since they are prime and their binary representations contain 2 `1`s only. 65537 is preferable, since large exponents guarantees higher security levels.

- **Private key optimizations**:
  The Chinese Remainder Theorem can be used to make private-key operations up to 4 times faster.

- **Dedicated keys**:
  When providing RSA functionalities as a service (e.g., from a server which accepts messages and encrypts/signs them) it is important to use separate keys for encryption and for signing. Otherwise an attacker could compute the hash of a message, ask to decrypt it and then claim it was actually signed with that particular key.

1.1.3 X.509 certificates

X.509 is a standard format for public key certificates, digital documents that securely associate cryptographic key pairs with identities such as websites, individuals, or organizations. It is based on the internationally trusted International Telecommunications Union (ITU) X.509 standard, which defines the format of public key infrastructure (PKI) certificates. They are used to manage identity and security in internet communications and computer networking. They are unobtrusive and ubiquitous, and we encounter them every day when using websites, mobile apps, online documents, and connected devices.

1.2 SEcube™

SEcube™ is an open-source security-oriented hardware and software platform.

1.2.1 Hardware

The core of the SEcube™ Hardware device family is a chip which embeds three hardware components: an STM32F429 processor (which includes an ARM Cortex M4 core and a True Random Number Generator), a Lattice MachXO2-7000 FPGA and an Infineon SLJ52G EAL5+ certified smart-card. The Lattice MachXO2-7000 device is based on a fast, non-volatile logic array whose main features are: 7,000 LUTs, 240 Kbits of embedded block RAM, 256 Kbits of user Flash memory. The communication between FPGA and CPU can be performed through a 16-bit internal bus.

1.2.2 Software

SEcube™ SDK is composed of:

- *Host*: code to be run on a desktop computer which provides libraries to control SEcube and expose its functionalities.
- *Firmware*: code to be run on a SEcube device which provide libraries for communicating with host and the implementations of symmetric cryptosystems and hashing functions.

The purpose of this project is to extend the SDK introducing the basics of asymmetric cryptography.

CHAPTER 2

Design and implementation

2.1 Hardware and software partitioning

Before working on the actual implementation it is necessary to decide what functionality should be implemented in hardware (on the FPGA) and in software (running on the CPU).

The most efficient solution in terms of performance and power consumption is to design at RTL the complete RSA module (composed of a key generation module and an encryption/decryption module), but the available FPGA provides 7000 LUTs only, therefore it is unfeasible to fit the entire design in HW.

In order to reduce the FPGA area occupation of the design, the natural solution was to identify the most critical parts, to be implemented in HW, and implement the rest in SW.

Since keys are generated once and used many times, the key generation module is less critical and it could be implemented in SW. The encryption/decryption module is basically a modular exponential: its critical part is the modular multiplier.

In this scenario it was necessary to find a reference architecture of a modular multiplier designed to be area efficient: the most widely accepted solution is the Montgomery multiplier, but all the implementations found in literature require a too high number of LUTs (e.g. for a 1024 bits multiplier, which is the bare minimum key length for a basic level of security: Blum's [2] smallest implementation requires 7572 LUTs, Zhang's [3] 9024 LUTs, Amanor's [4] 7680 LUTs, Daly's [5] 10806 LUTs).

The Tenca and Koc's [6] architecture is theoretically adaptable to any area at disposal: the more area is available, the higher performance is achievable.

Since the available area in the SEcube FPGA is very low and the time overhead for transferring data between CPU and FPGA would be non-negligible (in the smallest possible case, with 1024 bits keys, for a single multiplication it is needed to send 2048 bits for the multiplicands, 1024 bits for the modulus and receive 1024 bits of product, over a 16 bits bus), the performance gain would be low or null.

Moreover, since RSA typical applications are to encrypt symmetric keys or message digests, performance are not very critical: what it is really critical is the implementation security, which is easier to obtain in software, thanks to the easier verification process.

After all these considerations, the best solution appeared to be the usage of a well tested and trusted software library for the implementation of RSA functionality.

2.2 Firmware side

The SEcube firmware has been extended in order to support receiving, processing and responding to RSA functionality and key management and X.509 certificate management requests sent from the host.

2.2.1 RSA and X.509 library

The most convenient solution for getting core RSA functionalities (encrypt, decrypt, sign, verify) and X.509 certificates generation is to exploit an existing library. Since the library should run on an STM32F429 micro controller, it should have a small resources footprint and it should be implemented in C, being the only natively supported programming language. The library should also be included into the SEcube open source program, therefore it must have a permissive license. The most suitable libraries complying with the desirable characteristics are: `mbedtls` [7], `libtomcrypt` [8] and `BearSSL` [9]. Considering stability, security and trustfulness of the developer, `mbedtls` by `ARM` has been chosen as the underlying library for this project.

The library has been configured to run on the SEcube processor and all the unused files have been removed.

The SEcube True Random Number Generator is used whenever a random number has to be generated.

The digest for the digital signature is computed by the SHA256 algorithm.

2.2.2 Flash memory

RSA key storage

- **Key structure**

```
typedef struct se3_rsa_flash_key_ {
        uint32_t id;
        uint16_t key_size;
        uint8_t type;
        uint8_t public_only;
        uint8_t* N;
        uint8_t* E;
        uint8_t* D;
} se3_rsa_flash_key;
```

  `se3_rsa_flash_key` is the structure holding all the information needed for storing RSA keys to flash. In particular it contains:

  - `id`: the unique identifier of the key.
  - `key_size`: the size of `N`, `E` and `D` arrays.
  - `type`: it specifies the operations that can be performed with the key (crypto only, signature only or both) (see 1.1.2 for more details).
  - `public_only`: it is set to `true` when the key is composed of its public part only (therefore `D` content is ignored).
  - `N`, `E` and `D`: the raw key data.

- **Flash memory**
  In order to avoid code duplication for better maintainability and integration, RSA keys are stored to flash memory in the same flash node type (`se3_flash_key`) and in the same IDs space as symmetric keys, using the functions of `se3_key`.

  `se3_rsa_keys.c` includes all the functions required for managing RSA keys stored to flash. It operates as an interface for RSA keys over the functions provided by `se3_keys.c`: `se3_flash_key` contains a single data buffer, while `se3_rsa_flash_key` contains multiple data fields, therefore it is necessary to concatenate the different slices into a single buffer when writing to flash (performed by the `se3_rsa_to_plain_flash` function) and to split into the different fields when reading from flash (performed by the `se3_plain_to_rsa_flash` function).

X.509 certificates storage

The SEcube SDK did not provide any kind of certificate support, therefore a new flash node type have been implemented specifically for X.509 certificates, together with the functions needed for reading and writing to and from flash, in `se3_x509.c`.

2.2.3 Dispatcher

When the host sends a request to the SEcube device, the request is processed by the *Communication core* and forwarded to the *Dispatcher core* which is in charge of calling the requested functionality. Since the *Dispatcher core* can fit a limited number of functionalities, RSA functionalities are not directly called: *Dispatcher core* calls a dedicated *RSA dispatcher core*, which in turn calls the specific RSA functionality.

`se3_rsa`

`se3_rsa.c` exposes all the RSA functionalities to the dispatcher. All the `se3_rsa` functions have a similar structure:

1. Parse the request from the host, checking its validity.
2. Perform the RSA computation by exploiting the underlying RSA library.
3. Build the response to the host.

Among the request validity checks there is the key type check: if the request asks to perform a crypto operation using a signature-only key or a signature operation using a crypto-only key, an error code is returned, instead of performing the requested operation.

2.3 Host side

The SEcube host side code has been extended in order to add APIs for accessing RSA functionality and key management and X.509 certificate management of the SEcube device. These APIs also take care of checking correctness of inputs and outputs.

This extension has been performed while trying to maximize the integration with the existing APIs: some APIs have been reused (`L1FindKey`), some have been extended to support additional features (`L1KeyEdit`, `L1Encrypt`, `L1Decrypt`), while APIs peculiar to RSA and X.509 certificates have been added trying to comply with existing hierarchies and conventions.

2.3.1 Communication timeout

RSA key generation is a very complex operation, since it has to generate very long random numbers (at least 1024 bits) until it generates a prime number, therefore it may take a huge amount of time.

The communication between SEcube device and host was limited by a timeout of 10 seconds (defined in `L0.h` as `SE3_TIMEOUT`), resulting in an exception before RSA key generation could complete.

According to empirical measurements, 1000 seconds should be enough for generating 2048 bits keys, therefore the timeout has been set to that value.

It is advisable to use longer timeouts when generating longer RSA keys.

CHAPTER 3

API documentation

This section contains an overview of the level 1 APIs which expose asymmetric cryptography functionalities. More details can be found in the source code Doxygen-compatible comments.

3.1 SEcube RSA cryptography

Key management

```
typedef struct se3Key_ {
        uint32_t id;
        uint16_t dataSize;
        uint8_t* data;
        se3AsymmKey asymmKey;
} se3Key;

typedef struct se3AsymmKey_ {
        uint8_t* N;
        uint8_t* E;
        uint8_t* D;
        uint8_t type;
} se3AsymmKey;

struct RSAKeyType {
        enum {
                SE3_RSA_KEY_GENERIC,
                SE3_RSA_KEY_CIPHER,
                SE3_RSA_KEY_SIGN
        };
};
```

`se3Key` is the structure containing symmetric or asymmetric keys. `id` is a 32-bit unique identifier, `datasize` is the key size, `data` is the pointer to the symmetric key value and `asymmKey` is a structure containing the asymmetric key data. The asymmetric key may be of signature type, encryption type or a generic type.

```
struct KeyOpEdit {
        enum {
          SE3_KEY_OP_ADD = 1,
          SE3_KEY_OP_DELETE = 2,
          SE3_KEY_OP_ADD_TRNG = 3,
          SE3_KEY_OP_ADD_RSA = 5,
          SE3_KEY_OP_ADD_GEN_RSA = 6
```

```
        };
};

void L1::L1KeyEdit(se3Key& k, uint16_t op);
```

`L1KeyEdit` provides write access to RSA keys on SEcube device. It can perform three operations, depending on `op` value:

- `SE3_KEY_OP_ADD_GEN_RSA`: generate an RSA key `k.dataSize` Bytes long and store it to SEcube flash memory, associating it with the ID specified in `k.id`.
- `SE3_KEY_OP_ADD_RSA`: store the key provided in `k.asymmKey` to SEcube flash memory, associating it with the ID specified in `k.id`.
- `SE3_KEY_OP_DELETE`: delete the key associated with the ID specified in `k.id` from SEcube flash memory (can be a symmetric or an asymmetric key).

Note 1: Adding RSA key of size different from 1024, 2048, 4096 and 8192 bits is forbidden and generates an error.

Note 2: `SE3_KEY_OP_ADD` and `SE3_Key_OP_ADD_TRNG` are related to symmetric key functionalities.

```
void L1::L1FindKey(uint32_t keyId, bool& found);
```

`L1FindKey` sets `found` to `true` if a key (symmetric or asymmetric) associated with `keyId` is stored to SEcube flash memory, to `false` otherwise.

```
void L1::L1AsymmKeyGet(se3Key& k);
```

`L1AsymmKeyGet` reads from SEcube flash memory the public part (modulo N and public exponent E) associated with the ID specified in `k.id` and stores it to `k`.

Note: for security reasons there is no API for reading private exponent D of an RSA key stored to SEcube flash memory.

Cryptography

```
void L1::L1Encrypt(size_t plaintext_size, std::shared_ptr<uint8_t[]> plaintext,
        SEcube_ciphertext& encrypted_data, uint16_t algorithm,
        uint16_t algorithm_mode,
        se3Key key, uint8_t on_the_fly);
```

When `algorithm` is set to `L1Algorithms::Algorithms::RSA` the data stored to `plaintext` is encrypted using the RSA algorithm. If `on_the_fly` is set to `true` the key stored to `key` is used, while if it is set to `false` the key stored to SEcube flash memory associated with the ID `key.id` is used.

The generated ciphertext and other data related to the encryption operation are outputted to `encrypted_data`.

- Important note: `plaintext_size` can be at most `key_size - 2 - 2 * 32` bytes long, where `key_size` is the size of the key to be used in the encryption operation and `32` is the size of the output of the SHA256 hash function used in the encryption operation. Greater `plaintext_size` would raise an error.

  If the user needs to encrypt longer messages he is in charge of splitting messages into shorter chunks. This is discouraged due to performance and security issues [10]: consider using RSA for securely distribute a symmetric key to be used for encrypting the long message.

- Note: `algorithm_mode` is ignored when `L1Encrypt` uses the RSA algorithm.

```
class SEcube_ciphertext{
public:
        se3Key key;
        uint16_t algorithm;
        std::unique_ptr<uint8_t[]> ciphertext;
        size_t ciphertext_size;
        ...
};

void L1::L1Decrypt(SEcube_ciphertext& encrypted_data, size_t& plaintext_size,
        std::shared_ptr<uint8_t[]>& plaintext, uint8_t on_the_fly);
```

When `encrypted_data.algorithm` is set to `L1Algorithms::Algorithms::RSA` the data stored to `encrypted_data.ciphertext` is decrypted using the RSA algorithm. If `on_the_fly` is set to `true` the key stored to `encrypted_data.key` is used, while if it is set to `false` the key stored to SEcube flash memory associated with the ID `encrypted_data.key.id` is used.

The generated plaintext and its length will be stored to `plaintext` and `plaintext_size`.

Digital signature

```
void L1::L1Sign(const size_t input_size, const std::shared_ptr<uint8_t[]> input_data,
                const se3Key key, uint8_t on_the_fly, size_t &sign_size,
                std::shared_ptr<uint8_t[]> &sign);
```

`L1Sign` computes the RSA signature of `input_size` bytes of data stored to `input_data`. The result is stored to `sign` and its size to `sign_size`. If `on_the_fly` is set to `true` the key stored to `key` is used, while if it is set to `false` the key stored to SEcube flash memory associated with the ID `key.id` is used.

```
void L1::L1Verify(const size_t input_size,
                const std::shared_ptr<uint8_t[]> input_data,
                const se3Key key, uint8_t on_the_fly, const size_t sign_size,
                const std::shared_ptr<uint8_t[]> sign, bool &verified)
```

`L1Verify` checks the validity of the RSA signature stored to `sign` and `sign_size` bytes long, associated to the message stored to `input_data` of `input_size` bytes. If `on_the_fly` is set to `true` the key stored to `key` is used, while if it is set to `false` the key stored to SEcube flash memory associated with the ID `key.id` is used. If the signature is valid `verified` is set to `true`, to `false` otherwise.

3.1.1 Use cases

This section contains some code examples of the most common use cases of the RSA library.

See `src/examples` for the complete source code.

Key generation

The following code snippet shows how to generate an RSA key `KEY_SIZE` long (in Bytes) of type `KEY_TYPE` and store it to flash, associating it to `KEY_ID` id.

```
unique_ptr<L1> l1 = make_unique<L1>();

// login to the SEcube device
...
```

```
// specify the key ID, size and type
se3Key key = {
        .id = KEY_ID,
        .dataSize = KEY_SIZE,           // (in Bytes) can be 128, 256, 512 or 1024
        .asymmKey = {
                .type = KEY_TYPE        // can be any value of L1Key::RSAKeyType
        }
};

// send the request for generating the key and wait for the response
l1->L1KeyEdit(key, L1Commands::KeyOpEdit::SE3_KEY_OP_ADD_GEN_RSA);
```

Symmetric key distribution

The following code snippets show how to securely distribute a secret key from the *Sender* to the *Receiver*, according to these steps:

1. *Receiver* reads its public key from flash and sends it to *Sender*.
2. *Sender* encrypts the secret key using the *Receiver* public key and sends it to *Receiver*.
3. *Receiver* decrypts the secret key using its private key.

- *Sender* side:

```
unique_ptr<L1> l1 = make_unique<L1>();

// login to the SEcube device
...

// receive the RSA key to use for encrypting data from Receiver
se3Key asymmKey;
...

SEcube_ciphertext cipher;

// send the request for encrypting the symmetric key and wait for the response
// (assuming SYMM_KEY is a buffer containing the symmetric key sized
// SIMM_KEY_SIZE)
l1->L1Encrypt(SYMM_KEY_SIZE, SYMM_KEY, cipher, L1Algorithms::Algorithms::RSA,
                0, asymmKey, true);

// send cipher to Receiver
...
```

- *Receiver* side:

```
unique_ptr<L1> l1 = make_unique<L1>();

// login to the SEcube device
...

// specify the ID of the RSA key to use for encrypting data
se3Key asymmKey = {
        .id = ASYMM_KEY_ID
};

// send the request for getting the public part of the key to be used for
// encrypting the symmetric key and wait for the response
```

```
// (assuming an RSA key associated with ASYMM_KEY_ID is stored
// to the SEcube flash memory)
l1->L1AsymmKeyGet(asymmKey);

// send asymmKey to Sender
...

// receive cipher from Sender
SEcube_ciphertext cipher;
...

shared_ptr<uint8_t[]> symmKey;
size_t symmKeySize;

cipher.key.id = ASYMM_KEY_ID;

// send the request for decrypting the symmetric key and wait for the response
l1->L1Decrypt(cipher, symmKeySize, symmKey, false);
```

Digital signature

The following code snippets show how the *Signer* signs a message and the *Verifier* verifies it, according to these steps:

1. *Signer* reads its public key from flash and sends it to a trusted key storage (such as a key server).
2. *Signer* signs the message with its private key and sends the original message and its signature to the *Verifier*.
3. *Verifier* receives the message and its signature from the *Signer* and the *Signer* public key from the trusted key storage.
4. *Verifier* verifies the message.

- *Signer* side:

```
unique_ptr<L1> l1 = make_unique<L1>();

// login to the SEcube device
...

// specify the ID of the RSA key to use for signing data
// (assuming an RSA key associated with KEY_ID is stored
// to the SEcube flash memory)
se3Key key = {
        .id = KEY_ID
};

// send the request for getting the public part of the key to be used for
// verifying the message and wait for the response
l1->L1AsymmKeyGet(key);

// send key to the trusted key storage
...

shared_ptr<uint8_t[]> sign;
size_t signature_size;

// send the request for signing the message and wait for the response
```

```
// (assuming MESSAGE is a buffer containing the message sized
// MESSAGE_SIZE)
l1->L1Sign(MESSAGE_SIZE, MESSAGE, key, false, signature_size, signature);

// send MESSAGE, MESSAGE_SIZE, signature and signature_size to Verifier
...
```

- *Verifier* side

```
unique_ptr<L1> l1 = make_unique<L1>();

// login to the SEcube device
...

// receive message, message_size, signature and signature_size from Signer
// and key from trusted storage
shared_ptr<uint8_t[]> message;
size_t message_size;
shared_ptr<uint8_t[]> signature;
size_t signature_size;
se3Key key;
...

// send the request for verifying the signature and wait for the response
bool verified;
l1->L1Verify(message_size, message, key, true,
                signature_size, signature, verified);
```

3.2 SEcube X.509 certificates

The functions manage the certificates stored in the SEcube device. They build the request buffer according to the desired operation and they check and parse the response buffer after the SEcube computations. They throw exceptions in case of errors.

```
struct SEcube_certificate_info{
        uint32_t cert_id;
        uint32_t issuer_key_id;
        uint32_t subject_key_id;
        std::string serial_str;
        std::string not_before;
        std::string not_after;
        std::string issuer_name;
        std::string subject_name;
};
```

`SEcube_certificate_info` is the data structure containing all the information required for the generation of an X.509 certificate. In particular, the fields are:

- `cert_id` is the ID to associate the certificate with, when storing it to SEcube flash memory.
- `issuer_key_id` and `subject_key_id` are the IDs of the issuer and subject keys stored to SEcube flash memory.
- `serial_str` is a string containing the serial number of the certificate (in hexadecimal format).
- `not_before` and `not_after` are strings containing the start and the end of the validity period (in `YYYYMMDDhhmmss` format).

- `issuer_name` and `subject_name` are strings containing comma-separated list of OID types and values (e.g. `"C=UK,O=ARM,CN=mbed TLS CA"`) related to issuer and subject.

```
enum CertOpEdit {
        SE3_CERT_OP_ADD = 0,
        SE3_CERT_OP_DELETE = 1
};

void L1::L1CertificateEdit(const L1Commands::CertOpEdit op,
                const SEcube_certificate_info info);
```

`L1CertificateEdit` is the function for managing certificates, the operation to be executed has to be specified in the first parameter. The certificate can be added or deleted. The second parameter is the certificate. `L1KeyEdit` provides write access to X.509 certificates on SEcube device. It can perform three operations, depending on `op` value:

- `SE3_CERT_OP_ADD`: generate an X.509 certificate according to the information contained in `info` and store it to SEcube flash memory, associating it with the ID specified in `info.cert_id`.
- `SE3_CERT_OP_DELETE`: delete the certificate associated with the ID specified in `info.cert_id` from SEcube flash memory.

```
void L1::L1CertificateGet(const uint32_t cert_id, std::string &cert);
```

`L1CertificateGet` reads from SEcube flash memory the X.509 certificate associated with the ID specified in `cert_id` and stores it to `cert`, in `PEM` format.

```
void L1::L1CertificateFind(uint32_t certId, bool& found);
```

`L1CertificateFind` sets `found` to `true` if an X.509 certificate associated with `certId` is stored to SEcube flash memory, to `false` otherwise.

```
void L1::L1CertificateList(std::vector<uint32_t>& certList);
```

`L1CertificateList` returns the list of IDs of all the certificates stored to the SEcube device flash memory.

3.2.1 Use cases

This section contains some code examples of the most common use cases of the X.509 certificates library.

See `src/examples` for the complete source code.

Certificate generation

The following code snippet shows how to generate an X.509 certificate and store it to flash, associating it with `CERT_ID` id.

```
unique_ptr<L1> l1 = make_unique<L1>();

// login to the SEcube device
...

// specify the certificate information
SEcube_certificate_info info = {
        .cert_id = CERT_ID,
        .issuer_key_id = ISSUER_KEY_ID,
        .subject_key_id = SUBJECT_KEY_ID,
```

```
        .serial_str = SERIAL,
        .not_before = NOT_BEFORE,
        .not_after = NOT_AFTER,
        .issuer_name = ISSUER_NAME,
        .subject_name = SUBJECT_NAME
};

// send the request for generating the key and wait for the response
l1->L1CertificateEdit(L1Commands::CertOpEdit::SE3_CERT_OP_ADD, info);
```

APPENDIX A

SEcube SDK bugs

During the developement phase some bugs in the SDK code have been identified, fixed and reported to the maintainer.

A.1 SEcube firmware

A.1.1 `se3_key_read`

- *Severity*: critical
- *Cause*: buffer over-read: a wrong offset was used when reading from a buffer, therefore some useful data were ignored and some garbage values were read instead.
- *Consequence*: broken initialization of all algorithms requiring cryptographic keys.

A.1.2 `key_find`

- *Severity*: minor
- *Cause*:
  - buffer over-read: a wrong offset was used when reading from a buffer, therefore some useful data were ignored and some garbage values were read instead.
  - reversed response: the firmware sent 0 in case of hit and 1 in case of miss, but the L1 libraries expected the opposite.
- *Consequence*: broken `L1::L1FindKey`.

Bibliography

[1] K. Moriarty. Tech. rep. 2016. URL: https://datatracker.ietf.org/doc/rfc8017/.

[2] T. Blum and C. Paar. "Montgomery modular exponentiation on reconfigurable hardware". In: (1999). DOI: 10.1109/ARITH.1999.762831.

[3] Yuan-Yang Zhang et al. "An efficient CSA architecture for montgomery modular multiplication". In: (2007). ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2006.12.003.

[4] D. Narh Amanor et al. "Efficient hardware architectures for modular multiplication on FPGAs". In: (2005). DOI: 10.1109/FPL.2005.1515780.

[5] Alan Daly. "Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic". In: (2002). DOI: 10.1145/503048.503055.

[6] A.F. Tenca and C.K. Koc. "A scalable architecture for modular multiplication based on Montgomery's algorithm". In: (2003). DOI: 10.1109/TC.2003.1228516.

[7] ARMmbed. *mbedtls*. URL: https://github.com/ARMmbed/mbedtls.

[8] libtom. *libtomcrypt*. URL: https://github.com/libtom/libtomcrypt.

[9] Thomas Pornin. *BearSSL*. URL: bearssl.org.

[10] *What's the best block cipher mode of operation for RSA?* https://crypto.stackexchange.com/questions/66314/whats-the-best-block-cipher-mode-of-operation-for-rsa. Accessed: 2021/09/09.