POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

A security service provider for applications based on the SE*cube*™ Platform

Supervisor
prof. Paolo Prinetto

Candidate
Andrea Rubiolo

December 2021

Abstract

We live in times where Information Technology (IT) is constantly becoming more pervasive and ubiquitous, deeply permeating our society, from the citizens' private life to essential services, from industry to politics. The rapid development of new IT technologies brings great benefits, but poses even greater challenges and risks.
Cybersecurity is the discipline that deals with protecting valuable assets (i.e., data, devices, infrastructures, etc.) from risks such as unauthorized access or information disclosure. Cybersecurity is a process consisting of a series of practices, technologies and operations carried on by different means, ranging from conforming to simple rules (e.g., enable 2-factor authentication) to the usage of certified hardware devices aimed at granting specific levels of security.
Hardware Security Modules (HSM) are physical devices commonly used to safeguard valuable assets by means of dedicated capabilities in the realm of cryptography (i.e., strong and efficient data encryption, secure cryptographic key management, etc.) and secure data storage (i.e., anti tamper properties).
The **SE***cube*™ (Secure Environment cube) is an HSM designed according to a holistic approach to cybersecurity. The **SE***cube*™ is used in both civil and military applications; it offers an open source hardware design and both proprietary software and an open source SDK developed by European research institutions. Despite this, the open source SDK is affected by a partial lack of coherence and usability: it involves several libraries, tools and APIs that determine a significant learning curve to start leveraging the advantages of the security platform.

This thesis aims at improving the usability of the **SE***cube*™ platform by developing an intermediate software layer having two primary goals:
i) acting as a "logic glue" between the multiple libraries, tools, and APIs of the **SE***cube*™ SDK;
ii) acting as an easy-to-use interface towards the cybersecurity capabilities

of the **SE***cube*™ Security Platform.
The software developed in this thesis is compatible with Windows and Linux operating systems; it consists of multiple elements, including a GUI, that allow third-party applications to interact with the HSM using a standardized interface based on a simple request/response protocol.
Whenever a third-party application requires a functionality provided by the **SE***cube*™ HSM (i.e., data encryption), it can access that functionality thanks to the standardized interface provided by the software developed in this thesis. The software automatically deals with the complexity deriving from the interaction with the HSM, greatly reducing the learning curve of the **SE***cube*™ Security Platform.
This application leads to a simple, unified, and user-friendly environment that is compliant with strict security standards. This enables any developer who is willing to make use of the **SE***cube*™ Security Platform to implement applications that leverage strong cybersecurity capabilities.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The extensive digitization of services is impacting our lives on a daily basis, influencing processes of several fields, from the private life to public contexts. The development of new technologies leads to great benefits, but constitutes significant risks too.
Cybersecurity is the discipline that faces the problems related to protecting precious resources, such as devices and infrastructures, employing different kinds of technologies, practices and actions. Cybersecurity is a complex topic that requires specific expertise and know-how in order to leverage its advantages: during the development of secure tools the help of skilled personnel might be required, increasing costs and developing time.
Hardware Security Modules (HSM) are systems that integrate security processes in a comprehensive and seamless way, following the so-called holistic security approach. These platforms allow both developers and users to focus on exploiting the supported security services at a higher level, demanding the complex implementation and maintenance to security experts.
This thesis deals with the design and development of an application that improves the usability and reduces the learning curve of the **SE***cube*™ Security Platform. The **SE***cube*™ Security Platform is an instance of an HSM, a special-purpose device capable of performing cybersecurity functionalities (i.e., data encryption, key management system, etc.).
Platforms of this kind are worthwhile when arise the need of securing a local environment, where a general-purpose device does not provide an high level of security by itself and a dedicated reinforcing is required. It is necessary to point out that the **SE***cube*™ platform is not a stand-alone device, instead it cooperates with host machines (i.e., personal computers) that can provide the necessary resources.

The need to work on this particular topic can be better understood by exploring the details of the following themes:

1. The **SE***cube*™ environment complexity;
2. Pros and cons of adding an intermediary between the platform and the user.

The **SE***cube*™ platform offers an open source hardware design and an open source SDK. Over time, a considerable amount of libraries and APIs have been developed and included into the project, extending the functionalities provided by the platform. The problem, however, is that libraries and APIs contribute to increment the complexity of the platform, because they all have their own protocols and conventions.
The software architecture is structured into abstraction layers, generally referred to as L0, L1 and L2: each layer is presented as a service for the upper one and relies on services provided by the lower one.
The first level, L0, provides low-level functionalities to communicate with the **SE***cube*™ ; L1 implements login/logout functions and cryptographic algorithms. Finally, L2 is composed of three main libraries: **SE***key*™ , a key management system library; **SE***file*™ , a file system interface library; **SE***link*™ , a library focused on securing data in motion.
Furthermore, when an application needs a feature provided by the device, it is first necessary to manually include the required libraries; then the developer has to deal with the complexity of communicating with the device in the appropriate manner, writing code to integrate the platform capabilities into the application.
The **SE***cube*™ system is a sophisticated environment, growing in complexity as it is growing in size, and requires some time to be understood and to be used properly. As a matter of fact, an incorrect usage may lead to extend the developing phase delaying the release, or even worse to flaws that could compromise the integrity of the application and consequently of the whole system.
Given these premises, adding an intermediate layer between the platform and the user acting as a standardized interface (i.e., a middleware) leads to positive consequences on:

- scalability and flexibility, because it simplifies the process of including new functionalities;
- security, thanks to the 'security by design' principle the new layer has been designed to be foundationally secure;

- fault reduction;
- automation of recurrent routines.

On the other hand, this solution brings some disadvantages, such as:

- a slow-down factor to the execution time of applications;
- it might be necessary to rewrite parts of pre-existing code to integrate the new layer;
- it might require a somewhat long time to develop an adequate solution;
- new **SE***cube*™ functionalities will require additional time to be supported.

The application developed in this thesis is written in C/C++ and Qt, it acts as a "bridge" between the platform and third-party applications (also referred to as clients). Its main tasks are:

- to serve as a GUI to users, where they can connect to a **SE***cube*™ device and see details about the incoming requests from external applications (i.e., a logger);
- to authenticate client applications;
- to handle the requests coming from clients, acting as a scheduler.

One of the prerogatives of this work is to reach a level of security comparable with the one of the **SE***cube*™ platform. To allow communication between third-party applications and the middleware, several analysis on Inter Process Communication (IPC) methods were held, leading to the conclusion that there is not a intrinsic secure approach [3]: some of them (i.e., TCP sockets) send messages in clear and suffer from sniffing attacks; others, such as Unix-Domain sockets, are affected to unprivileged access from potentially malicious processes.
The solution adopted in this thesis is to use a TCP socket channel along with symmetric encryption. The communication is encrypted using AES-256 and session keys, in order to provide data confidentiality, and it is based on a simple request/response protocol: third-party applications send requests for a specific feature using a predefined format; the middleware arranges the

request in order to be compliant with the **SE***cube*™ standards and, once it is available, it sends back a response. Thereby, a typical "star architecture" is formed, with the middleware acting as the central node and applications as terminal ones.
This improvement allows to build a simple and user-friendly environment, which is at the same time compliant with strict security standards. Developers approaching the platform do not have to worry about low-level security details, nor they need to have a strong knowledge about cybersecurity principles and the **SE***cube*™ ecosystem.

The remaining of the thesis is organized as follows:

Chapter 2 gives an overview of the prevailing HSMs and their features, as well as the main Inter Process Communication (IPC) methods.

Chapter 3 provides a detailed view about the **SE***cube*™ Security Platform behaviour and its libraries.

In chapter 4 the application design and implementation processes are explained, covering the details of each key component, showing also relevant portions of code.

In chapter 5 the obtained result are presented, along with the problems faced and some use case models.

Finally, chapter 6 presents the conclusions and explores possible improvements.

Furthermore, appendix A provides a detailed list of the **SE***cube*™ functionalities supported by this work.

Chapter 2

State of the art

This chapter discusses the general features of hardware security modules, as well as their advantages and disadvantages. Moreover, since the software developed in this thesis involves the cooperation of multiple processes on the same host machine, inter-process communication methods (IPC) are analyzed as well.

2.1 Types of HSMs

The hardware security modules market offers a variety of solutions that differ on several aspects, such as certifications (i.e. FIPS [9]), cryptographic capabilities, performance, and cost. HSMs can be catalogued as proprietary or open source. The latter can be furthermore divided into three categories:

- System robustness evaluators: these modules focus on performing tests with the goal of exposing weaknesses, evaluating systems protection against external physical attack. SASEBO boards [18], for example, measure strength over side-channel attacks that are used to steal confidential data.

- Secure SW development supporters: platforms such as the Juno Development Board [5], that provide an environment to create general purpose software applications, supporting advanced security features.

- Embedded SoC: single chips that usually combine a CPU, an FPGA and a smart card that allow to implement cryptographic functionalities, like the Zynq board proposed by Xilinx [22].

Considering these three categories, Embedded SoC are, by far, the most capable. They usually provide a combination of hardware and software that is suitable to many different environments; however, they are usually expensive, meaning that prototyping based on Embedded SoC might not be always feasible. System evaluators, on the other hand, only act as guardians that do not implement any kind of cybersecurity functionality. Finally, Secure Software Development Supporters are general-purpose systems whose security elements usually cannot be controlled by the customer.
Another important distinction can be made among general-purpose and payment hardware security modules. The three categories previously described fall into the general-purpose HSMs, a macro-class that includes all the devices intended to serve different scenarios (i.e., government, military, companies, advanced users). Contrariwise, payment platforms are focused on the financial, banking, and commercial sectors.
Finally, since the cloud paradigm has become essential for many businesses, cloud-based HSMs are also available. In this case, they are usually defined as HSM-as-a-Service and provide services for:

- Rent a physical HSM in a off-site data center;
- Access a virtual environment sharing an HSM;
- Retrieve security functionalities of HSM vendor's devices.

This methodology cuts down a significant portion of expenses and allows customers not to manage the complexity that HSMs imply. On the other hand, customers do not have physical control of the devices.

Notice that the **SE***cube*™ platform is an example of general-purpose embedded SoC composed of three elements (a CPU, an FPGA and a smart card); furthermore, it provides an abstraction layer independent from the cryptographic operations, differently from other HSMs that use common packages (i.e., PKCS#11 [29] interface in public key infrastructures).

2.2 Inter-Process Communication Methods

An important aspect that must be considered for the implementation of this work is how to let communicate two or more applications (or processes) running on the same host machine. Modern operating systems allow processes to share data in several ways, named Inter-Process Communication (IPC) mechanisms. Normally, these mechanisms are adopted to implement a client-server connection, where clients request data and servers share them. Microsoft provides a list of questions [21] that can help to determine whether an application can benefit by using an IPC method and which one fits best:

> 1. *Should the application be able to communicate with other applications running on other computers on a network, or is it sufficient for the application to communicate only with applications on the local computer?*
> 2. *Should the application be able to communicate with applications running on other computers that may be running under different operating systems (such as 16-bit Windows or UNIX)?*
> 3. *Should the user of the application have to choose the other applications with which the application communicates, or can the application implicitly find its cooperating partners?*
> 4. *Should the application communicate with many different applications in a general way, such as allowing cut-and-paste operations with any other application, or should its communications requirements be limited to a restricted set of interactions with specific other applications?*
> 5. *Is performance a critical aspect of the application? All IPC mechanisms include some amount of overhead.*
> 6. *Should the application be a GUI application or a console application? Some IPC mechanisms require a GUI application.*

The answers to these question, keeping in mind the application that has to be built in this thesis, can be summarized as follows:

1. It is sufficient for the application to communicate only locally.

2. The application should be able to communicate with UNIX systems as well.

3. The application should be able to implicitly find its cooperating partners.
4. The communication is limited to a restricted set of interactions.
5. Performances are not a key point, however the application should not add too much overhead.
6. The application should provide a graphical interface.

These answers allow to reduce the set of IPC methods that cover the requirements of this works, whose details are analyzed in the following sections.

2.2.1 Sockets

A socket is a structure that acts as an end-point in a bidirectional channel, used to send and receive data, either locally or over networks. The lifetime of the socket is bound to the process that created it: when a process is destroyed, its sockets are destroyed too.
Network protocols define an Application Programming Interface (API) that specifies, among the others, the structure and properties of sockets. Moreover, the API is used to manage two fundamental structures: the socket address and the socket descriptor. The former is a combination of the network protocol adopted, the IP address of the counterpart and a port number, used to recognize the exact process we are communicating with; the latter is a unique identifier stored in the application's process memory, used to handle locally the existing sockets. The main disadvantage that this mechanism brings is that data are sent in clear, hence any kind of sniffing attack can be performed.

Sockets can be divided into three main categories:

- **Datagram Sockets**: the so-called connectionless sockets, which use the User Datagram Protocol (UDP). In this kind of implementation, there is no connection setup affected by the UDP layer, so packets are individually addressed and routed. It does not implement reliability or ordering functions: packets sent might arrive in a different order, or they might not arrive at all. Voice and video traffic allow occasional packet loss, hence it is generally transmitted using UDP.

- **Stream Sockets**: contrary to the previous ones, these are connection-oriented sockets which use Transmission Control Protocol (TCP). Stream sockets first establish a connection among the two communicating hosts, then data can be sent. There are no record lengths or boundaries, so a mechanism for distinguish information is needed (i.e., using TLV format). Since TCP guarantees reliability, packets ordering and no duplications, it is commonly used in those applications that require all data to be delivered, such as file transfer or emails.

- **Raw Sockets**: often used to test new protocol implementations, these sockets allow sending IP packets without any protocol-specific format. Raw sockets are usually used in security-related applications or in routing protocols such as Internet Group Management Protocol (IGMP).

2.2.2 Unix domain sockets

Unix Domain Sockets (UDS) are similar to the previously presented sockets, but data are exchanged only between processes running on the same host machine. They support both connectionless and connection-oriented protocols, while raw socket protocols are not supported.
The API for Unix Domain Socket is similar to the one of network sockets, but communication happens entirely in the OS kernel, instead of using a network stack. Hence, there is no need of checksum calculations or heavy headers, unlike TCP which requires them, allowing communication to be faster and less expensive in terms of computational resources.
Processes that implement UDSs use local files to communicate; socket descriptors are now common file descriptors, so the operating system controls who has read/write permissions on those files. The great advantage that this approach brings is that developers can arbitrarily assign access permission to a specific UDS file, allowing only certain users to use it.
The main problem of this solution is that file permissions can be managed at user level or group level only. This means that malicious processes running under a user (or group) allowed to access a UDS can access that UDS too, reading or writing data that might be private. Another problem related to this aspect is that in UNIX systems there is not an effective and portable kernel functionality that allows to get information about the user and his group (i.e., `int getpeereid(int s, uid_t *u, gid_t *g)` that retrieves user and group ids) [4].

2.2.3 Anonymous pipes

Anonymous pipes, sometimes simply called pipes, are unidirectional data channels that allow communication between two processes on the same host machine; the communication is held using standard input and output. This approach is typically used in multi-threaded applications, where the parent process opens a pipe and then creates one or more children processes that inherit the other end of the pipe. Bidirectional communication requires two anonymous pipes, otherwise data are written by one process to the write-end and buffered by the operating system until another process reads them from the read-end of the pipe.
In Unix-like operating systems, when creating a new pipe (i.e., by calling the `pipe` system call) a pair of file descriptors are returned: one refers to the read-end, while the other refers to the write-end, and any process that has a copy of the pipe handler can access it. This vulnerability could be exploited by an attacker that can enumerate a process handler table and duplicate a pipe handler poorly configured, leading to unauthorized interaction with the other end of the pipe [1].
I/O operations are, by default, blocking functions: when a process tries to read from an empty pipe, it is blocked until some data are available; if a process attempts to write to a full pipe (every byte is occupied), then it is paused until sufficient data has been read from the other end.
Pipes can also be applied in many modern operating systems to implement pipelines in shells. A pipeline is a set of processes chained one next to the other, so that the output of a process is served as input of the next one. Processes are executed concurrently, starting from the first; as soon as the result of the first process is ready, the second one can start. This mechanism lasts until all the processes have completed their operations: in that moment, the pipe is destroyed. Each command in a shell is executed by a process, the list of commands are stringed together, separated by the pipe character ('|') in order to form a pipeline.
The following example presents a pipeline that searches the PID of a process that is running Google Chrome [7] and kills it:

```
$ ps aux | grep chrome | awk '{print $2}' | xargs kill -
9
```

The result of each command is used as input of the next one, in particular:

- ps aux: outputs the list of running processes with some details;
- grep chrome: search the string "chrome" in the processes list;
- awk '{print $2}': the second world of each line in the processes list is the PID. This command takes that world;
- xargs kill -9: read from input (the result of the previous command) the PID and kills the process with that specific value.

2.2.4 Named pipes

Named pipes (also known as FIFO, since their behaviours are similar) are an expansion of the previously enunciated anonymous pipes, and are used as IPC mechanism to exchange data locally among processes on the same host machine. Instead of using standard input and output, named pipes use files similarly to Unix Domain Sockets: processes can append data to a file and read data from the first byte of it. Named pipes, however, use unidirectional channels, hence full-duplex communication requires two instances.
The great advantage of this mechanism is that it is not bound to processes lifetime, but it can last as long as the system is up; however, it can be deleted if it is no longer useful.
Windows named pipes differ from Unix ones in some aspects, in particular:

- No command line interface is available (except for PowerShell);
- Windows named pipes cannot be created as files under normal filesystems;
- Windows named pipes are volatile: once the last reference to them is closed, they are deleted.

This method, however, is affected by some vulnerabilities too, such as client impersonation in which named pipe server (the process that created the pipe) temporarily assumes the security context of the client, by using the `ImpersonateNamedPipeClient` API, a Windows feature that helps performing operations based on privileges of the client. This kind of action, if exploited properly by malicious applications, might let them attain full control of the system when connected to clients that have high privilege levels.

Chapter 3

The **SE***cube*™ Security Platform

This chapter provides an overview about the **SE***cube*™ Security Platform. Knowing the technology that is embedded into the **SE***cube*™ Security Platform is important to understand the details of the software that has been developed in this thesis, both in terms of implementations details as well as security features.

3.1 The **SE***cube*™ Hardware Security Module

The **SE***cube*™ Security Platform [23] is an open-source cybersecurity platform whose core is the **SE***cube*™ HSM, a hardware security module designed and produced by Blu5 Group [12]. The **SE***cube*™ is basically a small chip, a compact 3D SiP (System-in-Package) made of three main components: a low-power ARM Cortex-M4 processor, a high-performance FPGA, and an EAL5+ certified security controller (also referred to as smartcard). Figure 3.1 illustrates the composition of the **SE***cube*™ chip.
The main hardware products offered by **SE***cube*™ are:

- **SE***cube*™ Chip, above described;
- **SE***cube*™ DevKit, a development board;
- **USE***cube*™ Stick, a USB stick;

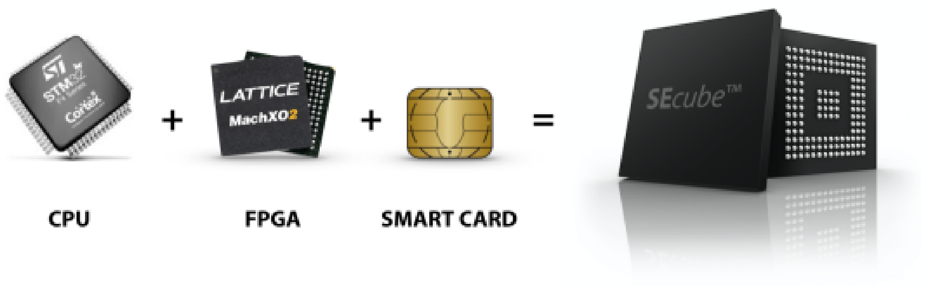
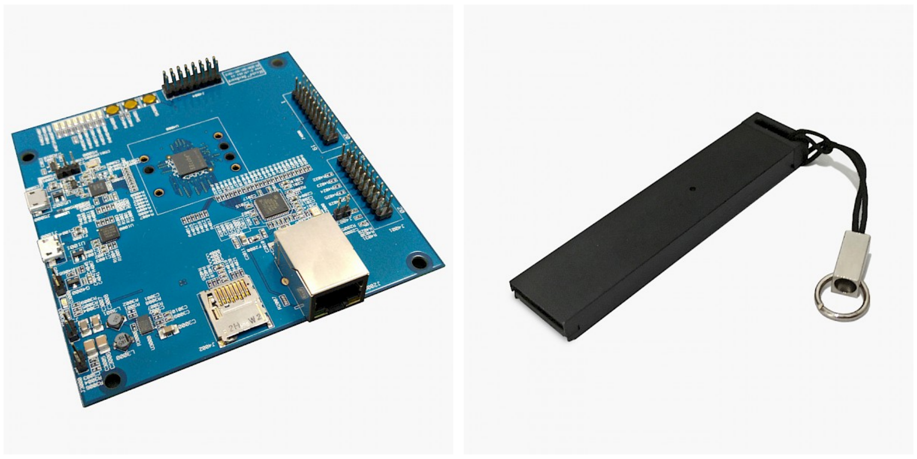


Figure 3.1: The components of the **SE***cube*™

The **SE***cube*™ DevKit, shown in Figure 3.2a, is a development board enabling the **SE***cube*™ Chip integration into hardware and software projects. It offers several peripherals that allow users to easily program and debug the chip, such as a USB 2.0 to UART port, a microSD card and an Ethernet 10/100 connection. Furthermore, it has a JTAG interface to flash new firmware versions on the device.
The **USE***cube*™ Stick (Figure 3.2b) is a multi-purpose USB 2.0 token that provides all the **SE***cube*™ functionalities in a small package, compatible with any operating system. For security reasons, rather than providing a JTAG interface, it comes with an embedded secure bootloader, useful to prevent unsolicited firmware manipulations.



(a) (b)

Figure 3.2: The **SE***cube*™DevKit (a) and the **USE***cube*™ Stick (b)

The CPU embedded into the **SE***cube*™ Chip is the STM32F429 [6], a high-performance ARM Cortex M4 RISC processor produced by ST Microelectronics. The chip provides the following features [20]:

- 256 KB of SRAM,
- 180 MHz clock frequency,
- 32-bit parallelism,
- 2 MB flash memory,
- Low power consumption.

Moreover, the CPU provides a True Random Noise Generator (TRNG) embedded unit, a hardware oriented mechanism of MPU (Memory Protection Unit), and a set of executions mode that differ in privilege level, which allow custom security approaches (i.e. key generation functions barriers, protected memory areas, etc.). Furthermore, it supports CMSIS-DSP libraries, to speed-up the execution of algorithms involving heavy floating-point operations.

The FPGA component is a Lattice MachXO2-7000 device based on a fast and non-volatile logic array, connected to the CPU through a 16-bit internal bus. It provides 47 general-purpose I/Os that can be used as a 32-bit external bus able to transfer data at 3.2 Gb/s [20, p. 15].

The third and last component of the **SE***cube*™ Chip is an SLJ52G EAL5+ certified security controller produced by Infineon [8]. It is connected to the CPU via a standard ISO7816 interface, and it does not provide any interface outside the chip, providing high-grade security primitives.

3.2 The SE*cube*™ software architecture

The **SE***cube*™ offers an open SDK [27] composed of a set of libraries for both the device and the host machine. The standard behaviour is well exposed in Figure 3.3: end-user applications interact with the host libraries, generating a request (i.e., a plaintext to be encrypted). The request is passed to the device side, where the provided libraries actually execute the code needed to satisfy the request; finally, the result is sent back to the host as a response (i.e., the ciphertext), which is presented to the application.

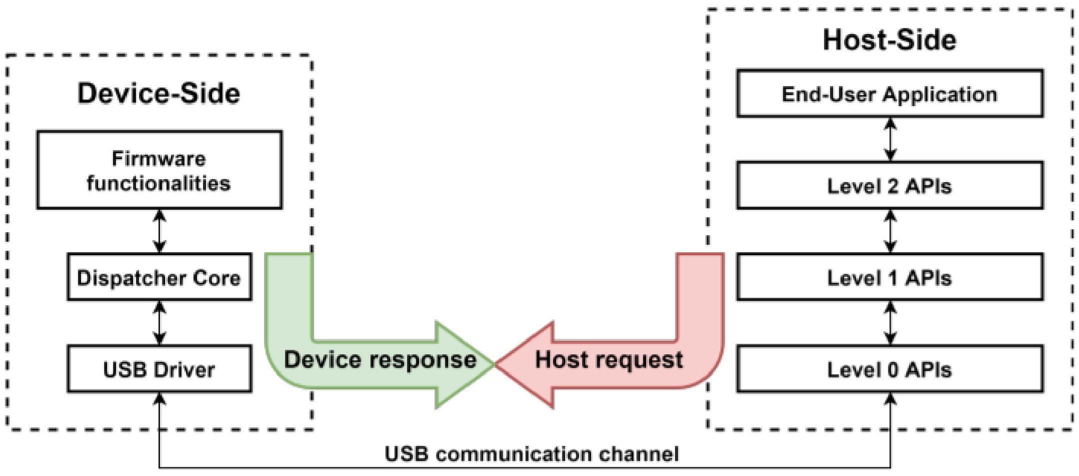


Figure 3.3: Request/response flow

Figure 3.3 shows that the SDK is composed of different elements and levels that are analyzed in detail in the following sections.

3.2.1 SE*cube*™ Device-Side APIs

The firmware is the core of the **SE***cube*™ platform. It is responsible of handling the requests coming from the host side, since the board is not provided with any operating system. The firmware performs a master-slave schema, where the **SE***cube*™ acts as the slave with respect to the host.
The firmware implements a basic request-response mechanism, where requests coming from the host side are organized according to an ID field that allows to identify the request type among the ones supported by the **SE***cube*™ . Since the firmware works in a sequential way, requests can be served one at a time: this is an important limitation that should always be kept in mind when developing host-side applications.
This drawback is a significant bottleneck in term of performances considering an architecture that must support a high load. For the purpose of this work, hence, having a centralized interface that cannot process requests concurrently is a notable limitation.
The firmware is divided into sub-modules, each of which deals with a specific set of elements, in particular:

- The Security Core manages the cryptographic functionalities (i.e., AES-256).

- The Communication Core processes data incoming from the host and prepares data that must be sent to the host.

- The **SE***key*™ Core handles the device-side interaction with the internal Key Management System (KMS) implemented through the **SE***key*™ library.

- The Smart Card Driver and the FPGA Driver deal with the interaction with, respectively, the smartcard and the FPGA.

Notice that the Smart Card Driver is not yet implemented, due to the sepcific know-how required to develop security modules of this kind, while the FPGA Driver is not exploited in the default configuration of the firmware. As a consequence, every implemented algorithm is software-based and does not leverage any kind of hardware enhancement that an FPGA would bring.

3.2.2 SE*cube*™ Host-Side APIs

The host-side software libraries are implemented into three hierarchical abstraction layers. Each layer provides a set of APIs and and it is built in order to represent a "service" for the upper level and to rely on "services" provided by lower ones [20, p. 23]. These layers are organized as follows:

- Level 0: the lowest layer implements basic functionalities such as communication with the platform, platform initialization, and so on. It is additionally divided into three families of APIs: provisioning, communication and commodities APIs.
  This layer is responsible of allocating and keeping up the communicating channel created through the USB interface.

- Level 1: this layer exploits basic security APIs required by secure applications, in particular: login/logout functions, key management functions (add/delete key) and cryptographic algorithms.

- Level 2: it offers optimized and easy-to-use APIs to create a secure environment. L2 APIs are part of some library whose integration must be effortless, without being forced to care about low-level details. This level includes three main libraries: **SE***key*™ [13], **SE***file*™ [14] and **SE***link*™ [15]. Since the software developed in this thesis is not designed to interact with **SE***key*™ , no further details about it are given.

Furthermore, a fourth layer (Level 3) has been added, it relies on L2 and provides advanced security APIs. The only library belonging to this layer is the Secure Database Library [17], it implements an encrypted SQL database to securely manage a database using the standard SQLite [28] C interface.
A complete overview of the software architecture, including both sides, is presented in the Figure 3.4. It clearly demonstrates how the host-side is developed in an hierarchical way, while the firmware is more fragmented and elaborated.

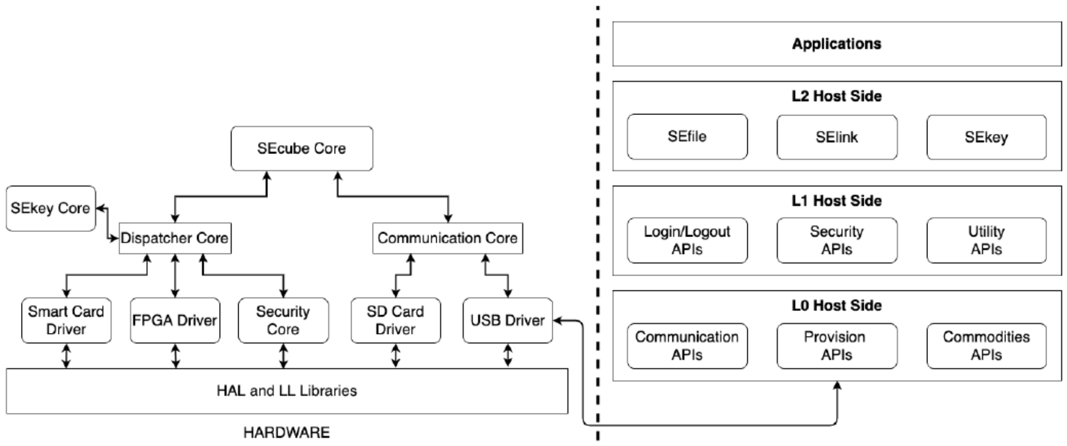


Figure 3.4: **SE***cube*™ complete software architecture

3.2.3 SE*file*™

Handling encrypted files in a host machine is a non trivial operation due to several security issues that might arise. The most significant one regards how and where to store the encryption key: the machine is not intrinsically reliable, hence the key cannot be stored in clear. Not storing the key and derive it from a password is not a feasible solution, considering that it only moves the problem from the machine to the user: the chosen password might be not strong enough and dictionary attacks could always be performed. The **SE***file*™ library natively solves this problem: keys are random-generated and are handled automatically by the **SE***cube*™ device, in such a way that they can be exploited regardless of the machine they are used on; moreover, files are permanently stored encrypted.
SE*file*™ is a L2 host-side library that implements a file system interface specific for the **SE***cube*™ platform, an OS-independent cryptographic wrapper

around the standard file system. It improves data-at-rest security, encrypting files and keeping them encrypted while used by third-party applications. **SE***file*™ only adds a small overhead to the original file, appending an identifier used to recognize the key used to encrypt. Figure 3.5 depicts a general overview of how **SE***file*™ works.

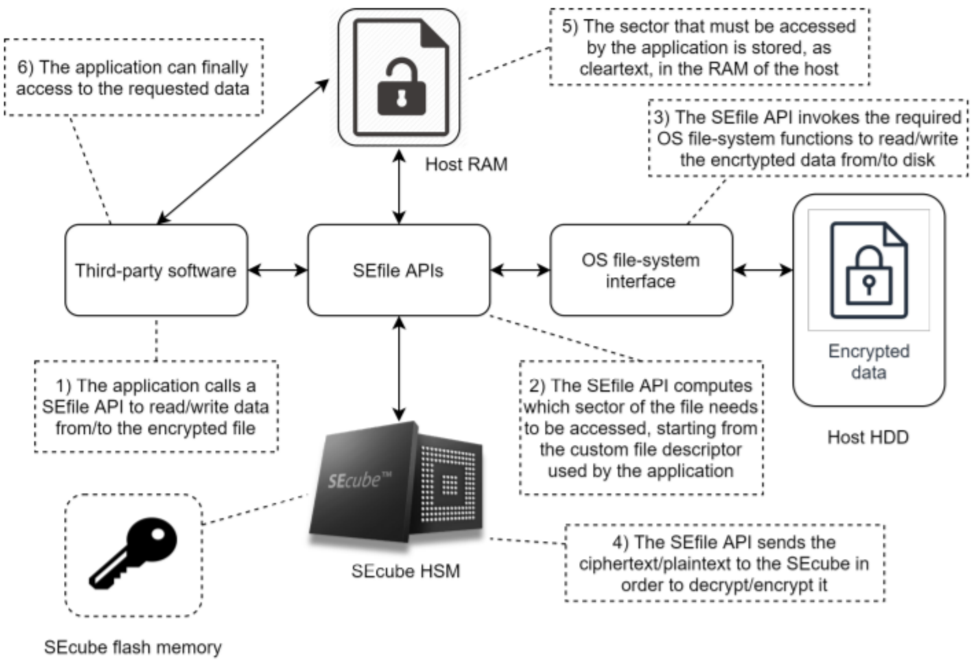

5) The sector that must be accessed by the application is stored, as cleartext, in the RAM of the host
6) The application can finally access to the requested data
3) The SEfile API invokes the required OS file-system functions to read/write the encrtypted data from/to disk
Host RAM
Third-party software
SEfile APIs
OS file-system interface
Encrypted data
Host HDD
1) The application calls a SEfile API to read/write data from/to the encrypted file
2) The SEfile API computes which sector of the file needs to be accessed, starting from the custom file descriptor used by the application
SEcube™
4) The SEfile API sends the ciphertext/plaintext to the SEcube in order to decrypt/encrypt it
SEcube HSM
SEcube flash memory

Figure 3.5: **SE***file*™ overview [19, p. 26]

A file protected with **SE***file*™ is divided into sectors: the first one is a header composed of metadata (i.e., name of the file and size, algorithm used, key ID, etc.), then a sequence of sectors contain the actual data. When a user requires to read a portion of a file, **SE***file*™ computes which sectors contain the required content considering the overhead added; those sectors are then decrypted and stored in the RAM memory. This approach lead to reduce the overhead time, maintaining a secure interaction.
The size of sectors can be customized, with the default size being 512 bytes. Each sector, except for a portion of the header sector, is encrypted. Moreover, in order to guarantee data integrity and authentication, each sector is signed with a signature computed using the HMAC-SHA256 algorithm. Finally, the name of the file is replaced with the SHA-256 digest of the original name, which is useful to increase entropy and obfuscation of information. The original filename, however, is maintained in an encrypted segment of

the header sector.
It is important to notice that padding might be required to fill the empty space in a sector that is not fully used. In order to avoid known plaintext attacks, **SE***file*™ uses random padding rather than using always the same pattern (i.e., zero padding). The typical structure of how **SE***file*™ builds files is illustrated in Figure 3.6.

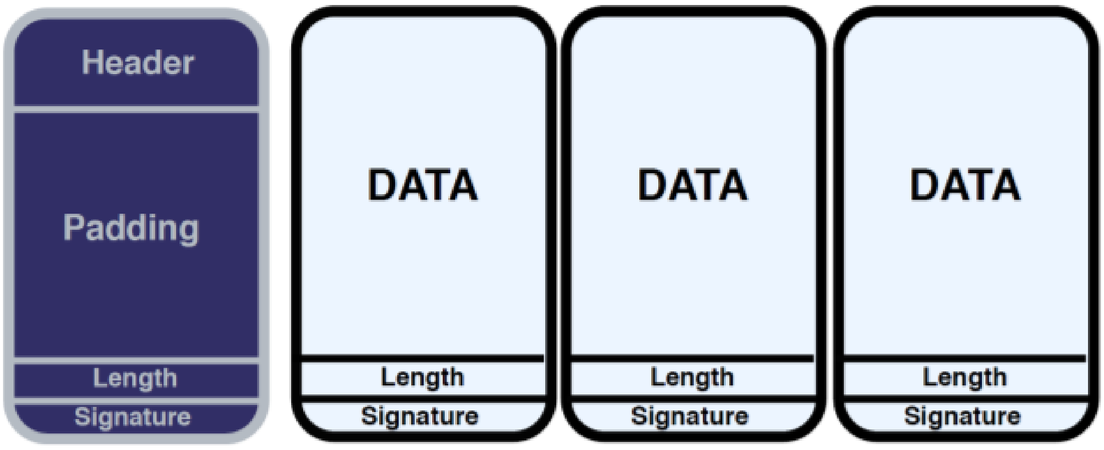


Figure 3.6: File organization under **SE***file*™ [30, p. 22]

3.2.4 SE*link*™

While **SE***file*™ focuses on data at rest, **SE***link*™ is a library developed to secure data in motion. The typical scenario **SE***link*™ was designed for, is an end-to-end communication channel among multiple parties. This library is independent from the protocol used to communicate and it offers specific APIs to serialize data to transmit them using a network protocol (i.e., TCP). Moreover, in order to guarantee confidentiality, integrity and authentication, data are encrypted using AES256-HMAC-SHA256. This library can be compared to a VPN, in which each exchanged message is guaranteed to be secure. A simplified overview of how **SE***link*™ works is shown in Figure 3.7.
It is important to point out that **SE***link*™ is not a tool used on top of preexisting software, it needs custom applications developed specifically to work with **SE***link*™ ; as a matter of fact, **SE***link*™ cannot be used without a **SE***cube*™ device.
The great advantage of this library is that it is easy to use; the provided APIs are only a few and developers do not need to care about low level security

details, thus it is simple to overcome the learning curve. On the contrary, it requires an effort to integrate the library into existing software.

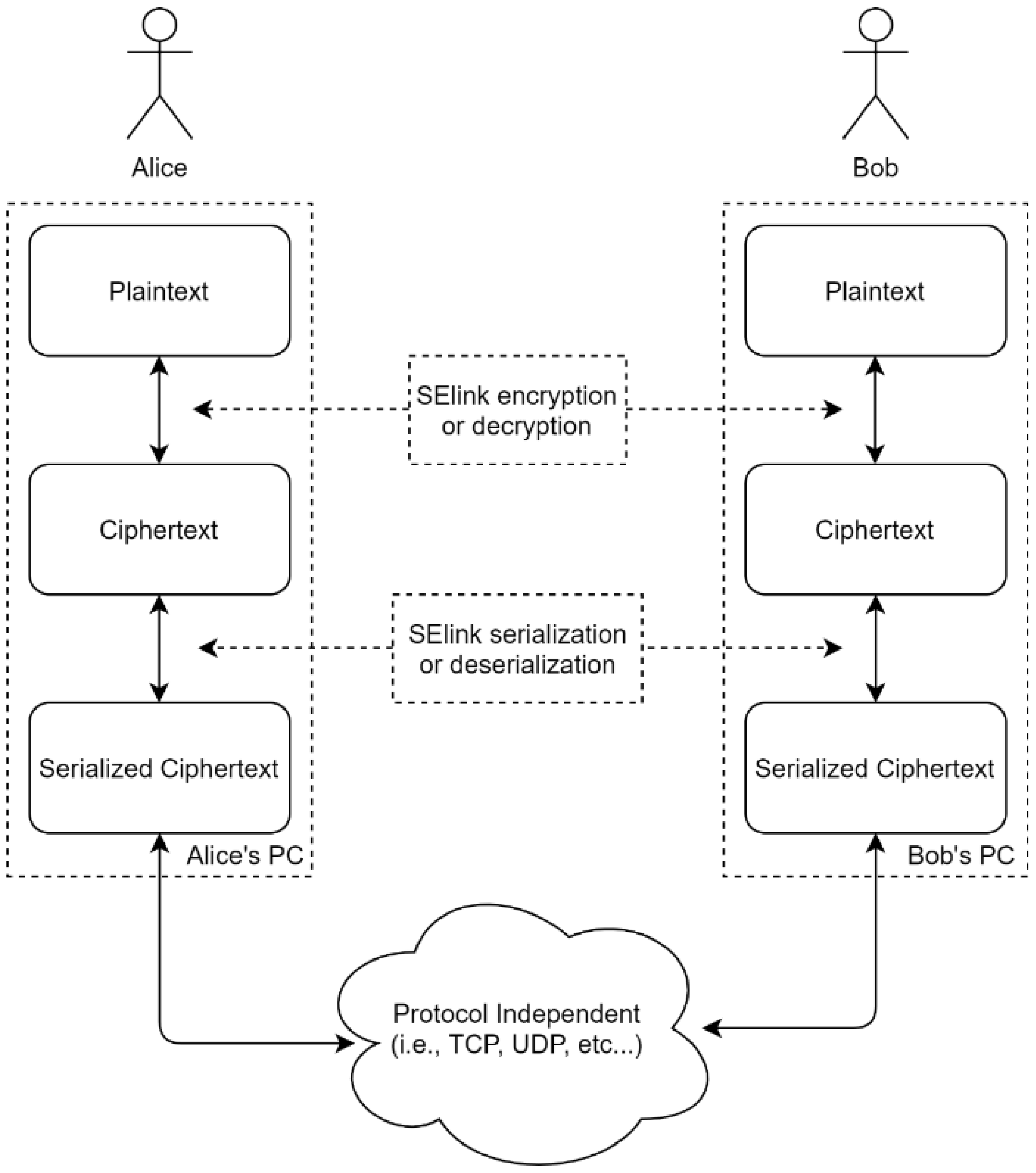

Alice
Bob
Plaintext
Ciphertext
Serialized Ciphertext
Alice's PC
Plaintext
Ciphertext
Serialized Ciphertext
Bob's PC
SElink encryption or decryption
SElink serialization or deserialization
Protocol Independent (i.e., TCP, UDP, etc...)

Figure 3.7: Simplified **SE***link*™ overview

Chapter 4

Application development

This chapter discusses the implementation of the software developed in this thesis, providing insights about the internal structure.
First, the main reasons that brought to the implementation of the software are described; then, the architecture and the implementation of the application are discussed.

4.1 Development motivations

The development of this work was mainly driven by the need of having a simple interface that manages different kinds of applications which rely on the capabilities of the **SE***cube*™ HSM, without having to deal with its low-level APIs, such as the ones included in L0 and L1.
Exploiting the functionalities of the **SE***cube*™ requires to manually deal with multiple libraries; developers need to study how libraries work and how to integrate them with existing software. Moreover, **SE***cube*™ libraries might be complex for developers who do not have a decent background in cybersecurity, leading to possible errors in the implementation and security flaws. The goal of this thesis is to move to an approach that is simpler, more scalable and less time consuming.
Another significant problem is that libraries, tools and APIs of the **SE***cube*™ SDK are not fully-integrated: most of the time, each library acts as "stand-alone" and there is not a system that encompasses all the functionalities together providing a standardized interaction. This leads, once again, to poor flexibility and scalability.
These few examples raise the difficulties related to not having a standard

interface in the **SE***cube*™ SDK:

- Increased learning curve of the **SE***cube*™ environment.
- **SE***cube*™ applications are more difficult to be developed.
- Application developers are required to have significant cybersecurity knowledge.
- Developers have to deal with low-level primitives, leading to possible security vulnerability and implementation errors;
- The manual management of libraries and tools is complex to handle in large projects, thus not flexible nor scalable.

These problems determine the need of an intermediate layer sitting between host-side applications and the **SE***cube*™ HSM, reducing the complexity of integrating the capabilities of the **SE***cube*™ in new software as well as existing applications.
The software developed in this thesis takes care of the complex handling of the **SE***cube*™ and its libraries, providing an interface that is simple to understand and use, likewise compliant with strict security standards. The most important feature is that it integrates most of the **SE***cube*™ functionalities available, allowing new applications to exploit them without knowing the implementation details.

4.2 Application high-level overview

This application works as a server in a TCP client-server communication, exploiting the functionalities offered by the `QTcpSocket` and `QTcpServer` classes [11] to handle the connections with clients. The communication is based on a simple request-response protocol, in which packets contain an identifier (i.e., an integer value) used to recognize the operation they refer to. Moreover, packets include a list of input parameters filled by clients and a list of output parameters prepared by the middleware, containing the result of the requested operation. In order to provide confidentiality and protect the communication from sniffing attacks, packets are encrypted using AES-256 and a session key, generated during the client's authentication phase.
Notice that any possible operation can be performed only when the following conditions are satisfied:

1. A **SE***cube*™ device is connected to the host machine;
2. The middleware is connected to a **SE***cube*™ device, either in admin or user mode;
3. Clients have authenticated themselves, proving that they know the PIN of the **SE***cube*™ .

Figure 4.1 illustrates the typical work-flow of how requests are handled by the middleware and how responses, including error ones, are managed. It is important to point out that the **SE***cube*™ device works sequentially and cannot serve multiple requests simultaneously, hence requests are managed as a FIFO queue and handled in the same order as they arrive, as shown in Figure 4.2, which depicts a use case scenario with several requests coming from different clients.

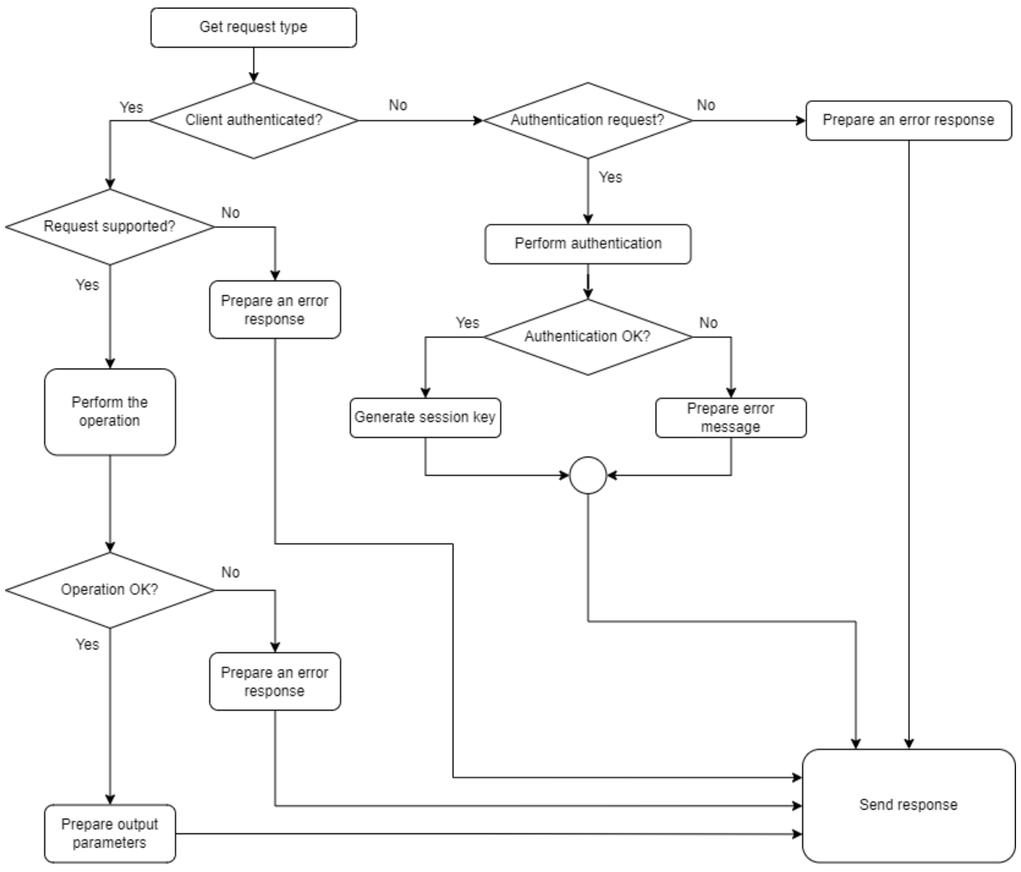


Figure 4.1: Request handling flowchart

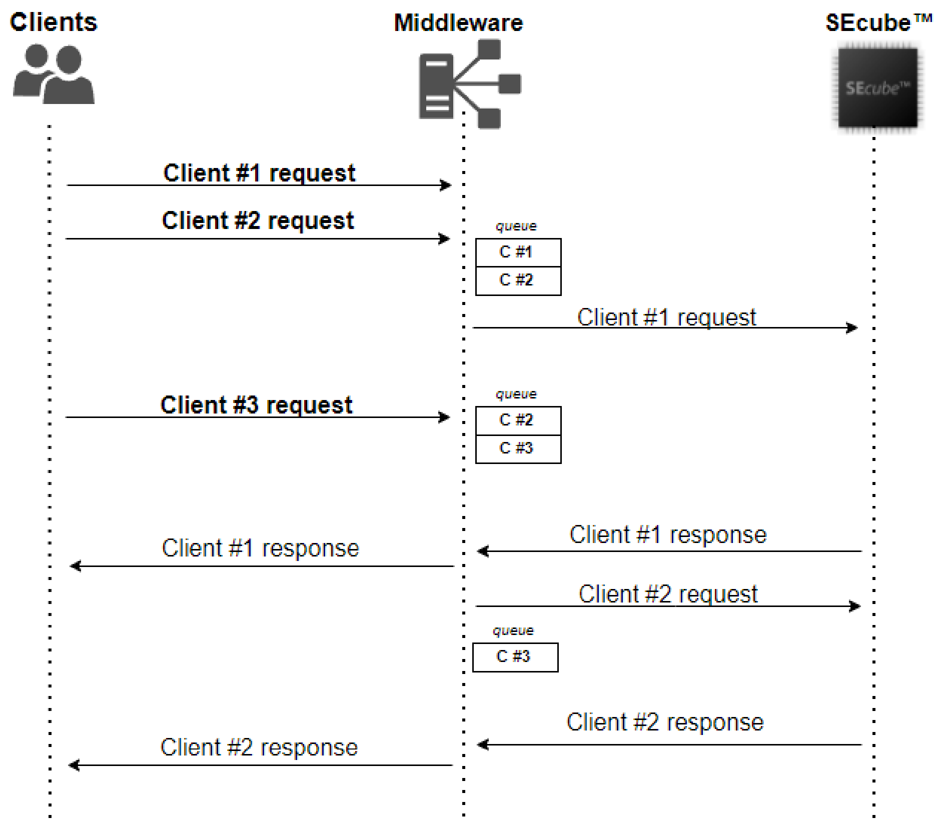


Figure 4.2: Simplified middleware FIFO implementation

4.3 Application implementation

The application has been developed in C/C++, it provides a GUI thanks to the Qt toolkit. It is compatible with Windows and Linux operating systems and it can work as a daemon application, serving requests in background. The following sections analyze the implementation details of the main components of this work, including the actual application developed and the modifications made on the firmware.

4.3.1 Application start up

The first thing to do when the application is started is to connect to a **SE***cube*™ device. Hence, the first window that appears is the `LoginWindow`, shown in Figure 4.3. Here a list of available devices is presented to the user, that has to select one in order to connect to it. Moreover, the user is asked to choose the login mode (user or admin) and to insert the PIN of the device. Once the user clicks on the login button, the challenge-based authentication process starts. If the authentication fails, an error message appears; otherwise the `MainWindow` is started and clients can can start opening connections.

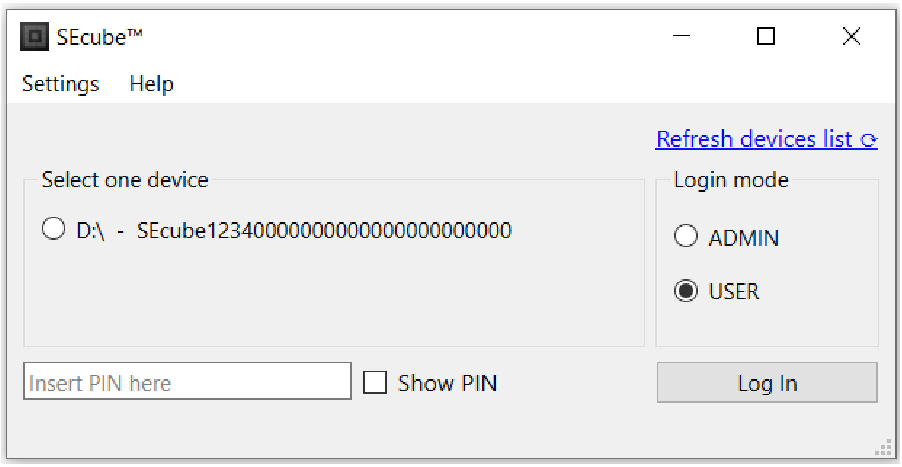


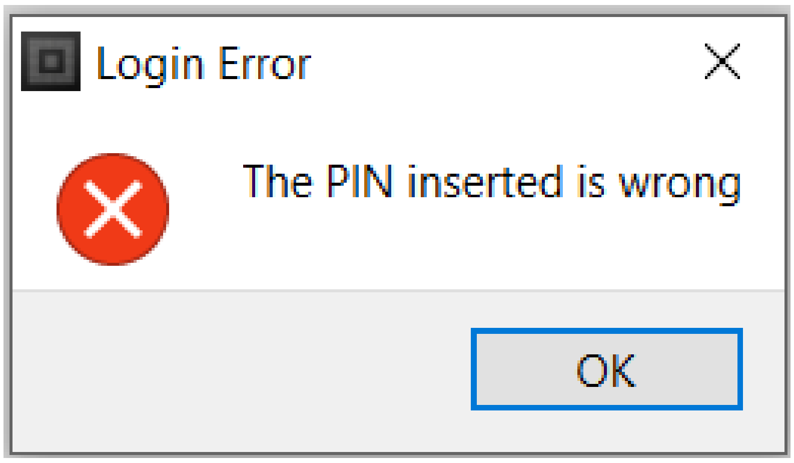
Figure 4.3: Login window

When the application is started, it is firstly populated the list of connected devices displayed in the login dialogue. Notice that the list can be refreshed any time, by simply clicking on the '*Refresh devices list*' label located in the top-right corner of the window.
Before clicking on the login button, the user has to:

1. Select a **SE***cube*™ device from the available ones;
2. Choose a login mode, admin or user;
3. Insert the PIN associated to the selected device;

Then, when the user clicks on the button, the login function is triggered and the challenge-based authentication provided by the library L1 is called. Once the login is successfully completed, another **SE***cube*™ functionality is exploited to calculate the SHA-256 digest of the inserted PIN. This operation is done in order to never store the plain value of the PIN in the flash memory, that would be potentially dangerous in case of memory leakage.
Notice that this is the only moment when the login API is called. In case of multiple login attempts on a device that has a user already logged in, a `L1AlreadyOpenException` is thrown, due to a constraint that the

SE*cube*™ imposes. Hence, third-party applications do not really perform the login, instead they only prove that they know the PIN by responding to a challenge. In order to achieve this behaviour, some changes were done on the **SE***cube*™ firmware, the details are analyzed in Section 4.3.3.
Once that these operations are completed, in case of error a message box appears providing some information (Figure 4.4) otherwise the login dialog is closed and the `MainWindow` is shown.


Login Error

The PIN inserted is wrong

Figure 4.4: Login error message

4.3.2 High-level message structure

Messages exchanged among middleware and client applications are sent over a TCP channel and are written in the TLV format (Type - Length - Value), in particular:

1. The type is an integer ID used to recognize which operation the message refers to;

2. The length refers to the size of the "value" field, since TCP does not provide fixed message size;

3. Value contains all the input/output parameters related to the operation that has to be performed and other parameters (i.e., return value and error message).

A predefined set of structures is provided to clients, each operation defines its own structure with the required parameters. The interaction, then, is

simple: once the connection in established, clients have to fill the structure corresponding to the operation they need (i.e., the `Digest_Params` struct to perform the digest of a string). Then the Cereal library [16] is used to serialize the message in the TLV format and send it over the socket.
The middleware executes the dual actions: firstly the request is deserialized, then the parameters are taken from the "value" field and the operation specified in the "type" field is executed on the **SE***cube*™ . A response with the results is composed in the same way as the request and sent to the client.
Serialized requests and responses are arranged into `uint8_t` arrays, and a set of constant values has been defined referring to the offsets of each field within the array. These values are shown in the code below, where an example of message structure is presented as well:

```
#define MINLEN   6 // at least 6 bytes (2B type + 4B length)
#define T_OFFSET 0 // type offset
#define T_SIZE   2 // type size
#define L_OFFSET 2 // length offset
#define L_SIZE   4 // length size
#define V_OFFSET 6 // value offset

struct KeyList_Params{
    int retvalue;
    std::string errmessage;
    std::vector<uint32_t> keys_id;
    std::vector<uint16_t> keys_len;

    void reset(){
        retvalue = 0;
        errmessage.clear();
        keys_id.clear();
        keys_len.clear();
    }

    template <class Archive>
    void serialize(Archive& archive){
        archive(retvalue, errmessage, keys_id, keys_len);
    }
};
```

4.3.3 Client authentication

Any **SE***cube*™ functionality supported can be exploited by third-party applications only after the authentication phase. If a client tries to send a request for an operation (i.e., file encryption) whilst not yet authenticated, the packet will be discarded and an error response will be sent. In particular, the client will receive as error message: '*Operation denied: client not authenticated*'.
As previously explained, clients do not perform an actual login (by exploiting the `L1Login` API), they are only demonstrating that they are aware of the shared secret value; the authentication phase is based on a challenge-response method, where the secret is the PIN of the **SE***cube*™ device. To perform this kind of operation, some changes have been made on the firmware too, adding two new functionalities, called `mw_challenge` and `mw_login`. These two functions work similarly to the two implemented in the actual login (called challenge and login) and the concept is similar: generating a challenge to send to the host and resolving its challenge. However, these new functions differ from the original ones in some aspects, such as:

- both functions do not return an error in case of host already logged in;
- mw_challenge does not derive a session key, unlike challenge;
- mw_login does not save the value of the token, unlike login.

On the host side the authentication has also been divided into two steps:

1. Prepare challenge: in this first step, the middleware communicates with the firmware in order to get a random value, called *cc1*, and the challenge; the first value will be used by the client to compute its challenge. A packet will be sent to the client containing these two values.
2. Compare challenge: the middleware receives from the client two values: *sRespExpected*, the expected response to client's challenge, computed using the PIN of the device and *cc1*, and *cResp*, the computed response to the challenge generated by the middleware. Then, the middleware will communicate with the firmware in order to check if the received values are the expected ones. If they are not, an error message is sent to the client; otherwise, a random 64-bit salt is generated and sent to the client. This value will be used on both sides to compute the session key used to encrypt the communication. More details about this key are given in Section 4.3.4.

Figure 4.5 shows how the client authentication works, displaying the parameters that each actor sends to the other and the operations performed on both sides.
It is important to point out that clients have also to specify the login mode which they are trying to authenticate with (i.e., user or admin mode). As a matter of fact, in order to prevent unprivileged access, connections in admin mode will be refused when the middleware is working in user mode.

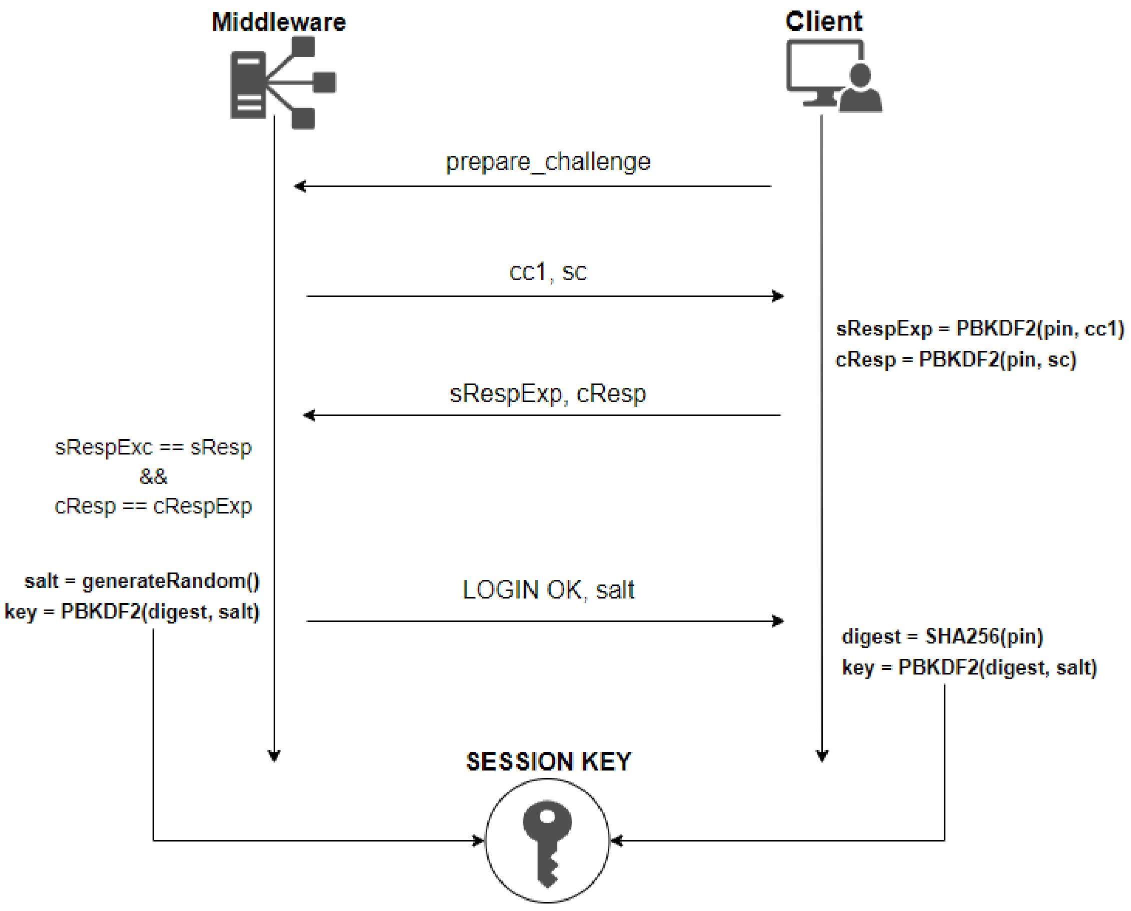


Figure 4.5: Simplified client authentication work-flow

4.3.4 Key generation

Network sockets do not provide data confidentiality natively: packets are sent in clear and sniffing attacks can be exploited. Considering that in the specific case of this work packets might contain confidential data (i.e., a plain text to encrypt), confidentiality should be always guaranteed to reduce attack surface.
Two methods have been analyzed to reach this goal:

1. Asymmetric encryption, building a public keys infrastructure into the middleware;
2. Symmetric encryption, using session keys;

Even if asymmetric encryption is stronger and more secure, some problems arise implementing this solution. In the first place, this approach increases the execution time because of its computational cost; furthermore, most importantly, the

SE*cube*™ platform does not yet support asymmetric encryption. For these considerations, the second approach has been carried on, exploiting symmetric encryption using AES-256.
Each request coming from a client, except for the ones related to the authentication, must be encrypted with a session key. Once a client is successfully authenticated, a `generateRandom` function generates a 64 bit integer random number. This value is used as cryptographic salt in the key derivation function (KDF). The key has to be generated on both sides in the same manner, in order to produce the same result. A PBKDF2 function is exploited to compute the key, using as input parameters:

- *Password*: the SHA-256 digest of the PIN.
- *Salt*: the previously described random value.
- *Iterations*: a constant value set to 150,000.

The final result is stored in the key variable, a 256-bit array.
Notice that the password used is the digest of the PIN instead of the actual PIN: this design choice, as previously described, was made in order to protect the plain value from any memory leakage that might expose it.
The number of iterations refers to the amount of times the function is executed. Its value can be arbitrary, but NIST suggests:

> *For PBKDF2, the cost factor is an iteration count: the more times the PBKDF2 function is iterated, the longer it takes to compute the password hash. Therefore, the iteration count SHOULD be as large as verification server performance will allow, typically at least 10,000 iterations. [2]*

Some performances tests were held and setting the value to 150,000 is a good compromise: the execution time is still acceptable and the strength of the key is sufficent. Considering a real case in which the middleware might have many connections opened with different third-party applications, a map of keys has been defined in the `MainWindow` class as follows:

```
std::map<QTcpSocket*, uint8_t[SESSION_KEY_SIZE]> session_keys;
```

In this way the key that refers to a specific socket can be immediately accessed through its socket descriptor. The socket descriptor can be retrieved thanks to the `sender()` method offered by the `QTcpSocket` class that returns a pointer to the object that sent a readyRead signal [10]:

```
QTcpSocket* socket = reinterpret_cast<QTcpSocket*>(sender());
auto key = session_keys[socket];
```

When the client closes the connection, the key is removed from the map since it is no more used, using again the map and the socket descriptor to access it directly.

4.3.5 Handling a request

Only authenticated clients are allowed to send requests to the middleware. Once a request arrives, the first check is performed on the client *status* to ensure he completed the authentication phase: packets coming from clients that do not satisfy this constraint are discarded, and an error response is transmitted to the sender.
To build a request three actions have to be performed:

1. The guest application has to fill a structure, providing the required input parameters;
2. The structure has to be serialized and written into an array following the TLV format;
3. The serialized structure has to be encrypted using the session key related to the communication;

Figure 4.6 illustrates an example of how a request to perform the SHA-256 digest of a string is built. Notice that for the sake of simplicity the content of the packet is shown in clear, while in a real case scenario it is encrypted with the session key.

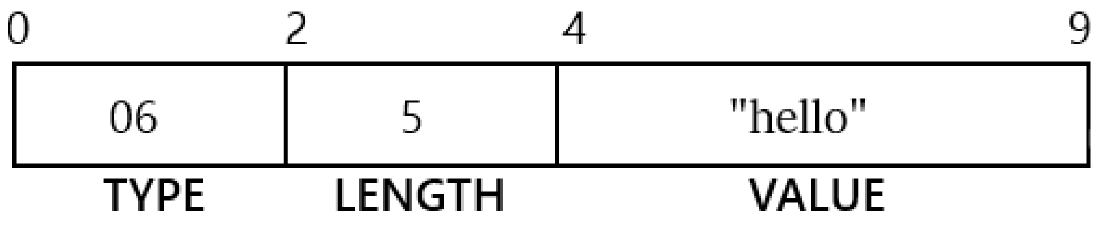
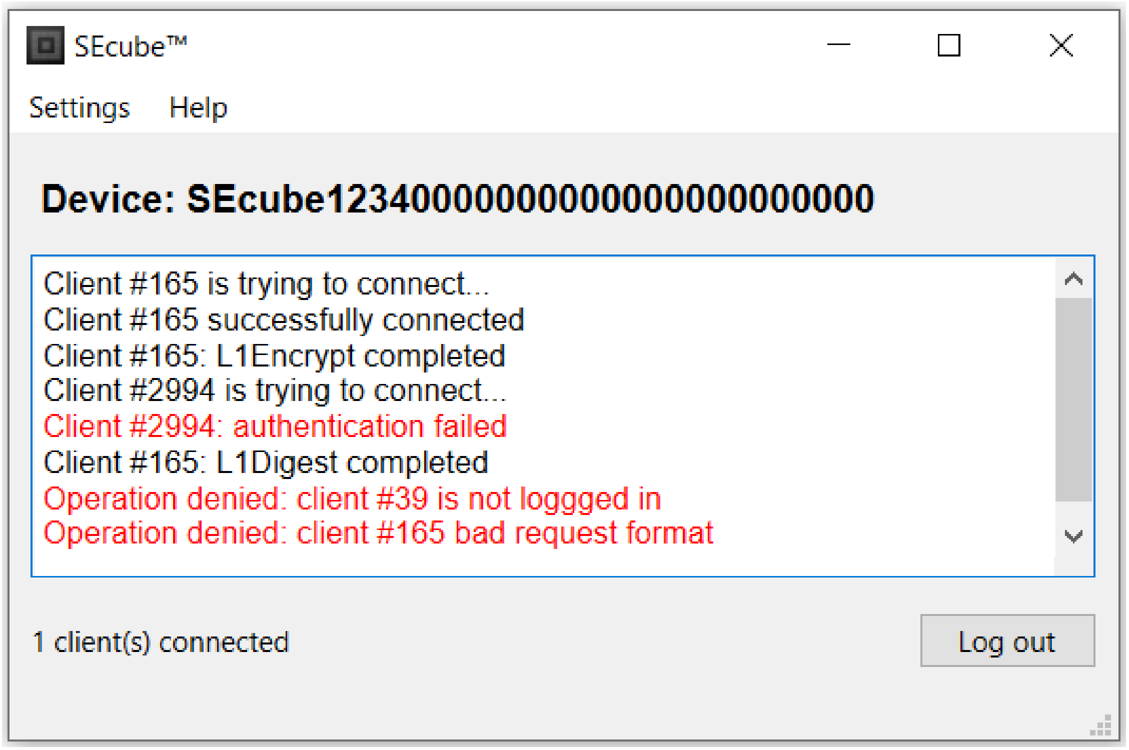


Figure 4.6: Request in TLV format

The middleware, on the other side, will perform the dual operation, decrypting the received packet. From the first bytes of the request, the middleware can read the operation ID and, thanks to the Cereal library, it can deserialize the packet and fill the corresponding structure with the input parameters. If the parameters are in a different format from the expected one, an error message is sent to the client.
Knowing which **SE***cube*™ functionality the client requires and the necessary parameters, the middleware can simply call the corresponding API and, when the result is ready, a response is built in the same way as the request and sent to the client.
During any kind of operation, the `MainWindow` is showed. It has a simple interface, shown in Figure 4.7, that contains:

- The serial number of the **SE***cube*™ device connected;
- A counter that shows how many clients are connected;

- A logger, showing information about the completed operations;
- A logout button.



Device: SEcube12340000000000000000000000

```
Client #165 is trying to connect...
Client #165 successfully connected
Client #165: L1Encrypt completed
Client #2994 is trying to connect...
Client #2994: authentication failed
Client #165: L1Digest completed
Operation denied: client #39 is not loggged in
Operation denied: client #165 bad request format
```

1 client(s) connected

Figure 4.7: Main window interface

4.3.6 Autostart the application

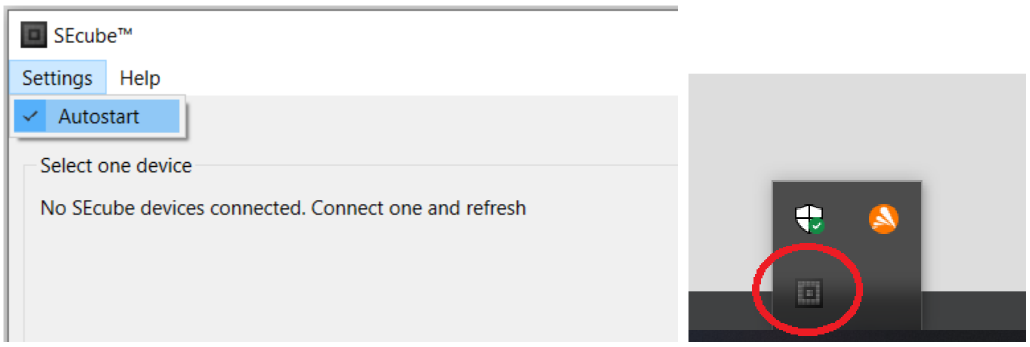
The middleware has also been developed as an autostarted daemon application. This means that:

1. The application is automatically executed at startup without involving the user;
2. The application can be executed in background without showing any window.

In order to autorun the middleware, some operations with the executable file of the application has to be performed. This procedure is different depending on the operating system, however it has been implemented in both Windows and Linux. In particular:

- On Winows systems, a string value containing the path to the file has to be added into the `C:\WINDOWS\Start Menu\Programs\StartUp` folder.
- On Linux systems, a `.desktop` file has to be created, containing some information about the application (i.e., application path, type). This file will be then added into the `home/.config/autostart` folder.

This is a significant feature enabled by default, but it can be disabled by the user if required. The *Settings* menu, shown both in the `LoginWindow` and in the `MainWindow`, contains a checkbox (Figure 4.8a) that allows to disable this functionality. By clicking on this checkbox, the executable file is removed from the mentioned folder. The reverse operation can be always performed by enabling again the checkbox.
When the application is running in background, a system tray icon (Figure 4.8b, Windows version) is shown: this icon is presented as a **SE***cube*™ icon and, if clicked, allows to display again the `MainWindow`, enabling the interaction with the user.



✓ Autostart

Select one device

No SEcube devices connected. Connect one and refresh

(a) (b)

Figure 4.8: Settings menu (a) and the system tray icon (b)

Chapter 5

Results and discussion

This chapter focuses on the results collected during the development of this thesis, along with the main characteristics and advantages of the solution implemented. The drawbacks that the chosen approach brings are then analyzed. Finally, in order to let the reader understand better how the system works, a list of the main use cases is given.

5.1 Results

The application developed in this thesis is based on the **SE***cube*™ Security Platform; the code is open source and available on GitHub [26]. This application exploits the main security-oriented functionalities offered by the **SE***cube*™ environment, along with other tools and capabilities of the Qt toolkit.
This application has been developed in order to incorporate most of the **SE***cube*™ capabilities in one single access point, building an ecosystem that is easy to use and secure. These needs come from the partial lack of coherence of the provided SDK: it involves a great amount of APIs and libraries that might determine a considerable learning curve to have full control of the advantages brought by the device.
The application can work with several third-party clients at the same time, but with one **SE***cube*™ module only. The user has to select which device to use during the login phase, then each client will be connected automatically to the selected one, without the privilege of changing.
Notice also that some functionalities might not be available in "user mode": users must login in "admin mode" in order to exploit all the supported functionalities.
Thanks to this application the interaction with a **SE***cube*™ device has become smoother both for experienced developers and for those approaching the platform for the first time, by means of a standardized interface that guarantees high-level security and a user-friendly environment, enabling any developer to implement applications that leverage strong cybersecurity functionalities.

Here is a summary of the main features offered by this work:

- Compatibility with Windows and Linux operating systems.
- A variety of APIs to manage low-level **SE***cube*™ functionalities.
- Support in the integration of a **SE***cube*™ device in already existing applications, to increase security.
- The security primitives are always performed within the **SE***cube*™ .
- Confidentiality among the application and clients is always guaranteed thanks to an encrypted communication based AES256 and a 32 byte symmetric key.
- Security issues and security primitives are handled automatically and are transparent to the users.
- Native compatibility with several **SE***cube*™ libraries, such as **SE***file*™ and **SE***link*™ .
- The implementation of an authentication method based on a challenge-response approach that does not expose the shared secret (the device's PIN);
- An easy-to-use and flexible interface that decrease the required time to fully understand how the **SE***cube*™ works.

5.2 Drawbacks

This application presents some drawback as well, such as:

- The **SE***cube*™ processes requests sequentially, making the system slower in case of several clients connected simultaneously.
- The **SE***cube*™ SDK only supports AES-256 for encryption: no support for asymmetric cryptography is given at the moment.
- Adding a new layer between the **SE***cube*™ and external applications increases the time required to perform the operations.
- New **SE***cube*™ functionalities require some time to be integrated and supported.
- Some Linux versions do not support system tray icons, making impossible to show an icon in the system bar while the application is running in background.
- The host to which the **SE***cube*™ is connected must be as secure as possible, since the device cannot detect if an attacker violates the host.

It is important to point out that some of the described drawbacks are consequences of designing choices: in order to reach a feasible solution some trade-off are inevitable.

5.2.1 Exposure of salt during session key generation

During client's authentication phase, explained in details in Section 4.3.3, a session key is generated and then used to encrypt the communication channel established among the middleware and the client. Since symmetric encryption is used, the same key has to be generated on both sides using a KDF that takes as input parameters the SHA-256 digest of the PIN and a salt, whose value is generated randomly by the middleware and sent to the client. Considering that the key does not exist yet, the salt has to be sent in clear, hence visible by any other process that is monitoring the communication.
A possible brute-force attack could be performed if an intruder intercepts the salt by sniffing the messages; the attack works in the following way:

1. The attacker studies how the communication works, since the code is open-source;
2. When a client opens a connection, the attacker starts to monitor the communication;
3. The attacker intercepts the packet that contains the salt sent in clear, and stores its value;
4. Once the client is authenticated, he starts to send encrypted requests: the attacker intercepts one of them, called C, and stores it;
5. The attacker performs a brute force attack to get all the possible values of the key: the shared secret is 256-bit long, hence he computes $key_i = KDF(digest_i, salt)$, where $i = 0, ..., 2^{256} - 1$.
6. The attacker decipher C trying every value of the key: $M_i = dec(C, key_i)$;
7. Finally, the attacker checks all the M_i values until he finds an acceptable plain text written in the expected format. When he finds it, he also gets the corresponding $digest_i$, which is the digest of the PIN that will be then used to exploit any future session key.

This is potentially a powerful attack because it can be performed offline and, once the digest of the PIN is exposed, it allows to decipher any message coming from any client by computing its session key. This attack will be effective until the value of the PIN is modified, thus a periodic change is recommended.
However, the time required to perform this kind of brute force attack should be considered as well. The KDF performs 150,000 iterations, increasing the execution time required to perform it; moreover, in the worst case scenario, the attacker, to get the correct key, has to execute the function 2^{256} times and, on average, 2^{128} times. The time required to execute the KDF depends on the hardware used (i.e., a GPU accelerator decreases the time) and can vary from microseconds to milliseconds. Considering a very optimistic case in which the KDF requires 1 µs to be executed, the average case scenario would require approximately 9×10^{24} years: a huge number

that makes this attack harmless at least with the current hardware. A real problem could arise in the future, with enhanced performances.

5.3 Sample use cases

Here are few use cases that show how to exploit some of the most useful features provided, to better understand how to integrate the middleware in pre-existing application or to build new ones. These use cases are deliberately simplified, some of them present figures and screenshots of running applications as well.

5.3.1 Client authentication

Executed by third-party applications that want to connect to a **SE***cube*™ device.
Preconditions: **SE***cube*™ connected to the host, middleware running and logged in the **SE***cube*™ .
Postcondition: the client is successfully connected to a device.

1. Create a socket and connect to the middleware on a specified port.
2. Send a `PREPARE_CHALLENGE` command specifying the access type (user or admin).
3. Compute the two challenges by calling the `PBKDF2HmacSha256` API, using as input parameters the two received values.
4. Send the two computed challenges by means of the `COMPARE_CHALLENGE` command.
5. Check the return value to be sure that the authentication was successfully, otherwise check the error message and close the socket.
6. Compute the SHA-256 digest of the PIN and call the `PBKDF2HmacSha256` API to compute the session key, providing the digest and the received salt.

5.3.2 Encrypt a message

Executed by third-party applications.
Preconditions: **SE***cube*™ connected to the host, middleware running and logged in the **SE***cube*™ , client authenticated.

1. Decide the plain text that has to be encrypted.
2. Decide which algorithm and mode should be used to encrypt.
3. Decide the ID of the key used to encrypt.
4. Send a `ENCRYPT` command, passing as parameters the text, its size, the algorithm and the key ID, encrypted with the session key and serialized.

5. The text is encrypted by the **SE***cube*™ and sent to the client.
6. Check the return value to be sure that the operation was completed without errors.

5.3.3 Decrypt a message

Executed by third-party applications.
Preconditions: **SE***cube*™ connected to the host, middleware running and logged in the **SE***cube*™ , client authenticated.

1. Decide the cipher text that has to be decrypted.
2. Decide which algorithm and mode should be used to decrypt.
3. Decide the ID of the key used to decrypt.
4. Send a `DECRYPT` command, providing as parameter the text, its size, the algorithm and the key ID, encrypted with the session key and serialized.
5. The text is decrypted by the **SE***cube*™ and sent to the client.
6. Check the return value to be sure that the operation encountered no problems.

5.3.4 SE*file*™ - Create a secure file

Executed by third-party applications.
Preconditions: **SE***cube*™ connected to the host, middleware running and logged in the **SE***cube*™ , client authenticated.
Postcondition: an encrypted file is successfully created.

1. Decide the name of the file.
2. Decide the ID of the key used on the file.
3. Decide the flags related to opening mode and file creation.
4. Send a `SECURE_OPEN` command, providing the decided parameters, encrypted with the session key and serialized.
5. The **SE***cube*™ will open the file, giving as name the SHA-256 digest of the decided name.
6. Check the return value to be sure that the operation was completed without errors. An integer value will be received to handle the file as a normal file with its file descriptor.

Notice that file should also be closed at the end, by means of the `SECURE_CLOSE` command, specifying the file descriptor.

5.3.5 SE*link*™ - Encrypt and serialize a message

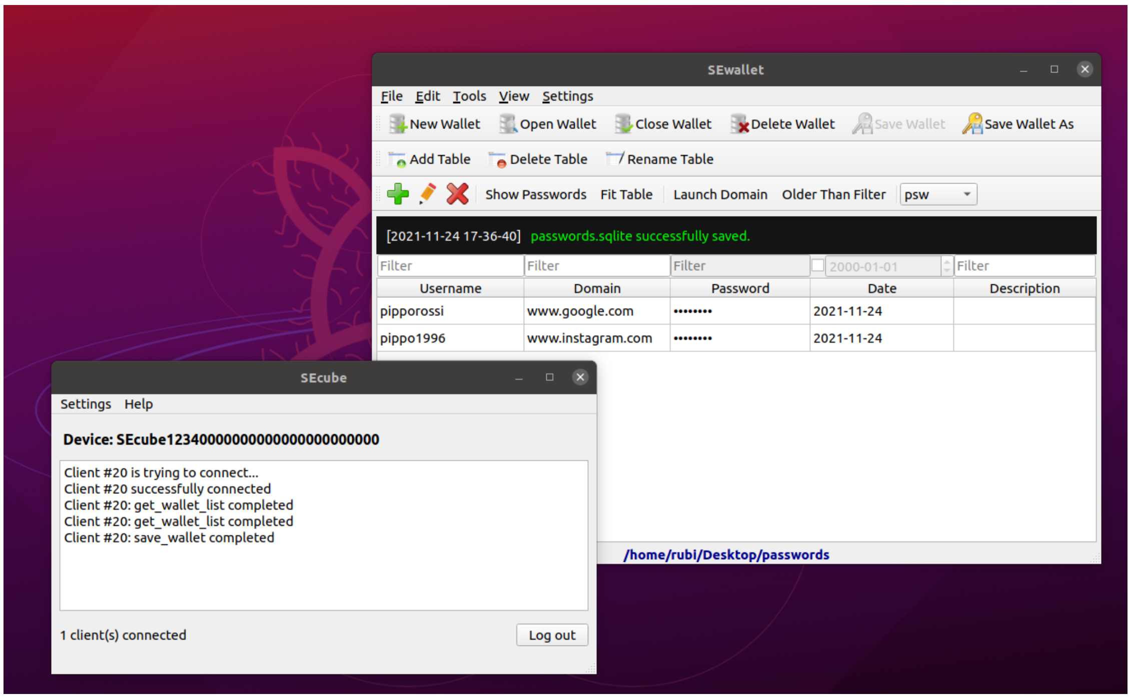
Executed by third-party applications.
Preconditions: **SE***cube*™ connected to the host, middleware running and logged in the **SE***cube*™ , client authenticated.

1. Decide the plain text that has to be encrypted and serialized.
2. Decide the ID of the key used on the plain text.
3. Send an `ENCRYPT_MANUAL` command, specifying the plain text, its size and the key ID as parameters, encrypted with the session key and serialized.
4. The **SE***cube*™ will serialize and encrypt the plain text, sending the serialized cipher text and its size as response.
5. Check the return value to be sure that the operation was successfully completed.

Notice that the dual operation that will decrypt and deserialize an encrypted cipher text can be performed by means of the `SELINK_DECRYPT` command.

5.3.6 SE*wallet*™ integration

SE*wallet*™ [30] is a password manager tool developed to be compatible with the **SE***cube*™ platform, with the goal of helping users to generate, store and use passwords in a secure way. The original version of **SE***wallet*™ has been developed integrating the functionalities offered by the **SE***cube*™ SDK directly into the project; the source code has been modified in order to support the communication with the middleware so that it can automatically handle the complexity related to security concerns. These modification has been realized to highlight the potential of this new interface: the integration into pre-existing applications is pretty intuitive and simple, making the applications more reliable since the security aspects are handled internally by the **SE***cube*™ in the correct way.
Figure 5.1 shows a use case in which the two applications are working together.



[2021-11-24 17-36-40] passwords.sqlite successfully saved.

Username	Domain	Password	Date	Description
pipporossi	www.google.com	••••••••	2021-11-24	
pippo1996	www.instagram.com	••••••••	2021-11-24	

/home/rubi/Desktop/passwords

Device: SEcube12340000000000000000000

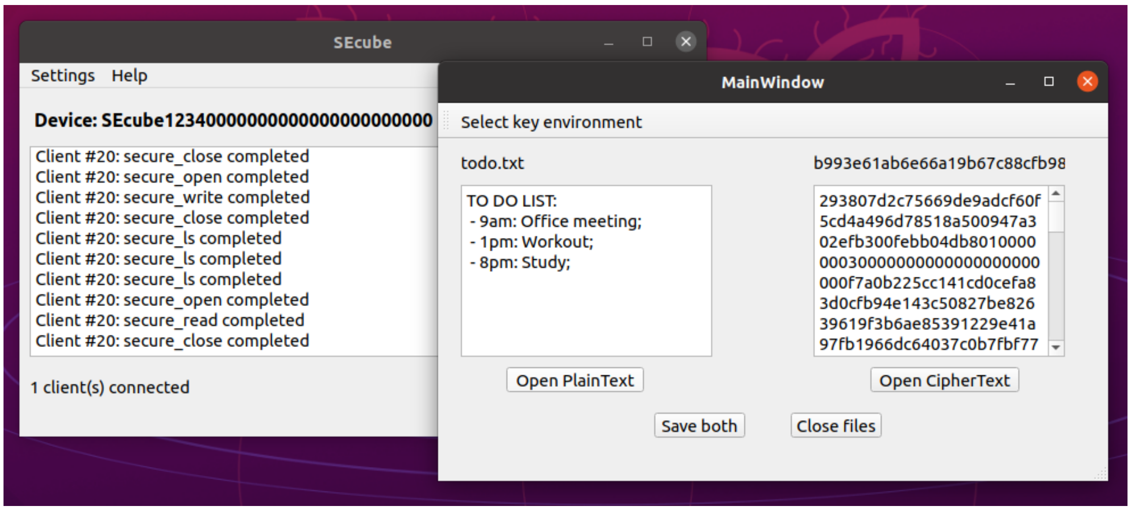
```
Client #20 is trying to connect...
Client #20 successfully connected
Client #20: get_wallet_list completed
Client #20: get_wallet_list completed
Client #20: save_wallet completed
```

1 client(s) connected

Figure 5.1: Example of **SE***wallet*™ running

5.3.7 Secure Text Editor integration

Secure Text Editor [25] is a demo project, developed as an example of GUI application working with **SE***file*™ . It helps to manage text files, operating as an editor for both plain and encrypted files. As an alternative to directly including the **SE***file*™ library into the project, the source code has been modified in order to make it communicate with the middleware and take advantage of it to exploit the **SE***cube*™ functionalities.
Figure 5.2 depicts the Secure Text Editor application running, showing the communication with the middleware as well.



Device: SEcube1234000000000000000000000

```
Client #20: secure_close completed
Client #20: secure_open completed
Client #20: secure_write completed
Client #20: secure_close completed
Client #20: secure_ls completed
Client #20: secure_ls completed
Client #20: secure_ls completed
Client #20: secure_open completed
Client #20: secure_read completed
Client #20: secure_close completed
```

1 client(s) connected

todo.txt

```
TO DO LIST:
 - 9am: Office meeting;
 - 1pm: Workout;
 - 8pm: Study;
```

b993e61ab6e66a19b67c88cfb98

```
293807d2c75669de9adcf60f
5cd4a496d78518a500947a3
02efb300febb04db8010000
00030000000000000000000
000f7a0b225cc141cd0cefa8
3d0cfb94e143c50827be826
39619f3b6ae85391229e41a
97fb1966dc64037c0b7fbf77
```

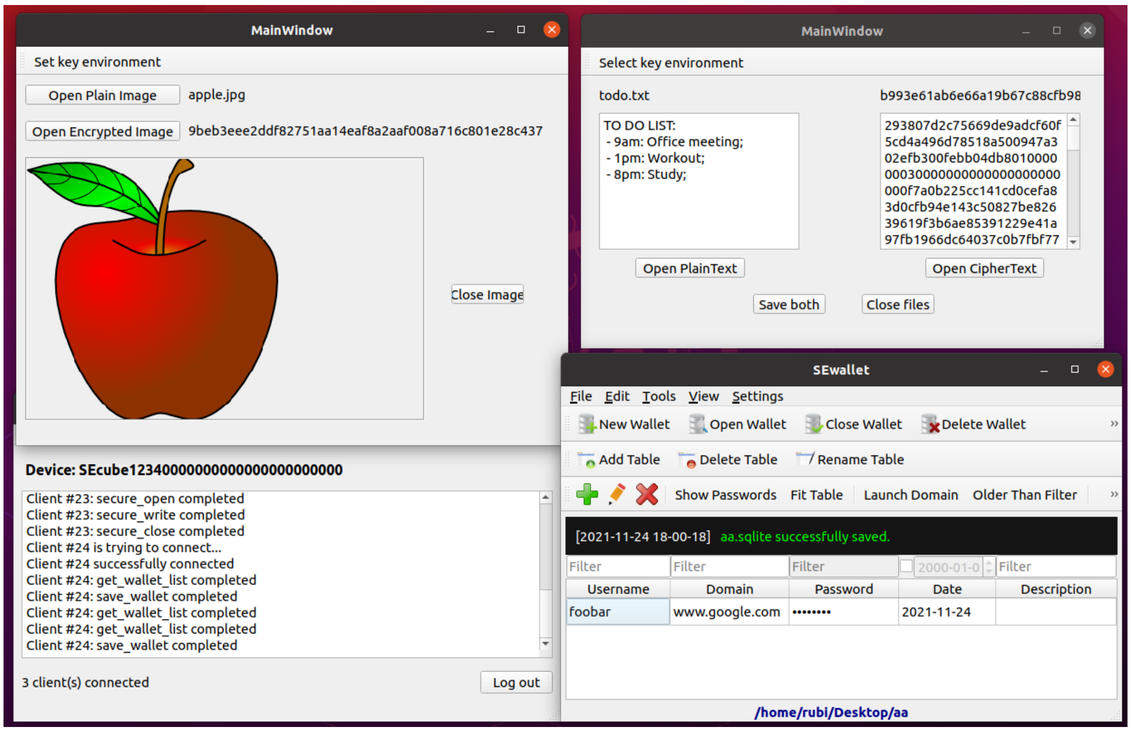
Figure 5.2: Example of Secure Text Editor running

5.3.8 Secure Image Viewer integration

Secure Image Viewer [24] is a demo project similar to the Secure Text Editor, it exploits the functionalities provided by the **SE***file*™ library in order to encrypt images. Moreover, it is composed by a GUI that allows to depict encrypted images. Likewise previously described projects, its source code has been improved in order to exploit all the required **SE***cube*™ features through the middleware instead of manually include the libraries and call the necessary APIs.
Figure 5.3 illustrates a more realistic and exhaustive scenario, where the user might run several applications at the same time. In this example, the following applications are running simultaneously:

1. The middleware, serving the incoming requests;
2. **SE***wallet*™ , with a newly created password wallet;
3. Secure Text Editor, with an encrypted file opened;
4. Secure Image Viewer, with a encrypted image opened;

Notice that in order to make third-party application to communicate to the middleware, it first has to be started and connected to a **SE***cube*™ device.



apple.jpg

9beb3eee2ddf82751aa14eaf8a2aaf008a716c801e28c437

Device: SEcube12340000000000000000000000

```
Client #23: secure_open completed
Client #23: secure_write completed
Client #23: secure_close completed
Client #24 is trying to connect...
Client #24 successfully connected
Client #24: get_wallet_list completed
Client #24: save_wallet completed
Client #24: get_wallet_list completed
Client #24: get_wallet_list completed
Client #24: save_wallet completed
```

3 client(s) connected

todo.txt

```
TO DO LIST:
 - 9am: Office meeting;
 - 1pm: Workout;
 - 8pm: Study;
```

b993e61ab6e66a19b67c88cfb98

```
293807d2c75669de9adcf60f
5cd4a496d78518a500947a3
02efb300febb04db8010000
0003000000000000000000
000f7a0b225cc141cd0cefa8
3d0cfb94e143c50827be826
39619f3b6ae85391229e41a
97fb1966dc64037c0b7fbf77
```

[2021-11-24 18-00-18] aa.sqlite successfully saved.

Username	Domain	Password	Date	Description
foobar	www.google.com	••••••••	2021-11-24	

/home/rubi/Desktop/aa

Figure 5.3: Example of multiple applications running

Chapter 6

Conclusions

This thesis aimed at developing an application acting as an intermediary between third-party applications and the **SE***cube*™ Hardware Security Module.
The need of such a solution derives from a partial lack of coherence and usability of the **SE***cube*™ SDK, which is open source and developed by European research institutions. As a matter of fact, the software architecture of the platform involves several libraries and APIs that require advanced skills and knowledge about various disciplines in order to leverage the advantages offered by the platform.
The application developed in this work allows to interact with a **SE***cube*™ device exploiting an easy-to-use interface towards its functionalities, in a way that is compliant with strict security standards adopting, for example, the 'security by design' principle.
The initial target of this work has been achieved, the application developed in fact provides a standard interface upon the **SE***cube*™ , allowing the interaction to be easier and conventional. Moreover, confidentiality among the application and clients is always guaranteed thanks to symmetric encryption based on a session key. Another great advantage is that the application can handle requests coming from many clients at the same time, implementing a FIFO queue. With this work, the **SE***cube*™ environment has been improved so that it can be easily integrated within third-party software as well as in pre-existing applications.
In conclusion, thanks to this application, the **SE***cube*™ offers a more versatile and flexible ecosystem. Developers can approach the platform without worrying about low-level security details and can exploit the provided functionalities in an easier way, building an environment that guarantees security features to personal computers or servers according to high-security standards.
The developed application still lacks on some key features that would make it a more reliable and complete product, improving performances as well. Some pre-existing limitations imposed by the **SE***cube*™ platform did not allow a significant performance enhancement (i.e., the sequential requests handling), however possible improvements can still be put in place.

6.1 Future works

6.1.1 White-listing filter

The current version of the application accepts connections coming from any kind of client, without performing some sort of security checks. Consequently, a potential malicious application could open a socket towards the middleware and try to exploit some vulnerability. A better approach requires to implement a white-listing filter: a mechanism that allows only a specific set of applications to connect to the **SE***cube*™ , denying by default all the others not explicitly authorized. During authentication phase, clients should append a predefined fingerprint to prove their identity (i.e., the process ID); the middleware stores a list of trusted processes and accepts only requests from those in the list. In order to implement such a solution, message integrity is also required to assure accuracy and completeness of the received fingerprint.

6.1.2 Session key expiration

This application has been designed in a way that uses symmetric encryption to ensure data confidentiality. The key used to encrypt the communication is generated starting from the SHA256 digest of the device's PIN and a random salt. Hence, its value is always different, even in the case of two connections opened by the same client, but it lasts till the communication is closed. The idea is to add an expiration to the key, so that its value is renewed periodically. The key duration can be defined in term of:

- time (i.e., 1 hour). This approach could be more difficult to implement because it requires perfect synchronization among middleware and clients;
- messages (i.e., 20 requests). This approach is easier since it only requires a circular counter implemented in the middleware.

This integration would make the whole system more reliable are robust against cyber attacks.

6.1.3 A priority scheduler

SE*cube*™ does not support parallel requests handling, hence operations are executed sequentially. The architecture of this application is based on FIFO queue, where requests coming from clients are served in the same order as they arrive. This mechanism is the simplest one but the throughput could be low. Other kinds of scheduling could be adopted in order to increase performances, such as a priority scheduler; with this approach, incoming requests are arranged in a queue ordered by priority: higher priority requests are served before than lower ones. This solution allows to build a hierarchical system where critical operations can be performed immediately.

6.1.4 SE*key*™ support

This application has been built in order to support the most useful libraries and APIs offered by the **SE***cube*™ SDK, except for **SE***key*™ . In the current version, the only APIs related to key management supported are those to manually add or remove a certain key, using its ID to access it. Supporting **SE***key*™ would allow to handle keys in a easier and more secure way, at the same time it would make the application more complete with respect to the **SE***cube*™ ecosystem, expanding the features covered.

6.1.5 SE*cube*™ smartcard integration

The **SE***cube*™ platform incorporates a smartcard with powerful cryptographic features, such as asymmetric encryption. This smartcard, however, is not yet supported by the **SE***cube*™ firmware, therefore it cannot be used in any application related. Integrating its capabilities would permit to benefit of great advantages, including key agreement protocols and asymmetric encryption.

Appendices

Appendix A

Supported functionalities

This appendix presents how to interface to the functionalities supported by this work. Table A.1 contains the list of the available commands, recognized by a 16-bit integer value (Command ID). IDs are divided in ranges depending on the library they refer to:

- 0-199 refers to the L1 library that manages generic operations;
- 200-399 refers to the **SE***file*™ library;
- 400-599 refers to the **SE***link*™ library;
- 600-65535 are unused.

A detailed description of how to deal with each functionality is also provided, including required parameters and an high-level workflow.

The supported L1 instructions are:

- PREPARE CHALLENGE: this is the first step to authenticate a third-party application that wants to connect to a **SE***cube*™ device. It requires as input parameter the access type (admin or user) and sends, as response, two 32 char arrays: `cc1` and `sc`, respectively used to calculate the expected device response to the client challenge and the client response to the device's challenge.
- COMPARE CHALLENGE: the client authentication's second phase. It has two input fields: `sRespExpected` and `cResp`, the two values calculated in the previous step. As response a 64-bit salt will be received, necessary to derive the session key.
- ENCRYPT: this command can be used to perform the encryption of a string. Client needs to send the plaintext, the algorithm and its mode (e.g., AES ECB) and the key ID used to perform the encryption. The response will be encapsulated in a `SEcube_cipher_t` struct variable, that contains the ciphertext and other information.

- DECRYPT: the dual operation of the previous one. A `SEcube_cipher` struct variable has to be filled with the ciphertext, its size and the key ID. A char vector will be received as response, containing the plain text.
- DIGEST: used to compute the HMAC-SHA256 digest of a string. It requires as input parameters the plain text along with its size, and a `SEcube_digest` object that contains information regarding the key and the algorithm to use. This object also contains an array where the computed digest will be placed.
- GET ALGO: this command does not require any input parameter, it simply returns a list of the algorithms supported by the device as a vector of `se3Algo` elements.
- KEYS LIST: returns the list of the stored keys, with their IDs and length. Notice that this function only returns manually inserted keys and does not support the interaction with SEkey.
- KEY EDIT: allows to manually edit a key. The user has to fill a `se3Key_t` object with the parameters of the key to edit and he has to specify which operation to perform (add or delete key) with an integer identifier.

The supported **SE***file*™ commands are:

- SECURE OPEN: it works similarly to the open syscall, but it deals with encrypted files. The user is asked to give the file name along with the algorithm and the key ID that will be used during encryption. Moreover, he has to specify the flags that refer to the access mode and the creation. The function will return a file descriptor as an integer value. Note that new files will be created with an encrypted name, which is the SHA256 digest of the given name.
- SECURE CLOSE: given a file descriptor, it closes a previously opened encrypted file.
- SECURE READ: this command in used to read an encrypted file. The user has to provide a file descriptor and the amount of bytes he wants to read. A string will be filled with the decrypted content of the file.
- SECURE WRITE: it is used to write in an encrypted file. User must provide a file descriptor and the plain text to write, that will be encrypted using the key and the algorithm specified during the file's opening phase.
- SECURE SEEK: the input parameters are a file descriptor, an offset and a whence. This function sets the position of an encrypted file's pointer to the specified offset starting from the given whence (e.g., from the beginning of the file).
- SECURE TRUNCATE: this function allows to resize an encrypted file. The user must specify the file descriptor and the new size, expressed in bytes.

- SECURE LS: given a string that contains the path of a folder, this function returns a list of pairs containing the encrypted name and the decrypted name of sub-directories and files present. Note that in case of a plain file, the two values will be equals.

Finally, the supported **SE***link*™ operations are:

- ENCRYPT MANUAL: this function allow to encrypt and serialize a plain text. The user gives as input parameters the plain text as `uint8_t` vector, its size and the ID of the key that will be used to perform AES256-HMAC-SHA256 encryption. As output a vector containing the serialized cipher text and its size are provided to the user.
- DECRYPT: the dual operation of the encrypt manual. Given an encrypted and serialized vector, this function decrypts and then deserializes the content of the vector.

Notice that the `selink_encrypt_auto` function is not yet supported, since it works using keys automatically managed by **SE***key*™ . The ID 402, however, has been already designed to support this functionality

Table A.1: Supported functionalities

Command ID	**Command name**
0	PREPARE_CHALLENGE
1	COMPARE_CHALLENGE
4	ENCRYPT
5	DECRYPT
6	DIGEST
7	GET_ALGO
8	KEYS_LIST
9	KEY_EDIT
201	SECURE_OPEN
202	SECURE_CLOSE
203	SECURE_READ
204	SECURE_WRITE
205	SECURE_SEEK
206	SECURE_TRUNCATE
207	SECURE_LS
401	ENCRYPT_MANUAL
403	DECRYPT

Bibliography

[1] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. 2006, p. 714.

[2] National Institute of Standards and Technology. *Digital Identity Guidelines, Authentication and Lifecycle Management*. 2017. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63b.pdf.

[3] Thanh Bui et al. “Man-in-the-Machine: Exploiting Ill-Secured Communication Inside the Computer”. In: *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bui.

[4] Daniel J. Bernstein. *Secure interprocess communication*. URL: https://cr.yp.to/docs/secureipc.html.

[5] Juno Development Board. URL: https://developer.arm.com/tools-and-software/development-boards/juno-development-board.

[6] STM32F429 Board. URL: https://www.st.com/en/microcontrollers-microprocessors/stm32f429-439.html.

[7] Google Chrome browser. URL: https://www.google.com/intl/it_it/chrome/.

[8] Infineon SLJ 52G Java Card. URL: https://www.infineon.com/cms/en/product/security-smart-card-solutions/smart-card-solutions/slj-52g/.

[9] FIPS certifications. URL: https://csrc.nist.gov/publications/fips.

[10] QObject Class. URL: https://doc.qt.io/qt-5/qobject.html#sender.

[11] QTcpSocket Class. URL: https://doc.qt.io/qt-5/qtcpsocket.html.

[12] Blu5 Group. URL: http://www.blu5group.com/.

[13] **SE***key*™ Library. URL: https://github.com/SEcube-Project/SEkey.

[14] **SE***file*™ Library. URL: https://github.com/SEcube-Project/SEfile.

[15] **SE***link*™ Library. URL: https://github.com/SEcube-Project/SElink.

[16] Cereal Library. URL: https://uscilab.github.io/cereal/.

[17] Secure SQL Database library. URL: https://github.com/SEcube-Project/Secure-SQL-Database.

[18] Toppan Ltd. *Side-channel Attack Standard Evaluation Board: SASEBO.* URL: http://www.toptdc.com/en/product/sasebo/.

[19] Fornero Matteo. *Development of a secure key management system for the SEcube Security Platform.* URL: https://webthesis.biblio.polito.it/14521/1/tesi.pdf.

[20] Fornero Matteo et al. ***SE****cube™ Open Security Platform.* URL: https://github.com/SEcube-Project/SEcube-SDK/blob/master/wiki/wiki_rel_012.pdf.

[21] Microsoft. *Interprocess Communications.* URL: https://docs.microsoft.com/en-us/windows/win32/ipc/interprocess-communications.

[22] Zynq UltraScale+ MPSoC. URL: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[23] **SE***cube*™ Security Platform. URL: https://www.secube.blu5group.com/.

[24] Secure Image Viewer project. URL: https://www.secube.blu5group.com/resources/open-sources-sdk/.

[25] Secure Text Editor project. URL: https://www.secube.blu5group.com/resources/open-sources-sdk/.

[26] **SE***cube*™ ™ GitHub Repository. URL: https://github.com/SEcube-Project.

[27] **SE***cube*™ Open SDK. URL: https://www.secube.blu5group.com/resources/open-sources-sdk/.

[28] SQLite. URL: https://www.sqlite.org/index.html.

[29] PKCS#11: Cryptographic Token Interface Standard. URL: https://www.cryptsoft.com/pkcs11doc/.

[30] Gallego Gomez Walter. *A Secure Password Wallet based on the* ***SE****cube™ framework*. URL: https://webthesis.biblio.polito.it/8201/1/tesi.pdf.