

POLITECNICO DI TORINO

Department of control and computer engineering



Master's Degree Thesis in Computer Engineering

Design and Development of a Hardware-based Key Management system

Supervisor

Prof. Paolo Ernesto Prinetto

Candidate

Vahid Eftekhari Moghadam

October 2020

Acknowledgements

I would like to express my deep gratitude and appreciation to Prof. Paolo Prinetto for giving me the opportunity to work on this thesis project and his support during this endeavor of mine.

I would also like to thank Nicolò Maunero and Gianluca Roascio for their help and support. Further gratitude to Politecnico di Torino, its directives, and staff for their support and opportunity of study for international students.

Finally, my deepest and sincere gratitude to my family for their continuous love and support.

Abstract

We are living in an era where communication and data exchange are at the center of everyday life and fundamental cornerstone of almost all the daily interactions. We are increasingly leveraging online services that are provided to facilitate our tasks. These actions involve a continuous exchange of private and sensitive information as in online payments, work data, medical information, and so on. Thus, information security and data privacy play paramount importance. Today, information is considered to be an important asset and has a huge impact on societies, governments, and perhaps the future of our home planet.

Cybersecurity, which is the practice of protecting information systems, networks, and programs from digital and physical attacks, helps in establishing a secure medium to data exchange and communication. Cryptography, one of the major topics in cybersecurity, emerged to solve the important pitfall of communication, the privacy, and its goal is to make information exchange between parties involved, private and undisclosed to others.

The mechanisms developed to establish security in cryptography are referred to as cryptographic algorithms, providing their users' features to maintain secure interaction and data protection. These features can be categorized into two main topics: encryption, the act of producing a cipher out of the user data, and decryption which is the act of retrieving information out of the cipher and is complementary to the encryption. The analogy for these processes can be shown as if a piece of information is locked within a box (encryption), and later it's been accessed through unlocking (decryption) and the only way to open the lock is by using a specific key. The digital representation of keys is known as *cryptographic keys* that have unique attributes, like their real-world counterparts. There is a wide range of algorithms that are developed to produce these digital keys, usually involving true random number generation, and define the procedures to encrypt and decrypt data, such as AES and RSA. Key management is an important aspect in this context. Due to the large number of interactions in everyday life and the unfeasibility of manual approaches to control and use keys, automatic systems are developed to uphold these needs which are referred under the term Key Management Systems (KMS).

This thesis aims at developing a set of device-side APIs for the SEcube™ open security platform together with a GUI application for it. The APIs are used to implement a distributed hardware-oriented KMS, in which every user possesses a SEcube™ Hardware Security Module (HSM). Each device functions both as KMS and as a cryptographic coprocessor, providing all the required cryptographic primitives. The implementation provides an environment in which clients can manage encryption keys, users, and groups, impose policies over them, and implement management schemes for their applications. The GUI application is supported under Windows and Linux and designed such that it provides a proof of concept for the KMS, allowing users to interact and use the entire system in a more simple and user-friendly way.

Table of Contents

| | |
|---|----|
| List of Tables | 8 |
| List of Figures | 9 |
| Listings..... | 10 |
| Introduction | 12 |
| State of The Art..... | 17 |
| 2.1 Key management system types | 17 |
| 2.2 NIST specifications for key management..... | 18 |
| 2.2.1 NIST physical level recommendations | 19 |
| 2.2.2 NIST logical level recommendations | 20 |
| 2.2.3 NIST cryptographic keys guidelines | 20 |
| 2.2.4 Other NIST guidelines | 22 |
| SEcube™ Security Platform | 24 |
| 3.1 SEcube™ hardware platform | 24 |
| 3.2 SEcube™ Software platform | 26 |
| 3.2.1 Firmware | 26 |
| 3.2.2 Software Architecture..... | 27 |
| 3.2.3 SEfile™ | 28 |
| 3.2.4 SELink™ | 29 |
| KMS implementation and GUI application development..... | 31 |
| 4.1 SEkey™ architecture..... | 31 |
| 4.1.1 KMS Cryptographic Keys | 34 |
| 4.1.2 KMS Users | 37 |
| 4.1.3 KMS Groups..... | 38 |
| 4.1.4 KMS Policies | 39 |
| 4.2 Firmware implementation | 39 |
| 4.2.1 se3_group Header..... | 40 |
| 4.2.2 se3_group Source | 41 |
| 4.2.3 se3_dispatcher_core integration..... | 45 |
| 4.3 Host-side Application | 48 |
| 4.3.1 Agents overview..... | 49 |
| 4.3.2 Authentication | 50 |
| 4.3.3 Main window | 50 |
| 4.3.4 Add Group window | 52 |
| 4.3.5 Group policy window | 53 |
| 4.3.6 Group user window | 53 |
| 4.3.7 Group Key window..... | 54 |
| 4.3.8 Read Group & Delete Group windows | 55 |

| | |
|---|----|
| Result and Discussion..... | 57 |
| 5.1 Results..... | 57 |
| 5.2 Drawbacks..... | 58 |
| 5.3 KMS sample uses cases..... | 60 |
| 5.3.1 SEcube™ initialization | 60 |
| 5.3.2 Add a group to the KMS..... | 60 |
| 5.3.3 Delete a group from the KMS | 61 |
| 5.3.4 Modify a key state within a group | 61 |
| 5.3.5 Retrieve a group from the KMS | 62 |
| 5.4 GUI application use case..... | 63 |
| Conclusion..... | 67 |
| 6.1 Future work..... | 68 |
| Bibliography | 70 |
| Appendix A, KMS data structures | 71 |
| Appendix B, Group data writing | 72 |

List of Tables

2.1 Key state and phases 21

2.2 Recommended cryptoperid 22

List of Figures

| | |
|--|----|
| Figure 1.1. KMS architecture | 14 |
| Figure 3.1.1. The components of the SEcube™ | 24 |
| Figure 3.1.2. The SEcube™ Hardware device Family | 24 |
| Figure 3.1.3. The SEcube™ DevKit..... | 25 |
| Figure 3.2.1. Request/Response flow | 26 |
| Figure 3.2.2. The device-side and host-side APIs architecture..... | 27 |
| Figure 3.2.3. Simplified step-by-step SEfile™ overview | 28 |
| Figure 3.2.4. SEfile™ File structure | 29 |
| Figure 3.2.5. SELink™ architecture | 29 |
| Figure 4.1.1: KMS context diagram..... | 32 |
| Figure 4.1.2: KMS actors use-cases..... | 32 |
| Figure 4.1.3: KMS class diagram | 33 |
| Figure 4.1.4: State transition scheme | 36 |
| Figure 4.2.1. Structure padding | 40 |
| Figure 4.3.1. Application overview | 49 |
| Figure 4.3.2. Login window | 50 |
| Figure 4.3.3. Main window | 51 |
| Figure 4.3.4. Add Group window | 52 |
| Figure 4.3.5. Group Policy window | 53 |
| Figure 4.3.6. Group User window | 53 |
| Figure 4.3.7. Group Key window..... | 54 |
| Figure 4.3.8. Delete group, Read group windows | 55 |
| Figure 5.4.1. Login window | 63 |
| Figure 5.4.2. Add group window..... | 64 |
| Figure 5.4.3. Group creation steps | 64 |
| Figure 5.4.4. information presentation in the main window | 65 |
| Figure 5.4.5. Invalid group credentials read, message | 65 |

Listings

| | |
|---|----|
| Listing 4.2.1: KMS firmware APIs | 41 |
| Listing 4.2.2: KMS group locating..... | 42 |
| Listing 4.2.3: KMS memory bank | 42 |
| Listing 4.2.4: KMS group creation | 43 |
| Listing 4.2.5: Key state change, part 1 | 44 |
| Listing 4.2.6: Key state change, part 2 | 45 |
| Listing 4.2.7: SEcubeTM security function prototype | 45 |
| Listing 4.2.8: SEcubeTM commands lookup table | 46 |
| Listing 4.2.9: function “group_edit” part 1 | 46 |
| Listing 4.2.10: function “group_edit” part 2 | 47 |
| Listing 4.2.11: function “group_edit” part 3 | 47 |

Chapter 1

Introduction

This thesis work purveys the design and implementation of a hardware-based key management system. The motivation to pursue such a work can be examined by the answers given to the following questions.

- Are cryptographic keys important?
- Do we need a key management system?
- Why a hardware-based key management system?

Cryptographic keys play an important role in today's human life. They are mainly used for encryption/decryption, authentication, and digital signing of messages. To secure the communication and information exchange, the cryptographic keys are heavily exploited. According to Kerckhoff's principle on the cryptography.

“security of a cryptosystem is not dependent on the security or confidentiality of any of its parts, but on the key(s) and the key(s) only.” [\[1\]](#)

Thus the answer to the first question is straightforward, yes, it is the most important element in the digital world that guarantees information security.

Considering the immense usage of the encryption in our daily interactions with the digital world and the dependence of these methods on the cryptographic keys, effective key management plays paramount importance on the confidentiality of any cryptosystem. It's obvious that the manual approaches are no more effective considering the management of the cryptographic keys. Key management systems are designed to have these in mind to properly manage the cryptographic keys. They are capable of generating and distributing cryptographic keys while providing the necessary mechanisms to effectively manage and protect them throughout their entire lifecycle.

Every key management system design has a different set of properties adapted to meet the specific target needs. The common goal pursued by each of the key management systems is to deliver the most secure environment in which the cryptographic keys are exploited to secure the information and communications of its users.

Thus the necessity of an effective key management system is evident. There are many different approaches regarding the implementations of the key management systems, from pure software to hardware-based solutions. Pure software implementations use the hardware resources of

host-machine, for example, a user's PC, to implement their functionalities. This approach could be potentially prone to defects since the underlying software(OS)/hardware resources may poses flaws in them that compromise the integrity of the information and an intruder can exploit these weaknesses to access users' information.

Hardware-based solutions are better in this sense since they are using two-factor authentication methods. They are providing another protection layer that intrinsically improves the security of the system. In a hardware-based solution, users are provided with a hardware module that is unique to them, while using the system, generally, users need to connect their device to a host-machine and provide the credentials required to login to the system, while in the software approaches the authentication is done by providing a password to get access to the system which makes them less secure.

The purposed solution by this thesis work is a hardware-based key management system that is built using the SEcube™ security platform while exploiting the features provided by the SEcube™ open-source SDK. The SEcube™ security platform uses the SEcube™ hardware security module that is a special-purpose module capable of performing encryption/decryption operations allowing the user to move its cryptographic operations away form a host-machine and perform them in a secure environment. It's worth mentioning that the security modules are not stand-alone machines, for their operation they need to be connected to a host machine that can provide the resources necessary for their operation.

The designed key management system is based on a distributed infrastructure where each actor possesses a SEcube™ device while there is a central administrator that works as a key distribution center. The device provides to the users of the key management system secure storage for cryptographic keys and all the functionalities required for their management. Moreover, it also provides the cryptographic primitives needed by the key management system itself or other applications.

The key management system is designed around four key elements.

- Groups
- Users
- Keys
- Policies

There are two main actors defined within the key management system. The admin and the users. Following the NIST guidelines on the separation of the roles and privilege levels defined for each role, the data access is restricted and the operations that can be performed by each actor are limited. In the designed key management system the admin has all the responsibilities to keep the system legitimate and updated. It's the admin's role to define the groups and impose policies over them. The assignment of the users and cryptographic keys are also done by the administrator of the system. Following the separation of roles, solely admin of the system has the capabilities to perform such actions. The users within the groups are only capable of performing encryption/decryption functionalities using the appropriate cryptographic keys belonging to them leveraging the functionalities provided by the SEcube™ device. The

discrimination of the roles is done upon login to the SEcube™ device. Since the administrator is the only person responsible for the management of the system, he/she keeps all the data in order to be able to restore the system in case of an error or malfunctions. Contrary to it, the users are only provided with the subset of information belonging to them.

The key management system comes with a simple graphic user interface application that can be used to perform the actions defined within the SEcube™ framework without the need of using the low-level APIs. While the software application is a complete management tool for the SEcube™ based key management system, the APIs provided within the framework can be integrated by developers in their own software to benefit from the functionalities provided by the SEcube™ open security platform. Figure 1.1 presents an overview of the architecture.

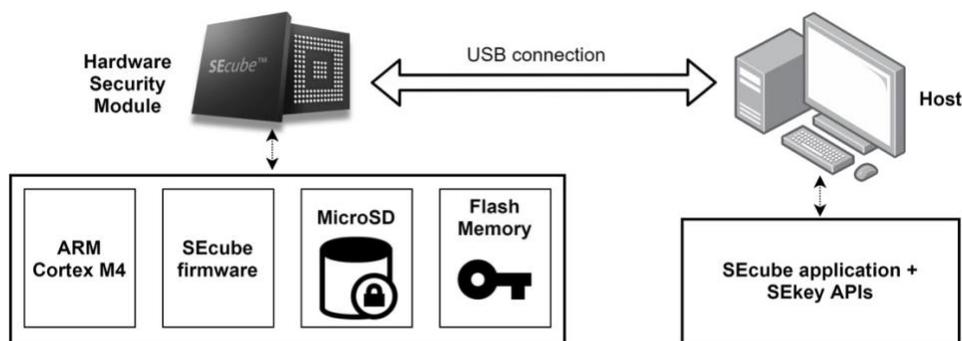


Figure 1.1. KMS architecture [11, p. 10].

Following is the thesis organization. It should be noticed that this thesis work is a joint collaboration between the author and Matteo Fornero on his thesis work titled “Development of a secure key management system for the SEcube™ security platform”. While the concept followed by the key management system shares the common ideas, the implementation path follows two different approaches resulting in different schemes and results. Due to this overlapping the ideas defined in chapter 2 and 3 follows the same structure and present similar concepts.

- Chapter 2

This chapter will overview the guidelines from the National Institute of Standards and Technology (NIST) on implementing a secure and reliable KMS.

- Chapter 3

This chapter will introduce the hardware and software resources used within this project.

- Chapter 4

The implementation details are discussed in this chapter. The general design scheme and some relevant portions of code are also examined in this section.

- Chapter 5

The obtained results together with drawbacks are discussed in this chapter.

- Chapter 6

The work conclusion together with some ideas on further improvements are reported in this segment.

Chapter 2

State of The Art

This chapter will overview the main types of key management systems. We will also examine the NIST recommendations on the cryptographic key management.

2.1 Key management system types

Key management systems can be categorized as follow [\[2\]](#).

- **Software**
In this approach, the KMS is implemented purely in software. It uses the software resources provided by the underlying machine to implement its functionalities.
- **Virtual**
The software resources of the KMS are provided alongside the KMS itself, virtualizing the environment in which it is operating. Underlying resources such as hardware or the medium in which the virtualization is operational is responsibility of its user.
- **Appliance**
The KMS is provided as an integrated hardware/software solution. The implemented hardware-based KMS falls under this category in which there is hardware component, the SEcube™ HSM, and the software component, the SEcube™ open-source SDK.
- **Service**
In this approach the KMS is provided as a cloud-based service to its users. Since this approach don't require the hardware/software resource managed by its user.

It's worth mentioning that each kind of key management system has advantages and disadvantages regarding their use cases. Generally, the software-based solutions are more flexible and the operation cost could potentially be less. Among the solutions, the key management systems that are provided as a cloud-based service are more convenient to its users since the management of the resources is not the user's responsibility. Consequently, the lack of control over the resources could be potentially hazardous and less secure. The solutions that are leveraging special-purpose hardware (e.g. a hardware security module) are generally more secure but they provide less flexibility and can be more expensive.

2.2 NIST specifications for key management

Regardless of features and designs difference, almost every key management system developed following the recommendations of the NIST¹.

According to the NIST [\[3\]](#):

“The proper management of cryptographic keys is essential to the effective use of cryptography for security. Keys are analogous to the combination of a safe. If a safe combination is known to an adversary, the strongest safe provides no security against penetration. Similarly, poor key management may easily compromise strong algorithms. Ultimately, the security of information protected by cryptography directly depends on the strength of the keys, the effectiveness of cryptographic mechanisms and protocols associated with the keys, and the protection afforded to the keys. All keys need to be protected against modification, and secret and private keys need to be protected against unauthorized disclosure. Key management provides the foundation for the secure generation, storage, distribution, use and destruction of keys.”

The documentation published by the NIST addresses every issue that a key management system should take care of, the following is the most important ones.

- **Life cycle**
Each key is associated with a state that determines the usability of it. For example, a key state could prevent it being used for encryption purposes.
- **Access**
The importance of the access to the data regarding the cryptographic keys and the data of the key management system both at physical and logical level impose a necessary protection measurements for them. The NIST suggests using dedicated hardware solution (i.e. a HSM) to physically protect the data. Practices such as encryption, access control and ..., can be done to achieve logical protection.
- **Separation of roles**
It's important to sperate the roles of the actors within the key management system both physically and logically. Logical separation is an important idea from a functional point of view. It can be achieved by defining different privileges for every actor within the key management system. For example, critical functions within the key management system can be limited to be accessed by only an admin level user. Physical separation is another important idea that should be followed. For example, physical accesses to the location in which the important data are stored should be limited to set of authorized people. Locations such as key servers where they hold crucial data should be monitored and limited to access by unauthorized users.

¹ National Institute of Standards and Technology of the US Government

2.2.1 NIST physical level recommendations

Upon developing a key management system from scratch or adapting an already developed key management system solution, it's important to evaluate the environment in which the KMS operates. For example, the storage location of the cryptographic keys or protection mechanisms used are very important. The NIST gives the definition of physical security [4, p. 66]:

“Measures taken to protect systems, buildings, and related supporting infrastructure against threats associated with their physical environment.”

The physical security risk assessment includes: physical access to buildings and facilities, fire safety of buildings where the data are stored, structural integrity of the buildings, facilities backup strategies in case of failure (i.e. electrical power failure, hardware cooling failure, air conditioning failure, etc.), physical displacement of cables in order to limit data interception, mobile device management (i.e. no mobile devices can be introduced in the facilities) [5]. The physical security assessment obviously involves also the hardware that is used to run the key management system, in particular the hardware (i.e. HSMs and general purpose devices) may be certified according to the following criteria [6, p. 1, 2, 3]:

- **Security Level 1:** “No specific physical security mechanisms are required in a Security Level 1 cryptographic module beyond the basic requirement for production-grade components. An example of a Security Level 1 cryptographic module is a personal computer (PC) encryption board. Security Level 1 allows the software and firmware components of a cryptographic module to be executed on a general purpose computing system using an unevaluated operating system.”
- **Security Level 2:** “Enhances the physical security mechanisms of a Security Level 1 cryptographic module by adding the requirement for tamper-evidence, which includes the use of tamper-evident coatings or seals or for pick-resistant locks on removable covers or doors of the module. Requires, at a minimum, role-based authentication in which a cryptographic module authenticates the authorization of an operator to assume a specific role and perform a corresponding set of services.”
- **Security Level 3:** “Intended to have a high probability of detecting and responding to attempts at physical access, use or modification of the cryptographic module. The physical security mechanisms may include the use of strong enclosures and tamper detection/response circuitry that zeroizes all plaintext CSPs (Critical Security Parameters) when the removable covers/doors of the cryptographic module are opened.”
- **Security Level 4:** “The physical security mechanisms provide a complete envelope of protection around the cryptographic module with the intent of detecting and responding to all unauthorized attempts at physical access. Penetration of the cryptographic module enclosure from any direction has a very high probability of being detected, resulting in the immediate zeroization of

all plaintext CSPs. Security Level 4 cryptographic modules are useful for operation in physically unprotected environments.”

2.2.2 NIST logical level recommendations

In the design of a key management system it's very important to define the role of the each actor within the system from the logical point of view. The following principle definitions from NIST are of paramount importance.

The 'Separation of Duties' [7, p. 19]:

“A security principle that divides critical functions among different staff members in an attempt to ensure that no one individual has enough information or access privilege to perpetrate damaging fraud.”

The 'Split Knowledge' [3, p. 19]:

“A process by which a cryptographic key is split into n key shares, each of which provides no knowledge of the key. The shares can be subsequently combined to create or recreate a cryptographic key or to perform independent cryptographic operations on the data to be protected using each key share. If knowledge of k (where k is less than or equal to n) shares is required to construct the key, then knowledge of any $k - 1$ key shares provides no information about the key other than, possibly, its length.”

The 'Least Privilege' [7, p. 17]:

“A security principle that restricts the access privileges of authorized personnel (e.g., program execution privileges, file modification privileges) to the minimum necessary to perform their jobs.”

These three principles draw an important guideline in the development of a key management system. A key management system could be compliant with these guidelines in their design or they could drop some due to criteria and environment in which they operate. For example, in the design of the hardware-based key management system subject to this thesis, the 'Least Privilege' and "Separation of Duties" are adopted while the "Split Knowledge" principle due to implementation complexity and hardware resource limitation not followed.

2.2.3 NIST cryptographic keys guidelines

Cryptographic keys are at the heart of any key managements system, following are the important considerations about their management. [8, p. 14].

- if a key is exposed by any entity having access to it, then all data protected by that key are compromised.

- The probability of exposing the value of a key is proportional to the number of entities having access to that key.
- The probability of a key to get compromised is proportional to the time span during which it is used and to the amount of data that it protects.
- The entity that is entitled to generate and distribute cryptographic keys must be trustworthy. The same rule applies to the recipients of the keys. A strong authentication mechanism to assess the identity of those entity is preferred.
- The communication channel that is used to distribute the keys and the related information must be protected.

These rules should be considered in the design and implementation of any key management system. There are many other guidelines regarding the cryptographic keys specification that NIST documentation provides, such as key states. The NIST recommends six possible states that can be grouped into four operational phases [3, p. 98], as listed in Table 2.1.

| State | Operational Phase |
|----------------|-------------------|
| Pre-activation | Pre-operational |
| Active | Operational |
| Suspended | Operational |
| Deactivated | Post-Operational |
| Compromised | Post-Operational |
| Destroyed | Destroyed |

Table 2.1. Key states and phases

The state definitions can be used to model the cryptographic keys lifecycle. The state assignment and transitions between them can be tailored to meet the specific requirements of each key management system.

Another important parameter that is closely connected to the cryptographic keys state is cryptoperiod. The definition is, “The time span during which a specific key is authorized for use or in which the keys for a given system or application may remain in effect.” [3, p. 10].

The cryptoperiod is usually the sum of two components: the originator usage period (OUP) and the recipient usage period. The OUP is, “The period of time in the cryptoperiod of a key during which cryptographic protection may be applied to data using that key.” [3, p. 16] while the recipient usage period is, “The period of time during which the protected information may be processed (e.g., decrypted).” [3, p. 17].

The cryptoperiod plays a fundamental role in the management of the keys, since it may be the source of security flaws and potential risks that may compromise the security of the data. Accordingly, the NIST also suggests a possible cryptoperiod for cryptographic keys, depending on their type and usage. In the Table [2.2](#), suggested cryptoperiods of the key types that are used in SEkey is listed. More information can be found in NIST documentation [[3](#), p. 50]. The reported values are intended as a guideline, one should adopt the values according to their criteria.

| Key Type | Originator Period | Recipient Period |
|---------------------------|-------------------|------------------|
| Symmetric Authentication | <= 2 years | <= OUP + 3 years |
| Symmetric Data Encryption | <= 2 years | <= OUP + 3 years |
| Symmetric Authorization | <= 2 years | <= 2 years |
| Symmetric Key-Wrapping | <= 2 years | <= OUP + 3 years |

Table 2.2. Recommended cryptoperiods.

2.2.4 Other NIST guidelines

The documents published by the NIST about cybersecurity are extremely detailed and complex. Many other topics and problems should be addressed, here is a couple of additional issues to be considered [[5](#)].

- Business continuity, defined as the capability of delivering a service even in case of a disruptive event. In the case of SEkey, there is a recovery procedure that allows to restore the data of the SEcube HSM of a specific user, but no other solutions have been implemented to grant business continuity on the administrator side.
- Hot failover, consisting in having a backup infrastructure for the main elements of a specific network or environment. In the case of SEkey, and only for the administrator, this would require a SEcube device with a real-time backup of the data of the KMS and possibly another host computer. Again, this is not really an issue for a simple key management system such as SEkey.

These issues will not be discussed anymore because they are advanced features essential for commercial, safety critical, and mission critical systems; they are less important for open-source and/or academic solutions. On the other hand, the recommendations mentioned in Section [2.2.1](#), Section [2.2.2](#), and Section [2.2.3](#) will be discussed in detail along with the actual solutions implemented by the key management system presented in this thesis.

Chapter 3

SEcube™ Security Platform

This chapter introduces the underlying hardware/software resources used for this work. Information provided here is necessary to understand the organization and implementation of the system implemented.

3.1 SEcube™ hardware platform

The core of the SEcube™ hardware device is a 3D SiP (System in Package). The three main components are illustrated in Figure 3.1.1: a low-power ARM Cortex-M4 processor, a high-performance FPGA, and an EAL5+ certified security controller (commonly known as a smartcard). SEcube™ is a unique security environment for a great variety of applications.



Figure 3.1.1. The components of the SEcube™ [9, p. 9].

The SEcube™ chip is available on a family of devices, from chip to PCI-express Board, and namely (Figure 3.1.2), [9, p. 10].

- The Chip, named: SEcube™ Chip, or simply SEcube™
- The Development Board named: SEcube™ DevKit
- The Stick, named: USEcube™ Stick
- The Phone, named: SEcube™ Phone
- The PCI-express Board named: SEcube™ PCIe.



Figure 3.1.2. The SEcube™ Hardware device Family [9, p. 10].

The SEcube™ DevKit is an open development board (Figure 3.1.3) designed to support developers to integrate the SEcube™ Chip in their hardware and software projects.

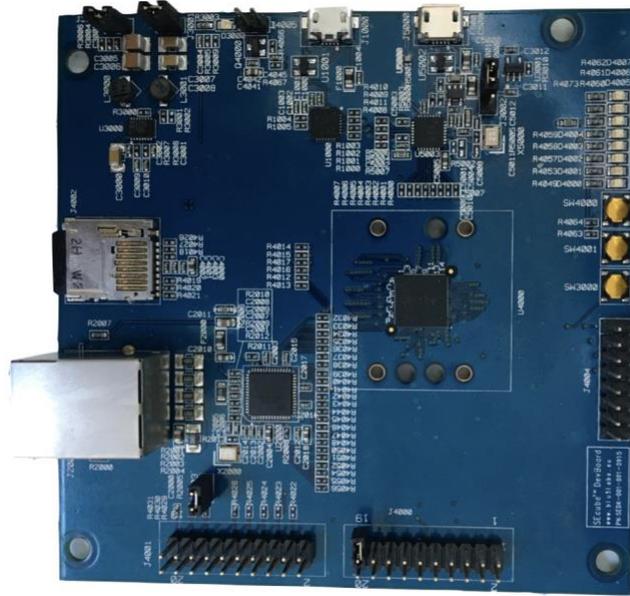


Figure 3.1.3. The SEcube™ DevKit [9, p. 19].

The processor adopted within the SEcube™ is the STM32F429², produced by the ST Microelectronics™, which includes a high-performance ARM Cortex M4 RISC core and provides the following features [9, p. 17].

- 2 MiB of Flash memory
- 256KiB of SRAM
- 32-bit parallelism
- Operating frequency of 180MHz
- Low power consumption

This CPU offers several features that make it suitable for high-performance and security-oriented solutions. For example, it supports the Cortex CMSIS implementation that provides, among the others, the CMSIS-DSP libraries: a collection with over 60 DSP functions for various data types. The CMSIS-DSP library allows developers to implement complex, real-time operations using the embedded hardware Floating Point Unit.

Moreover, the CPU provides several peripherals such as SPI, UART, USB2.0, and SD/MMC, which ease the hardware integration in diverse devices. For example, a secure USB device can be easily realized using the USB2.0 and the SD card interfaces, respectively.

² <https://www.st.com/en/microcontrollers-microprocessors/stm32f429-439.html>

On the security side, a TRNG (True Random Noise Generator) embedded unit, hardware mechanisms like MPU (Memory Protection Unit), and privileged execution modes allow implementing the security strategies required by a certified secure controller (e.g., privileged memory areas, key generation, etc.).

For programming, debug, and testing operations, the CPU provides a standard JTAG interface that can be permanently disabled once the development cycle is over, protecting all the sensitive information through a physical hardware lock. The FPGA element, a Lattice MachXO2-7000 device, is based on a fast, non-volatile logic array. The FPGA exposes 47 general-purpose I/Os which may be used as a 32-bit external bus, able to transfer data at 3.2 Gib/s.

The third component of the SEcube™ Chip is an EAL5+ certified security controller, based on a secure chip produced by Infineon. Within the SEcube™ Chip, the SmartCard is connected to the CPU via a standard ISO7816³ interface. The smartcard does not expose any interface outside the chip. This architectural solution provides high-grade and certified security functionalities behind a simpler and easy-to-use application interface. The work of this thesis and all the feature designed for SEkey™ have been developed resorting only to the MCU.

3.2 SEcube™ Software platform

3.2.1 Firmware

The SEcube™ platform is associated with its own dedicated firmware, the core that manages and handles all the requests coming to the device from a host computer. This interaction can be represented as a Client-Server scheme in which the device is the server for the incoming commands. For example, a user can ask the device to encrypt/decrypt a given payload and the device will respond with the result of the operation. The flow is depicted in Figure 3.2.1.

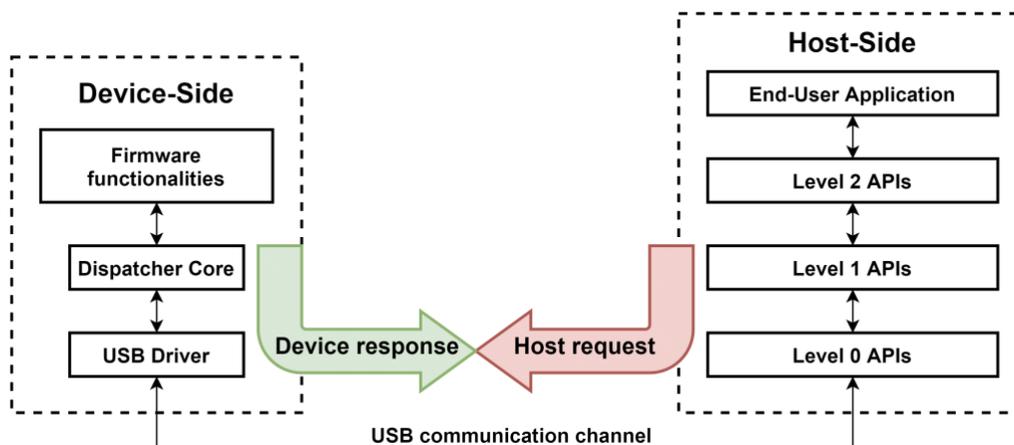


Figure 3.2.1. Request/Response flow [11, p. 23].

³ <https://www.iso.org/standard/54089.html>

3.2.2 Software Architecture

The SEcube™ open-source software is structured in several abstraction layers as they are depicted in the figure: 3.2.2, usually referred to as L0, L1, L2, and Host Application, respectively. At each Abstraction Layer, several sets of APIs are provided. These APIs are required to communicate with the SEcube™ and exploit its features. The SEcube™-side APIs are executed on the embedded processor of the SEcube™ - based hardware device (e.g., the USEcube™), whereas at the External-Side the software is tailored for existing devices (e.g., laptops or Desktop PC).

- Level 0: The closest layer to the device and implements basic functionalities such as communication with the device, initializing the SEcube™ serial number, transmission/reception of data between the host machine and the device.
- Level 1: The basic functionalities required for the security applications are provided in this layer, for example, login/logout operations, encryption, and decryption of payload, and so on.
- Level 2: This level of APIs provides functionalists to implement a secure environment in which details about underlying software/hardware are abstracted from its user point of view. A library such as SEfile™ and SELink™ are examples of this level of APIs.

The structure of these APIs is hierarchical. At each level, each component represents a service for the upper level and relies on services provided by lower-level APIs. this scheme is applicable only for the host-side set of APIs, the device-side scheme is more complex and distributed.

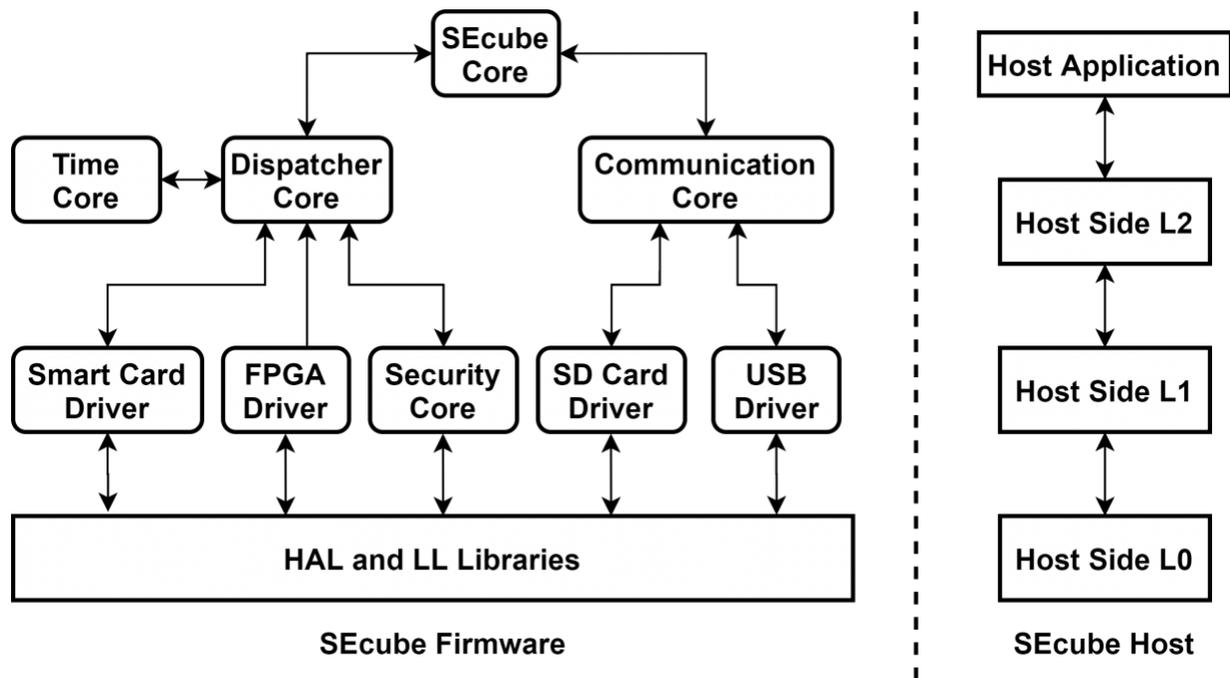


Figure 3.2.2. The device-side and host-side APIs architecture [9, p. 11].

3.2.3 SEfile™

SEfile™ aimed at providing a secure environment in which users can perform basic file operations on files. SEfile™ design is done in such a way that it hides from the host device the details about the location and the operation kind done on a file, thus it provides a secure medium in which a user can perform his/her action in insecure host machines. A general overview of how the SEfile™ works, is depicted in figure [3.2.3](#)

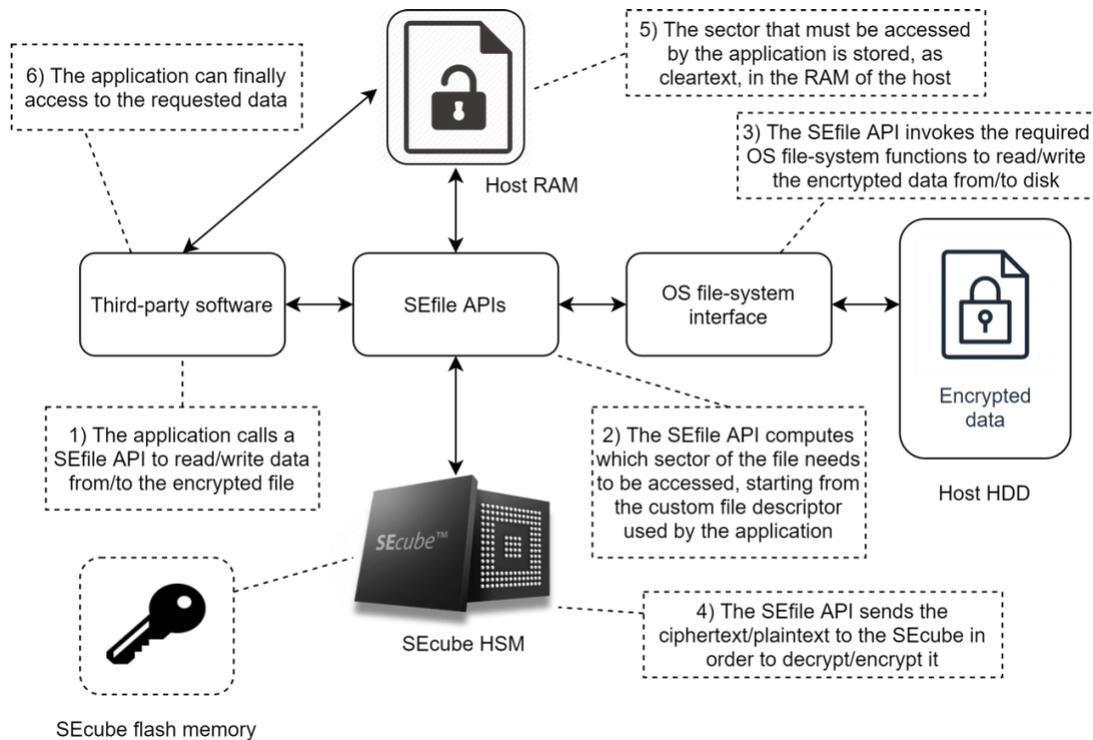


Figure 3.2.3. Simplified step-by-step SEfile™ overview [11, p. 26].

The structure of a secured file is shown in figure [3.2.4](#). The file data is divided into sectors and each sector is encrypted and signed by the specific encryption algorithms. The first sector poses metadata and is referred to as a header. When a user needs to access the file for reading/writing, only the portions of the file required to be accessed are encrypted or decrypted, the rest will be untouched. This will result in time overhead reduction while maintaining secure interaction with the file. SEfile™ uses the AES encryption algorithm established by the U.S. National Institute of Standards and Technology (NIST) for file encryption. “For each data sector, AES-256-CTR is used, while the header sector is encrypted using AES-256-ECB”. [9, p. 42].

For authentication, “each sector, including the header, is signed using an authenticated signature obtained with SHA-256-HMAC, meaning that the signature depends on both the data contained in the sector itself and on a chosen encryption key. To use two different keys to encrypt data and to digest authentication, a feature increasing overall system security, SEfile™ leverages on the pbkdf2() function already implemented within the SDK.” [9, p. 43].

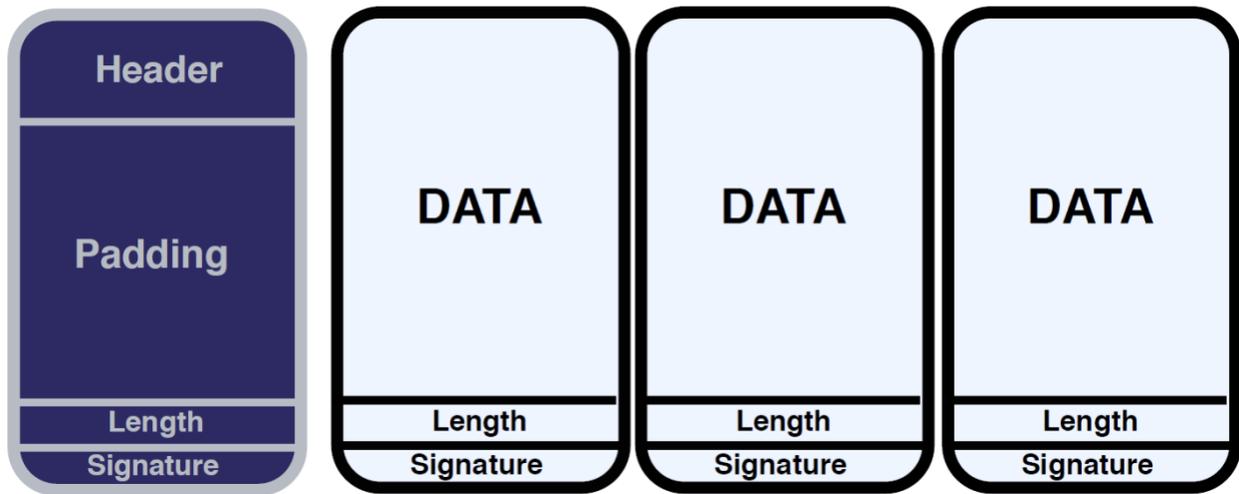


Figure 3.2.4. SEfile™ File structure. [10, p. 22].

3.2.4 SElink™

SElink™ is an application implemented using the SEcube™ open-source platform securing the network traffic of the user. It encrypts the streams of data that the user application sends over the network. The encryption is agnostic to the application-level protocol. By exploiting the SElink™, users can benefit from a secure network layer added to their application without applying changes to their design. For example, a user can integrate the SElink™ to his/her network-enabled software and customize the SEcube™ platform security features according to his/her needs and benefit from security functionalities. Since the SElinks™ interacts at the lower-level level of communications, the application will not notice the presence of the SElink™ and can continue to function as it's been designed.

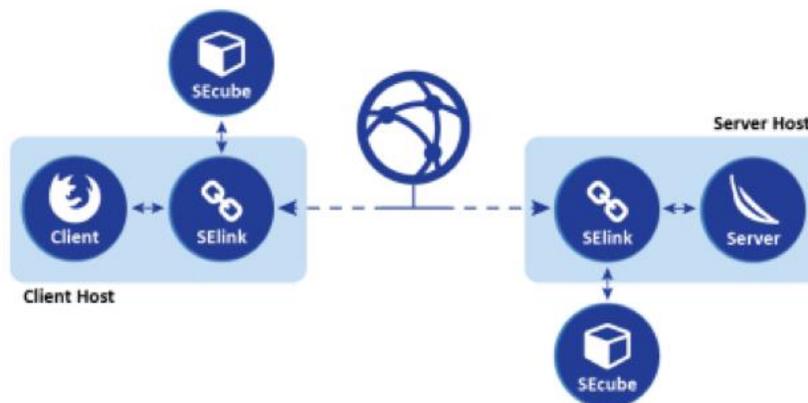


Figure 3.2.5. SElink™ architecture [9, p. 56].

Chapter 4

KMS implementation and GUI application development

This chapter presents the implementation details of the developed hardware-based key management system according to the insights and criteria explained in the NIST standard on the topic of key management systems and the management of their lifecycles. We will see in greater detail the internal structure of the group's data together with SEcube™ open-source firmware APIs that are exploited to achieve the desired outcome. The graphical user interface application internal organization will be elaborated and discussed in further detail to give an in-depth insight into the underlying software which is developed for the key management system.

4.1 SEkey™ architecture

The key management system scheme that is employed in our design enables the users of it to securely communicate and exchange data between themselves. To be able to establish this interaction the users must be within the same system meaning that all the users should employ a SEcube™ based device for their communication.

The hardware-based key management system is implemented using four main key concepts, which are as follow:

- Keys
- Users
- Groups
- Policies

The designed key management system collects the users and assigns them to specific groups. Internally, the key management system assigns to each group a specific symmetric key upon request of group creation or modification of an already existing user group. These keys can be used later by the users belonging to the same group to securely communicate and exchange data. More specifically, they are used for encrypting or decrypting data.

Through this mechanism a user can communicate with others who share at least one common group with him/her, employing the cryptographic key that belongs to their common group. In cases where a user shares more than one group with a specific target user, the system arbitrates among common groups and chooses to secure communication by selecting a cryptographic key that belongs to the least numerous common group among them. This approach imposes a more secure way of interaction since it hides the users to be seen by other users that are not belonging

to the same group and since it establishes communication using the cryptographic keys belonging to a smaller set of groups, in case of an intruder, it intrinsically reduces the chance of information leak. In the context in which the key management system is present, multiple SEcube™ based devices are available and they all share the same key management system and it's distributed among them.

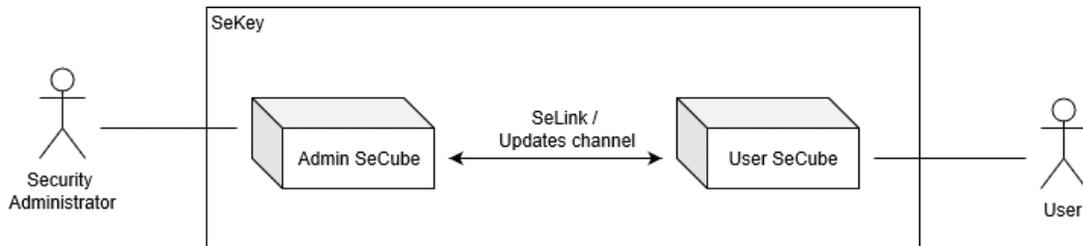


Figure 4.1.1: KMS context diagram [9, p. 74].

As it can be seen from the KMS context diagram, the admin's SEcube™ acts as a server to the SEcube™ devices of the users. To keep the communication legitimate and secure, user's SEcube™ devices should frequently connect to their admin's SEcube™ device and update themselves with the most recent configuration and data available to them. The communication between the devices should be protected by using the master key specific to each user. The admin role in this scheme is the management of the information and organization regarding the groups, such as group creation, elimination, modification. Users assignment to their corresponding groups, key assignment to each group, and keeping track of their status, imposing policies on the groups, and act on these whenever necessary to keep the communication secure and legitimate.

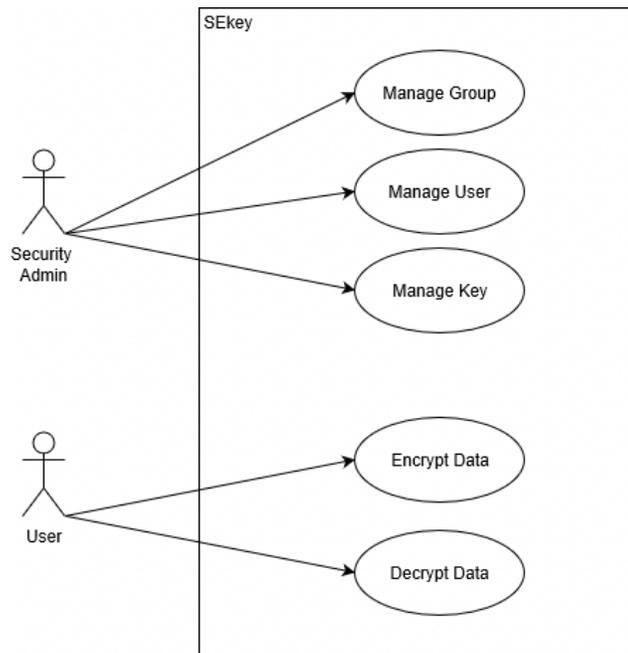


Figure 4.1.2: KMS actors use-cases [9, p. 75].

As it can be seen from the figure 4.1.2, the differentiation of functionalities available to each actor of the key management system are according to their prospective tasks. The users can communicate and exchange data using their SEcube™ devices while securing their data by functionalists available to them, data encryption and decryption.

Following is the class diagram representation of the building blocks regarding the implemented key management system. As stated before there are multiple SEcube™ devices involved in the system. There is always a single SEcube™ based device for admin purposes, the rest are dedicated for the users of the key management system. The SEcube™ devices of the users only poses the information regarding their scope which means they have access to the information that only belongs to them, thus the admin SEcube™ device has privileged access to all the groups. It is the admin responsibility to keep track of all the updates necessary to each groups, thus it stores all the information of the system regarding the groups, users, keys and policies within its SEcube™ device.

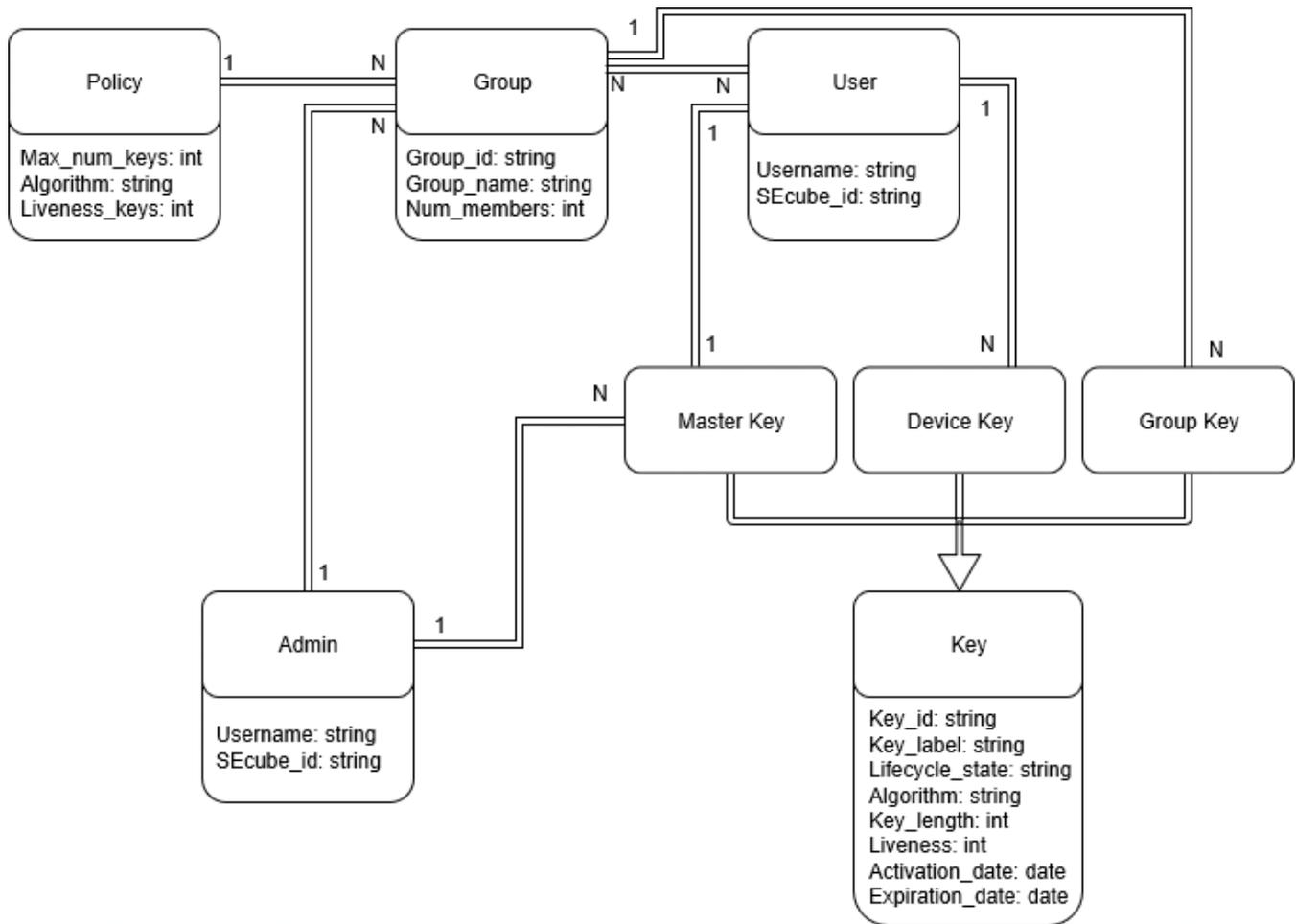


Figure 4.1.3: KMS class diagram [9, p. 74].

As can be seen from figure [4.1.3](#), there are three min key groups used to regulate the interactions between the users and protecting the system from unauthorized accesses.

- **Master Key**

A master-key is a symmetric cryptographic key specific to each SEcube™ device. They are used for transferring updates and are employed for each device uniquely for admin-user communications.

- **Device Key**

A device-key is a cryptographic key that is located within the SEcube™ device and is used for internal data encryption. Their data should not be exposed to the outside world, thus they need to be managed carefully since they protect the data at rest located within the SEcube™ device.

- **Group Key**

These kinds of keys are heavily used in the key management system for communication. They are shared between users of each group and are updated most frequently to keep the communication and data exchange between them more secure.

4.1.1 KMS Cryptographic Keys

The implemented key management system employs cryptographic keys to encrypt or decrypt communications between the users of the SEcube™. The encryption keys used in this communications are static kind, which is used in symmetric encryption algorithms. Contrary to the asymmetric encryption/decryption algorithms in which the users communicated using public and private keys to secure their data transfer, this approach only uses one key both for encryption and decryption of users' data. The motivation of choosing such a method is due to the support of the SEcube™ open-source SDK on symmetric encryption, although it can be expanded further to support other encryption methodologies. Following is the list of attributes used by the key management system for the cryptographic keys.

- **Key Identifier**

A unique alphanumeric string of characters representing a key in KMS. Using this tag one can retrieve a key form a set of keys.

- **Key Label**

A human-readable text describing the key (i.e. key_1_group_1).

- **Owner Identifier**
A unique alphanumeric string of characters representing the group which the key belongs to.
- **State**
Indicates the cryptographic key state according to the predefined options (pre-active, active, and ...). This value determines the usability of the cryptographic key. For example, a key in an active state can be used to encrypt data.
- **Cryptographic Algorithm**
This value determines the kind of algorithm, supported by the firmware, to be used for encryption or decryption.
- **Length**
Since cryptographic algorithms use a different kind of key lengths, this value represents the length of the keys in Bytes.
- **Key Value**
This field holds the data of the cryptographic key. The key data is not available to the user. It's generated internally and could be used with APIs such as SEfile™, for encryption/decryption purposes.
- **Generation Time**
The timestamp at which a key has been created.
- **Activation Time**
The timestamp at which a key has been activated. For example, a key could be created but not used, until the already active key becomes deactivated or suspended.
- **Expiration Time**
The timestamp at which a key will be deactivated. This field will be compared internally with the Cryptoperiod value of the key and based on the comparison outcome, the key will be deactivated.
- **Cryptoperiod**
The time interval during which the cryptographic can be actively used for communication and data exchange.

Each of the above mentioned attributes is stored alongside the keys in the SEcube™ as metadata. As mentioned above, the attributes that are used for identification, are created once and cannot be changed later since they may cause integrity issues. At the moment the APIs provided by the SEcube™ firmware supports for the AES encryption/decryption algorithms. As will be discussed later in the source code analysis, the values and size of the fields are designed considering these criteria in mind. The cryptoperiod field is an important value that is used to keep the

communication secure. As a key can be created anytime without making it active for encryption purposes, the value represented by this field is used to calculate the time in which the key must be deactivated for further use starting from transition to an active state. The expiration time will be compared with this value. The decision on the key state would be evaluated by comparison between expiration time and time calculated by the cryptoperiod plus the activation time. The lower value would determine the state of the cryptographic key.

The key state is another important factor that makes the use of cryptographic keys more secure. Following the recommendations about the key lifecycle from the NIST, during its lifecycle, a key can pass through a few states. The state of the key determines the action that can be done, using that specific key. Following is the figure that represents a possible state transition.

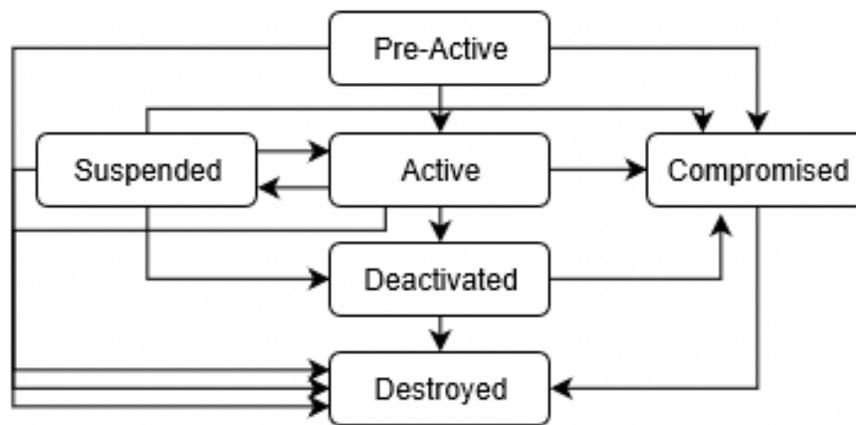


Figure 4.1.4: State transition scheme [3, p. 87].

- Pre-active**
 It's the default state when a key is generated. Since it's not yet activated, it can be used neither for encryption or decryption of data. As state transition depicts, a pre-active key can pass to a compromised or destroyed state without being activated.
- Active**
 A key in this state can be used to protect the data and secure communication. Encrypting data can be done only with a key in this state.
- Suspended**
 In case a key state becomes suspended, it shouldn't be used to encrypt data, however, it can be used to decrypt the data that was previously encrypted using the key. A key in this state can go back to the active state as the state transition diagram demonstrates.
- Compromised**
 A compromised key should not be used for encryption purposes. Since the name suggests, the key information could've been leaked outside the SEcube™ device due to some human error or by an act of an intruder. It can be exploited to process protected data

encrypted using that key under caution with knowledge of possible consequences. The data protected by the key in this state is no more secure and should be re-encrypted using another key whose information is not exposed, thus has the legitimacy of usage.

- **Deactivated**

When a key reaches this state, it cannot be used for encryption anymore. Unlike a key in the compromised state, the deactivated key can be continued to be used decrypting data. Automatically when a key reaches its expiration, the key state becomes deactivated.

- **Destroyed**

As a key becomes no longer needed, it should pass to this state. The key in this state is not part of the key management system anymore, however, its metadata could be preserved. The key value in this state does not exist, thus all the data encrypted using the key-value are lost and out of reach.

The state declarations that are presented here are one of the many possible solutions available for keeping track of encryption keys. One can omit a specific state or add more states according to their needs. It's important to the user of the implemented key management system noticing the administrator role. Since the key management system uses a server-client scheme as depicted in figure [4.1.1](#), it is the admin's responsibility to keep track of changes relating to the state of the keys. Failing to do, will cause incompatibility issues related to the use of cryptographic keys or data leaks due to poor management.

4.1.2 KMS Users

Each user (including the admin) in the key management system is associated with a SEcube™ based device. Although functionalities available to the admins' SEcube™ are different than those of users' SEcube™, each user is characterized using the following attributes.

- **User ID**

A unique alphanumeric string of characters representing a user in KMS. Using this tag one can retrieve a user from a set of users.

- **Username**

A human-readable text describing the user (i.e. user_1).

- **SEcube™ SN**

The serial number of the user's SEcube™.

- **Update key**

As stated before in figure [4.1.3](#) an update key, also known as "Master Key", is the encryption key used by the administrator to encrypt the data sent to the user.

There are some functionalities built into the key management system regarding the actions related to the user. Following is the listing of the functionalities.

- New user creation
- User assignment to a group
- User elimination from a group
- User credentials modification

4.1.3 KMS Groups

In the key management system, a group is a medium in which, users can communicate and exchange data between themselves. Each group is represented by the following attributes.

- **Group ID**
A unique alphanumerical string of characters representing a group in KMS. Using this tag one can retrieve a group from a set of groups.
- **Group Name**
A human-readable text describing the group (i.e. group_1).
- **Number of Users**
- **Number of Keys**
- **Policy**
Each group upon creation has a predefined set of rules. As will be described in the next section, the policy values are set according to the context and criteria.
- **User List**
As a group cannot be empty, each group has a list containing the users' information belonging to that group. Only users in the group list can communicate with each other.
- **Key List**
Each user in a group uses one of the keys available in the group key-list to communicate data. The selection of a proper key depended on the type of action and data itself.

Following is the list of functionalities provided by the key management system for group related actions.

- Group creation
- Group elimination
- Group modification
- User insertion to a group

- User elimination from a group
- Key insertion to a group
- Key modification within a group

It is worth mentioning that user insertion to a group or user elimination from a group will result in the creation of a new group containing all the information as of the original group reflecting the changes made to it.

4.1.4 KMS Policies

A policy is a set of predefined rules imposed on each group within the key management system. These policies are defined according to the context on which a group operates.

- **Maximum Number of Keys**
An upper limit is defined on the number of keys each group can poses.
- **Algorithm**
A default algorithm which is used by the group to secure its communication. The encryption key algorithm attribute overrides this value.
- **Liveness**
The default value which is used by groups' keys to determine their cryptoperiod. The encryption key cryptoperiod attribute overrides this value.

4.2 Firmware implementation

As it was discussed in the introduction, the SEcube™ open-source SDK software architecture provides different functionalities in different levels. To be able to successfully implement a task through an application, one should use level 0 APIs to establish communication with the device. User authentication also needs to be performed through level 1 set of APIs. In the key management system, these APIs are heavily exploited.

The firmware expansion of SEcube™ open-source SDK resulted in the addition of two files. These files can be located under the “Device” subfolder of the main source files.

- se3_group.h
- se3_group.c

4.2.1 se3_group Header

In this header file, the data structures are defined according to the implementation scheme discussed in section 4.1. Starting from the size limit defined for each attribute, one can change their limit by modifying the corresponding definitions. It's worth mentioning that any change in size limit would modify the corresponding payload that is sent on each communication. When a SEcube™ sends or receives information, the communication is done by using a buffer. The data representation in this buffer starts from a section right after the payload information. The fetching of the data both in the device-side and host-side depends on the sizing of each element. Any change to the sizing would imply a corresponding modification on the address of each element within the payload. It's the responsibility of the actor to follow the changes in case of code alteration. Appendix A presents the structure definitions. In the development of the key management system, there was an attention to the integration with the current version of the SEcube™ firmware. The current version of SEcube™ open-source SDK provides to its users the AES cryptographic algorithms, and the supported key size is up to 256 bits. Therefore the "key_value" is defined as a 32-bit array of 8-bit width unsigned integer type. As discussed in section 4.1.1, cryptographic key states are represented, as an enumeration that is used to set their corresponding states.

Since the key management system stores, all the data related to groups, users, keys, and metadata within the flash of the device, optimum usage of space is crucial. To improve the performance of data transactions, compilers often add one or more empty spaces to data represented as structures to align them in memory when they are allocated in memory. This concept is known under the term, "structure padding". Due to this optimization, the structure size is not always compliant with the expected size of the data types. Following is the representation of such a phenomenon.

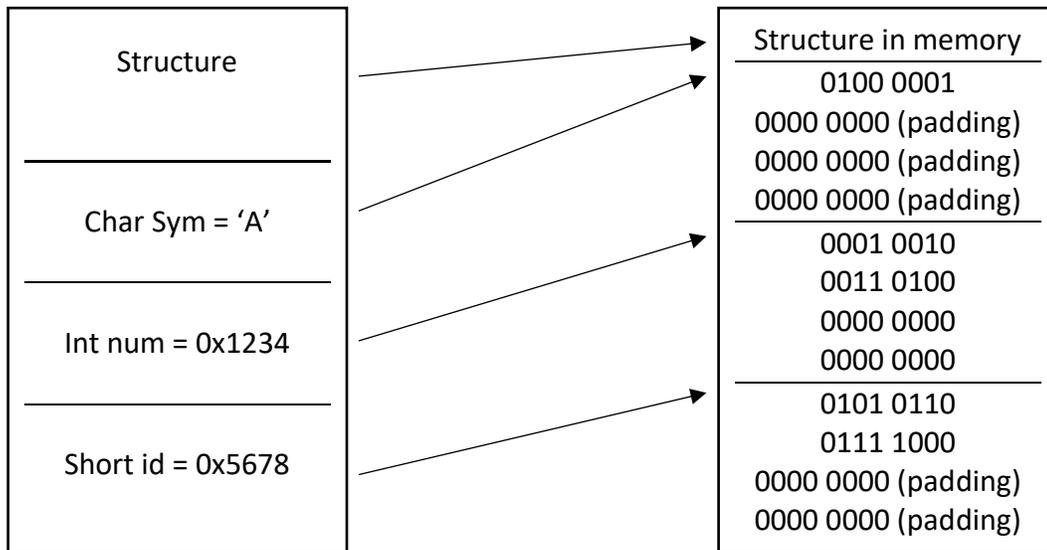


Figure 4.2.1. Structure padding

To overcome this optimization behavior, one can use “__attribute__((packed))”. This feature can be applied to each specific field. Since we have limited buffer space for data exchange, it’s applied to the whole structure.

According to the functionalities defined for managing the groups and the keys within the key management system. The following, are the functions designed to achieve the desired actions. As stated before, the actions related to managing the users within the groups are not present in the firmware since any alteration in the users' list of groups will result in a creation of a new group. The implementation of those functionalities is done in the host-side APIs. For example, if a user needs to be added to a group, the user of the SEcube™ (The admin of the system has such privilege to alter the group and their related metadata), can read the data related to the desired group, impose the users' list of the group by adding a new user to it and adding the newly created group by the APIs that implements the task.

```
1. bool se3_group_find(char* id, se3_flash_it* it);
2. // given group_id, it checks for presence of group in device flash.
3.
4. bool se3_group_new(se3_flash_it* it, se_group* group);
5.
6. bool se3_group_write(se3_flash_it* it, se_group* group);
7. // Notice --> this function is only used in device side.
8. // Notice --> comment this function on host side library.
9.
10. uint16_t se3_group_read(se3_flash_it *it, uint8_t *resp_buffer);
11.
12. bool se3_group_change_key_state(se3_flash_it* it, char* key_id,
    uint8_t state);
13.
14. bool se3_group_change_key_remove(se3_flash_it* it, char* key_id);
15.
16. bool se3_group_change_key_add(se3_flash_it* it, se_key* new_key);
17.
18. bool se3_group_active_key(se3_flash_it* it, se_key* new_key);
```

Listing 4.2.1: KMS firmware APIs

As the naming suggests, these APIs are can be used considering the actions for groups and keys. In the following section we will discuss in detail their implementation details.

4.2.2 se3_group Source

In this section, we will discuss the implementation details of the key management system following the objectives defined in section 4.1. Following the state definitions and the transition diagram depicted in figure 4.1.4, the “check_transition” function has been defined to evaluate the legitimacy of a change in the key states. Followings are the acceptable state transitions.

- A key in a “pre-active” state can be set to “active”, “suspended” or “destroyed” state.
- A key in an “active” state can be set to any state except “pre-active” state.
- A key in a “suspended” state can be set to any state except “pre-active” state.
- A key in a “compromised” state can be set only to “destroyed” state.
- A key in a “deactivated” state can be set to both “compromised” or “destroyed” state.
- A key state in a “destroyed” state cannot be changed to any other state.

The “check_transition” function is a static function and is used internally to evaluate the state transitions and its scope is limited to the “se3_group.c” file. To check the existence of a group within a SEcube™, a user can use the “se3_find_group” function. Upon a successful outcome, the user will have access to the location of the group within the flash of the SEcube™ device. Following is the function definition.

```

1. bool se3_group_find(char* id, se3_flash_it* it)
2. { char group_id[5]={'0','0','0','0','\0'};
3.   se3_flash_it_init(it);
4.   while (se3_flash_it_next(it)) {
5.       if (it->type == SE3_TYPE_GROUP) {
6.           strcpy(group_id, it->addr);
7.           if (strcmp(group_id,id)==0) {
8.               return true;
9.           }
10.        }}
11. return false;}

```

Listing 4.2.2: KMS group locating

It is important to notice the “se3_flash_it” structure since it’s heavily exploited for the memory operations within the SEcube™. The flash memory of the SEcube™ is located within the CPU core, STM32F29™. The internal flash organization consists of several banks which in turn divided into segments. More information on the internal organization of the flash memory can be found in the STMicroelectronics™ manual⁴. The current version of SEcube™ open-source SDK uses two main sectors of the internal flash storage for its functionalities.

```

1. #define SE3_FLASH_S0_ADDR ((uint32_t)0x080C0000)
2. #define SE3_FLASH_S1_ADDR ((uint32_t)0x080E0000)

```

Listing 4.2.3: KMS memory bank

⁴ https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf , page 79

Each sector used within the SEcube™ has been managed using flash nodes. The detailed information on how they are organized and their implementation details can be found in the “se3_flash.h” header file. The “se3_flash_it” structure defines an iterator that allows locating data within the sector fast and straightforward. The “se3_group_find” function uses this structure to locate a target group within the SEcube™ devices of the key management system. The “se3_group_find” function accepts as a search pattern the group-id. Using the flash iterator functionalities of the SEcube™ firmware, it searches the internal flash storage to locate the target.

The creation of a group is done using the “se3_group_new” function. This function accepts as an input the group metadata information. It checks for storage space availability. Upon successful evaluation, it creates a group within the SEcube™ flash memory with the provided data.

```
1. bool se3_group_new(se3_flash_it* it, se_group* group)
2. {
3.     uint16_t size= (33+(sizeof(se_user)*(group->num_users))+(sizeof(se_key)*(group->num_keys)));
4.
5.     if(size > SE3_FLASH_NODE_DATA_MAX){
6.         return false;
7.     }
8.
9.     if (!se3_flash_it_new(it, SE3_TYPE_GROUP , size)) {
10.         SE3_TRACE("E group_new cannot allocate flash block\n");
11.         return false;
12.     }
13.
14.     return se3_group_write(it, group);
15. }
```

Listing 4.2.4: KMS group creation

The “se3_group_new” function exploits the “se3_group_write” function internally to transfer the data related to the groups to the flash memory of the SEcube™ device. Appendix B presents the implementation. This function will not succeed unless it finishes all the data writings. Failing to do so in any stage of writing will result in an unsuccessful compilation. It’s the user’s responsibility to issue a group delete command after an unsuccessful group creation command, providing the same credentials. This action will deallocate the space for further writings.

To read group data, one can use the “se3_group_read” function. Due to similarities in the implementation with respect to the “se3_group_write” function, the source implementation is not reported here. It’s worth mentioning that the “se3_group_read” function accepts as an input a flash node iterator and the requested result is written back to a response buffer. A response buffer is a way the SEcube™ communication core uses for its communication with a host device. Following each command to a SEcube™ device, a response is received. The communications are established using a predefined payload buffer while the structure of the buffer is agreed on both

the host and the device side. Since this function needs a correct value of flash node iterator, the user should run the “se3_group_find” function prior to the execution of this function. Failing to do so will result in incorrect data response.

Considering the actions related to the cryptographic keys in the implemented key management system, it’s important to notice a hardware drawback of the way the information is stored in the SEcube™ device. A flash storage class of devices are not able to change a memory bit cell after it has been written. Due to performance optimization, it will also not enables a user to erase an individual bit cell and rewriting to it. Due to this hardware behavior, any operation regarding a change in the key metadata of a group’s key list, the APIs have to read all the data related to that specific group and enforce the changes according to the user specifications and writing back the group data into the SEcube™ device.

To change a key state, one can use the “se3_group_change_key_state”. Similar to the “se3_group_read” function, the user needs to call the function “se3_group_find” prior to this function. The following listing shows the key state changing action. While copying the keys information the state transition is affected. The function “check_transition” is used to check the legitimacy. Upon successful evaluation, the desired change will be executed.

```
1. // copy keys
2. if(users_count>0){
3. temp.key_list=(se_key*)malloc(sizeof(se_key)*(keys_count));
4.   for(uint8_t i=0; i<keys_count; i++){
5.       memcpy((void*)((temp.key_list+i)->se_key_id), (void*)(it-
6. >addr+byte_count), 5);
7.       byte_count += 5;
8.   memcpy((void*)((temp.key_list+i)->se_key_label), (void*)(it-
9. >addr+byte_count), 20);
10.      byte_count += 20;
11.      byte_count += 5;
12.   if(strcmp(((temp.key_list+i)->se_key_id), key_id)==0){
13.       uint8_t current_state;
14.       memcpy((void*)&(current_state), (void*)(it-
15. >addr+byte_count), 1);
16.       if(check_transition(current_state, state)){
17.           memcpy((void*)&((temp.key_list+i)-
18. >se_key_state), (void*)&(state), 1);
19.       }
20.       else
21.           return false;}
22.   else{
23.       memcpy((void*)&((temp.key_list+i)-
24. >se_key_state), (void*)(it->addr+byte_count), 1);
25.   }
26.   byte_count += 1;
```

Listing 4.2.5: Key state change, part 1

At the end of each function definition regarding the key modifications, the following pieces of code are repeated. Their presence is due to flash storage writing drawbacks.

```

1. se3_flash_it_delete(it);
2. se3_flash_it_init(it);
3. if (!se3_group_find(temp.group_id, it)) {
4.     it->addr = NULL;}
5. if (!se3_group_new(it, &temp)) {
6.     SE3_TRACE(("[Lld_group_edit] se3_group_new failed\n"));
7.     return SE3_ERR_MEMORY;}
8. free(temp.user_list);
9. free(temp.key_list);
10. return true;}

```

Listing 4.2.6. Key state change, part 2

Similarly, the function “se3_group_change_kay_add” and “se3_group_change_kay_remove” are used for adding or removing a key from a group. Following the modifications made by these functions, the only difference in implementation with respect to “se3_group_change_key_state” is the modification in the number of keys and the key list of the group structure, thus their implementation details are not reported here.

4.2.3 se3_dispatcher_core integration

The key management system uses a SEcube™ based device to implement its functionalities. According to the nature of the implementation and the SEcube™ communication core, every interaction with the device follows the client-server scheme. Due to this, to establish a communication with a SEcube™ device, a user needs to issue a command. A SEcube™ device will check the command upon arrival and act accordingly. The SEcube™ open-source SDK provides a set of APIs that allows a user to define a command for a SEcube™ device and its implementation.

The definitions of commands are done within the “se3_dispatcher_core.h” file. The supported commands should follow a generic function prototype that is defined as follow:

```

1. /** \brief Security function prototype. */
2. typedef uint16_t(*se3_cmd_func)(uint16_t, const uint8_t*, uint16_t*,
    uint8_t*);

```

Listing 4.2.7. SEcube™ security function prototype

Each command definition should be included within the SEcube™ commands lookup table. The lookup table is implemented using a matrix in which each row describes the hardware on which the functionality (hardware/software) is implemented, while each column describes the functionality itself. Since the SEcube™ consists of three main blocks, the following is the organization of the lookup table.

```
1. se3_cmd_func handlers[SE3_N_HARDWARE][SE3_CMD1_MAX]
```

Listing 4.2.8: SEcube™ commands lookup table

- Row 0: Security core
- Row 1: FPGA
- Row 2: Smartcard

It's the user's responsibility to include the functionality in its corresponding hardware block location. Considering the key management system, every operation regarding the key management system is handled using a simple "group_edit" command to a SEcube™ device. Since the key management system uses the security core of the SEcube™ device, this command is included in row 0 of the lookup table matrix. Any communication from the host side regarding the key management system will be handled using this function. By issuing the group edit command to a SEcube™ device, the function "dispatcher_call" will execute the "group_edit" function accordingly. Following is the implementation of the "group_edit" function.

```
1. uint16_t group_edit(uint16_t req_size, const uint8_t* req, uint16_t*
   resp_size, uint8_t* resp)
2. {
3.
4.     uint16_t operation, temp_offset=0;
5.     // operation is used to decide the type of operation to be
   executed.
6.     memcpy((void*)&(operation), (void*)req, sizeof(uint16_t));
7.     // copy the first 2-byte of request data which holds the operation
   value.
8.
9.     se_group device_side;
10.    // se_group structure to hold the request values.
11.
12.    // bool equal;
13.    se3_flash_it it = { .addr = NULL};
14.
15.    if (!login_struct.y) {
16.        SE3_TRACE("[Lld_group_edit] not logged in\n");
17.        return SE3_ERR_ACCESS;
18.    }
19.
20.    memcpy((void*)(device_side.group_id), (void*)(req+2), 5);
21.    // copy group_id from buffer(request), 5 characters including
   '\0'.
22.    memcpy((void*)(device_side.group_name), (void*)(req+7), 20);
23.    // copy group_name from buffer(request), 20 characters including
   '\0'.
```

Listing 4.2.9: function "group_edit" part 1

As mentioned before, communication with the SEcube™ device is done by using a communication buffer. The commands to a SEcube™ and the data related to it are provided using the request

buffer. Upon receiving a command regarding the key management system a temporary group structure is created and its data are placed within the structure accordingly. It's worth mentioning that a request buffer could have empty group data. Any empty data will result in having a group structure filled with random data. Since it will be discarded later, the empty data will cause no undesired effect within the key management system. The operation type is fetched from the request buffer upon arrival and copied to the "operation" variable. Prior to any execution, a user login status is checked as it's shown in the listing [4.2.9](#).

The value of the cryptographic keys are randomly generated. Upon a key generation, this value is generated and stored alongside key metadata. Following is the listing demonstrating that.

```
1. if((device_side.num_keys)>0){
2. // this check is to prevent to do malloc(0), which is implementation
   defined.
3. device_side.key_list=(se_key*)malloc(sizeof(se_key)*(device_side.num_ke
   ys));
4. // allocate space for saving the keys.
5. time_t t;
6. srand((unsigned) time(&t));
7. for(int i=0;i<device_side.num_keys;i++){
8. uint8_t random_key_value[4];
9.     for(int i=0; i<4; i++){
10.         random_key_value[i] = (uint8_t)rand();
11.     }
12. // copy key data fields for each key to the group key_list.
13. memcpy((void*)((device_side.key_list+i)->se_key_id),
   (void*)(req+35+(sizeof(se_user)*(device_side.num_users))+(i*sizeof(se_k
   ey))), 5);
14. // . . . missing lines
15. memcpy((void*)&((device_side.key_list+i)->key_value[0]),
   (void*)(random_key_value), 32);
16.     // copy key_value from buffer(request).
```

Listing 4.2.10: function "group_edit" part 2

To check for the existence of a group within the SEcube™, prior to any action a flash node iterator is created and the flash memory is checked.

```
1. se3_flash_it_init(&it);
2. if (!se3_group_find(device_side.group_id, &it)) {
3.     it.addr = NULL;
4. }
```

Listing 4.2.11: function "group_edit" part 3

According to the fetched command from the request buffer, the following operations are executed.

- SE3_GROUP_OP_INSERT
This will result in the creation of a new group within the SEcube™ device. If a group exists with the same “group_id” within the device, it fails and responds to the user using the response buffer an error message, “OVERWRITING EXISTING DATA”.
- SE3_GROUP_OP_DELETE
This will result in the elimination of a group from the SEcube™ device.
- SE3_GROUP_OP_READ
Upon successful execution, the group data will be provided on the response buffer. Failing to do so the error message, “DATA NOT FOUND” will be provided.
- SE3_GROUP_OP_DELETE_ALL
This will result in the elimination of all the groups’ data within the SEcube™ device.
- SE3_GROUP_OP_READ_ALL
The response buffer will be filled with all the groups’ data.
- SE3_GROUP_OP_KEY_CHANGE_STATUS
This will result in a state change of a provided key ID. Upon successful state change, the message “KEY STATE UPDATED” will be provided in the response buffer.
- SE3_GROUP_OP_KEY_REMOVE
This will result in the elimination of a key from a group key list.
- SE3_GROUP_OP_KEY_ADD
This will result in the addition of a new cryptographic key to a group key list.

4.3 Host-side Application

As stated in section [3.2](#), the SEcube™ software architecture defines a set of APIs that are used to communicate with a SEcube™ device. These APIs are divided into different levels of abstractions that hide the low-level implementation details from the application user standpoint. In the host-side application, we exploited these APIs for communication and authentication purposes. Following is a short reminder of the mentioned APIs.

- Level 0 APIs are designed to establish communication with a SEcube™ device.
- Level 1 APIs are designed to provide higher levels of functionalities, such as Login/logout, encryption/decryption, and so on.

The SEKey™, host side, APIs are developed leveraging on Level 0 and Level 1 functionalities and are placed at Level 2 in the SEcube™ software stack. They provide the set of functionalities required by the KMS and to perform all the actions described in Listing 4.2.1. Developers can easily integrate these APIs in application to benefit from the SEkey™ hardware based key management system.

The developed key management system comes with a graphical interface, that is a perfect example of an application that take advantage of the SEkey™ APIs. The following will be dedicated to explaining the graphical user interface application of the key management system.

4.3.1 Agents overview

The graphical user interface is implemented using abstraction for each action of interest. Each abstraction is called agents. The tasks that each agent performs are partial and limited, thus allowing them to be fast and less error-prone. This approach gives the possibility to a developer to modify or expand the codebase in the future much easier and faster. To implement the application functionalities the following agents are defined:

- Agent_add_group
- Agent_get_group_id
- Agent_set_group_keys
- Agent_set_group_policy
- Agent_set_group_users
- Group_window_agent

In figure 4.3.1 is presented is an overview of the application and the transitions between each state.

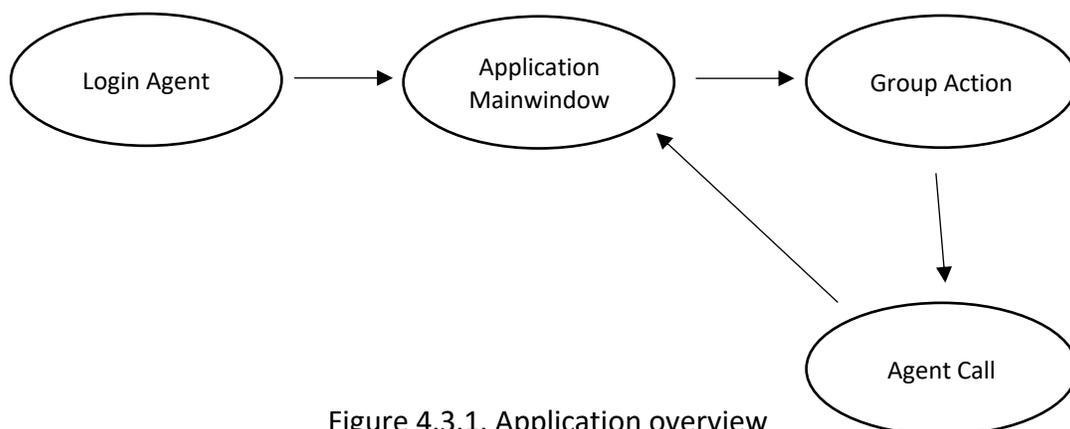


Figure 4.3.1. Application overview

As an entry point, the login agent always performs user authentication. Upon successful evaluation, the user will be directed to the application main window. A user can perform the actions defined to its privilege level. By performing an action, the corresponding agent is called.

Upon finishing the user's requested action, the user will be redirected to the application's main window.

4.3.2 Authentication

When a user runs the GUI application, the first window that appears to him/her is the "Login" window as shown in figure 4.3.2. It is possible to choose among two types of logins, by choosing the admin mode, a user has access to all the functionalities defined within the key management system while the user mode would only have read access to the SEcube™ device. By providing the password and clicking the "Sign in" button, an authentication process between the host device and the user's SEcube™ device will be initiated. Upon password mismatch or device detection failure, a user will be prompted for the authentication failure and redirected to the login page again. Upon successful authentication, the user will have access to the application main window.

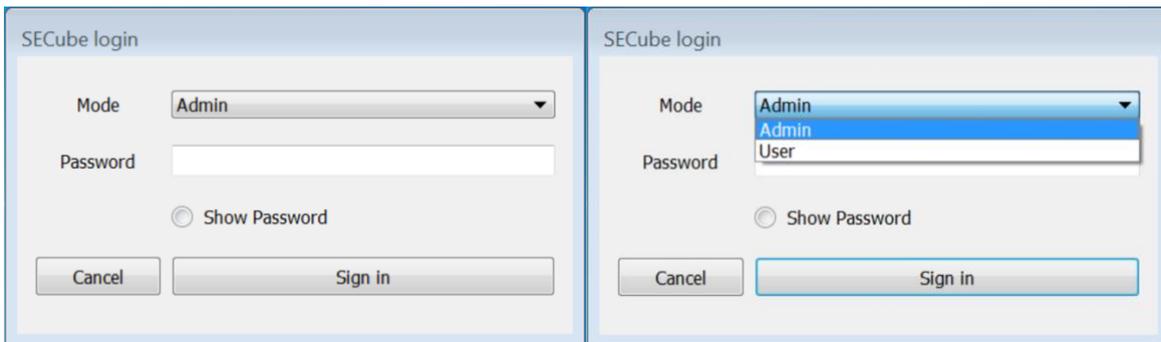


Figure 4.3.2. Login window

Any authentication process starts with discovering a SEcube™ device connected to the host machine. The host-side Level 0 APIs provide these functionalities, thus they are used within the login window agent for its implementation. It's worth mentioning that the admin and user passwords setup is done in the device initialization phase. Since the password is stored with the SEcube™ devices, one cannot use the GUI application to change the users' credentials.

4.3.3 Main window

The key management system GUI application's main window is designed to provide all the functionalities available within a simple interface as shown in figure 4.3.3. As a user connects its SEcube™ device to the computer and successfully logs in, the application main window is presented, it's the medium through which the user can interact with the SEcube™ and access the functionalities of the KMS. Following is the list of commands available to a user at the admin level.

- ADD
- DELETE
- DELETE ALL
- READ

A user who has no admin privileges can only issue a read command to get access to the information it needs. A user can use this command, providing a group ID, to retrieve a list of users that are located within the group or see a list of cryptographical keys and their status that are available to that specific group.

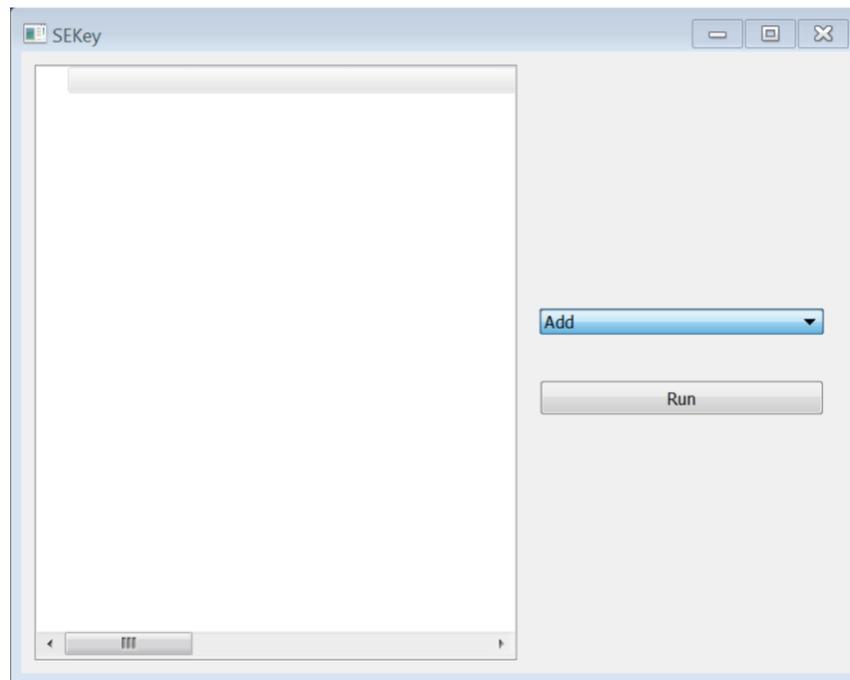


Figure 4.3.3. Main window

The main window is organized into two main parts. Their purposes are as follows.

- **Action panel**
This panel, which is located on the right side of the main window is used to provide the actions a user can do within the key management system. By choosing the type of action from the options list available to a user and clicking the “Run” button, the corresponding operation will be executed and the result will be shown on the information panel.
- **Information panel**
When a user interacts with a SEcube™ device asking for information about a group, the group’s data will be presented in this section. Following any unsuccessful information retrieve from a SEcube™ device, an “Error” message will be shown as a result instead.

4.3.4 Add Group window

In order to add a new group to the key management system, one can choose the “ADD” option from the options list and hit the “Run” button. By issuing this command user will be prompted by the “Add Group” window. The following figure depicts such a window.

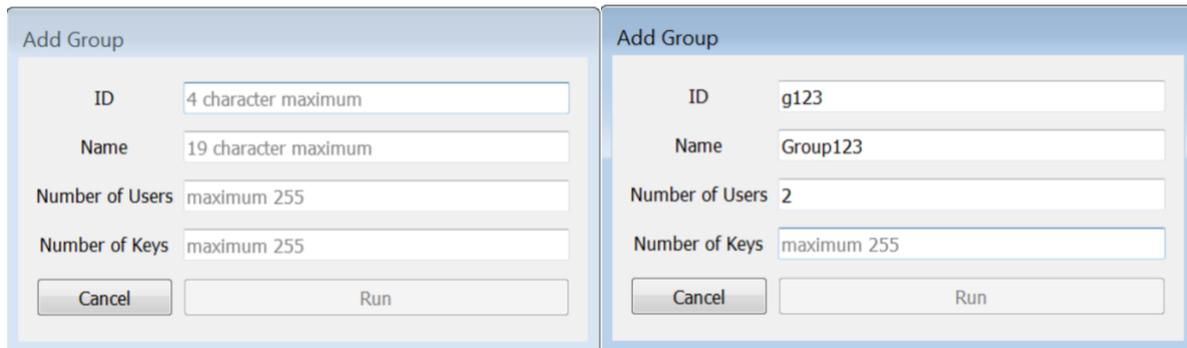


Figure 4.3.4. Add Group window

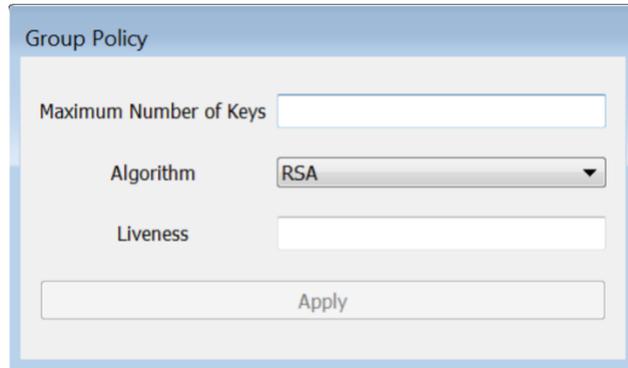
According to the definitions of the data structures used for representing the elements of the key management system, each of the agents that are retrieving the information, have the following elements in their panels.

- **Data fields**
The user will be asked to provide the information necessary to build a group within the key management system.
- **Filters**
Each data field is associated with a filter that checks the user input and compares it with a valid set of ranges. This protection mechanism will guarantee data integrity upon group creations and lowers the risk of wrong data input that may cause wrong data storage.

Due to filters assigned to data input fields, the administrator needs to follow specific patterns foreseen for each field. For example, an administrator cannot input a character value in the fields “Number of Users” and “Number of Keys”. By doing so, the filter will not register the user input. It’s also necessary to provide all the data and not leave any field empty. Failing to do so will disable the “Run” button and the user will not be allowed to register the group’s data within the key management system. By providing all the data necessary for group creation, the user will be prompted by a “Group Policy” window.

4.3.5 Group policy window

The group policy window is the interface that the agent handling it will provide to its user to retrieve the information regarding the policy of a group. Figure [4.3.5](#) depicts such a window.



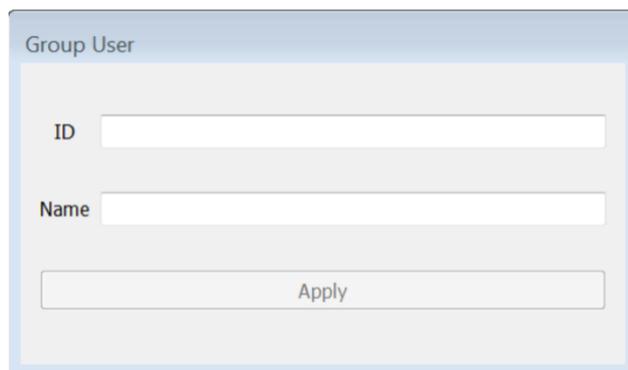
The screenshot shows a window titled "Group Policy". Inside the window, there are three input fields: "Maximum Number of Keys" (a text box), "Algorithm" (a dropdown menu with "RSA" selected), and "Liveness" (a text box). Below these fields is an "Apply" button.

Figure 4.3.5. Group Policy window

It's worth mentioning that all the data represented here will affect the overall behavior of the newly created group. For example, The "Algorithm" chosen by a user will be a default type and the agent that handles the cryptographic keys will act accordingly and sets this value to a key by default.

4.3.6 Group user window

The group user window is the interface that the agent handling it will ask for the required information. According to the data provided on the number of users, the agent will retrieve that exact number of data.



The screenshot shows a window titled "Group User". Inside the window, there are two input fields: "ID" and "Name". Below these fields is an "Apply" button.

Figure 4.3.6. Group User window

4.3.7 Group Key window

Following the same logic that is applied for the users, the group key agent will require the administrator to input the key information belonging to a group. The interface for the group's key is depicted in the Figure [4.3.7](#).

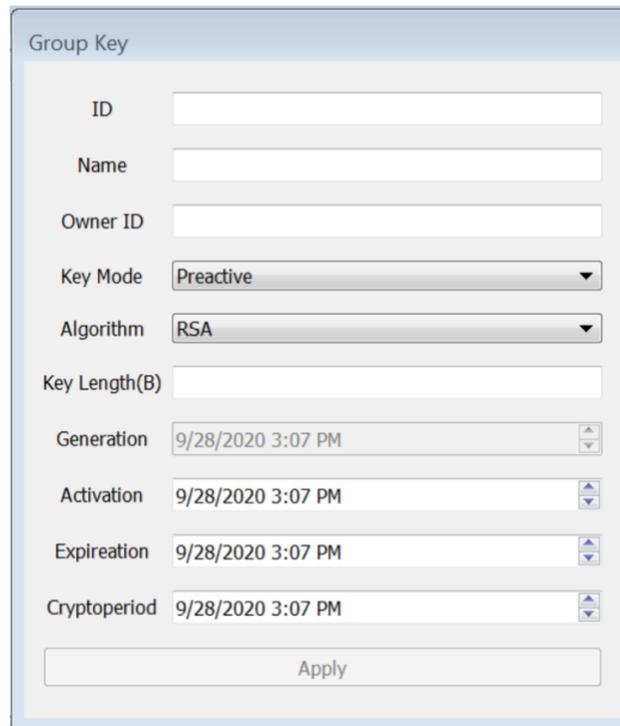


Figure 4.3.7. Group Key window

To provide data integrity and eliminate the risk of incorrect data storage within a SEcube™ device, users' inputs are checked against filters. The following are the important filters that are implemented within the agent.

- **Algorithm**
According to the group Policy, the default encryption algorithm used by the cryptographic keys within a group is set at the beginning of the group creation. The algorithm tag represented in figure [4.3.7](#), will always show the default option for the keys. A user can change the default algorithm by choosing a proper option or it can go without changing this value. By doing so the default value will be set for each key and the user doesn't need to set this value for each cryptographic key.
- **Generation time**
This value of this tag is not editable. It's represented to assist the user to set a proper time-related data of the cryptographic keys. This value is set internally according to the system time.

- **Time tags**

The “Activation”, “Expiration” and “Cryptoperiod” time tags cannot hold a value lower than the “Generation” time. Doing so, will not register user input.

4.3.8 Read Group & Delete Group windows

Due to similarities, reading, and deleting group agents have similar interfaces. In the Figure [4.3.8](#) are shown the two windows, they both ask the user or the administrator to enter the unique group identifier.

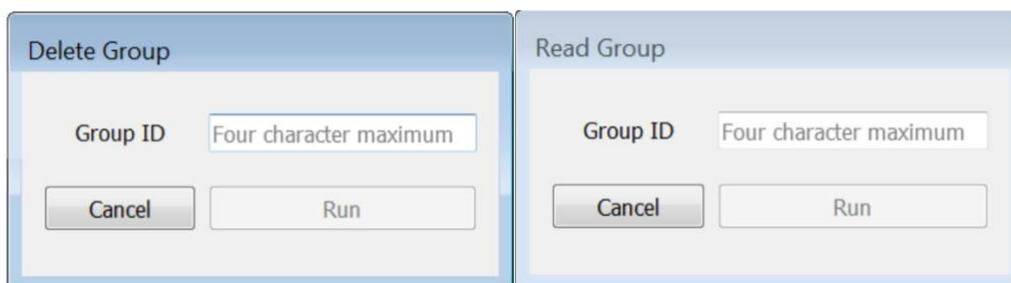


Figure 4.3.8. Delete group, Read group windows

Chapter 5

Result and Discussion

In this chapter, we will examine the result obtained by the thesis work. Together with features explanations, we will discuss the drawbacks of the implementation. In the end, some use cases will be examined to give the reader an understanding of how the system works.

5.1 Results

The hardware-based key management system developed by this thesis work is a proof of concept that demonstrates the idea of implementing a secure environment in which the users are able to communicate and exchange data securely leveraging on an automated system for the cryptographic key management. The key management system exploits the functionalities provided by the SEcube™ hardware security module to manage and protect cryptographic keys.

There are two main categories of users defined within the key management system, the administrator of the system and the users. The system is designed in such a way that all the management and organization are done by an administrator, he/she has all the privileges and access regarding the data within the key management system and it's under his/her responsibility to manage the system and distribute all the data necessary to the users of the system. Normal users without admin privileges are exposed to a subset of functionalities and cannot, by any means, modify anything in the key management system. This approach follows the "Least Privilege" principle that protects the system from unauthorized access to data.

The discrimination between an admin-level user and a normal user of the system is done by enabling a different set of functionalities and privileges upon login into the system. Only using the SEcube™ admin access pin reveals the whole set of functionalities defined with the key management system. The data regarding the elements of the key management system, like groups and their metadata, are stored within the flash of the SEcube™ device. This is valid for both admin's and users' devices with a difference in the amount of information available to them.

The developed graphical user interface application for the key management system is built on top of the features provided by the set of host-side APIs that are developed for the key management system. The GUI application is a complete and fully functional stand-alone application that can be used to access the key management system features. The developed software and device firmware have the following characteristics and functionalities.

- Set of APIs allowing a security administrator to manage the groups, keys, users within the KMS, and impose policies over them.
- All the cryptographic primitives are provided and executed within the SEcube™ itself, Providing strong isolation for sensitive data from the main system.
- Low-level security actions and communication are abstracted and handled automatically.
- The data regarding the KMS are stored locally in the flash of the SEcube™ device protecting them from possible compromises.
- The cryptographic key values are not available to any type of user of the KMS.
- The cryptographic key value is generated internally in a random fashion without any interaction from outside.
- The discrimination of the roles of admin and user is established following the definition by the NIST on the “Least Privilege” principle.
- Usability of the KMS by the users in offline mode, exploiting their local copy of data.
- Compatibility with other SEcube™ framework libraries such as SEfile™.
- Integrity of the data upon compromise, since the data available to the users, are limited and the management is done centrally by an admin.
- The GUI application convenient to establish a secure environment without exploiting the low-level APIs.
- The open-source nature of the project that allows third parties to exploit the codebase and expand the functionalities.

5.2 Drawbacks

In this section, we will go over, explaining the limitations and drawbacks of the developed key management system considering both the APIs and the GUI application.

- **SEcube™ SDK flash storage APIs limitation**
As it's been shown in the listing 4.3.3, the key management system exploits the already existing SEcube™ opensource SDK to read/write the data to a device's internal flash memory. In the implemented APIs there are two sectors reserved for the users' data. It's also worth mentioning that among those sectors only one sector can be active at a time for usage. This is a major limitation in the amount of storage space available to store the

data regarding the key management system. Since the cryptographic keys within the key management are heavily exploited and changed rapidly to keep the environment and communications secure, this will cause the flash storage to wear out. Due to the flash storage nature, it's not possible to change the specific bits within storage cells. In addition to that, the flash storage is implemented in such a way that it only allows the user to reset the storage cells in large portions. These limitations are imposed to improve the performance of flash devices. As discussed in section 4.3 any action regarding a modification within the key management system will result in moving the whole group's data, changing the data of interest, and writing back them to a new location within the flash memory. This approach will have a huge impact on the read/write cycle of the sectors used for the key management system.

- **Operation cost**

All the users within the key management system are required to poses a SEcube™ device to allow them to secure their communications and data exchanges. This will impose a potential cost of operation in larger environments.

- **Scalability limitation**

By exploiting all the available space within the SEcube™ flash storage, there is still a maximum limit on the number of elements that can be stored within the SEcube™ device.

- **Encryption algorithms**

The SEcube™ open-source SDK provides to its user the symmetric encryption/decryption algorithm. To be more specific the APIs only support symmetric cryptographic keys and not having support for asymmetric cryptography can be a limitation in some scenarios.

- **Single point of failure**

Since the users have access to a subset of information within the key management system, it's the admin's SEcube™ device that holds all the information regarding the groups and their metadata. This will imply a possible threat to the key management system in case of data loss due to possible hardware failure or malicious intervene.

- **OS dependability**

The current version of the Level 0 APIs is exploiting the OS-specific APIs for communication purposes. As there is only support for Windows and Linux operating systems, one cannot use the current version of APIs under other operating systems such as macOS. Although the QT framework is multiplatform, the usage of the level 0 APIs within the application implementation will limit the availability of the application under the other platforms.

- **GUI application**

Since this thesis developed as a proof of concept to demonstrate a possible implementation using a hardware-based key management system, the elements used

within the graphical interface aren't elegant and lack the common approaches used in modern GUI designs.

It's worth mentioning that some of the drawbacks and limitations described above are due to the design choices and do not constitute real problems. Limitations such as scalability are due to choosing to implement a hardware-based key management system. Although it may limit the scalability, it also provides higher security following the recommendations suggested by the NIST.

5.3 KMS sample uses cases

In this section, we will go over some uses cases that will give useful insight into how a user can use the APIs developed for the key management system and include them in their own design. Alongside the APIs' use cases, the application outcome is also depicted.

5.3.1 SEcube™ initialization

All the initialization processes are done specifically by the system's admin.
Precondition: The SEcube™ device is not initialized before.

- The SEcube™ should be connected to a host machine.
- Execute the LO_discover_init() function to discover the connected SEcube™ device.
- Execute the login command to the SEcube™ using the default admin PIN.
- call the sekey_admin_init() function, providing the desired admin and user PIN.
- Disconnect the SEcube™ from the host machine.

At this point, the admin can reconnect the SEcube™ device to the host machine and run the login command again providing the newly set credentials to check that the setup process was effective.

5.3.2 Add a group to the KMS

This action can only be performed by an admin.
Precondition: The SEcube™ device is initialized before.

- The SEcube™ should be connected to a host machine.
- Execute the LO_discover_init() function to discover the connected SEcube™ device.

- Execute the login command to the SEcube™ using the admin PIN.
- Create a temporary data structure representing a group data.
- Fill out the information regarding the policy of the group.
- Fill out the information regarding the users within the group.
- Fill out the information regarding the keys within the group.
- Execute the L1_group_edit() function providing the command SE3_GROUP_OP_INSERT.
- Check the return message from the SEcube™ response.
- Disconnect the device.

5.3.3 Delete a group from the KMS

This action can only be performed by an admin.

Precondition: The SEcube™ device is initialized before.

- The SEcube™ should be connected to a host machine.
- Execute the L0_discover_init() function to discover the connected SEcube™ device.
- Execute the login command to the SEcube™ using the admin PIN.
- Create a temporary data structure representing a group data, the important data that is necessary is the group_id, the other fields of the temporary data structure could be left unfilled.
- Execute the L1_group_edit() function providing the command SE3_GROUP_OP_DELETE.
- Check the return message from the SEcube™ response.
- Disconnect the device.

5.3.4 Modify a key state within a group

This action can only be performed by an admin.

Precondition: The SEcube™ device is initialized before.

- The SEcube™ should be connected to a host machine.
- Execute the LO_discover_init() function to discover the connected SEcube™ device.
- Execute the login command to the SEcube™ using the admin PIN.
- Create a temporary data structure representing a group data.
- Create a temporary data structure representing a group data, the important data that is necessary is the group_id, the other fields of the temporary data structure could be left unfilled.
- Execute the L1_group_edit() function providing the command SE3_GROUP_OP_READ
- Fill out the information regarding the policy of the group from the response buffer.
- Fill out the information regarding the users within the group from the response buffer.
- Fill out the information regarding the keys within the group from the response buffer.
- Modify the value regarding the key state.
- Execute the L1_group_edit() function providing the command SE3_GROUP_OP_KEY_CHANGE_STATUS.
- Check the return message from the SEcube™ response.
- Disconnect the device.

5.3.5 Retrieve a group from the KMS

This action can only be performed by an admin.

Precondition: The SEcube™ device is initialized before.

- The SEcube™ should be connected to a host machine.
- Execute the LO_discover_init() function to discover the connected SEcube™ device.
- Execute the login command to the SEcube™ using the admin PIN.

- Create a temporary data structure representing a group data, the important data that is necessary is the group_id, the other fields of the temporary data structure could be left unfilled.
- Execute the L1_group_edit() function providing the command SE3_GROUP_OP_READ.
- Check the return message from the SEcube™ response.
- Disconnect the device.

The sample use cases described in section [5.3](#) are useful to understand the steps necessary to be performed when a user wants to interact with the key management system using the APIs provided by the framework. One can choose to interact with the key management system using the GUI application provided alongside the framework. The following will be dedicated to showing a sample interaction with the GUI interface of the key management system. Among all the actions done by the administrator to manage the system, the creation of a new group is the most complete one, thus it's used demonstrated here as a sample interaction with the GUI application.

5.4 GUI application use case

- Use case: Add a new group to KMS
- Precondition: The SEcube™ device is already initialized and connected to the host machine.

The admin will be prompted with the login window as depicted in the figure [5.4.1](#). Upon providing the login password the admin will be directed to the main window of the application.

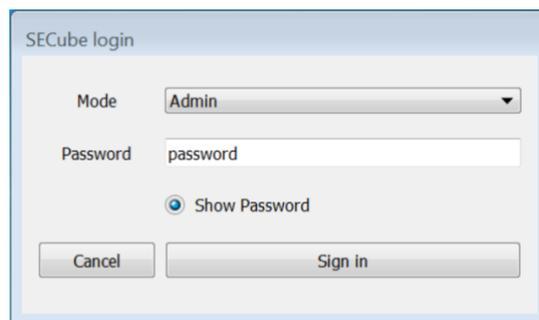
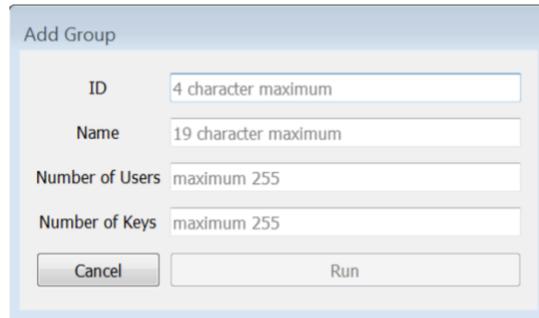


Figure 5.4.1. Login window

By choosing the “ADD” option from the option list the admin will be prompted by the add group window. Figure [5.4.2](#), depicts such a window.

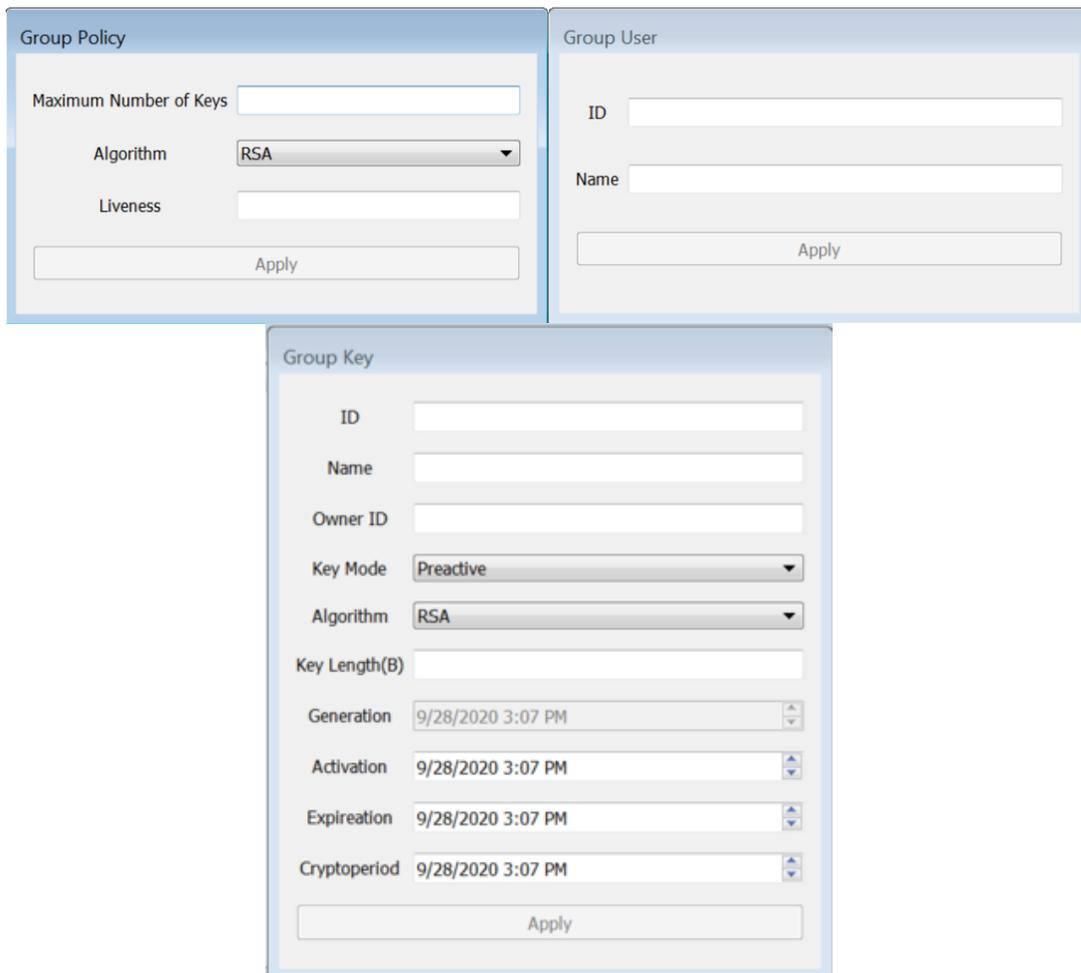


The 'Add Group' dialog box contains the following fields and buttons:

- ID: 4 character maximum
- Name: 19 character maximum
- Number of Users: maximum 255
- Number of Keys: maximum 255
- Buttons: Cancel, Run

Figure 5.4.2. Add group window

Providing the required information the admin needs to fill out all the data regarding the Policy, users, and keys. These steps are done in sequence and cannot be skipped.



The figure shows three sequential dialog boxes for group creation:

- Group Policy:**
 - Maximum Number of Keys: [text input]
 - Algorithm: RSA (dropdown)
 - Liveness: [text input]
 - Apply button
- Group User:**
 - ID: [text input]
 - Name: [text input]
 - Apply button
- Group Key:**
 - ID: [text input]
 - Name: [text input]
 - Owner ID: [text input]
 - Key Mode: Preactive (dropdown)
 - Algorithm: RSA (dropdown)
 - Key Length(B): [text input]
 - Generation: 9/28/2020 3:07 PM (spinner)
 - Activation: 9/28/2020 3:07 PM (spinner)
 - Expiration: 9/28/2020 3:07 PM (spinner)
 - Cryptoperiod: 9/28/2020 3:07 PM (spinner)
 - Apply button

Figure 5.4.3. Group creation steps

By providing the last key information, the admin can hit the “Apply” button and the operation will be completed. The admin can check the submission of the group within the key management system by issuing a read command. Upon providing a valid group_id, the information will be presented on the information panel of the main window. The following figure depicts that.

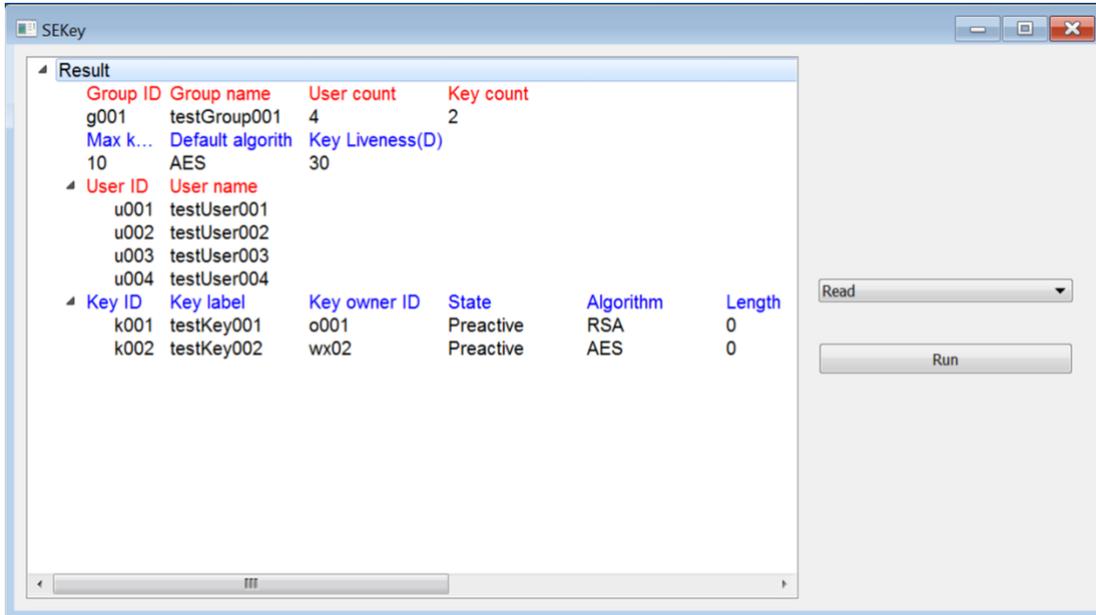


Figure 5.4.4. information presentation in the main window.

If the provide group credentials are not valid, an error message will be shown in this panel.

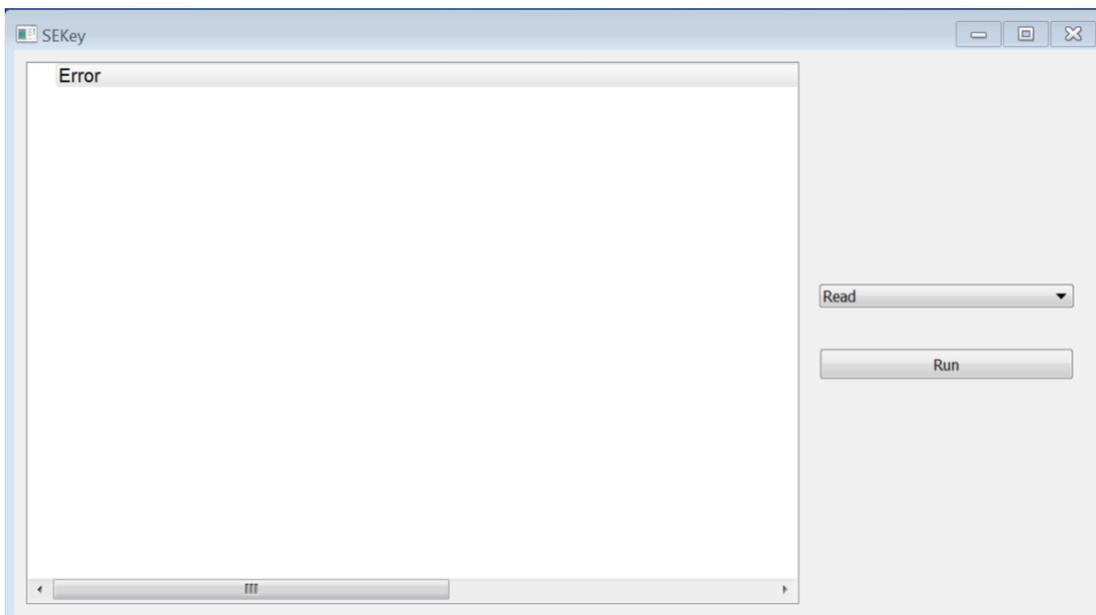


Figure 5.4.5. Invalid group credentials read, message.

Chapter 6

Conclusion

This thesis aimed at designing and developing a hardware-based key management system leveraging on the functionalities and unique characteristics of the SEcube™ hardware security module. SEkey™ was developed on top of the already available software stack, adding new functionalities and enhancing old ones, and providing to developers a new set of APIs for the management and interaction with the underlying key management system. Moreover, in this thesis, a graphical user interface has been developed, providing an easy way of interaction with the KMS and a good example of how the SEkey™ API can be integrated into different applications

The unfeasibility of a manual approach to managing the cryptographic keys is the reason that key management systems are born in the first place. The well-known growing need for higher security in preserving and managing the cryptographic keys motivated the implementation of the hardware-based key management system. The SEcube™ framework offers both the hardware and software elements necessary to establish a robust and reliable environment to manage such needs. The toolchain offers reliable data encryption and the available software libraries simplify the work of non-expert developers in cryptography to design and build security-related applications.

The integration of the developed key management system with the SEcube™ framework resulted in a reliable and easy to manage key management system that can be exploited to secure communications and data. The privileged accesses defined within the key management system are another aspect of the design. Considering the administrator and user roles, the privileges are defined in such a way that the admin of the system has full control over the data and functionalities defined for the system while the user privilege limits its access to the system's data and functionalities. This approach follows the NIST, "Least Privilege" principle paradigm.

Following the objectives defined for the SEkey™ framework, the implemented key management system is capable of providing the functionalities required to manage the groups, users, and cryptographic keys. The administrator of the system can exploit these functionalities to create groups and assign users to them. By defining cryptographic keys for each group and assigning policies to regulate them, the admin can build a secure environment for communication and data exchange. The users of groups can exploit these cryptographic keys and protect their data.

The graphical user interface application that comes alongside the key management system has designed to be a functional application that abstracts the user from the underlying mechanism of cryptography and key management. It's worth mentioning that the software application is designed to be a proof of concept and still lacks some features that are required in real-world work environments.

6.1 Future work

The following are the few ideas that can be investigated to further expand the feature set of solutions based on the SEcube™ devices.

- **Database integration**
The developed key management system uses the SEcube™ device internal flash storage for its storage purposes. Following the separation of the metadata of the groups and the cryptographic keys, it would be hugely beneficial to move the data regarding the users and the groups out of the internal storage of the device.
- **SElink™ and SEfile™ integration**
The integration of the SEkey™ with the other solutions developed for the SEcube™ open security platform, namely SEfile™ and SElink™ (see [chapter 3](#)), will be a key objective for the future development, providing a fully secure environment for SEcube™ users.
- **SEcube™ firmware expansion**
The current version of the SEcube™ firmware imposes a few limitations on the potential uses cases regarding the SEcube™ based devices. For example, the firmware only supports the AES encryption algorithm with a fixed key size. It can be further expanded to support more algorithms schemes such as asymmetric encryption. It can also be investigated to include the other approaches in cryptography that don't exploit the cryptographic keys for securing the environment, the zero-knowledge based proof are the examples of such solutions. It's worth mentioning that the resources regarding the performance and storage of the SEcube™ are quite limited, any improvement needs to evaluate the tradeoffs of such solutions.
- **Operating system integration**
Possible integration of the SEcube™ firmware features together with an embedded operating system (RTOS) can provide a secure environment for the development of IoT applications.
- **Smartcard integration**
The SEcube™ comes with a smartcard built into its package. The features that a smartcard provides regarding the encryption/decryption can be exploited and used within the firmware for encryption purposes. Since asymmetric cryptography is considerably resource consuming, the smart card is an appropriate target for its implementation with the features provided by the SEcube™.

Bibliography

- [1] The Role of the Key in Cryptography & Cryptosystems. URL: <https://study.com/academy/lesson/the-role-of-the-key-in-cryptography-cryptosystems.html>
- [2] Selecting The Right Key Management System. URL: https://www.cryptomathic.com/hubfs/Documents/White_Papers/Cryptomathic_White_Paper_-_Selecting_The_Right_Key_Management_System.pdf
- [3] National Institute of Standards and Technology. Recommendation for Key Management: Part 1 – General. URL: <https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-5/archive/2019-10-08>
- [4] M. Nieves, K. Dempsey, and V. Y. Pillitteri. An introduction to information security. National Institute of Standards and Technology, June 2017. doi: 10.6028/nist.sp.800-12r1. URL: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-12r1.pdf>
- [5] Townsend Security. The Definitive Guide To Encryption Key Management Fundamentals. 2016. URL: <https://info.townsendsecurity.com/definitive-guide-to-encryption-key-management-fundamentals>
- [6] Security requirements for cryptographic modules. National Institute of Standards and Technology, May 2001. URL: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-2.pdf>
- [7] Recommendation for Key Management – Part 2: Best Practices for Key Management Organization. National Institute of Standards and Technology, Aug. 2005. URL: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-57p2.pdf>
- [8] E. Barker and W. Barker. Recommendation for Key Establishment Using Symmetric Block Ciphers. National Institute of Standards and Technology, July 2018. URL: <https://csrc.nist.gov/CSRC/media/Publications/sp/800-71/draft/documents/sp800-71-draft.pdf>
- [9] Varriale Antonio, Prinetto Paolo, Maunero Nicolò, and Roascio Gianluca. The SE- cube™ Open Security Platform. URL: https://www.secube.eu/site/assets/files/1152/secube_sdk_v1_4_1_wiki_rel_009.pdf
- [10] Gallego Gomez Walter. A Secure Password Wallet based on the SEcube™ framework. URL: <https://webthesis.biblio.polito.it/8201/1/tesi.pdf>
- [11] Matteo fornero. Development of a secure key management system for the SEcube Security Platform. URL: <https://webthesis.biblio.polito.it/14521/1/tesi.pdf>

Appendix A, KMS data structures

```
1.
2. #define ID_SIZE_LIMIT 5
3. #define LABEL_SIZE_LIMIT 20
4. #define NAME_SIZE_LIMIT 30
5. #define MAX_KEY_SIZE_IN_BYTES 32 // support up-to 256-bit length, key.
6. #define SE3_TYPE_GROUP 111
7.
8. enum {preactive=0, active=1, suspended=2, deactivated=3, compromised=4,
   destroyed=5};
9. enum {rsa=0, aes=1};
10.
11. typedef struct se_key{
12.     char se_key_id[ID_SIZE_LIMIT];
13.     char se_key_label[LABEL_SIZE_LIMIT];
14.     char se_key_owner_id[ID_SIZE_LIMIT];
15.     uint8_t se_key_state;
16.     uint8_t se_key_algorithm;
17.     uint16_t se_key_length;
18.     uint8_t key_value[32];
19.     time_t se_key_generation;
20.     time_t se_key_activation;
21.     time_t se_key_expiration;
22.     time_t se_key_cryptoperiod;
23. } __attribute__((packed)) se_key;
24.
25. typedef struct se_user{
26.     char se_id[ID_SIZE_LIMIT];
27.     char se_username[NAME_SIZE_LIMIT];
28. } __attribute__((packed)) se_user;
29.
30. typedef struct {
31.     uint8_t max_num_keys;
32.     uint8_t algorithm;
33.     uint32_t liveness_key;
34. } __attribute__((packed)) group_policy;
35.
36. typedef struct {
37.     char group_id[ID_SIZE_LIMIT];
38.     char group_name[LABEL_SIZE_LIMIT];
39.     uint8_t num_users;
40.     uint8_t num_keys;
41.     group_policy policy;
42.     se_user* user_list;
43.     se_key* key_list;
44. } __attribute__((packed)) se_group;
45.
```

Appendix B, Group data writing

```
1. bool se3_group_write(se3_flash_it* it, se_group* group)
2. {
3.     bool success = true;
4.     uint16_t data_offset = 0;
5.
6.     if (!se3_flash_it_write(it, data_offset, (uint8_t*)(group->group_id), 5)) {
7.         success = false;
8.     }
9.     // copy group_id.
10.    data_offset +=5;
11.    // update offset for next data field.
12.
13.    if (!se3_flash_it_write(it, data_offset, (uint8_t*)(group->group_name), 20)) {
14.        success = false;
15.    }
16.    // copy group_name.
17.
18.    // missing lines
19.    // ...
20.    // ...
21.    // ...
22.    // ...
23.
24.    if (!se3_flash_it_write(it, data_offset, (uint8_t*)&((group->key_list+i)-
>se_key_cryptoperiod), 4)) {
25.        success = false;
26.    }
27.        // copy se_key_cryptoperiod for each key.
28.        data_offset +=4;
29.        // update offset for next data field.
30.    }
31.
32.    if (!success) {
33.        SE3_TRACE(("[se3_group_write] cannot write to flash block\n"));
34.    }
35.
36.    return success;
37. }
```