POLITECNICO DI TORINO

Master's Degree Course in Computer Engineering

Master's Degree Thesis

Development of a secure key management system for the SEcube Security Platform

Supervisor
prof. Paolo Ernesto Prinetto

Candidate
Matteo Fornero

Academic Year 2019/2020

Abstract

The massive digitization of services and the widespread use of electronic devices pose important problems in terms of privacy and data security, leading to significant investments by governments, companies, and organizations in cybersecurity solutions.
Cybersecurity is a complex topic covering many issues; one of the most addressed of these is cryptography. The purpose of cryptography is to transform data into a form that is not intelligible by anyone except those who know a special value called “key”, whose role is like that of the key of a padlock which closes a box. The process of closing the box with the padlock is called encryption, while the reverse action is called decryption; the internal mechanism of the padlock is called algorithm. Just as in the case of a padlock, also in cryptography protecting the key is very important.
This issue is known as key management, any flaw in handling the keys could make the entire encryption process useless. In most environments, the quantity of keys that must be handled is so significant that manual management is unfeasible and not secure. In order to address this issue, special solutions called Key Management Systems (KMS) are constantly being developed.
This thesis aims at developing a simple KMS, called SEkey, based on the SEcube security platform. The SEcube is a hardware security module (HSM) used to execute security primitives such as encryption and decryption. SEkey is particularly focused on distributed environments, where every actor resorts to a SEcube device as his security-oriented coprocessor. SEkey targets the management of the life cycle of encryption keys, the administration of users and of their groups and the supervision of multiple security policies.
Leveraging open-source libraries such as SQLite, and the pre-existing SEcube software ecosystem, consisting of the device firmware and a set of computer-side APIs, the KMS was developed to be as compliant as possible with the best security standards.
SEkey resorts to algorithms such as AES-256 and SHA-256 to guarantee confidentiality, integrity and authentication. The encryption keys are physically protected inside each SEcube device, whose access is limited according to the policies established by the administrator. The KMS is compatible with Windows and Linux, and it offers APIs that are easy to integrate in third-party software. SEkey can be used out-of-the-box with native SEcube applications, moreover it can be customized in order to meet ad hoc specifications.
Because of specific requirements and SEcube hardware/software limitations, trade-offs were made, such as the mandatory ownership of a SEcube HSM for every user of the system.
Additionally, the development of the KMS led to substantial improvements in the pre-existing SEcube software stack and to the creation of a library to manage encrypted SQL databases with the HSM.

SEkey allows developers to create security configurations where the keys are automatically and safely distributed to the entitled users, moreover the users can be organized in groups with different properties, security policies and privilege levels, in order to enforce the physical and logical separation of the actors operating in a certain environment. Thanks to SEkey, the SEcube offers a solution to deal with the key management issue without requiring a deep knowledge of cybersecurity.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis deals with the design and development of SEkey, a key management system (KMS) based on the SEcube Security Platform.
A key management system is used to generate, manage, distribute and protect cryptographic keys through their life cycle, adopting the highest security standards.
The SEcube Security Platform is based on the SEcube hardware security module (HSM), which is a special-purpose device dedicated to performing cryptographic operations, such as encryption and decryption, so that they do not have to be executed by a general-purpose machine (e.g. a personal computer).
A hardware security module mainly has two goals: to speed-up the execution of security primitives and to increase the security of the environment where it is used. A hardware security module is not a stand-alone device, in fact it works together with a host machine that is usually a personal computer or a server.
Before starting to analyse the details of SEkey, few fundamental issues need to be addressed:

- Why and when do we need a key management system?
- What are the advantages and disadvantages in using a key management system?

Key management systems are used to solve a problem that is becoming more and more significant with the ubiquitous use of encryption: the proper management of cryptographic keys. There is no doubt that encryption is a powerful means to provide security at multiple levels; however, it must be implemented correctly to avoid critical losses in terms of information, reputation, and money.
Different key management systems may have different targets and security properties, but they all share a common goal: allowing the most secure, transparent, flexible, and easy management of cryptographic keys. The most important advantage given by a KMS is that it should be able, by design, to handle keys guaranteeing the same security properties independently from the quantity of keys managed. Scalability is the feature that differentiates significantly a KMS from the manual management of keys; the manual management in fact is complex, time consuming, less secure and error prone.
In conclusion, the answer to the first question is quite simple: the complexity of many environments has grown so much that solutions to manage cryptographic keys automatically are required. Notice that these environments are found in big companies as well

as very small companies, because almost everyone has some data to protect resorting to encryption. Moreover, the current legislation imposes to adopt security measures against the loss or leakage of sensitive data, this problem is usually solved through encryption and this implies that there are keys that must be managed properly. Independently from the type of data to be protected, such as intellectual properties or healthcare information, encryption and key management are extremely relevant.
The advantages of a KMS are inherited from the aspects previously mentioned:

- scalability, because many keys can be handled;
- security, because a KMS is built to be secure (security by design);
- automation of recurrent procedures;
- fault reduction;
- increased transparency of the security layer inside applications;
- flexibility and portability (mostly for purely software-based key management systems).

The disadvantages brought by the adoption of a key management system are:

- it might be expensive to purchase or to develop from scratch;
- it may require a long time to be developed;
- it may require to modify the pre-existing security infrastructure, especially if dedicated hardware security modules are involved;
- it might be necessary to modify or rewrite some software in order to exploit the functionalities of the KMS.

In conclusion every key management system is a trade-off; however, the disadvantages are usually surpassed by the advantages granted by the KMS.
There are many ways to implement a key management system; SEkey adopts a mixed hardware-software approach where the security properties are supplied by the SEcube (i.e. AES-256 encryption and SHA-256 cryptographic hash) while the software provides a set of APIs to exploit the KMS. Most of the features offered by the SEcube are implemented in software but the device provides also an FPGA which can be customized in order to implement specific algorithms in hardware. For sake of simplicity, this thesis relies on security primitives which are completely software-implemented.
SEkey is focused on distributed environments, where ‘distributed‘ means that each actor of the environment is provided with a dedicated SEcube device to be used with the KMS. This is a crucial aspect that has had important consequences on the choices that were made during the development of the KMS. SEkey was developed mainly because the SEcube platform needed a tool to automatically distribute cryptographic keys among several SEcube devices without requiring a constant, cumbersome, and potentially not secure interaction with the person in charge of managing the devices.
SEkey has been designed to be secure, flexible, portable, and easy to use. These properties

cannot fully coexist in a hardware-based KMS because the hardware itself implies limitations with respect to a purely software-based solution.
The SEcube hardware security module is the centre of the key management system, it stores the keys of the KMS and their metadata. The security of the keys is enforced by a strong two factor authentication, where the first factor is the physical possession of the SEcube device, and the second factor is the knowledge of the PIN to access to the HSM. This qualifies the SEcube as a multi-factor cryptographic device, according to the NIST recommendations the SEcube is compliant with the FIPS 140-3 Level 4 [8] in terms of authentication mechanism because «*The multi-factor cryptographic device is something you have, and it shall be activated by either something you know or something you are.*».
The keys are stored in the flash memory of the SEcube (thus being protected by the strong authentication mentioned before); on the other hand, the metadata are stored in a database on the microSD card of the SEcube. This database is constantly encrypted, even when the KMS is accessing to it.
SEkey is built around four basic concepts: keys, users, groups of users, and security policies. This logical structure is managed by a single security administrator, who has read/write access to the data of the KMS. On the other hand, every user only has 'read' access to a limited portion of data according to the rules established by the administrator. Both the administrator and the users can exploit the KMS with the SEkey APIs, a software library compatible with Windows and Linux which allow the complete control of the key management system.
Because of the distributed architecture, the data and the keys of the KMS are constantly synchronized from the SEcube of the administrator to the SEcube devices of the users, in this way the users have all the data necessary to exploit the KMS locally, without the constant support of an administrator or another actor. The basic architecture of the system is showed in Figure 1.1, it is valid both for the administrator and for the users.

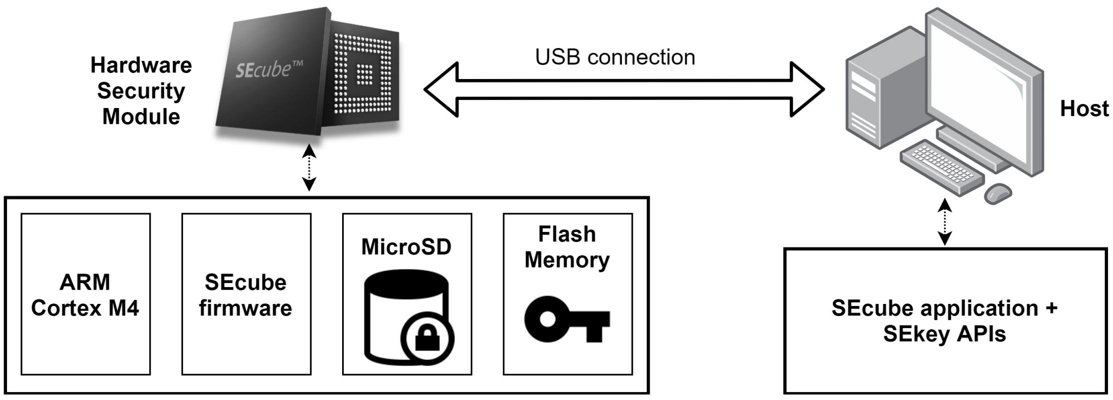

Figure 1.1. The basic architecture of the key management system

In the next chapter we will briefly describe the most important types of key management systems, along with the guidelines from the National Institute of Standards and Technology (NIST) to build a secure and reliable KMS. The remaining chapters will initially cover the details about the SEcube Security Platform and about other libraries that were used for the development of the key management system. We will move on analysing in detail all the features of the KMS, describing its architecture and how it works. Then the results of this thesis will be presented along with a recap of the main aspects of the key management system. Finally, the conclusions will be presented, together with some suggestion about possible future improvements.

Chapter 2

State of the Art

In this chapter we will analyse the main types of key management systems along with their features, then we will look at the recommendations of the NIST for the management of encryption keys.

2.1 Different kinds of key management systems

Key management systems can be divided into four main categories [13]:

- Software: this kind of KMS is purely software-based, it is basically a software that implements its own protocol or is compliant to a specific protocol in order to manage cryptographic keys. The software runs on the operating system of a machine that is not shipped with the KMS (e.g. a pre-existing server built by the customer of the KMS).
- Virtual: a pre-installed virtual machine that runs in a virtualized environment. In this case the operating system that hosts the KMS is shipped with the key management system but the hardware where the VM runs is not included in the KMS, so it is under the complete control and responsibility of the customer.
- Appliance: this is the category of SEkey. In this case the KMS is shipped as an integrated hardware and software solution, for example a server with certified hardware running the KMS on a certified operating system. Another possibility is a KMS that runs on a hardware security module.
- Service: a key management system offered as a cloud-based service that can be exploited by the customer without the need of dedicated hardware or virtual machines. This is also known as KMS-as-a-Service, it is a very popular solution because of the fast migration towards complex cloud structures where a single company may rely on several cloud server providers.

Keeping in mind these four categories, we can evaluate advantages and disadvantages of each one of these looking at the Table 2.1 [13]. Notice that this table is simply a guideline and cannot be applied to specific cases, every KMS needs to be analysed individually in

Type	Pros	Cons
Software	Wide compatibility Runs on pre-existing hardware Runs on common OS Easy to fix and update	HW and OS provided by customer Hardware may not be certified OS may not be certified Usually weaker
Virtual	Wide compatibility Runs on pre-existing hardware Easy to run multiple installations OS provided with the KMS Easy to fix and update	HW provided by customer Hardware may not be certified Usually weaker Virtualization overhead
Appliance	HW and SW provided with the KMS Turnkey installation All-in-one solution Usually more secure	Lower flexibility Difficult to fix or update HW limitations Usually more expensive
Service	No installation required Easy to use No local resources required Flexible in terms of usage and payments	Keys stored in the cloud No physical control

Table 2.1. Pros and cons of different kinds of KMS

strict relation with the surrounding hardware and software environment.
In general, key management systems based on certified special-purpose hardware (e.g. a hardware security module) are more secure even though they are usually less flexible and more expensive. A purely software-based KMS is much more flexible and possibly also cheaper but it is generally less secure than a hardware-based KMS.
Notice also that a virtualized KMS is very similar to a software KMS, the main difference is that the producer supplies also the operating system to be virtualized.
Finally there is the 'as-a-Service' approach where the KMS resides on the cloud, the customer simply has to pay a subscription plan in order to get access to a dedicated KMS that exposes APIs to operate with the cloud-based system. The disadvantage is that the customer does not have under his direct control the hardware and software used by the KMS, on the other hand the customer does not have to care about anything but paying the subscription and integrate the APIs of the KMS in his applications.
Notice that the key management system based on the SEcube falls into the 'Appliance' category because it is built with a mix of special-purpose hardware and dedicated software. A KMS built with this approach may be limited by the hardware chosen for the implementation, because hardware cannot be changed 'on the fly' to withstand different or higher requirements.

2.2 NIST specifications for key management

Despite having different features and architectures, almost every key management system is developed according to the recommendations of the National Institute of Standards and Technology of the US Government.
According to the NIST [9]:

> *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 most important aspects are:

- Key life cycle: independently from its type, a key is characterised by a state that determines its use. For example, a key may be in a state that prevents it from being used while another key may be in a state that makes it usable for encryption and decryption.

- Physical and logical access to the locations where the keys and the other data of the KMS are stored: this is of paramount importance because the data that are managed by the KMS must be protected both at the physical and logical level. For this reason, the NIST recommends using dedicated hardware solutions (i.e. a HSM) to physically protect the keys. The logical protection includes measures like encryption, authentication, integrity, access control, and accountability in order to protect the keys from unwanted modification, unauthorized access, and unauthorized usage.

- Physical and logical separation of the roles of the actors within the KMS: the roles of different actors should be clearly established in order to avoid conflicts and security flaws. The logical separation is important from a functional point of view, the physical separation is also crucial because access to the physical assets must be limited and monitored. For example, only selected personnel may have permission to physically access to the facilities where important hardware resources (i.e. key servers, backup servers, etc.) are located. Similar protection must be applied from a logical perspective implementing different privilege levels based on a strong authentication mechanism. For example, an actor with high privilege may access to critical features of the KMS while an actor with low privilege may only use basic features.

2.2.1 NIST physical level recommendations

When planning to develop a KMS from scratch or when choosing which KMS to purchase, evaluating the characteristics of the environment where the KMS must operate is very important. An in-depth analysis must be performed in order to assess where the keys and every other data will be physically stored, where the hardware will be located, how it will be protected and so on.
The NIST gives the definition of physical security [10, 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) [15].
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 [12, 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

The design of a key management system is particularly delicate at the logical level, where the roles of the actors involved in the system must be established. In particular, the NIST defines three basic principles that are of paramount importance.

The 'Separation of Duties' [11, 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' [9, 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' [11, 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.*

The combination of these three principles constitutes an important guideline for the development of a key management system. Notice that a KMS may or may not be compliant with these guidelines. For example, the 'Split Knowledge' principle is very difficult to apply because it requires a more complex architecture and more complex code to handle the KMS. On the other hand, the 'Separation of Duties' is easier to implement because, in general, the architecture of a KMS is designed having already in mind the roles of the actors involved.
In the case of SEkey, the 'Separation of Duties' and the 'Least Privilege' principles were adopted, while the 'Split Knowledge' principle was not considered because of the underlying complexity.
In conclusion, the NIST recommends to clearly identify the roles of the actors involved in the system, along with the operations that they can perform and the data to which they are allowed to access. This process is extremely important to avoid security flaws in the architecture of the key management system.

2.2.3 NIST guidelines about cryptographic keys

Cryptographic keys are the most important element of any key management system; therefore, it is worth recalling some considerations about the keys to be managed [3, 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 drive the design and development of any key management system. Moreover, the NIST documentation specifies many other guidelines and details related to the keys. In particular, a KMS should allow the management of several types of keys that can be symmetric or asymmetric, additionally each key has a specific usage field depending on its type (i.e. a key may be used only to apply data confidentiality while another key may be used for authentication).
Another important property of a key is its state. The NIST recommends six possible states that can be grouped into four operational phases [9, p. 98], as listed in Table 2.2.

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

Table 2.2. Recommended key states and phases

These states are useful to model the life cycle of a key. The transitions across the different phases are regulated by the specific requirements of each KMS, a certain system may even avoid using some of these states.
There is another parameter that is strictly connected to the state of a key, the so-called cryptoperiod. The cryptoperiod is defined as «*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.*» [9, 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.*» [9, p. 16] while the recipient usage period is «*The period of time during which the protected information may be processed (e.g., decrypted).*» [9, p. 17].
The cryptoperiod assumes a fundamental role in the management of the keys because it

may be source of security flaws and potential issues that may undermine the security of the protected data. For this reason, the NIST specifies also the suggested cryptoperiod of keys, depending on their type and usage; in the Table 2.3 are listed the suggested cryptoperiods of the key types that are used in SEkey. For more detailed information refer to the NIST documentation [9, p. 50].
Notice that these values must be intended as a guideline for the development of a KMS, the actual cryptoperiod should be chosen depending on the environment where the system is supposed to operate.

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 2.3. Recommended cryptoperiods for the types of keys used in SEkey

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 [15]:

- 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

The SEcube Security Platform and the software background

In this chapter we will briefly analyse the basic hardware and software resources that have been used in this thesis work. This chapter is useful to understand the foundation of SEkey, in terms of features offered by the SEcube hardware security module and by the software libraries that were exploited.

3.1 SEcube hardware specifications

The SEcube is a very compact System-in-Package (SiP) made of three main components as illustrated in Figure 3.1: a low-power ARM Cortex-M4 processor, a high-performance FPGA and an EAL5+ certified security controller (commonly known as smartcard). The SEcube is a unique security environment for a great variety of applications [2].

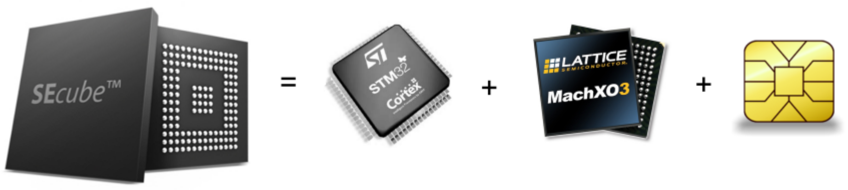

Figure 3.1. The components of the SEcube.

The SEcube chip is available on a family of devices but, for this thesis work, we only focus on the SEcube Development Kit, which has been used to develop SEkey, and on the SEcube USB Stick (also called USEcube Stick), which is the final product to be used by the administrator and the users of the key management system. The devices are shown in Figure 3.2.
The SEcube USB Stick is much more compact than the development kit and is intended to be the ‘final product‘ to be sold to the customers. The essential difference between them

is that the Development Kit has a JTAG interface in order to easily flash a new firmware on the device during the development; the SEcube USB Stick is not provided with the JTAG interface for security reasons, it comes instead with an embedded secure bootloader that prevents unwanted firmware modifications. This is a very important aspect because it is not possible to build a secure KMS based on a HSM that does not have a locked bootloader.

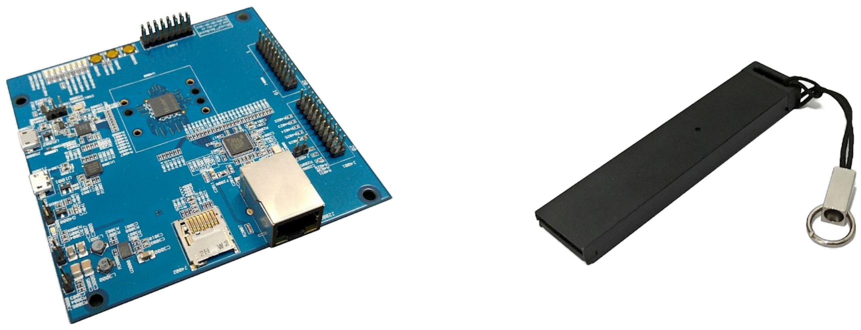
Figure 3.2. The SEcube Development Kit and the SEcube USB Stick.

The processor included in the SEcube chip is an ARM Cortex M4 RISC of the family STM32F429 produced by ST Microelectronics, the chip comes with the following features [2]:

- BGA 9x9mm form factor,
- 2 MB of flash memory,
- 256 KB of SRAM,
- 32-bit parallelism,
- 180 MHz clock frequency,
- optimized for low power consumption.

Other important components included in the SEcube are a True Random Number Generator (TRNG) embedded unit, a microSD card, and hardware mechanisms oriented to security like a Memory Protection Unit (MPU). Moreover, the Cortex M4 processor supports highly optimized CMSIS-DSP libraries to perform complex floating-point operations in order to speed-up the execution of many algorithms, and also a set of privileged execution modes which allow the implementation of custom security strategies (i.e. protected memory areas, fences around key generation primitives, etc.)[2].
The most important hardware components related to SEkey are the ARM Cortex CPU, the TRNG, and the flash memory. The TRNG is essential for the generation of cryptographic keys; the flash memory instead is probably the most significant bottleneck to the KMS as we will see later.

For the sake of this thesis, no additional details are provided about the embedded FPGA and Smart Card because they are not currently exploited by SEkey.

3.2 SEcube firmware

The firmware is the core of the SEcube board. This HSM is not provided with any operating system (e.g. an embedded OS like FreeRTOS), so the firmware is responsible of handling the requests coming from the host device. In this configuration the SEcube is the slave, while the host is the master.
The firmware of the board implements the master-slave configuration with a very basic schema: the only task of the SEcube is to wait for a request coming from the host and execute the corresponding action (i.e. the host sends a payload to be encrypted, the SEcube sends back the encrypted data). This behaviour is shown in Figure 3.3.

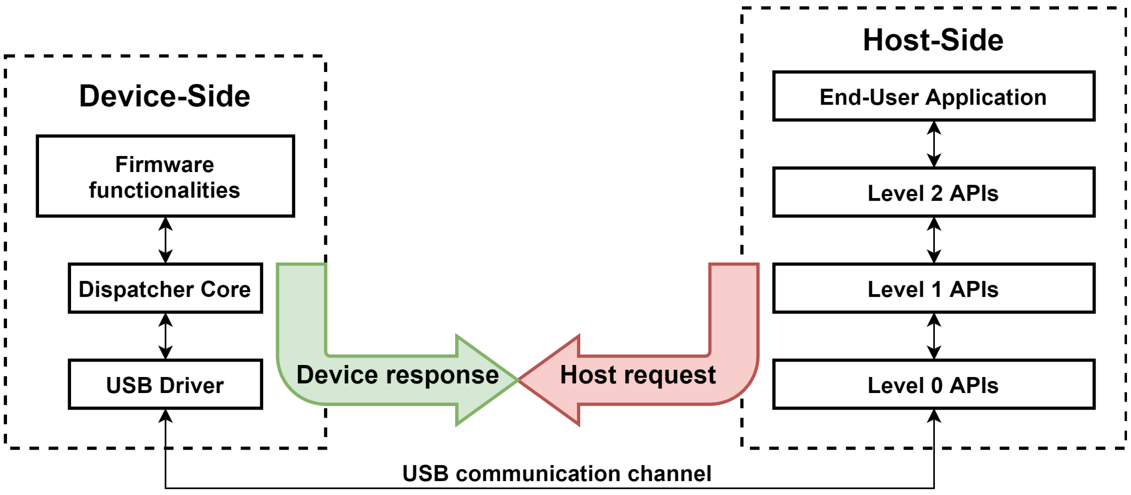

Figure 3.3. The flow of requests and responses involving host and device.

Notice that the firmware is sequential, meaning that the host must wait for the response of the SEcube before sending another command to the device. This is an important limitation that must be considered when writing SEcube host-side applications.
The sequential implementation of the SEcube firmware poses a significant bottleneck on every architecture where a SEcube is supposed to provide security properties to a server that must withstand a high load. In the case of a centralized key management system with a single key server implemented on the SEcube, the firmware would not be able to process concurrent requests and the performance of the key server would be very limited. This is an example of hardware limitation imposed by the HSM; it is significant because it influences the design of the KMS.
The current firmware implementation implies another bottleneck, because the firmware is not optimized to exploit the entire flash memory of the SEcube. In particular, only ~128 KB of the original 2 MB of flash memory are actually usable, therefore SEkey can only store a quite limited amount of cryptographic keys on the device (the theoretical limit

is 4096 256-bit keys but, considering some overhead, a realistic estimate is ~3000 256-bit symmetric keys).
An optimized firmware is currently being developed as a thesis work by another student and will be released later in 2020 in order to enhance the capabilities and performance of SEkey. Here is a simple use case that shows the workflow of the firmware.

Use Case: interaction between the host and the firmware of the SEcube

Preconditions: SEcube connected to the host, user logged in the SEcube.

1. The user calls a host-side API to store a new key in the flash memory of the SEcube.
2. The host-side application prepares a packet filled with the command identifier and the data (if any) to be sent to the SEcube.
3. The host-side application sends the packet to the SEcube through the USB interface.
4. The firmware parses the content of the packet looking for the command identifier.
5. The function associated to the command identifier is called, in this case the function to create a new key.
6. The firmware creates a new random key using the embedded TRNG and stores the key in the flash memory of the HSM.
7. A return code is sent from the firmware to the host.
8. The host-side application checks the return code and reacts accordingly.

3.3 SEcube Level 0, Level 1, and Level 2 APIs

The host-side software available for the SEcube runs on the computer to which the HSM is connected; it consists of three categories of APIs identified as Level 0, Level 1, and Level 2. These APIs are required in order to communicate with the SEcube and exploit its features.

- Level 0: the closest layer to the device, oriented to the implementation of basic functionalities such as the communication with the HSM, the discovery of connected devices and so on. This layer is responsible of allocating, maintaining and deallocating the communication channel through the USB interface that is used to send commands and receive responses. L0 is divided into: Provisioning APIs, Communication APIs, Commodities APIs.
  Other features of Level 0 include: initializing the SEcube serial number, handling multiple SEcube connections on the same host device, handling transmission/reception of data packets between the host and the HSM.
- Level 1: this layer implements the basic functionalities to be exploited by secure applications and secure libraries, for example: login/logout operations, encryption and decryption of data, retrieving the list of supported encryption algorithms and so on.

- Level 2: this is essential to create a secure environment easy to use also for developers who are not expert about cybersecurity. The SEkey APIs belong to L2, the same holds for SEfile and SElink APIs. L2 APIs are usually part of some library (e.g. SEfile), they must be easy to integrate in any third-party software without the need to care about low-level security details because these issues are automatically managed.

The structure of these APIs is hierarchical, upper levels are built on top of lower levels therefore each layer sees lower layers as 'services' which can be exploited (i.e. L2 relies on L1, and L1 relies on L0). This hierarchy is valid only for the host-side software, the firmware in fact is different and generally more fragmented.
The Figure 3.4 clearly shows the structure described above, where the host-side is hierarchical while the device-side (the firmware) is more complex and fragmented.

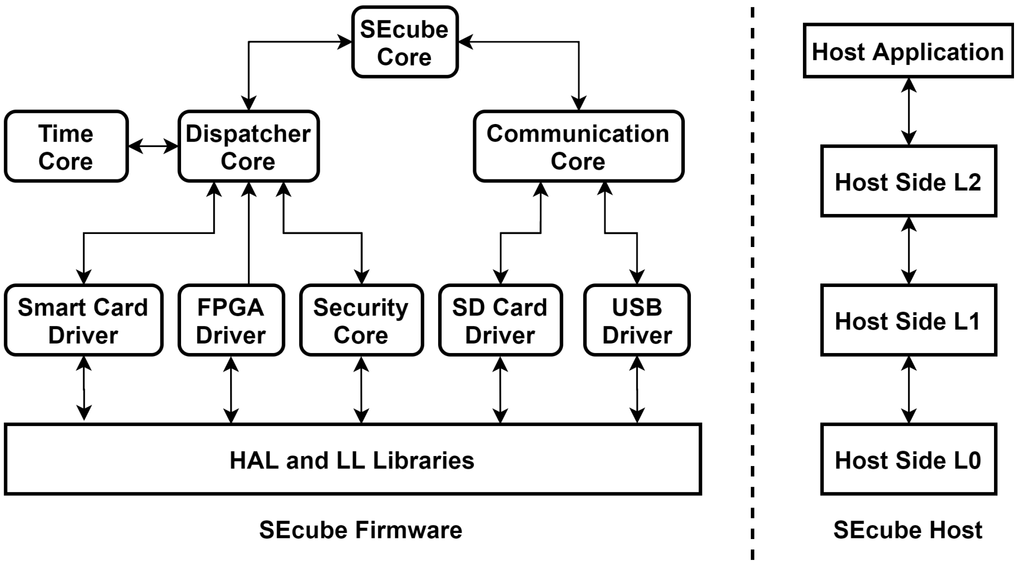

Figure 3.4. The device side and host side software organization [2, p. 11].

3.4 SEfile

SEfile is a Level 2 library which runs on the host to which the SEcube is connected. It works as a cryptographic wrapper around the standard file system interface available on Windows, Linux and Mac.
The goal of SEfile is to improve the security in the data-at-rest scenario not only encrypting files but, most importantly, by keeping them encrypted even when they are being used by any application. The simplified workflow of SEfile is depicted in Figure 3.5.
SEfile is independent from the algorithm chosen to encrypt the data; it simply keeps track of the security context (i.e. the encryption key and the crypto algorithm) used with a specific file, this context is then used by Level 1 APIs to perform operations such as encryption, decryption, and digest computation.

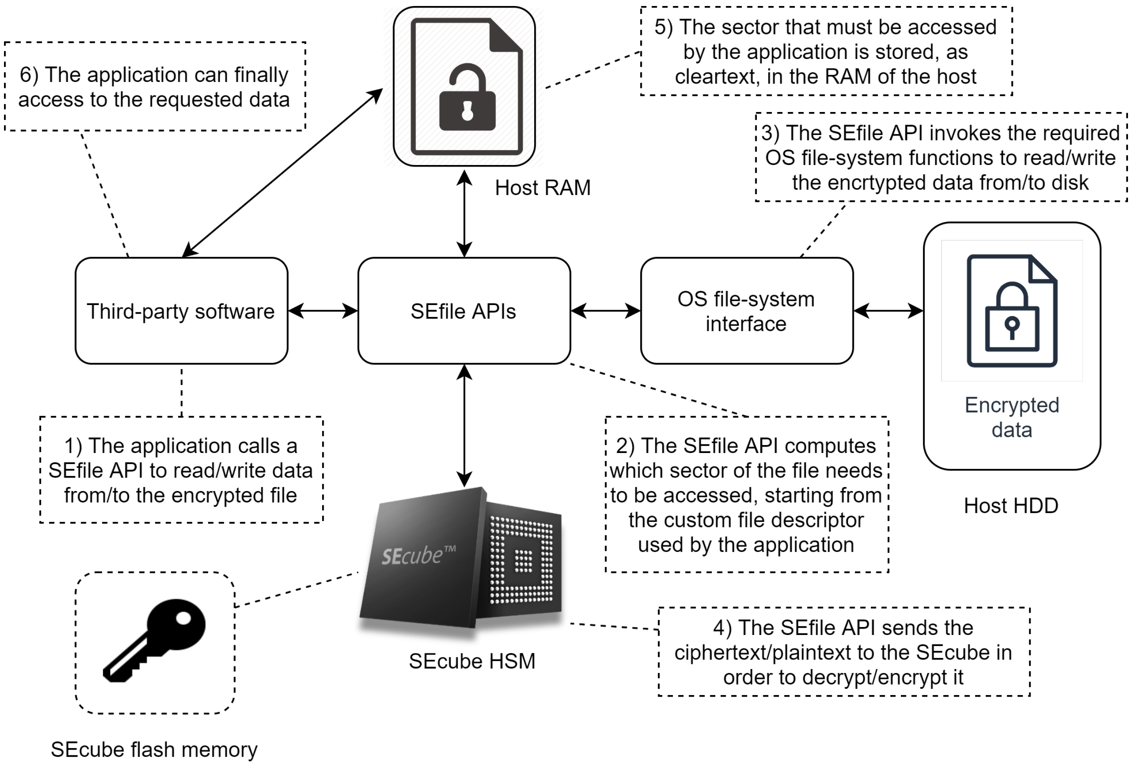
6) The application can finally access to the requested data
Host RAM
5) The sector that must be accessed by the application is stored, as cleartext, in the RAM of the host
3) The SEfile API invokes the required OS file-system functions to read/write the encrtypted data from/to disk
Third-party software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

Figure 3.5. A simple overview of how SEfile works.

The advantages of SEfile are:

- the data are always encrypted on disk;
- it grants confidentiality, integrity and authentication for encrypted files;
- it is based on AES256-HMAC-SHA256;
- it is portable across different operating systems;
- being based on the SEcube, it enforces the strong two factor authentication;
- it automatically tracks the security context used for a certain file (encryption key and crypto algorithm, this feature was added while developing SEkey and is not present in the original SEfile library).

The disadvantages of SEfile are:

- applications must be rewritten to use specific SEfile APIs;
- it only works with the SEcube;
- it introduces an overhead in terms of computation time and size of the file;

- when an encrypted file is accessed for a read or write operation, the cleartext of the file sector is stored in the RAM memory of the host.

The structure of a file encrypted with SEfile consists of a header sector that contains some metadata (i.e. the name of the file, its length, the ID of the key used for encryption, the type of algorithm used for encryption, etc.); every other sector contains the actual data of the file. The typical structure of a file protected with the SEfile library is illustrated in Figure 3.6.

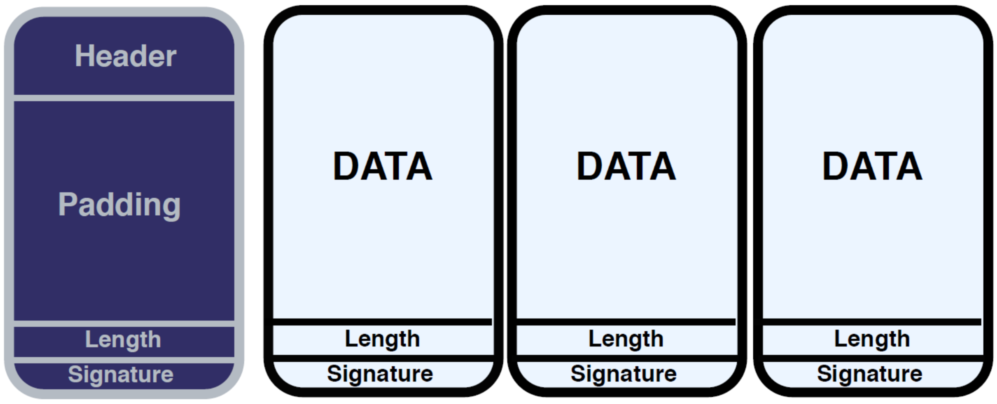

Figure 3.6. The organization of file sectors inside SEfile [16, p. 22].

The standard sector size is 512 bytes, but it can be easily customized in the source code of SEfile. Every sector of the file is encrypted to guarantee data confidentiality, then an authenticated digest of the sector is computed with HMAC-SHA256 to guarantee also integrity and authentication. Notice that the header sector is encrypted with AES-256-ECB, all the other sectors are encrypted with AES-256-CTR. Moreover, the name of the file is moved into the header sector, then it is replaced by an apparently random string which is the SHA-256 digest of the original name.
Because of the custom sector structure introduced by SEfile, random padding may be required to fill the space not used by the valid data inside the sector. This happens in the header, but it may occur elsewhere if the sector is bigger than the quantity of data that must be stored. The padding used by SEfile is always random, in order to avoid 'known plaintext' attacks which would otherwise be possible if the padding was always the same (i.e. zero padding).
Finally, is important to remind that the security primitives required by SEfile in order to work correctly (i.e. encryption, decryption, digest computation) are always performed inside the SEcube.

3.5 SQLite

SQLite is a SQL database engine optimized to be small, fast, and reliable but at the same time able to offer all the features of SQL [14]. SEkey exploits the SQLite C interface to store the metadata of the key management system inside a traditional SQL database, which is then encrypted with the SEfile library.
According to the SQLite website, the SQLite database engine is the most widespread database engine in the world. It can be found in billions of devices such as: every Android phone or tablet, every Apple product (i.e. iPhone, Mac, iPad etc.), every Windows 10 PC, and it is used also by well-known software like Mozilla Firefox, Google Chrome, Apple Safari, Apple iTunes, Dropbox, and many more [7]; even many embedded systems use SQLite.
The fact that SQLite is so widespread is very important because it means that SEkey uses a database engine which is proved to be stable and reliable, moreover the SQLite C APIs are very well documented making the development and the customization of SEkey easier. Another paramount advantage is that SQLite was already used by the developers of SEfile in order to build a demo application with a secure version of the popular SQLite Browser tool [1]. SQLite allows the developer to implement a custom virtual file system which replaces (or better, acts as a wrapper around) the real file system of the machine where the SQLite engine runs. For all these reasons, SQLite appeared to be the most obvious choice.
Even though SQLite is a very complex and powerful database engine, SEkey exploits a tiny fraction of its potential. The main features required by SEkey are the A.C.I.D. properties and manual transaction handling. SQLite includes also many possible optimizations to be implemented in order to make the database engine faster and more efficient; however, at the moment SEkey does not exploit any SQLite advanced feature because the goal is to build a system as simple and as reliable as possible.
Finally, is important to notice that the SQLite version used by SEkey is the 3.19.3, released on 2017-06-08. This is the most recent version that appears to be fully compatible with the SEfile library. In the next chapter, more information will be given about this choice and about how the SQLite database engine is integrated within SEkey, along with the SEfile library.

Chapter 4

SEkey development and implementation details

This chapter covers the steps of the development of the key management system and gives detailed insights on the internal structure of SEkey.
We will see how the KMS addresses the issues posed by the NIST standards about the management of the keys and their life cycle, the separation of the roles of the actors, the physical security of the environment.
We will also analyse how the data of SEkey are organized at the physical and logical level, how they are distributed to the users, how they are protected. Finally, we will look at the implementation details of SEkey in terms of the integration of pre-existing SEcube libraries to support the key management system.

4.1 The reasons for the development of SEkey

The development of SEkey was mainly driven by the necessity of creating a library to easily manage and share cryptographic keys on multiple SEcube devices without having to deal with low-level APIs such as Level 1 and Level 0.
Before the development of SEkey, exploiting the SEcube using only the L0 and L1 APIs was difficult because it was required to manually handle many low-level security details.
With the original SEcube firmware and without SEkey, when a security manager wanted to create a simple system with few SEcube devices sharing few cryptographic keys among them, he had to manually inject the same key in all devices and the Level 1 SEcube APIs did not even provide a way to generate a sufficiently random key value. This process was very time consuming, error prone, not secure and clearly not adequately scalable. Moreover, the metadata (i.e. the validity period, the algorithm, etc.) about the keys had to be manually distributed to every SEcube involved. The distribution of the metadata posed several issues about the scalability and security of this fully manual system.
Another significant problem arose with the SEfile library, which was written to allow the encryption and decryption of files in the data-at-rest scenario.
The issue was that the library did not give any support to remember the ID of the key used to encrypt a file, so the user was supposed to manually keep track of all the encryption

operations performed by the SEcube. This approach was not flexible nor scalable with a huge number of files; it was also very error prone because it totally relied on the user. In the case of the SEfile library the user was also supposed to choose and remember the encryption algorithm.
These few examples raise many problems related to the absence of a key management system in the SEcube Security Platform:

- Greater difficulty in developing SEcube applications.
- Greater difficulty in learning how to properly use SEcube applications (because the user must interact at low-level with the SEcube).
- Every SEcube user and every SEcube developer is supposed to have a solid cybersecurity background.
- Forcing the user to interact with low-level SEcube features and security primitives is error prone and leads to security vulnerabilities.
- The manual approach to the management of cryptographic keys is not feasible in complex scenarios, it is not flexible, and it does not scale well.
- Regarding SEfile, the user is supposed to remember the ID of the key and the algorithm which are used for each SEfile operation, because the library does not provide any automatic system to store this information.

These problems highlight the necessity of a mechanism to remove the complexity from the user side, addressing it in a transparent way.
SEkey is the solution to the problem, it allows an easy and straightforward management of security issues, especially concerning keys and encryption. The SEkey library takes care of the complexity once demanded to the user and automatically handles keys in order to always provide to the user the right key to be used.
Here is a use case of the SEfile library before and after the introduction of the KMS.

Use Case: SEfile without SEkey

Preconditions: user logged in the SEcube.

1. The user chooses an algorithm and a key, the key must be stored in the SEcube.
2. The user initializes SEfile with the chosen algorithm and key.
3. The user calls the SEfile API to create a new encrypted file.
4. The SEcube creates the encrypted file.
5. The user notes down somewhere (i.e. on a notebook) the chosen algorithm and the ID of the key used, because the library does not keep track of them.

Use Case: SEfile with SEkey

Preconditions: user logged in the SEcube, SEkey started

1. The user calls the SEkey API to retrieve the most secure key to be used, based on the security perimeter that the user wants to obtain (i.e. encrypt the file for himself, encrypt it so that selected people can access to it, etc.).
2. The user initializes SEfile with the key returned by SEkey, the algorithm is automatically retrieved depending on the ID of the key.
3. The user calls the SEfile API to create the encrypted file.
4. The SEcube creates the encrypted file.
5. The SEfile library automatically writes in the header of the encrypted file the ID of the key and the algorithm used for encryption. The user is not supposed to do anything else than checking the returned value to detect any error.

4.2 High-level SEkey overview

SEkey is structured around four simple concepts: keys, users, groups of users, and security policies. The idea is to build a system where the users are organized in groups, every group is characterised by a specific security policy and the keys managed by the KMS are uniquely associated to the groups.
The users have access to the keys associated to the groups to which they belong; moreover, the users are connected by a network determined by the intersections of their groups. Two users know each other if they have at least one group in common, otherwise they are not aware of each other.
The keys associated to a group must be compliant with the security policy of the group, the security policy determines the security level of a group (i.e. a group that requires AES-256 encryption is more secure than a group that requires 3-DES encryption).
This approach is finalized at modeling real world environments where the security administrator of a company may want to separate the employees who work in a specific department from people who are in charge of other tasks, moreover the security policy allows to distribute the personnel on several security levels depending on their job.
The KMS is focused only on symmetric cryptography, mainly because the SEcube does not support asymmetric cryptography. Within symmetric cryptography, there are many different types of keys that may be used for different purposes: data encryption, authorization, authentication, key derivation, key wrapping. For sake of simplicity, SEkey always assumes that the managed keys are used for symmetric data encryption. This assumption comes from the fact that there is not any SEcube library currently requiring other types of keys. However, the basic support for many types of keys is included in the code of SEkey and it will be activated as soon as other SEcube-based libraries that require other key types will be developed.

The figure 4.1 shows a very simplified view of the SEkey system with its main building blocks, notice that multiple SEcube devices are involved in this schema: there is a single SEcube for the administrator and there is a SEcube for each user of the key management system.

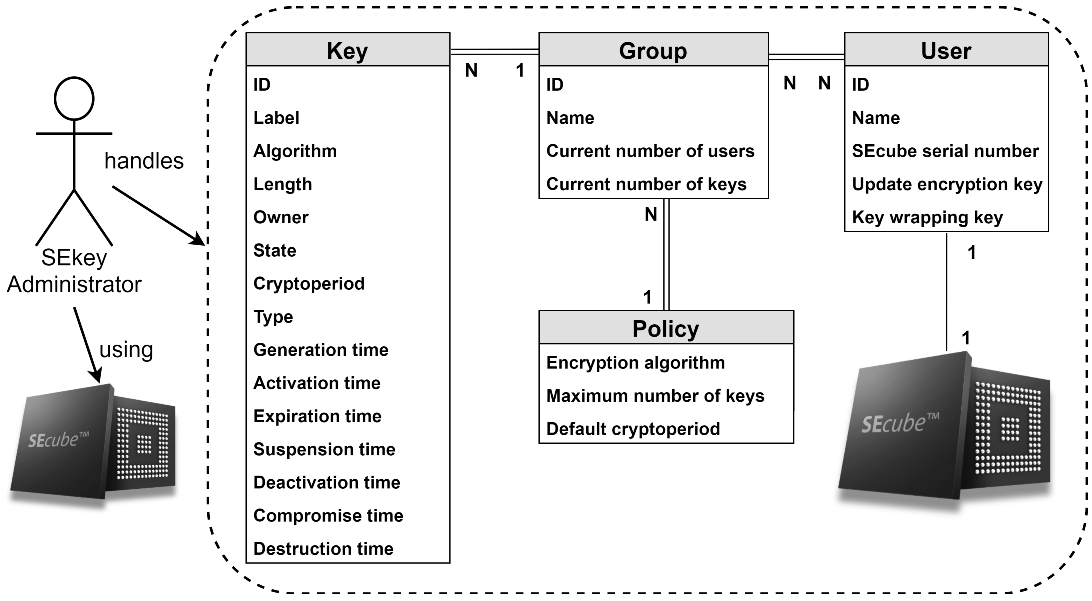

Figure 4.1. SEkey basic building blocks.

4.2.1 Key attributes and supported actions

At the moment, SEkey is only capable of managing keys for symmetric data encryption, mainly because the software and the libraries available for the SEcube are focused on symmetric encryption. However, the source code of the KMS supports also other types of keys (i.e. key wrapping keys, symmetric authorization keys, etc.) but there is not any API of SEkey explicitly written to deal with keys intended for purposes other than symmetric data encryption. Every key is characterised by several attributes:

1. ID: a unique alphanumeric string that identifies the key within the KMS;
2. label: a human-readable string to describe the key or its purpose;
3. algorithm: the crypto algorithm to be used in association with the key;
4. length: the length in bits of the key;
5. owner: the group that is associated to the key;
6. state: the current condition of the key (i.e. active, compromised, destroyed, etc.);
7. cryptoperiod: the time span during which the key can be used for encryption;

8. type: the type of the key (currently only symmetric data encryption keys are supported);
9. generation time: the time at which the key has been created;
10. activation time: the time at which the key has become active (0 if the key has never been activated);
11. expiration time: the time at which the key will be deactivated (used for automatic deactivation, computed as activation time + cryptoperiod);
12. suspension time: the most recent time at which the key has been suspended (0 if the key has not been suspended);
13. deactivation time: the time at which the key has been deactivated (0 if the key has not been deactivated);
14. compromise time: the time at which the key has been compromised (0 if a key has never been compromised);
15. destruction time: the time at which the value of a key has been physically destroyed (0 if the key has not been destroyed).

Notice that the ownership of the key is static, meaning that the owner is set once and never changed so the ownership property cannot be transferred. The owner of a key is always a group and never a single user; however, a group might not own any key.
Among all attributes, the cryptoperiod is very important. As specified by the NIST [9], the cryptoperiod is made of two values: the originator usage period and the recipient usage period. The first one determines for how long a key can be used for encryption, the second one for how long a key can be used for decryption.
The approach of SEkey is simpler, considering the cryptoperiod as the same of the originator usage period, therefore the cryptoperiod is the time span during which a key managed by SEkey can be used to encrypt data. On the other hand, there is no limit to the recipient usage period. A key may be used for an indefinite amount of time to decrypt data, the revocation or destruction of a key is a responsibility of the security administrator that must execute it as soon as a key is compromised or is not needed anymore.
The state is another crucial attribute of a key, it determines the operations that may be performed using the key, and it describes the entire life cycle of the key. SEkey follows the recommendations about the key life cycle reported by the NIST [9], applying some simplification where possible. Here are the key states:

- Pre-active: a key in this state cannot be used neither for encryption nor for decryption. This is the default state for every new key.
- Active: a key in this state can be used for encryption and decryption. This implies that it was activated in the past and that it will be deactivated in future.
- Suspended: a key can be in this state only if it was active and then it was suspended for some reason. A suspended key can be used for decryption but not for encryption, it can also be activated again. Notice that the time during which the key is suspended still contributes to the time that determines the expiration of the key.

- Deactivated: a key in this state cannot be used for encryption but only for decryption of old files once encrypted with the key when it was active. Notice that a key may be set to the deactivated state even without being active in the past, on the other hand a key which is active will be set to deactivated once it reaches its expiration time.
- Compromised: a key in this state may have been stolen by some attacker or may have been leaked outside of SEkey. This key is not secure anymore, therefore it must not be used for encryption; it should be used as soon as possible to decrypt the files which were encrypted, in order to encrypt them again with a new, secure key. Old files related to the compromised key should be deleted.
- Destroyed: a key in this state is not part of SEkey anymore. Its metadata are still retained in the SEkey database, but the actual key value has been deleted and cannot be recovered so all files which are still encrypted with a destroyed key are lost forever.

There are specific rules about the state transitions, as illustrated in Figure 4.2:

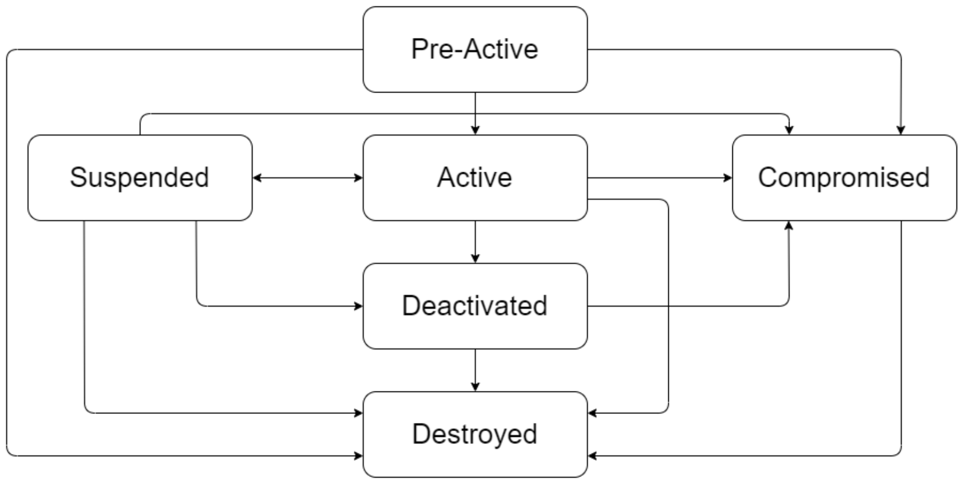

Figure 4.2. Key state transition schema.

1. A key in ‘pre-active‘ state can change to active, compromised, destroyed.
2. A key in ‘active‘ state can change to suspended, compromised, deactivated, destroyed.
3. A key in ‘suspended‘ state can change to active, deactivated, compromised, destroyed.
4. A key in ‘compromised‘ state can only change to destroyed.
5. A key in ‘deactivated‘ state can change to compromised, destroyed.
6. A key in ‘destroyed‘ state cannot change to any other state.

Notice that the only state always assumed by a key is the ‘pre-active‘, because it is set automatically at the creation of the key; all the other transitions are optional (e.g. a key may also stay forever in the default pre-active state).

The compromised state, for example, is never used unless a key is stolen by an attacker or is leaked outside of the SEcube where it is stored; the same holds for the ‘suspended‘ state that is not mandatory (a key may be activated and deactivated without assuming the intermediate suspended state). Finally, not even the destroyed state is mandatory; it is used only when the SEkey administrator is sure that a key will never be required again for decryption so it can be safely deleted.
The figure 4.3 illustrates a possible key life cycle in a very simple case where the key is created, activated, and it finally expires. Additional state transitions (i.e. to suspended and destroyed) are not mandatory.

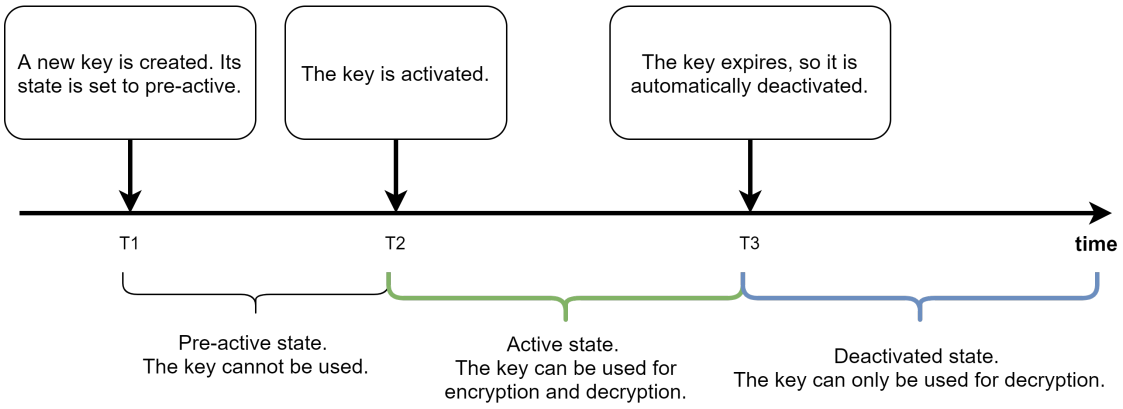
A new key is created. Its state is set to pre-active.
The key is activated.
The key expires, so it is automatically deactivated.
T1
T2
T3
time
Pre-active state. The key cannot be used.
Active state. The key can be used for encryption and decryption.
Deactivated state. The key can only be used for decryption.

Figure 4.3. An example of a possible key life cycle.

These are some of the operations that can be done using the APIs of SEkey:

1. Key creation: a new key is generated using the True Random Number Generator of the SEcube, its state is set to pre-active, the creation timestamp is computed, the owner attribute is set according to the request of the caller and the other attributes are set to the required values.

2. Key activation: the activation timestamp is set and the key becomes usable for encryption, the expiration time of the key is set according to the cryptoperiod of the key (or the cryptoperiod of the security policy of the group if this is shorter than the cryptoperiod of the key), the state of the key becomes active.

3. Key suspension: the state of the key is changed to suspended and the key is not usable for encryption, the suspension time is set.

4. Key deactivation: the state of the key is changed to deactivated, the deactivation time is set, from this moment on the key is not usable anymore for encryption but it may be used for decryption.

5. Key destruction: the state of the key is set to destroyed, the destruction time is set, the value of the key in the flash memory of the SEcube is erased, the metadata of the key are retained.

6. Key revocation: the state of the key is set to compromised, the compromise time is set, the key is not usable for encryption, every file encrypted with it should be decrypted as soon as possible in order to be encrypted with a new key.

Notice that the key activation involves the setting of the expiration time of the key; however, the key can be deactivated or revoked before its expiration time is reached. The administrator may decide to revoke a key if it is leaked outside of the KMS, similarly he may decide to deactivate a key simply to replace it with a new key. If the administrator does not perform any of these actions, a key is deactivated automatically when its expiration time is reached.
The cryptoperiod used to compute the expiration time of a key is the shortest cryptoperiod between the one of the group that owns the key and the one specified as parameter to the key creation function. The latter is optional, therefore by default the cryptoperiod of the security policy of the group is used, unless a shorter cryptoperiod is specified during the creation of the key.

4.2.2 User attributes and supported actions

The users of the KMS are provided with a dedicated SEcube HSM, therefore every user is associated to a specific device. Every user is characterised by the following attributes:

1. ID: a unique alphanumeric string that identifies the user within the KMS;
2. name: the name and surname of the user;
3. SEcube serial number: the serial number of the SEcube device that belongs to the user;
4. Update encryption key: symmetric data encryption key shared with the administrator to encrypt the data sent by the administrator to the user;
5. Key wrapping key: key used to encrypt the keys that are sent from the administrator to the user.

More information about the last two attributes will be given in Section 4.5. Here are some of the operations that can be done:

1. creation of a new user;
2. insertion of a user into a group;
3. deletion of a user from a group;
4. deletion of a user from the entire KMS, therefore from all groups.

4.2.3 Group attributes and supported actions

A group consists of a set of users, every group is characterised by the following attributes:

1. ID: a unique alphanumeric string that identifies the group within the KMS;
2. label: a human-readable string to describe the group or its purpose;
3. current number of keys;
4. current number of users;
5. security policy: the set of security rules applied specifically to the group.

Notice that a group includes zero, one or multiple users. These are some of the operations available for the groups:

1. group creation (by default a group is empty);
2. group deletion (every user is automatically removed, every key owned by the group is deactivated);
3. user insertion;
4. user removal;
5. key insertion (when a key is created a group is set as owner).

4.2.4 Security policy attributes and supported actions

A policy is a set of security rules which are applied to a specific group, every group is associated to a policy. A policy consists of:

1. crypto algorithm to be used by all the keys owned by the group associated to the policy;
2. maximum number of keys that the group associated to the policy can own;
3. default cryptoperiod of the keys owned by the group associated to the policy.

Notice that the cryptoperiod of the security policy is applied to the key of a group only if it is shorter than the specific cryptoperiod of the key. In this way the security administrator still has the possibility to create a key with a cryptoperiod shorter than the general policy of the group.
Notice also that the policy of a group can be modified in terms of maximum number of keys and default cryptoperiod (the change is not retroactive), but the algorithm cannot be changed. If the security administrator needs to modify the algorithm, a new group with a new policy must be created.

4.3 The roles of the administrator and of the users

SEkey is structured according to a master-slave approach where the administrator is the master, and the users are the slaves.
The roles are clearly distinct, there is only one SEkey administrator who is the only actor with enough privilege to modify (or create) the data stored in the KMS, performing operations on the keys, groups, users, and security policies. The users do not even know that the administrator is there, their job is to use the system as it is, they do not care about how it is managed.
The administrator has full access to SEkey (read and write privilege) while the users only have the right to read the data concerning them. According to the 'Least Privilege' principle of the NIST, the administrator is the only actor who has a global view of the system, the users can only see a fraction of the data of the KMS because they are limited by the groups to which they belong.
Let us consider the example shown in Figure 4.4, where the administrator has added to the KMS three users named Alice, Bob, and Eve. He also created two groups, the first group includes Alice and Bob while the second group includes Bob and Eve. Every group owns a key which is active and can be used to encrypt the data shared among the members of the group.

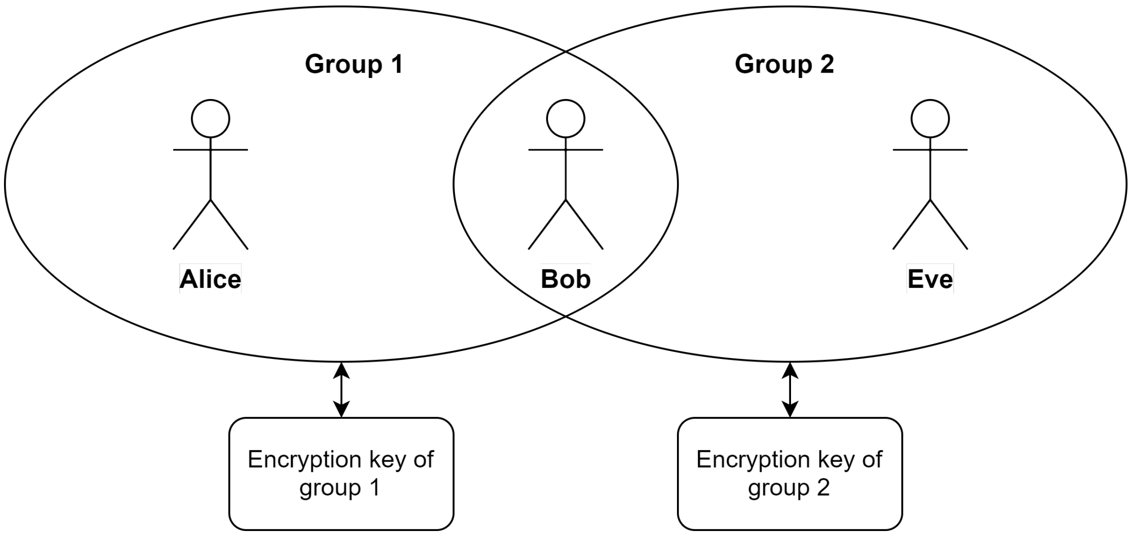

Figure 4.4. A simple configuration of users and groups.

- Alice knows Bob because they are both members of Group 1, she also knows the encryption key of Group 1. She does not have any knowledge about Group 2, its encryption key and its members (Eve). Finally, she is not aware of the fact that Bob is also a member of Group 2.
- Bob knows both Alice and Eve because he is a member of both Group 1 and Group 2. He also knows the keys of both groups.

- Eve knows Bob because they are both members of Group 2, she also knows the encryption key of Group 2. She does not have any knowledge about Group 1, its encryption key and its members (Alice). Finally, she is not aware of the fact that Bob is also a member of Group 1.

The limitations imposed to the users, following the 'Least Privilege' principle, are also related to the features of the KMS. Most of the APIs of SEkey are available exclusively to the administrator, because they are used to change the configuration of the key management system (i.e. add a user, add a key, add a group, etc.). The users only have access to the bare minimum APIs that are needed to use the KMS; moreover, they are not allowed to perform any change on the data of the system.
This policy is implemented thanks to the login levels of the SEcube. Since every actor involved in the KMS uses his personal SEcube to exploit the system, the login executed on the SEcube is used to determine a privilege level.
The SEcube device allows two types of login: administrator and user. Each login type is protected by a PIN up to 32 bytes in length, depending on the login type some APIs of SEkey and some features of the SEcube firmware are disabled or enabled. The values of the two PIN codes used to access to the SEcube are set during the physical initialization of the device, that happens after flashing the firmware.
The administrator of SEkey oversees the initialization of every device, including his own. There is a specific API to initialize the SEcube of the administrator and then there is another API to initialize the SEcube devices of the users. The PIN codes of the devices must be unique and they are established by the administrator, they are not provided automatically by SEkey because every system integrator may want to adopt a different policy regarding PIN codes (i.e. forcing the users to remember very complex PINs or allowing passphrases that are easier to remember).
Notice that, given a SEcube device, SEkey does not store its PIN codes anywhere; they are only stored in the flash memory of that SEcube. This means that the administrator must remember his admin PIN and the users must remember their user PINs. Moreover, the administrator must not disclose to the users the admin PIN set for their SEcube devices, because this would compromise the entire architecture of SEkey based on different privilege levels. If required, the administrator might store the admin PINs of the SEcube devices of the users in a file encrypted with SEfile, but this feature is not implemented.
Finally, in order to address also the accountability issue, the administrator and all users automatically generate an encrypted and authenticated log file that can be used to analyse the history of the key management system in order to identify potential security flaws and other problems. The log file of a user is encrypted with a key known only by that user and the administrator; the log file of the administrator is encrypted with a key known only by himself.

4.4 The physical organization of the data of SEkey

The data architecture introduced in Section 4.2 and Section 4.3 can be reduced to two categories:

- Key values: the actual cryptographic keys managed by the KMS;
- SEkey metadata: all the information about the users, groups, keys, and policies which are needed to support the functionalities of the KMS.

These data are physically stored not only in the SEcube of the administrator but also in the SEcube devices of the users. The administrator is the only one who has, in his SEcube, the data about every key, every user, every group, and every security policy. The users only have a subset of these data, according to the limitations imposed by the groups to which they belong.
For example, all the data about keys, groups, and users related to groups to which a user does not belong are not included in the data that are sent to the SEcube of that user. Going back to Figure 4.4, notice that Alice's SEcube does not store any information about Group 2 because she does not belong to it, therefore she does not know anything about that group, the user Eve and the encryption key owned by Group 2.
This strategy is compliant with the 'Least Privilege' principle, implemented storing in the personal SEcube of every user only the minimum information that is needed to run the key management system.
This organization of the data is perfectly coupled to the distributed architecture of SEkey (a SEcube for every actor), it allows to fully exploit the SEcube devices of the users to build a system with greater redundancy and reliability.
The most evident consequence of this architecture is that SEkey is not a centralized KMS, which would require a central server that can be queried with requests about keys and other data. A centralized KMS is usually based on a key server, whose job is to process the requests of the users asking for a certain key or asking for access to a certain resource or service; another important task of the key server is to assert the identity of who issues a request.
This kind of approach, in SEkey, would imply a significant bottleneck because the central server would be based on the SEcube, whose firmware is sequential and cannot handle concurrent requests. In a high-load scenario this would be a problem, undermining the high-availability of the key management system.
Thanks to the distributed architecture, every user can exploit the KMS without the need of querying the administrator to retrieve a certain key or any other information, simply because the user already has all the data that he needs in his personal SEcube. Clearly, a specific data distribution protocol has been implemented in order to automatically distribute the data of the KMS to the users, more information on this protocol can be found in Section 4.5.
In conclusion, SEkey is a KMS that can properly work also in absence of a centralized architecture. This allows the users to work in 'offline' scenarios, where they have all the data that they need in their personal HSM. This architecture also prevents the users from harming the authoritative source of the data of the KMS, which is the administrator. If the SEcube of a user is compromised and the attacker wipes out all the data or modifies

something, the attack will not affect the administrator, whose task would simply be to declare as compromised every key that was stored in the SEcube of the user.
In the next two sections we will see how the cryptographic keys are physically managed and how the metadata of SEkey are handled. The Figure 4.5 illustrates the detailed data organization that we are going to describe.

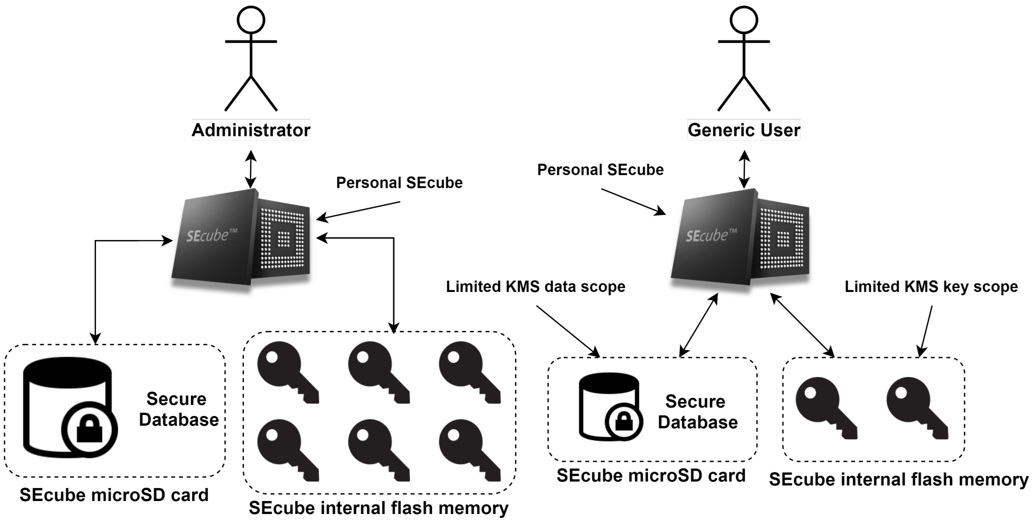

Figure 4.5. The data split between keys in the flash memory and the secure database on the SD card.

4.4.1 Physical management of encryption keys

Let us analyse how SEkey physically generates and manages the keys. Please notice that the details expressed in this section are valid both for the SEcube of the administrator and for the SEcube devices of the users.
The cryptographic keys managed by SEkey are stored, as cleartext, in the flash memory of the SEcube. The keys are generated only in the SEcube of the administrator, they cannot be generated by any other actor or in any other SEcube. The keys are generated using the True Random Number Generator embedded in the SEcube, therefore the value of any key is compliant with the specifications of the NIST [4, p. 9, 10] in term of randomness.
There is an upper boundary to the number of keys that can be stored in the KMS, in fact, due to the firmware implementation only 128 KB out of 2 MB of the internal flash memory can be used. Moreover, some space is reserved for other purposes and the data structure used to store a key in the flash memory involves a significant overhead; therefore a reasonable guess for the number of keys in a single SEcube is ~2250 256-bit keys.
In order to overcome this problem, an optimized SEkey firmware is currently being developed as main topic of another thesis work. The goal of the optimized firmware is to reduce the space required by each key stored in the flash memory and, possibly, extend the usable memory up to ~2 MB. Supposing that the optimized firmware will grant to SEkey ~1.5 MB of available storage, the SEcube would be able to store ~41000 256-bit keys (a little

overhead has been taken into account because each key must be associated to its ID on 4 bytes and its length on 2 bytes).
The keys stored in the flash memory of a SEcube cannot leave the device as cleartext. The firmware of the SEcube, in fact, does not allow to retrieve the value of a key unless it was previously wrapped with another key. This feature is accessible only if the person using the SEcube is logged in as administrator, therefore the users of the KMS will never be able to intentionally expose a key outside of their SEcube because they do not know the PIN to login as admin.
On the other hand, the SEkey administrator uses this feature to retrieve the keys that must be distributed to the users. Notice that the key is already wrapped with another key when it leaves the SEcube of the administrator, and it will be unwrapped only when it will reach the SEcube of the user; consequently, it is protected along the entire path. More details about this mechanism will be explained in Section 4.5. The most important elements to remember about the physical management of the keys are:

- Every key is generated with a True Random Number Generator.
- The keys are stored in the flash memory of the SEcube, as cleartext.
- Even though the keys are stored as cleartext, the SEcube still requires a strong two factor authentication to be used (the physical device and the PIN).
- The administrator exposes the values of the keys only if they are wrapped by another key, the wrapping takes place in the SEcube of the administrator so that the host never sees the cleartext value.
- The users cannot expose the value of a key, not even wrapped.
- The SEcube currently supports up to ~2250 256-bit keys, potentially it could support ~41000 256-bit keys.

4.4.2 SEkey metadata management with the Secure Database

SEkey would not work without a complex set of additional data about the users, the groups, the keys, and the security policies of the KMS. These data are organized into a secure database implemented applying the SEfile library to the SQLite database engine.
The secure database is stored on the microSD card of the SEcube and it is encrypted with a key that is specific to each SEcube of the system, not even the administrator can decrypt the database stored on the SEcube of a user. Notice that the key used to encrypt the secure database is never exposed outside of any SEcube, in fact it must be kept strictly private. The database of the administrator includes all the metadata of SEkey, on the other hand the databases of the users only contain the bare minimum data required by the users, according to the ‘Least Privilege‘ principle.
The idea of using a secure database to store the metadata of the KMS comes from the fact that, when the SEfile library was released, it was published with a secure version of the SQLite Browser tool which was able to work with SQLite databases that where constantly encrypted on disk.
Leveraging the work that was done by the developers of SEfile, the idea was ported to

SEkey in order to store all the data of the KMS (except the actual key values) into a traditional SQL database. The secure database consists of five tables: Users, Groups, Keys, UserGroup, Recovery.

- Users: this table contains information about each user of the system, for example their IDs and names (and many other info which are not relevant in this context).
- Groups: this table contains information about each group of the system (i.e. ID, name, number of users, etc.).
- Keys: each entry in this table contains all the information required for a key (i.e. ID, label, algorithm, length, state, creation time, activation time, expiration time, etc.).
- UserGroup: each row of this table correlates a user with a group to which the user belongs (i.e. Bob, Group 1).
- Recovery: this table is used to implement an automatic recovery procedure used only by the administrator of SEkey. This is a kind-of 'Disaster&Recovery' tool that is used by the administrator to restore the integrity of the data stored in the SEcube of a user in case of severe problems.

Thanks to the security layer provided by the SEfile library, the file of the database is not identifiable as a traditional SQLite database because the name is changed to the SHA-256 of the original name. Moreover, the structure of the file is totally modified according to SEfile, keeping each sector of the file encrypted and authenticated. Notice that, even if the key to decrypt the database file was found, it would be useless because the file structure would not be recognized by the traditional SQLite tools since it can be understood only by SEfile.
In conclusion, the secure database is a traditional SQL database managed with SQLite and SEfile, therefore the properties of confidentiality, integrity, and authentication are granted; moreover, the database is protected against SQL Injection attacks because it has been implemented resorting exclusively to parameterized queries.

4.4.3 A low-level analysis of the Secure Database

The secure database is based on the interaction between the SEfile library and the SQLite database engine, implemented thanks to a software layer called custom virtual file system. SQLite allows to specify a customized file system interface to be used by the database engine in place of the default one (i.e. the Windows file system interface) or in absence of any interface (i.e. for embedded systems).
Since SEfile is a cryptographic wrapper around the standard file system interfaces of Windows, Linux, and Mac; SEfile can be exploited by the SQLite engine using a custom virtual file system interface designed around SEfile itself. With this system, every SQLite function that interacts with the database is forced to use the SEfile APIs, making SQLite able to natively work on encrypted and authenticated databases. The interaction of SEkey, SQLite, SEfile, and the SEcube is clearly shown in Figure 4.6.
The first release of the secure database for the SEcube dates to 2017, it was published as a demo application for SEfile and it was based on the SQLite Browser software.

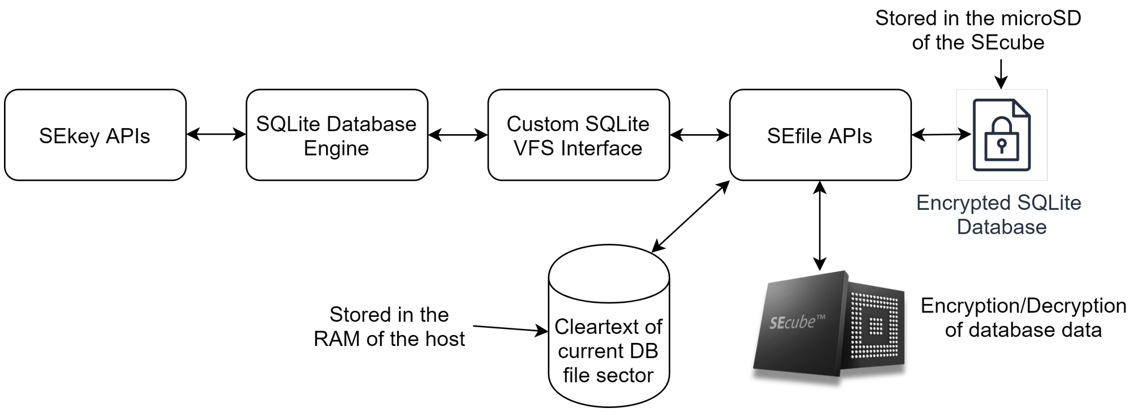
Stored in the microSD of the SEcube
SEkey APIs
SQLite Database Engine
Custom SQLite VFS Interface
SEfile APIs
Encrypted SQLite Database
Stored in the RAM of the host
Cleartext of current DB file sector
SEcube™
Encryption/Decryption of database data

Figure 4.6. The hierarchy of the Secure Database architecture.

This version was also used by the student who developed SEwallet, a password manager based on the SEcube [16], as his thesis work. In the SEwallet thesis, the developer mentioned a bug that led to the corruption of the SQLite database file. The origin of the bug was unknown, and no solution was found; however, the developer found a workaround consisting in using an empty additional table to avoid the corruption [16, p. 90].
During the development of SEkey the same bug arose again; moreover, the workaround mentioned above proved to be useless for databases with several tables and many entries. After a deep analysis of the SEfile library and of the SQLite source code, it was possible to determine that the bug probably arose from a low-level incompatibility of the SEfile library with the SQLite database engine. In fact, SEfile uses its own file sector which is a wrapper around the standard file sector defined by the operating system; at the same time SQLite uses a complex mechanism of paging and caching.
Considering that the SQLite library was also modified in order to use a custom virtual file system interface (VFS), necessary to exploit SEfile instead of the traditional OS calls for the file system, it appeared immediately clear that the problem was lying at the joint between SEfile and SQLite.
In order to solve the problem once and for all, a significant modification of the SEfile sector structure had to be done. The size of the original SEfile sector must be a power of 2 and can be configured in the SEfile library, the default value is 512 bytes. Inside each sector, 2 bytes are reserved for the length attribute (how many bytes of actual data there are inside the sector) and 32 bytes are reserved for the signature of the sector. In conclusion, 478 bytes are available for the actual data of the file.
SQLite uses a paging mechanism based on a page size which is variable but is always a power of 2. Since SEfile does not provide to the actual file an available space within the sector that is a power of 2, the most straightforward deduction is that SEfile may violate the assumptions of SQLite. The rule established by SEfile is:

$$FreeSpace = SectorSize - 32 - 2$$

where 32 bytes are reserved for the signature and 2 bytes are reserved for the length attribute. Notice that, since *SectorSize* is a power of 2, *FreeSpace* cannot be a power of 2. In order to grant a *FreeSpace* value which is a power of 2 while keeping the security properties of SEfile intact (confidentiality, integrity, authentication), the SEfile sector had to be redefined introducing a padding segment. The only way to assign to both *FreeSpace* and *SectorSize* a value which is a power of 2, using the minimum overhead, is to define the first one as half the second one. The new SEfile sector size, valid when SEfile is used with SQLite database files, follows the rules illustrated in Figure 4.7:

$$FreeSpace = SectorSize/2$$

$$Padding = (SectorSize/2) - 32 - 2 - 14$$

$$SectorSize = FreeSpace + Padding + 32 + 2 + 14$$

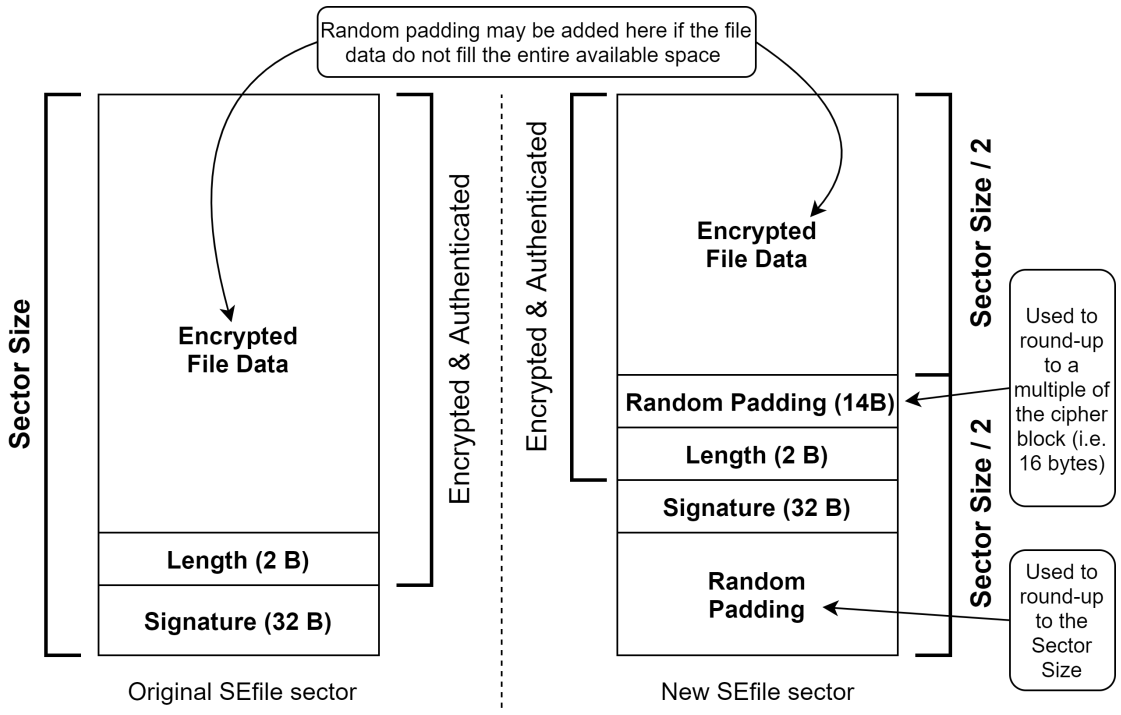
Random padding may be added here if the file data do not fill the entire available space
Sector Size
Encrypted File Data
Length (2 B)
Signature (32 B)
Encrypted & Authenticated
Original SEfile sector
Encrypted & Authenticated
Encrypted File Data
Random Padding (14B)
Length (2 B)
Signature (32 B)
Random Padding
New SEfile sector
Sector Size / 2
Sector Size / 2
Used to round-up to a multiple of the cipher block (i.e. 16 bytes)
Used to round-up to the Sector Size

Figure 4.7. The original SEfile sector and the SEfile sector used for the Secure Database.

Notice that the new SEfile sector uses two padding segments. The first segment always has a size of 14 bytes, it is random and it is used to round up the size of the data to be encrypted and authenticated to a multiple of the cipher block, which is 16 bytes (SectorSize/2 + 2B of length attribute is not a multiple of 16 so 14 bytes of random padding are needed). The second padding segment, always random, is simply used to reach the actual size of the sector; its real size depends on the size chosen for the SEfile sector.
For example, when *SectorSize* is equal to 1024 bytes, the space reserved to SQLite data is 512 bytes, the first padding segment is 14 bytes, the length field is 2 bytes, the signature is 32 bytes, and the second padding segment is 464 bytes.
The consequence is a significant overhead in the file size (double of the size that a cleartext SQLite file would require); however, the overhead is negligible because the database hardly reaches a size greater than few MB. Moreover, this new design of the SEfile sector also

allowed to upgrade the SQLite library from version 3.14.3 to version 3.19.3 (more recent versions showed another bug, still to be solved).
After having fixed this bug, the SEfile library was ported from C to C++, then it was significantly optimized adopting an object-oriented approach, the RAII paradigm, and the C++ smart pointers. This optimization led to the detection and solution of other significant problems related to the memory usage that may have caused security flaws.
Notice that the secure database is not simply something that was developed to support SEkey, it is also a ‘product‘ on its own. The SEfile library specialized for SQLite, in fact, can be used for any application that needs to handle an encrypted database resorting to the SEcube. The library works as a black box, so the application simply has to include the special SQLite/SEfile library (instead of the traditional SQLite source code) and then all the standard APIs from the SQLite C interface can be used without the need of messing around with SEfile or the SEcube.

4.5 Distribution of keys and metadata

In Section 4.4 we stated that the data of the key management system are distributed among all SEcube devices involved in the KMS, including the devices of the users. Since the administrator is the only one who has all the data of the KMS on his SEcube, a protocol to distribute the data from the administrator to the users is required. The data that must be distributed consist of the cryptographic keys and the metadata saved into the Secure Database.
As specified by the NIST [5, p. 52, 53], the data must be distributed using a protected channel that allows also for the mutual authentication of the parties involved. SEkey implements this protected channel using files, simply called ‘updates‘, which are encrypted and authenticated using AES256-HMAC-SHA256 thanks to the SEfile library. The data distribution protocol follows these rules:

1. Every update file is generated specifically for a single user (1-to-1 communication).
2. Every update file is encrypted with a symmetric key shared only by the specific user and the administrator, guaranteeing confidentiality, integrity and authentication thanks to AES256-HMAC-SHA256.
3. Every update file is automatically created or modified by the administrator following the execution of a SEkey API, the administrator is the only actor who can write data to an update file.
4. A user can only read an update file that has been specifically written for him, he cannot read the update files of other users because he does not have the key to decrypt them.
5. The users automatically process their update file as soon as they are available (i.e. when they start SEkey or when they call a SEkey API).
6. The user is responsible for deleting the update file once it has been processed, otherwise the administrator will continue appending new changes to it.

7. The users and the administrator know how to distinguish which update file is dedicated to each user, because the name of the file is based on the serial number of the SEcube of the users.

Notice that every time that the data of the KMS concerning a specific user are modified by some API, the very same API takes care of generating a new update file for that user in order to propagate the changes from the administrator to the involved user. In case there was already an update file available for that user, then the new changes are simply appended to the file.
The generation of the update files follows the same 'Least Privilege' principle that drives the design of the entire KMS, therefore a user is not notified of changes that are not related to the groups he does not belong to. For example, let us consider again the Figure 4.4; if the administrator removes Eve from Group 2, Alice is not notified of the change in her update file because she does not belong to that group so she is not related to Eve. On the contrary, Bob will be notified of the change through his update file because he belongs to Group 2 and he was related to Eve. The architecture that drives the distribution of the data is clearly shown in Figure 4.8.

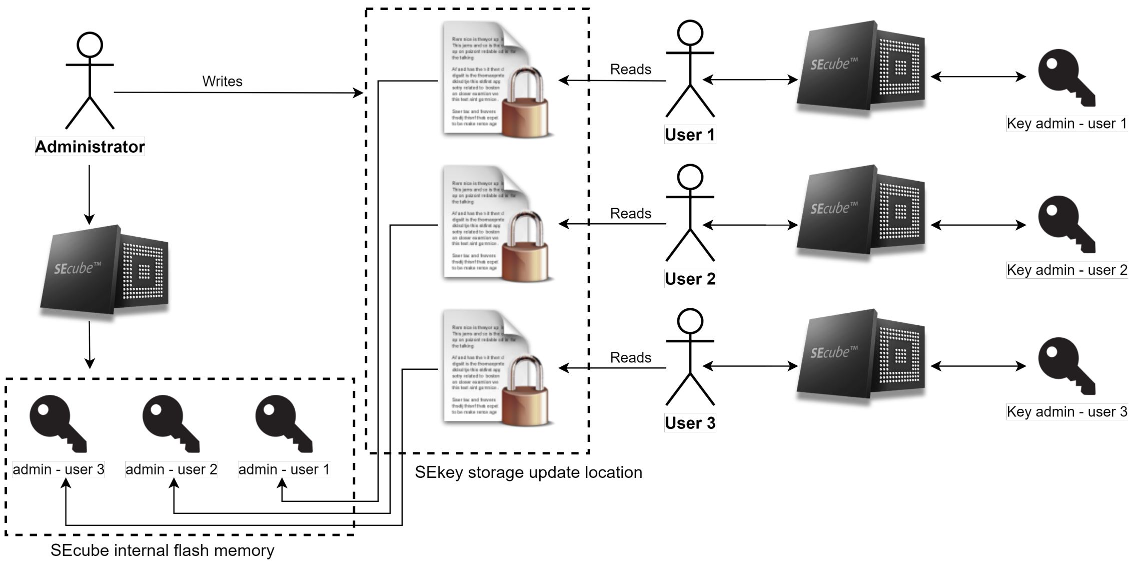

Figure 4.8. Data distribution logic.

Now we come to the physical distribution of the update files. SEkey has been implemented such that it is completely independent from the actual technique that is used to distribute the update files. When an update is generated by the administrator, it can be sent to the user in many ways: through Internet, on a local network, using a NAS, on a shared folder, with a USB drive and so on. The only thing that the system integrator must do is to setup the path where update files are stored. Basically, who wants to use SEkey must provide a way for the administrator and the user to implement a traditional producer-consumer schema. SEkey does not care if this is straightforward (i.e. administrator and user working directly in the same folder) or if it requires additional layers (i.e. synching the update files

to/from a server).
Notice that the location set to store the update files should not be accessible to anyone, except the users and the administrator; consequently, a suitable authentication mechanism (i.e. login with username and password) must be implemented. Without a proper authentication method, any attacker could download the encrypted update files and perform offline brute force attacks against them. This aspect is critical because if an attacker can perform an offline brute force attack, then the window of exposure of the system never closes.
Here is a use case that describes how the administrator automatically creates the update file for a certain user. The details mentioned in this use case will be covered in the next sections; this is one of many possible use cases, the update mechanism always works in the same way.

Use Case: update file generation mechanism

Preconditions: admin logged in the SEcube, SEkey started.
Postconditions: the update files of all the users involved are written to the desired location.

1. The administrator calls a SEkey API that implies a change in the data of the KMS.
2. The SEkey API modifies the KMS according to the request of the administrator.
3. The SEkey API generates an update file for each user of the system involved by the changes. This is necessary because the data on the SEcube of the users may not be coherent with the data of the administrator. Each update file is encrypted with a unique key.
4. If the SEkey API fails while generating the update file for a user, an automatic mechanism will perform a complete KMS data recovery for that user.
5. The users will process their files, updating the data stored on their SEcube devices.

The content of an update file is made of records, each record has four elements as depicted in Figure 4.9: type, counter, length, and value.

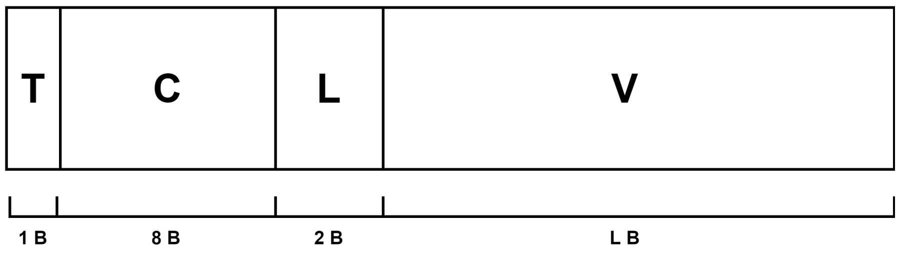

Figure 4.9. SEkey update record structure.

- Type: 1-byte field used to identify the type of the current record (a record may be a SQL query or may contain other data for another type of operation on SEkey).
- Counter: 8-byte field used by the user to check if he is aligned with the updates generated by the administrator.

- Length: 2-byte field containing the length of the value segment.
- Value: it could contain several types of data (i.e. a SQL query, the value of a new encryption key, etc.). Its length is the 16-bit unsigned integer value of the L field.

4.5.1 Protection of the update files

The update files are used to distribute the keys and the metadata managed by the KMS to the users. Protecting the update files is essential to maintain the integrity and security of the system. Every update file is protected (granting confidentiality, integrity, authentication) with two keys, which are unique for every pair of administrator and user.
The first key is used to encrypt the whole update file with the SEfile library; the second key is a ‘key wrapping key‘ since it is used to wrap the keys to be exported from the SEcube of the administrator to the SEcube of the user. Both these keys are pre-shared secrets, they are generated into the SEcube of the administrator using the TRNG and then they are injected, as cleartext because of the lack of other methods, into the SEcube of the user.
The combined action of these keys allows to obtain a double defence: an outer channel encrypted with the first key, and an inner channel encrypted with the second key. The inner channel is exploited only by the keys that must be distributed; on the other hand, the outer channel protects also the metadata. Notice that, in this way, each key is encrypted two times using two different keys. This configuration is illustrated in Figure 4.10.

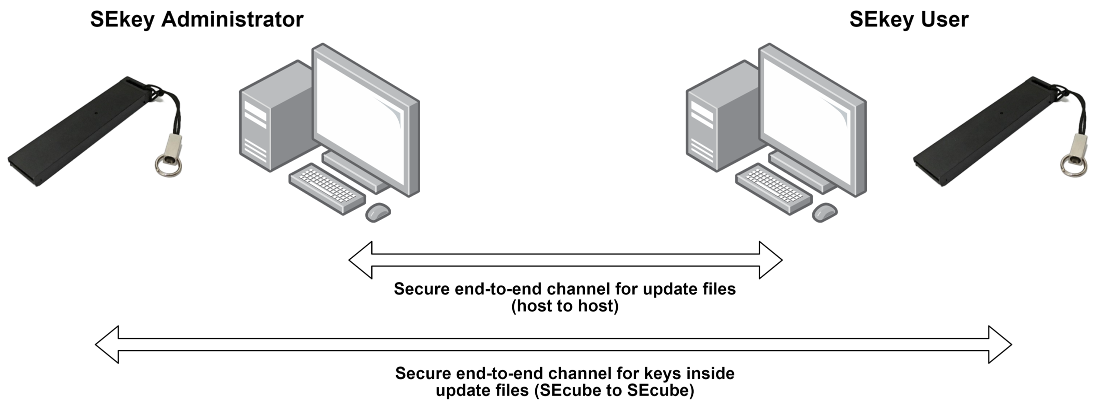
SEkey Administrator
SEkey User
Secure end-to-end channel for update files (host to host)
Secure end-to-end channel for keys inside update files (SEcube to SEcube)

Figure 4.10. Encrypted data distribution channels.

This mechanism is similar to a simple VPN. The outer encrypted channel goes from the host of the administrator to the host of the user; inside it the metadata are visible as cleartext, but the keys are already encrypted. The inner encrypted channel in fact goes from the SEcube of the administrator to the SEcube of the user, implementing a secure SEcube-to-SEcube communication that prevents even both hosts to see the cleartext value of a key.

4.5.2 Processing of the update files

The next step consists in understanding how the encrypted update files can be processed by the users in order to store the keys and the other data inside their SEcube. Let us suppose that the user application exploiting SEkey has already been configured with the correct path where to find the update files.
The processing of update files is executed automatically by the KMS whenever needed (i.e. when SEkey is started or when the user asks to the KMS for a key).
The distributed structure of the KMS, where each user relies on his local data when calling any API of SEkey, implies that a user may exploit the key management system while having outdated data on his SEcube. For this reason the update interval of the KMS must be short, forcing the user to update his local SEkey data as frequently as possible in order to avoid a mismatch of the data with respect to the latest information stored in the SEcube of the administrator.
A user should always be able to check if the administrator has prepared an update for him, if this is not possible or if the system is configured to check for updates with long time intervals (i.e. once a day at 12 AM), the possibility for the local SEkey data to be outdated with respect to the administrator is higher and problems may arise.
The SEkey update mechanism is transparent to the user, there are three possible types of update files: standard update, recovery update and ‘init‘ update. All the three types share the same record structure (Figure 4.9) so from the user point of view there is not a real difference among them because they are processed in the same way.
Each type has a different priority, in particular:

- ‘Init‘ update: top priority, must be processed before any other update file. This is the update generated when the administrator creates a new user and initializes his personal SEcube device with a unique serial number. This update contains the basic information about the current user (i.e. ID, name, SEcube serial number, etc.), it is generated only once by the administrator and applied only once by each user.

- Recovery update: intermediate priority, it is used to restore the data of the KMS that a user is supposed to have in his SEcube, this may be useful in case of data corruption. This update is generated by the administrator after an explicit request of the user (to be done manually, not implemented in SEkey) or it could be generated automatically by the administrator when he is not able to create a standard update file. Since the generation of a recovery update is done only in case of severe problems, in normal operating conditions this kind of update file is not used.

- Standard update: lower priority, used by the administrator to propagate any change in the data of the KMS to the involved users. Standard update files are the most used, if the KMS is highly dynamic (i.e. a change is made every few minutes) the activity around the standard update files will be intense. In normal working conditions and after the execution of the ‘init‘ update, this is the only type of update file that is used.

The execution order is: first ‘init‘ update (only once), then recovery update (whenever it is necessary), finally standard update. If one of these files cannot be found, SEkey assumes that that update file is not needed, and it goes on with the types which have lower priority.

If the processing of an update file fails, SEkey retries a limited amount of times with the same update file; if the update still fails then SEkey is locked, lower priority updates are not processed, the user privilege to access specific SEkey APIs is revoked. In this situation, the user should manually ask to the administrator to issue a recovery update. The only way to unlock SEkey is executing correctly a normal update that previously failed or a recovery update just released by the administrator. The Figure 4.11 illustrates the flow of the update mechanism as it is performed by the users of SEkey.

4.5.3 Problems related to the use of obsolete data

SEkey is built to be reliable and to keep its data safe from external attackers; however, these properties cannot always be granted if a user has outdated KMS data on his SEcube. Imagine a scenario where two users, Alice and Bob, want to communicate encrypting their messages. A necessary prerequisite is that Alice and Bob have at least one common group with at least one key eligible to be used for encryption; suppose that this condition is met but suppose also that Alice has the latest data of SEkey (thus is perfectly synchronized with the administrator) while Bob has old SEkey data (maybe because he was not able to perform the update of SEkey).
In this situation, it may happen that the key chosen by Alice to encrypt her messages is not the same key chosen by Bob, because Bob is not aligned with the data of the administrator. The result of this possible mismatch is that Bob will not be able to decrypt Alice's messages so the communication will not be possible.
A more dangerous possibility is that a compromised key may be chosen by one of the users involved in the communication. This is the biggest security issue related to the use of outdated SEkey data because it may make the entire encryption process useless.
For this reason, the update policy of SEkey is very strict, blocking the functionalities of SEkey as soon as a possible mismatch with the data managed by the administrator is detected. When the SEkey APIs for a user are blocked, a specific error code is returned. In this case the only operation that can be done is to issue the update again or restart SEkey; notice that technically normal SEkey APIs are still callable but they will check for the updates again and they will keep the KMS blocked in case the update is not completed correctly. The user is also suggested to manually ask to the administrator to issue a recovery update.

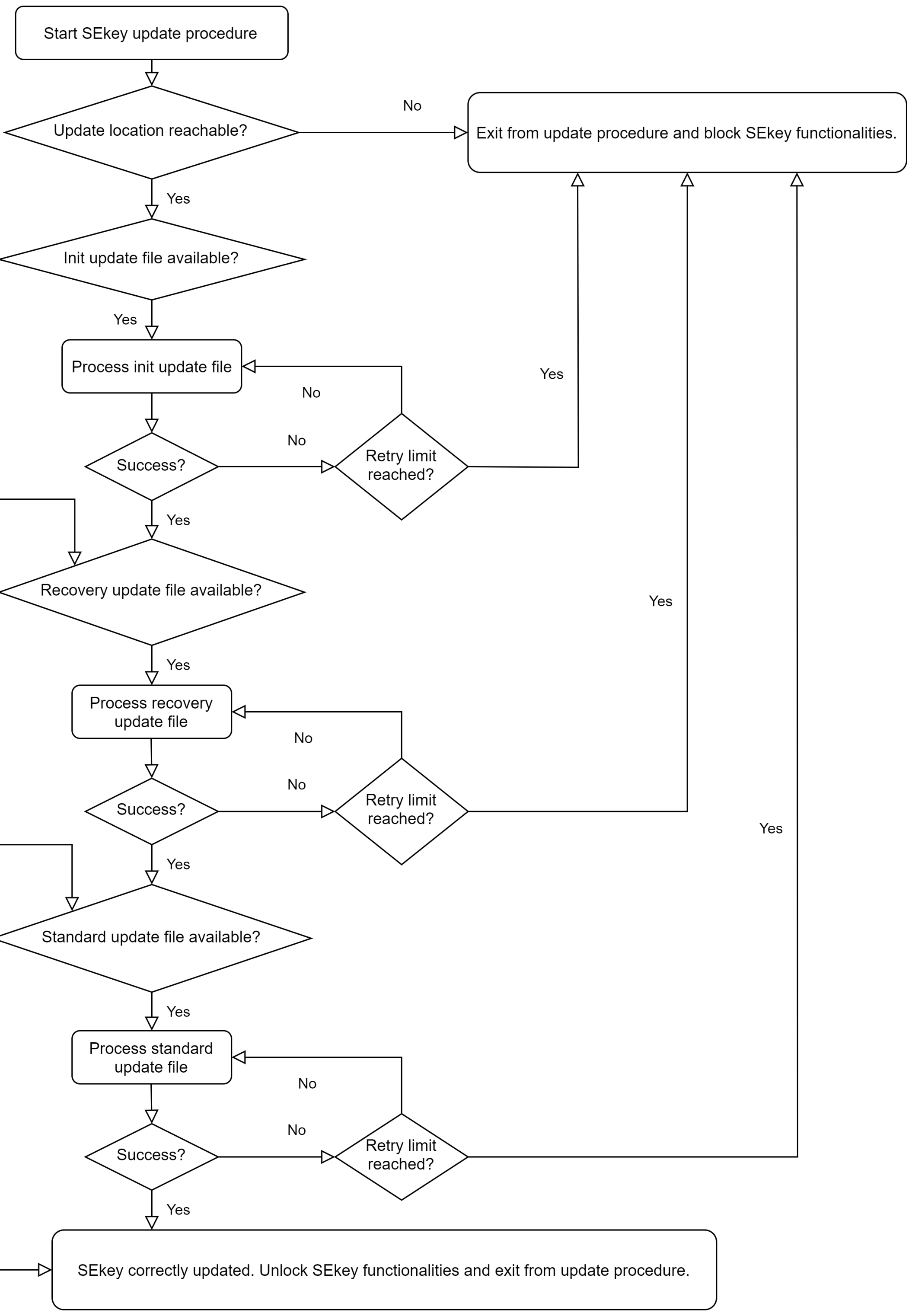
Start SEkey update procedure
Update location reachable?
No
Exit from update procedure and block SEkey functionalities.
Yes
Init update file available?
Yes
Process init update file
No
Success?
No
Retry limit reached?
Yes
No
Yes
Recovery update file available?
Yes
Process recovery update file
No
Success?
No
Retry limit reached?
Yes
No
Yes
Yes
Standard update file available?
Yes
Process standard update file
No
Success?
No
Retry limit reached?
No
Yes
SEkey correctly updated. Unlock SEkey functionalities and exit from update procedure.

Figure 4.11. Flowchart of the update mechanism executed by the users.

Chapter 5

Results and discussion

In this chapter we will discuss the results of this thesis, along with the main features and advantages of the key management system that was implemented. The drawbacks of the adopted solution will be highlighted, then the list of the APIs developed to support the KMS will be presented. Finally, few use cases will be included in order to let the reader understand how the software works.

5.1 Results

The key management system developed in this thesis is based on the SEcube Security Platform. The KMS is designed according to a distributed architecture, where each actor owns a personal SEcube hardware security module and is provided with a portion of the data of the KMS.
The actors operating in the system are either administrator or users; however, the KMS is designed to work with only one administrator. The administrator has access to all the data of the system, he is responsible for managing the KMS and distributing the data to the users. The SEcube of the administrator contains every key and every other data managed by the KMS.
The users can exploit a reduced set of the features of the KMS, as well as a reduced set of the data of the system since they are limited by the 'Least Privilege' principle.
In order to distinguish the roles of the administrator and of the users, SEkey implements two privilege levels that are determined by the login mode of the SEcube. Only the login executed with the 'admin PIN' of the SEcube allows to access to the most advanced features of the key management system.
The keys managed by the KMS are stored in the flash memory of the SEcube, every other data needed by the KMS is written in an encrypted SQL database located on the microSD of the SEcube. This organization is valid both for the SEcube of the administrator and for the SEcube devices of the users.
The distribution of the keys and other metadata from the administrator to the users is automatically executed by the key management system. This procedure is designed to be secure and reliable because every data that is transmitted is always encrypted and authenticated, moreover a data recovery strategy takes place if required.

Thanks to the distributed architecture, SEkey can easily work in an offline scenario, where the user does not need to get in touch with the administrator or any other source, provided that the user is aligned with the latest data of the administrator.
Here is a recap of the main features of the key management system.

- Broad set of APIs to manage and use the KMS.
- Support for the management of keys, users, groups, and security policies.
- Compatibility with Windows and Linux.
- Support for AES-256 (encryption), AES-256 (encryption) + SHA-256 (integrity), AES-256 (encryption) + HMAC (authentication) + SHA-256 (integrity).
- Low-level security issues and security primitives are handled automatically and are completely transparent to the users.
- The security primitives are always performed within the SEcube.
- The keys and the data of the KMS are physically stored in different non-volatile memories (the SEcube flash memory and the SEcube microSD).
- The administrator and the users cannot access to the cleartext value of any key managed by the KMS.
- Every data of the KMS that is stored on the microSD of a SEcube (meaning everything except the key values) is encrypted with AES256-HMAC-SHA256, using a key known only by that SEcube.
- The roles of the users and the administrator are compliant to the ‘Least Privilege‘ principle of the NIST.
- The users can exploit the KMS using their local copy of the data, without constantly interacting with the administrator.
- The distribution of the data of the KMS from the administrator to the users is automatically managed, it is encrypted, it grants integrity and mutual authentication, and it provides an automatic data recovery procedure in case of failure.
- The encryption keys managed by the KMS are generated exclusively inside the SEcube of the administrator using a TRNG.
- Native compatibility with the SEfile encryption library for the SEcube.
- Even though the changes applied to the KMS are not immediately system-wide, because they need to be propagated to the users by means of the update files, the users are forced to fetch the latest system-wide data distributed by the administrator before they can execute any SEkey API.
- Embedded audit and accountability system, protected with SEfile to grant confidentiality, integrity and authentication.
- If the SEcube of a user is compromised, the integrity of the data of the KMS is preserved because the authoritative source of the data is the administrator.

5.2 Drawbacks

Here are the drawbacks of the proposed solution.

- Every user is required to possess a SEcube HSM, making the system potentially expensive.
- The SEcube hardware security module limits the flexibility and the scalability of the system, mainly because the hardware imposes strict limitations on the maximum number of keys that can be managed.
- The current SEcube firmware is not optimized to support the KMS, but a new version is under development.
- The free SEcube software stack only supports AES-256 for encryption along with HMAC and SHA-256 for authentication and integrity. There is no support for asymmetric cryptography.
- The developer who wants to integrate SEkey in his application must provide a location to SEkey where to store and fetch the update files (i.e. a shared folder, a server, etc.).
- The administrator is a single point to attack because he has read/write access to all the data of the system (i.e. keys, users, groups, etc.).
- If an attacker violates the host to which the SEcube is connected, the HSM is not able to detect the violation and some data may be stolen; for this reason the host to which the SEcube is connected must be as secure as possible, possibly implementing specific features are requested by the NIST [5, p. 70, 71].
- It is not a KMS-as-a-Service or a plug-and-play solution; it requires the development of applications designed to work with the SEcube.
- Due to the distributed architecture, there is not any key usage count mechanism.
- SEkey does not provide any ‘Disaster&Recovery‘ utility to backup and restore the keys and the metadata of the KMS stored in the SEcube of the administrator.

Notice that some of the drawbacks of the KMS do not constitute real problems because they are the consequence of a design choice, trade-offs are inevitable. For example, deciding to use a hardware security module to implement the system limits the flexibility and scalability of the solution but, at the same time, it allows to pursue the highest security standards recommended by the NIST.

5.2.1 Exposure of keys outside of the SEcube

The best way to implement the distribution of the keys managed by the KMS would be to avoid sending their entire value, resorting instead to a Key Derivation Function. On the SEcube this is not feasible for several reasons:

- the SEcube firmware does not support any key agreement algorithm;

- the SEcube firmware does not support asymmetric cryptography;
- deriving a key shared by multiple users exploiting a KDF requires a shared secret among all users, this is not feasible since the users of SEkey can be organized in countless groups.

Because of these limitations, sending the actual key values to the users is inevitable. However, when a key leaves the SEcube device of the administrator, it is already wrapped with another key thanks to a feature implemented for SEkey in the SEcube firmware.
Notice that the firmware functionality that allows to expose a wrapped key outside of the SEcube is available only when the login is executed with the admin PIN. This feature is available only to the administrator because the users do not know the admin PIN of their SEcube devices. This approach is compliant with the NIST guidelines [5, p. 58], [9, p. 108].
The exposure of a wrapped key outside of the SEcube implies a vulnerability related to the sub-optimal implementation of the SEcube firmware. When a key is returned to the host, already wrapped with another key, the wrapping key is not established by the SEcube but is decided by the host. If the wrapped key is then sent to the SEcube as parameter of a low-level API to decrypt a payload, specifying the same key wrapping key as input of the crypto algorithm, then the cleartext of the key can be obtained by the host of the administrator.
This vulnerability will be addressed in future releases of SEkey, the optimized firmware may implement several limitations to the use of low-level features in order to achieve stronger security.
This is not the only problem related to the SEcube. In order to encrypt the update files and to wrap the keys as explained in Section 4.5.1, the administrator shares a different pair of symmetric keys with every user. These keys work as pre-shared secrets, which are established during the physical initialization of the SEcube of every user.
Given the SEcube of the administrator and a brand-new SEcube of a user that must be initialized, they do not share any pre-existing secret, and they do not have any mechanism to generate a shared secret without exposing it (because there are not key agreement algorithms available).
Therefore, the only way to create a shared secret among two SEcube devices is to manually inject it in both of them, this clearly implies that the shared secret is exposed as cleartext at least for some time during the injection process. This kind of situation is critical in terms of security [9, p. 107, 108], nevertheless a manual approach is the only feasible solution, even according to the NIST recommendation [3, p. 17].
In the case of SEkey, the injection is implemented generating the shared secret on the SEcube of the administrator, then the secret is retrieved in an encrypted form from the flash memory of the SEcube. It must be injected as cleartext, so the encrypted secret is decrypted by the administrator (this is feasible because is the administrator that tells to the SEcube which key to use for encryption) and it can be sent to the SEcube of the user. This process must be done on the host machine of the administrator, which must be a trusted third-party.

5.2.2 Replacement of private SEcube keys and pre-shared keys

SEkey is mainly based on three elements: the Secure Database, the keys stored in the flash memory of the SEcube, and the update files used to carry the information from the administrator to the users. The Secure Database is encrypted with a key which is stored only in the memory of that particular SEcube; similarly, the update files are encrypted with two keys that are unique for each pair administrator, user (to understand why two keys are needed, please refer to Section 4.5.1).
In the case of the Secure Database, the key is generated independently by every SEcube during the device initialization. In the case of the keys for the updates, each pair of keys is manually injected by the administrator in the SEcube devices because that is the only way to store a shared secret in multiple HSMs (refer to Section 5.2.1 for more details).
Due to the nature of these keys, replacing them is extremely costly and dangerous, as mentioned also by the NIST [9, p. 40].
In particular, replacing the key of the Secure Database would require to create another database file where the information stored in the original one would be moved. This operation exposes the database to data loss and data corruption, moreover it may be time consuming in case of a large database.
The same problem affects the replacement of the pre-shared keys. Their replacement would require to decrypt and encrypt again the update files and also the audit file generated by every user. Moreover, the administrator would need to physically collect all SEcube devices to manually inject new shared keys (sending them as a normal update would not be secure since they would be encrypted with the same keys that they are meant to replace).
Notice that, for these reasons, SEkey does not specify any cryptoperiod for these keys; therefore they can be indefinitely used. At the moment, if the administrator wants to change these keys, he is supposed to manually collect the SEcube devices to generate or inject new keys. This operation is time consuming and expensive, therefore it must be evaluated carefully.

5.3 The SEkey APIs

SEkey does not natively provide any high-level software capable of exploiting the KMS. However, there is a plan to develop a GUI for Windows and Linux in order to use SEkey without having to deal with the development of customized software.
Instead of providing an application already fully developed, SEkey provides a set of APIs, written in C++, that can be used to integrate the KMS within pre-existing software or to create a dedicated application (such as the GUI mentioned above).
Notice that most of the APIs are reserved to the SEkey administrator because they imply changes in the data of the key management system. The APIs available to the users are the ones that do not modify anything inside the KMS, such as the APIs to retrieve the list of the known users and groups, to retrieve the security policy of a group, to find the most secure key to be used in a certain communication and so on.
The SEkey library consists of many APIs, it is quite complex because it requires a lot of additional code that is used by the APIs for internal purposes. The Table 5.1 shows a simplified prototype and explanation of the most significant APIs. Further details are available in the Doxygen documentation that has been written to help everyone who wants

to use SEkey or customize it. The complete source code of the SEkey KMS is included in the SEcube open-source SDK, which can be downloaded from the website of the SEcube manufacturer [6].

API	**Functionality**
sekey_start()	Start the SEkey KMS
sekey_stop()	Stop the SEkey KMS
sekey_admin_init()	Initialize the SEcube of the administrator
sekey_user_init()	Add a user to SEkey
sekey_update_userdata()	Update the KMS of the current user
sekey_user_get_info()	Retrieve info about a user of SEkey
sekey_user_get_info_all()	Retrieve info about all users of SEkey
sekey_group_get_info()	Retrieve info about a group of SEkey
sekey_group_get_info_all()	Retrieve info about all groups of SEkey
sekey_key_get_info()	Retrieve info about a key of SEkey
sekey_key_get_info_all()	Retrieve info about all keys of SEkey
sekey_delete_user()	Delete a user from SEkey
sekey_add_user_group()	Add a user to a group
sekey_delete_user_group()	Delete a user from a group
sekey_add_group()	Add a group to SEkey
sekey_delete_group()	Delete a group from SEkey
sekey_add_key()	Add a key to SEkey
sekey_activate_key()	Activate a key (it becomes usable to encrypt data)
sekey_set_key_state()	Set the state of a key to a new value
sekey_expired_check()	Check if any key is expired
sekey_find_key_v1()	Find the most secure key given a pair of users
sekey_find_key_v2()	Find the most secure key given a user and a group
sekey_find_key_v3()	Find the most secure key given a set of users

Table 5.1. List of the most important SEkey APIs

Notice that the three APIs to retrieve the best encryption key to be used given certain parameters, are by far the most important. These APIs are based on the assumption that the caller needs to encrypt some data that must be sent or shared with somebody else, therefore SEkey must provide to the caller the ID of the key which is considered to be the most secure.
If the caller specifies a group as ‘destination‘, then the most secure key of that group is chosen. If the caller specifies one or more users as ‘destination‘, then the most secure key belonging to the smallest group common to all users is chosen. Notice that, if the caller

specifies himself as ‘destination‘, then the key belonging to the smallest group is chosen; this implies that the administrator should create a group for each user that is allowed to encrypt data only for himself.
The choice of the most secure key to be used is done according to the following rules:

1. the state of the key must be active;
2. if the caller did not specify an entire group as ‘destination‘, then the key must belong to the smallest group that includes all the users involved;
3. if there is more than one key satisfying the previous conditions, then the key related to the strongest crypto algorithm is chosen (currently only AES-256 is available);
4. if all the previous conditions are met by multiple keys, then the key with the shortest cryptoperiod is chosen (the assumption is that a shorter cryptoperiod implies that a key is used less times);
5. if the previous requirements are not enough, then the key with the smallest ID is chosen (because the ID is a number).

For example, let us imagine that Alice and Bob belong to Group 1 and Group 2, where Group 1 is the smallest having only 2 users. Suppose that Group 1 has two active keys that use the same algorithm, but a key has a cryptoperiod of 1 year and the other one has a cryptoperiod of 1 month. The KMS will choose as ‘most secure‘ the active key belonging to Group 1 with a cryptoperiod of 1 month.
Notice that, in order to guarantee higher security, the usage count of a key would be fundamental to evaluate if the key is secure or not. Unfortunately, since SEkey is distributed and these APIs run without the centralized control of the administrator or of a key server, maintaining a system-wide key usage counter is difficult. A local key usage counter would be useless because even if a user does not use a key so often, the same key may be used more frequently by someone else. The implementation of a system-wide key-usage counter is one of the things that must be done in future.

5.4 Partial collection of SEkey use cases

Here are few use cases that are useful to understand how the SEkey APIs can be exploited to build an application dedicated to the KMS or to embed the KMS in a more complex software. Notice that these use cases are intentionally simplified, more details about how to use the APIs can be found in the Doxygen documentation included in the SEcube open-source SDK [6].

5.4.1 Initialization of the SEcube of the administrator

Executed only by the administrator.
Precondition: the SEcube of the administrator has not been initialized yet.

1. Connect the SEcube to the host.
2. Login to the SEcube with the default administrator PIN.
3. Call the sekey_admin_init() API, providing the new admin PIN and the new user PIN to be set.
4. Disconnect and reconnect the SEcube.
5. Login to the SEcube to check if the admin PIN was set correctly.
6. Logout from the SEcube.
7. Disconnect the SEcube from the host.

5.4.2 Initialization of the SEcube of the user

Executed only by the administrator.

1. Connect the admin SEcube to the host.
2. Login to the SEcube with the administrator PIN.
3. Call the sekey_start() API.
4. Call the sekey_init_user_SEcube_stepA() API, providing the required data.
5. Call the sekey_stop() API.
6. Disconnect the admin SEcube from the host.
7. Connect the user SEcube to the host.
8. Call the sekey_init_user_SEcube_stepB() API, providing the required data.
9. Disconnect the user SEcube from the host.
10. Connect the admin SEcube to the host.
11. Login to the SEcube with the administrator PIN.

12. Call the sekey_init_user_SEcube_stepC() API, providing the required data.
13. Call the sekey_stop() API.
14. Logout from the SEcube.
15. Disconnect the SEcube from the host.

5.4.3 Start SEkey, use SEkey, stop SEkey (admin-side)

Executed only by the administrator.

1. Connect the SEcube to the host.
2. Login to the SEcube with the administrator PIN.
3. Call the sekey_start() API.
4. Perform the required operations on the KMS using the available APIs.
5. Call the sekey_stop() API.
6. Logout from the SEcube.
7. Disconnect the SEcube from the host.

5.4.4 Start SEkey, use SEkey, stop SEkey (user-side)

Executed by the users.
Precondition: the SEcube of the user is already initialized.

1. Connect the SEcube to the host.
2. Login to the SEcube with the user PIN.
3. Call the sekey_start() API.
4. Perform the required operations on the KMS using the available APIs.
5. Call the sekey_stop() API.
6. Logout from the SEcube.
7. Disconnect the SEcube from the host.

5.4.5 Adding a new key to the KMS

Executed only by the administrator.
Preconditions: SEcube connected to the host, admin logged in the SEcube, SEkey started.
Postcondition: the key is successfully added to the KMS.

1. Decide to which group the new key should belong.
2. Decide the ID and the name of the new key.
3. Call the sekey_add_key() API, passing to the function the ID, the name and the group of the new key.
4. The key value will be generated automatically by the SEcube, without letting the host know anything about it. The other key metadata will be stored in the Secure Database.
5. Check the return value to be sure that the key was added correctly.

5.4.6 Adding a new group to the KMS

Executed only by the administrator.
Preconditions: SEcube connected to the host, admin logged in the SEcube, SEkey started.
Postcondition: the group is successfully added to the KMS.

1. Decide the attributes of the group: ID, name, security policy.
2. Call the sekey_add_group() API, passing to the function the attributes established in the previous step.
3. The group is added to the Secure Database, it is empty, and it does not have any key. The SEcube device is not involved.
4. Check the return value to be sure that the group was added correctly.

5.4.7 Adding a new user to the KMS

Executed only by the administrator.
Preconditions: SEcube connected to the host, admin logged in the SEcube, SEkey started.
Postcondition: the user is successfully added to the KMS.

1. Decide the ID and the name of the user to be added.
2. Call the sekey_user_init() API, passing to the function the attributes established in the previous step.
3. The user is added to the Secure Database, by default the user does not belong to any group. The SEcube of the administrator automatically generates a pair of symmetric keys to protect the updates for the new user. The SEcube of the user is not involved and can be initialized later.
4. Check the return value to be sure that the user was added correctly.

5.4.8 Adding a user to a group

Executed only by the administrator.
Preconditions: SEcube connected to the host, admin logged in the SEcube, SEkey started.
Postcondition: the user is successfully added to the group.

1. Decide which user should be added to which group.
2. Call the sekey_add_user_group() API, passing to the function the ID of the user and the ID of the group.
3. The user is added to the group inside the Secure Database, the updates for the users involved by this change are automatically generated.
4. Check the return value to be sure that the user was added correctly to the group.

5.4.9 Retrieve a key from the KMS

Executed by the users.
Preconditions: SEcube connected to the host, user logged in the SEcube, SEkey started.
Postcondition: the KMS has given to the user the ID of the most secure key to be used.

1. Decide the 'perimeter' of the encryption (i.e. the encrypted data must be visible only by the same user who encrypt them, they must be visible also to another user or to another set of users, etc.).
2. Call the required sekey_find_key() API.
3. The KMS will search for the best key to be used (the most secure key) and will return its ID.
4. Check the return value to be sure that the key is valid.

Chapter 6

Conclusions

This thesis aimed at developing SEkey, a hardware-based key management system, leveraging the features offered by the SEcube Security Platform. This platform consists of the SEcube hardware security module and of a set of software libraries.
The need of a key management system for the SEcube platform arises from the lack of pre-existing support of even basic key management policies. The capability of properly managing cryptographic keys is critical in modern scenarios because a manual approach is not feasible. SEkey addresses this specific issue, exploiting an architecture based on the SEcube hardware security module and on a set of APIs developed to use the KMS.
The key management system is focused on distributed environments where every actor is provided with a dedicated SEcube.
SEkey allows an easy management of cryptographic keys, at the same time it is secure and reliable. The NIST guidelines for key management have been followed in order to be compliant to the best security standards. For example, the 'Least Privilege' and the 'Separation of Duties' principles were adopted, designing an architecture where the roles of the administrator and of the users are clearly separate.
However, pre-existing limitations imposed by the SEcube firmware implementation were difficult to circumvent, therefore the firmware was modified adding new features. Anyway, significant improvements are still needed to overcome some limitation that undermines the scalability and efficiency of the key management system.
The initial target of SEkey has been fully achieved, the KMS in fact is able to manage the life cycle of cryptographic keys as well as security configurations. Moreover, SEkey allows to handle many users, dividing them in groups with different security policies, enforcing also rules to restrict the access to the keys and to the features of the system. Another great advantage is that SEkey can work offline, without resorting to a key server, because the data are automatically distributed to the users thanks to a custom secure data distribution protocol. With SEkey, the SEcube platform can provide security features to any environment, because the SEcube software ecosystem has been improved in order to be easily integrated within third-party software as well as in dedicated applications.
The key management system is not the only result achieved in this thesis; in fact the Secure Database is a standalone library which can be easily integrated in any other software. The Secure Database library is based on the union of the SEfile library with the SQLite database engine, it was developed to safely store the metadata of SEkey.

In conclusion, thanks to SEkey, the SEcube offers a more complete software ecosystem. With SEkey, any developer can build an environment where multiple SEcube devices provide security features to any personal computer or server, managing at the same time many cryptographic keys in a smart and easy way, according to high-security standards. The users do not need to worry about low-level security details; at the same time the security administrator can move from ‘how‘ to ‘who and what‘, focusing on who is allowed to access to data, who can access to certain functionalities, which data are involved, which operations are performed.

6.1 Future work

6.1.1 SEcube firmware optimized for SEkey

The current firmware of the SEcube is not optimized for SEkey; it has many limitations, the biggest one being the inefficient use of the flash memory. Since SEkey has been developed as a set of host-side APIs, the APIs are still exploiting the original firmware. Consequently, the potential of SEkey is undermined by software limitations on the SEcube side. The target of the optimized firmware is to remove these limitations, targeting optimal flash memory usage, flexibility, scalability, and reliability of the low-level SEcube software.
Another important goal of the optimized firmware is to protect the keys stored in the flash memory of the SEcube exploiting encryption, clearly this would require to store the cleartext of the original key somewhere. In addition to confidentiality, also the integrity of the keys must be granted. The optimized firmware is currently under development, it is expected to be released later in 2020.

6.1.2 SEkey GUI

SEkey does not natively provide any ‘ready to use‘ software for the KMS. What SEkey provides is a set of APIs which can be used to control the system. A software with a GUI dedicated to the use of the key management system is currently under development, it will allow the administrator and the users to easily work with the KMS.

6.1.3 Split knowledge and dual control

The architecture of SEkey is based on a single administrator who oversees the management of the entire KMS. A better approach requires that nobody has access to all the data of the system. This goal may be pursued splitting the role of the administrator in two: a first administrator that focuses his actions on the metadata and a second one that focuses his work on the keys. In this way nobody would have access to all the data of the KMS. A further optimization would include two administrators that work on the keys, the first administrator that works on the first half of the key and the second administrator that works on the second half.

6.1.4 Centralized KMS

SEkey is focused on distributed environments; however, in many cases a centralized approach is simpler and efficient. In a centralized KMS there is not any need of distributing the data to the users because everything is done replying to specific requests of the users. Every data of the KMS would stay in the SEcube of the administrator; the users would send authenticated requests to the administrator in order to retrieve the required data, such as an encryption key. Particular attention would be required in protecting the communication between the users and the administrator, adopting strong encryption and authentication while guaranteeing the integrity of the data.

6.1.5 Upgrade the SQLite library to the latest version

SEkey heavily relies on SQLite in order to implement the secure database. Even though the current implementation is working and is secure, it is sub-optimal because it is based on SQLite 3.19.3, which was released on 2017-06-08. Due to a bug still to be solved, SEfile is not compatible with the latest release of SQLite; the cause of the bug is likely hidden in the custom virtual file system used by SQLite. Solving this problem would allow to update the SQLite library to its latest version, making the system more secure and reliable.

6.1.6 SEcube smartcard integration in SEkey

The SEcube board embeds a smartcard with interesting cryptographic capabilities, especially considering asymmetric encryption and certificates. At the moment, the smartcard is not used by SEkey or any other SEcube related software because it is not even supported by the SEcube firmware. The development of a smartcard applet is an important suggestion for future work. The advantage of this integration would be the ability to use asymmetric cryptography algorithms in order to support new features like key agreement protocols.

6.1.7 A secure operating system

The SEcube is based on a firmware that lacks many features, for example a file system interface to manage the files on the microSD. The idea is to replace the firmware with a real-time operating system oriented to security. Some open-source operating system with the required features is already available; the goal is to integrate the features of the firmware into the secure operating system, possibly implementing new functions in order to enhance the potential of the SEcube. The adoption of an operating system on the HSM would make the SEcube platform simpler, more efficient, and capable of tackling many different issues.

Bibliography

[1] Varriale Antonio, Prinetto Paolo, Carelli Alberto, and Trotta Pascal. *SEcube™: Data at Rest and Data in Motion Protection.* URL: https://www.secube.eu/site/assets/files/1137/02_-_sam16_-_data_at_rest_and_data_in_motion_protection.pdf.

[2] Varriale Antonio, Prinetto Paolo, Maunero Nicolò, and Roascio Gianluca. *The SEcube™ Open Security Platform.* URL: https://www.secube.eu/site/assets/files/1152/secube_sdk_v1_4_1_wiki_rel_009.pdf.

[3] 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.

[4] E. Barker and A. Roginsky. *Recommendation for Cryptographic Key Generation.* National Institute of Standards and Technology, July 2019. DOI: 10.6028/NIST.SP.800-133r1. URL: https://doi.org/10.6028/NIST.SP.800-133r1.

[5] E. Barker, M. Smid, D. Branstad, and S. Chokhani. *A Framework for Designing Cryptographic Key Management Systems.* National Institute of Standards and Technology, Aug. 2013. DOI: 10.6028/nist.sp.800-130. URL: https://doi.org/10.6028/nist.sp.800-130.

[6] Blu5 View Pte. Ltd. *SEcube Open Source SDK.* URL: https://www.secube.eu/resources/open-sources-sdk/.

[7] *Most Widely Deployed and Used Database Engine.* URL: https://www.sqlite.org/mostdeployed.html.

[8] National Institute of Standards and Technology. *CMVP Approved Authentication Mechanisms.* URL: https://csrc.nist.gov/publications/detail/sp/800-140e/draft.

[9] 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/draft.

[10] M. Nieles, 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://doi.org/10.6028/nist.sp.800-12r1.

[11] *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.

[12] *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.

[13] *Selecting The Right Key Management System*. URL: www.cryptomathic.com/hubfs/Documents/White_Papers/Cryptomathic_White_Paper_-_Selecting_The_Right_Key_Management_System.pdf.

[14] *SQLite Home Page*. URL: https://www.sqlite.org/index.html.

[15] Townsend Security. *The Definitive Guide To Encryption Key Management Fundamentals*. 2016. URL: https://info.townsendsecurity.com/definitive-guide-to-encryption-key-management-fundamentals.

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