TU WIEN Informatics

Towards a Scalable Secure Element Cluster

A Recommendation on Hardware Configuration

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Timo Hinterleitner, BSc
Matrikelnummer 01425998

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.-Prof. Dipl.-Ing. Mag. Dr.techn. Edgar Weippl
Mitwirkung: Dipl.-Ing. Dr.techn. Georg Merzdovnik
Dipl.-Ing. Wilfried Mayer
Dipl.-Ing. Dr.techn. Thomas Rössler

Wien, 12. April 2020

Timo Hinterleitner | Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Towards a Scalable Cluster of Secure Elements

A Recommendation on Hardware Configuration

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Timo Hinterleitner, BSc
Registration Number 01425998

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof. Dipl.-Ing. Mag. Dr.techn. Edgar Weippl
Assistance: Dipl.-Ing. Dr.techn. Georg Merzdovnik
Dipl.-Ing. Wilfried Mayer
Dipl.-Ing. Dr.techn. Thomas Rössler

Vienna, 12th April, 2020

Timo Hinterleitner | Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Timo Hinterleitner, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 12. April 2020

Timo Hinterleitner

Acknowledgements

Without the help of many kind people, this thesis would not have been possible.

First of all, I want to thank Thomas Rössler for giving me the chance to participate in the ESPRESSO research project, introducing me to the involved parties, and being of great help whenever I had questions. The test hardware was provided by Martin Bonato, Alexander Chaloupka, and Tim Kovrigar, working at PrimeSign GmbH and CRYPTAS it-Security GmbH, respectively. I greatly appreciate their support.

I also want to give special thanks to Helmut Hammer and Thomas Popp from Yagoba GmbH, who developed the FPGA-based prototype. It allowed me to gather better test results in this thesis.

I would like to express my great gratitude to Edgar Weippl, Georg Merzdovnik, and Wilfried Mayer for supporting my thesis and giving valuable suggestions and feedback.

Further, I want to acknowledge the help of my friends and family who supported me throughout all the years at Vienna University of Technology.

Kurzfassung

In vielen Rechenzentren und IoT-Anwendungen sind die sichere Speicherung von Schlüsselmaterial und deren hardwarebeschleunigte Verarbeitung eine Grundanforderung. Derzeit werden dafür meist Smartcards oder Secure Elements für kleine Anwendungen, oder Hardware Security Modules für Enterpriseapplikationen verwendet. Für Einsatzzwecke mit Anforderungen zwischen diesen beiden Extremen ist die verfügbare Hardware daher zu unter- oder überdimensioniert und damit unwirtschaftlich.

Diese Diplomarbeit stellt einen neuen, skalierbaren Ansatz zur sicheren Speicherung und Verarbeitung von Schlüsselmaterial vor. Dafür wird ein Gerät auf Basis von geclusterten Secure Elements entwickelt, das je nach Anforderungen an Performance, Langlebigkeit, Lastverteilung, Partitionierung des Schlüsselmaterial und Kosten dimensioniert werden kann. Damit ist für jede Art von Anwendungen ein optimales Kosten-Nutzen-Verhältnis gegeben. Nachdem die Architektur und Funktionsweise des Secure Element Clusters vorgestellt wurde, werden zwei verschiedene Prototypen mitsamt deren Software- und Hardwarekonfiguration beschrieben. Eine neu entwickelte PKCS #11 Library kapselt die Cluster-Funktionalität und bietet damit optimale Kompatibilität mit bestehenden Softwarelösungen. Im Gegensatz zu bereits existierenden Secure Element Grids müssen die Anwendungen keine Verwaltungsaufgaben des Clusters übernehmen.

Die Eigenschaften der Cluster-Prototypen werden genau analysiert, um die Clusterarchitektur in weiteren Entwicklungsphasen zu verbessern. Basierend auf Geschwindigkeits- und Langlebigkeitsmessungen, die während dieser Analysephase durchgeführt werden, wird eine mathematische Formel zur optimalen Dimensionierung eines Secure Element Clusters entwickelt.

Abstract

Hardware protected storage of key material and secure processing of cryptographic operations are required in data centers as well as IoT applications in the field. Currently, the available hardware satisfies this demand only poorly. Small-scale applications use a smart card or secure element to satisfy their needs. Large-scale enterprise deployments make use of specially designed Hardware Security Modules. These two options provide only a minimal choice and offer no solution for demands between those configurations. The possibilities are either too weak or too large-scaled. Therefore, the existing solutions are unsuitable for medium-sized use cases.

This paper describes a new, scalable approach for storing key material securely and performing cryptographic operations during changing demands. The solution introduces a device based on clustered secure elements to provide configuration options for performance, longevity, load distribution, partitioning, and costs. After describing the overall cluster architecture, the thesis presents two prototype builds with their complete hardware and software stack. All cluster functionality of the prototypes is encapsulated in a newly developed PKCS #11 library, providing far better compatibility with software solutions than existing secure element grids. The properties of the prototypes are studied in detail to improve the final cluster design. Based on performance and durability analyses of the prototype, the thesis introduces a scaling scheme for determining the optimal cluster configuration for given load requirements.

Contents

CHAPTER 1

Introduction

The need for secure key storage and fast processing of cryptographic operations has been present for decades. In the past, key material was often stored unprotected alongside other application data for various reasons. Hardware protected key storage was not available, too expensive, unintuitive to use, or the developers were just not aware of the risks. Nowadays, different kinds of protected key stores are available to serve different use cases. Secure Elements (SEs) are implemented in embedded devices and smartphones to protect key material and offload cryptographic operations to a dedicated processor. Smart cards and tokens are popular for portable use cases like credit cards, health insurance cards, and corporate identity cards. General-purpose Hardware Security Modules (HSMs) are used in data centers, as PCI Express (PCI-e) cards or standalone network-attached appliances, to serve high availability and high volume needs. Some areas like payment or critical infrastructure services have unique requirements, and therefore special-purpose HSMs were developed as in [1].

While general-purpose HSMs can store a large number of key objects as well as perform cryptographic operations in fractions of a second, they come with the downside of high costs. General-purpose HSMs are commonly used for managing the key material of a public key infrastructure [2], for electronically signing contracts, or for TLS offloading. Tiny and cheap SE chips [3] only provide limited speed, storage space, and lifetime. The underlying storage technology (most often flash or EEPROM) mainly limits the lifetime of SEs.

Although different kinds of protected hardware are available, the OWASP Top 10 of 2017 [4] still ranks "Sensitive Data Exposure," which includes key material exposure, in third place. It may be a minor fraction, but some problematic implementations arise from requirements that still cannot be satisfied by existing hardware. An example of such a case is a deployment that requires more performance than a single SE, token or smart card can deliver, but its constraints on size and energy consumption make the use of a

typical HSM infeasible. In other deployments, the performance of an SE is sufficient, but its expected lifetime is not.

These drawbacks motivate the introduction of scalable SEs. A scalable approach combines the advantages of general-purpose HSMs with those of SEs by providing a configurable trade-off between performance, lifetime, and cost efficiency. The main idea is combining multiple SEs in a cluster, allowing improvement of cryptographic operation throughput, enhancing the storage space, and extending the lifetime of chips. These performance parameters can be tuned separately to meet the required performance demands while optimizing costs. A possible architectural design of such a cluster is described in Chapter 4. The resulting configuration is small in size and does not necessarily need to be operated in a data center. Especially deployments in the field benefit from this approach.

The improved properties allow for new applications. For example, the resulting device can be integrated into roadside assistance systems for autonomous vehicles or even inside vehicles as outlined in [5]. These systems are designed to be long-lasting and require excellent cryptographic performance while enforcing limits on energy consumption, cooling, size, and costs - which existing general-purpose HSMs or a single SE cannot achieve.

An essential feature of the cluster is the abstraction of single SEs, distinguishing it from existing grids of SEs [6]. A grid of SEs adds multiple SEs to the device, where each of them can be used for different purposes or the same. The application logic has to manage each device individually, including initialization, key generation, cryptographic operation scheduling, and keeping track of which SEs are used and their state. While it follows the key separation principle by storing keys of different applications in separated hardware environments, it comes with significant implementation overheads. If the separation of keys is no requirement, as it is in many Internet of Things (IoT) devices specifically designed for a particular task, individually managing all SEs turns into a disadvantage. Using a grid for load balancing of cryptographic operations requires to synchronize key material among SEs and scheduling of the operations within the application.

An SE cluster transfers this complexity into an intermediate software or hardware layer. The cluster can be used as a single security module while consisting of multiple building blocks. The number of building blocks can be chosen according to the requirements and may even vary over time. When requiring more cryptographic operation throughput, the number of building blocks, meaning SEs, can be increased and the load is distributed to a higher number of cryptographic processors. If the lifetime of an SE needs to be extended, adding additional SEs can reduce the load on each element and thereby achieving the desired result. The solution stays cost-effective as it can be scaled, in theory, precisely to the performance levels required. Scalability in practice is examined in Chapter 6. The complexity of synchronization and cryptographic operation scheduling is fully encapsulated and not visible from the outside. Failing elements can degrade the performance of the cluster, but as long as the key material is available on another SE, the cluster stays functional. When replacing failed elements or adding additional elements

to the cluster, the keys are synchronized automatically and performance is recovered or improved further.

The following chapters describe a potential SE cluster and evaluate its performance. In Chapter 2, a short overview of technologies used in the later chapters is given, including more detailed definitions of HSM and SE. Chapter 3 presents the state of the art and provides a synopsis of existing publications on the topic of SE clusters. The following Chapter 4 describes a possible cluster architecture based on SEs or smart cards. It outlines how the single cryptographic elements are connected and managed as well as how its key material is generated and used. In Chapter 5, the cluster setup is implemented in a software library and described in detail, allowing an easy rebuild of the configuration.

Chapter 6 evaluates the characteristics and pitfalls of the cluster and compares the properties of two different proof-of-concept implementations. These characteristic properties include cryptographic performance and durability, among other key performance indicators. The recommendation scheme found in this chapter provides a suggested cluster configuration. Based on a formula, the optimum number of SEs is calculated to match a given performance level or durability requirements. Example calculations for known use cases based on the previous measurements are included. Finally, Chapter 7 discusses the findings and gives an outlook on possible future research opportunities and open questions. A summary of the arguments brought forth, as well as the findings of this work, conclude this thesis.

CHAPTER 2

Background

The following chapter gives a short overview of the used technologies in this thesis. By highlighting the most important characteristics of different hardware devices, software products, and standards, it is the author's goal to make this thesis accessible also for readers without experience in the field of secure hardware and use thereof. A basic understanding of computer technology is assumed.

Furthermore, this chapter describes essential features of hardware security modules, characteristics and communication protocols of smart cards, differences between smart cards and SEs, the Java Card specification, card management using GlobalPlatform, and the PKCS #11 standard.

2.1 Hardware Security Modules

An HSM is a tamper-resistant hardware device specifically designed for secure key storage and cryptographic operations [7, p.384]. Their protection mechanisms allow secure key usage also in hostile environments. While "Hardware Security Module" is the most common name of this device group, one can also find the terms Tamper-Resistant Security Module (TRSM) and Network Security Processor (NSP) used within the field [8].

The required properties of HSMs are very similar across all form factors. Some definitions of the term HSM also apply to cryptographic USB tokens, smart cards, and SEs as they achieve similar functionality. In this thesis, the term HSM is explicitly distinguished from tokens, smart cards, and SEs because of drastically different costs, processing capabilities, and use cases. The distinction allows to easily refer to certain device classes, without one class being a subgroup of the other. HSMs in the sense of this thesis include network-attached appliances, PCI-e cards, and USB-attached appliances as long as they provide similar performance. Examples for these device groups are the SafeNet Luna Network HSM, the Safenet Luna PCIe HSM, and the SafeNet USB HSM. However, the

SafeNet USB HSM provides significantly fewer signatures per second. Other vendors like nCipher and Utimaco provide similar products.

HSMs are used in a wide variety of use cases, ranging from Public Key Infrastructures (PKIs) and health record protection over TLS offloading to payment service processing. Therefore, HSMs have hardware accelerated and hardware protected mechanisms for [8]

- random number generation
- symmetric and asymmetric key generation and derivation
- protected key storage of symmetric keys and private keys
- signing data
- encrypting and decrypting data

The supported algorithms vary between different devices, but the most common key types, including 3-DES, AES, RSA, and ECC, are widely supported.

2.1.1 Physical Security of HSMs

HSMs are certified for different specifications and implement different standards because they are used in many security-critical areas, and auditing the device and verifying its software is not possible for every company. The most commonly found certification is for Federal Information Processing Standard (FIPS) 140-2, defined by the National Institute of Standards and Technology (NIST). The FIPS 140-2 defines four levels of security which a device can implement. Table 2.1 outlines these levels.

The CC standard is defined in the publicly available ISO/IEC 15408. According to the assurance levels, a certain extent of documentation must be available and the product must be tested differently [10, p.15ff]. The following list gives an overview of the defined assurances levels:

- EAL1: functionally tested
- EAL2: structurally tested
- EAL3: methodically tested and checked
- EAL4: methodically designed, tested, and reviewed
- EAL5: semiformally designed and tested
- EAL6: semiformally verified design and tested
- EAL7: formally verified design and tested

Level	Description
1	Level 1 is the lowest level of security. It provides basic cryptographic functions implemented in software or hardware.
2	Hardware implementations must use tamper-evident seals and pick-resistant locks. Role-based authentication is required. Code for cryptographic operations can be executed on general-purpose CPUs as long as the operating system meets Common Criteria (CC) Protection Profile (PP) requirements and is evaluated to at least CC EAL2.
3	In addition to tamper-evidence, level 3 requires tamper-resistance for the Critical Security Parameters (CSPs). Identity-based authentication is required. Code for cryptographic operations can be executed on general-purpose CPUs as long as the operating system meets CC PPs, Trusted Path requirements, and is evaluated to at least CC EAL3.
4	Level 4 is the highest level of security. All illegal physical access attempts to the cryptographic module are detected and handled. Level 4 devices can be used in physically insecure environments. The operating system must meet all requirements of level 3 and must be evaluated at least at CC EAL4.

Table 2.1: Definition of the four FIPS 140-2 security levels [9]

.

The previously used terms tamper-evidence and tamper-resistance are defined in ISO/IEC 19790 and referenced in ISO/IEC 30104 [11].

A device is tamper-evident if it protects against substitution with a forged or compromised device. Additionally, the device shall be constructed in a way that any successful attack leads to physical damage to the device or requires it to be removed from its authorized location [8].

A device is tamper-resistant if its design protects the content while in the original location as well as when moved to another location with specialized attack equipment. Any modification of the content shall not be possible without specialized equipment and involves damaging the device in a way such it is no longer operable. The monitoring of the device shall be made impossible by shielding it against (electromagnetic) emissions. The device must be able to detect tamper events regardless of the power state [8].

Attacks can be of different types, including

- physical tampering by opening the device,
- under- and overvoltage,
- low and high temperature,
- electromagnetic emission and injection, and
- power, timing, and other side-channel attacks.

The components of HSMs are structured in several groups to make their implementation tamper-resistant. Most importantly, they are distinguished into a tamper-resistant core and its non-critical surroundings. The tamper-resistant core contains all sensitive data and performs sensitive operations. It is protected against the attacks mentioned above. Typically, a battery-backed memory holds the master key and is zeroized immediately when an attack is detected. However, the attack must not erase the security configuration, e.g., authorized users, according to ANSI X9.24-1.

Attack detection can be physical (e.g., opening the device releases a button), optical (e.g., a light sensor detects the opening of the device), or electrical (e.g., a circuit in wire-mesh must be interrupted to gain access). Various types of sensors (temperature, voltage, current, electromagnetic, tilt) detect changes in the surrounding. While each type of protection can be circumvented with some effort, the combination of all mechanisms ideally renders attacks infeasible [7, p.388ff].

2.1.2 HSM and Key Management

Many tasks for managing the HSM require manual user interaction and authentication. Authentication can be performed locally, i.e., with physical access to the device, or remotely over the network. As HSMs are installed in data center racks and high-secure areas, remote access is used where possible. Remote access offers several advantages, making the process of managing the HSM faster and cheaper.

As it is best practice to implement roles with separated duties, performing tasks on the HSM involves multiple persons. It is a complex process to bring these individuals, who often are in a higher management position, together at the same time. When physical access is required, additionally, these persons must be in the same location. In the case of outages of the HSM infrastructure, a remote access possibility can reduce the resulting downtime [7, p.393ff]. Since large enterprises are required to have their systems up and running at all times to prevent significant financial loss, and because they tend to have their personnel spread worldwide, remote access often is their only choice. On the other hand, having remote management enabled increases the attack surface, because additional systems and networks are involved. Many deployments still require remote access and actively depend on it.

Apart from the initial installation and configuration of the HSM, authentication is required for [7, p.393f]

- creating new users and roles,
- modifying user and role permissions,
- key generation and backup,
- configuration and clustering,
- auditing, and
- updating software and firmware.

HSMs manage several different key types with each possibly having a great number of key instances. The main idea is that the key material never leaves the secure core. Some implementations weaken this constraint by allowing the export in an encrypted form. As long as the encryption is performed using a secure algorithm and the decryption key is stored securely, i.e., in the HSM, smart cards, or similar, the security of the exported key can be assumed.

While it should be avoided where possible, some cases require the export of key material in cleartext. This can be the case when switching the HSM vendor and no compatible import/export functionality is available. In these cases, the plain key must be split into two or more components and secured using an appropriate procedure, including physical separation and split knowledge.

HSMs have three main categories of keys [7, p.395f]:

- **Storage keys** are used to encrypt key material at rest. Multiple storage keys can be applied sequentially to separate the protected keys into partitions, e.g., a host master key protects the entire content of the HSM. Additionally, specific partition keys separate the stored keys.
- **Transport keys** are used to encrypt key material during key exchange, e.g., when building a cluster environment, the key material is synchronized between the cluster members using a key encrypting key,
- **Functional keys** are used to perform cryptographic functions and compute values, e.g., card verification key or PIN verification key.

Security policies define management rules for the HSM and its key material. An HSM administrator chooses the available algorithms, PIN length, PIN complexity, the number of key splits, the requirement of a second/additional factor for authentication, key backup and export functionality, key usage, and further details according to the policies [7, p.394].

As briefly mentioned before, key material can either be stored inside the secure core or be encrypted using a storage key and stored outside in a database or filesystem. The first method has advantages in performance and security, as the key is inside the secure core at all times. The disadvantages of this approach include the limited available storage space as well as synchronization to other HSMs and backup of the key material. The SafeNet Luna HSM family typically uses this method. The second method can store the encrypted key material in any available cheap storage without further needs for hardware-enforced protection. When needed, the keys are loaded into the HSM and decrypted using the unwrapping key kept inside the secure core. Loading the keys on use introduces additional latency and dependencies to the external storage. The nCipher nShield HSM family typically uses this method.

2.2 Smart Cards

Smart cards store key material on a small, portable chip card. They are mainly used in applications requiring key material to be removed, relocated, and in physical possession while enforcing tamper-resistant properties similar to HSMs. Therefore, smart cards are commonly used as banking cards, SIM cards in mobile phones, and identity cards. Storing the personal key material for bank transactions on a secure, portable device allows withdrawing money from foreign ATMs as well as securing no other person has access to the key material and bank account [7, p.3ff].

While different types of smart cards exist, starting with simple ID cards and memory cards, cryptographic applications require secured memory cards or secured microcontroller cards. Secured microcontroller cards offer secure data storage together with a programmable processor. The ability to deploy so-called applets to the card makes them incredibly versatile[7, p.18]. The Java Card specification provides programmers with a programming language and class library specifically designed for smart card applets. As with traditional Java programs, Java Card applets run within a virtual machine. The Java Card VM abstracts the underlying hardware allowing the same Java Card applet to run on smart cards with different specifications. Section 2.4 gives a more detailed description of the Java Card platform.

Modern smart cards offer GlobalPlatform support for card and applet management. With this, multiple applets can be deployed on a single card and are shielded from each other by a hardware firewall. A more detailed description can be found in Section 2.5.

Having a processor run a virtual machine and Java Card program on a small and portable device comes with requirements and constraints on the power supply and communication performance. Smart cards are typically equipped with contact or contactless interfaces, with some cards providing both options via a dual interface. Contact interfaces transmit data over an exposed contact pad and receive power from a card reader on the same pad. Contact cards are standardized in ISO/IEC 7816 1-3. Contactless smart cards have an antenna inside the card and need to harvest power via resonant inductive coupling. Sending data to the card is done by modulating the amplitude (Amplitude-Shift Keying

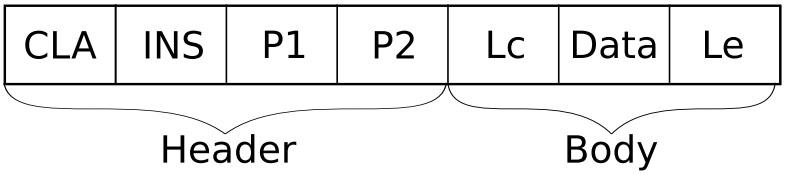

Figure 2.1: Command APDU structure [12]

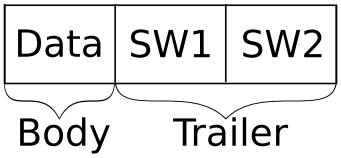

Figure 2.2: Response APDU structure [12]

(ASK)). For sending data back to the reader, the card uses load modulation due to power supply constraints. The contactless interface is standardized in ISO/IEC 14443. Both interface types communicate using the same Application Protocol Data Units (APDUs), standardized in ISO/IEC 7816-4 [7, p.13ff]. APDUs are an essential part of the cluster implementation in Chapter 4. Subsection 2.2.1 gives a more detailed description of APDUs.

2.2.1 Application Protocol Data Unit

APDUs are low-level datagrams to communicate with smart cards and smart card applications, standardized in ISO/IEC 7816-4. APDUs are split into command and response APDUs, for simplicity called C-APDU and R-APDU. C-APDUs start with a class byte (CLA), defining the application type and valid commands inside the application. The instruction byte (INS) defines the instruction to execute. Two parameter bytes (P1 and P2) provide additional information for the instruction, e.g., the offset for write access. CLA, INS, P1, and P2 form the header of the C-APDU and are always present. The optional body contains the fields Lc, Data, and Le, which can also be optional. The Data field contains data sent from the reader to the card. Lc defines the length of the Data field and must be present when the Data field is specified. Le defines the length of the response data to be sent from the card to the reader, i.e., the response length [7, p.15f]. The C-APDU structure is shown in Figure 2.1.

R-APDUs are structured in an optional body containing data depending on the executed instruction and a mandatory trailer. The two-byte trailer has two 8-bit status words, SW1 and SW2, indicating the result of the executed instruction. For humans, SW1 and SW2 values are often grouped into a 16-bit word to give a shorter representation. Status 0x9000 indicates the successful execution of the command. Other values are interpreted as error codes [7, p.15f]. An overview of the structure of R-APDUs is shown in Figure 2.2.

PC/SC provides a de-facto standard interface for programmers to interact with smart cards and transmit APDUs. The specification of the PC/SC Workgroup was originally implemented in Microsoft Windows' WinSCard, but with PC/SC lite, a free implementation for Linux and Mac OS X exists. Functions like SCardConnect and SCardTransmit allow implementing smart card communication efficiently. Programmers do not have to deal with physical data exchange directly. I.e., apart from specifying if protocol T=0, T=1, T=CL, or if auto-detection is used, the APDU encoding is transparent to the programmer.

As mentioned before, smart cards have a strong need for tamper-resistance. While HSMs are also protected against tampering, they are usually located in physically secure locations, and it is hardly possible to remove them without somebody noticing.

Smart cards are small, portable, and often carried in the pocket. Therefore, stealing them is easier, and their absence may go unnoticed. It is crucial to protect banking and identity cards against third-party attackers, however, tamper-protection is also necessary against the owner. A tampered identity card may allow the card holder to impersonate another person. Smart cards include sensors for tamper-detection and techniques to reduce side-channel leakages to protect themselves. Physical tampering can be detected by light, temperature, voltage, and current sensors. Additionally, the chip layout is scrambled, and data lines are encrypted. Side-channel leakages are prevented by drawing a constant amount of power, adding noise to the power consumption and timing as well as constant-time algorithms. These tamper protection mechanisms typically lead to smart cards evaluated to EAL4+ [7, p.19].

2.3 Secure Elements

Secure Elements (SEs) are designed to behave very similarly to smart cards in all aspects, essentially emulating smart card behavior. They are small, tamper-resistant single-chip devices designed for key storage and cryptographic operations, with the main difference being their embedding in other devices. Therefore, SEs do not need to be portable, allowing them to be more integrated and compact. While physical wear of contacts and plastic is a serious consideration for smart cards, it is not very important for SEs.

SEs require secure memory to store cryptographic keys and trust anchors, cryptographic functions to use the key material for signing and encryption without extracting the keys, and a secure environment for code execution. It is possible to emulate the features of SEs in software, leveraging CPU features like the ARM TrustZone. However, a specially designed embedded device offers more and better ways of protecting the key material and code.

The physical connection of the SE depends on the used chip. Analogously to when using a Universal Integrated Circuit Card (UICC), the SE can be connected using Single Wire Protocol (SWP). Alternatively, SigIn-SigOut-Connection (S2C) can be used for embedded SEs as well as the protocols mentioned in ISO/IEC 7816 [7, p.357].

Communication with applications on the SE is handled using the APDU format described in Subsection 2.2.1. Also, the management of SEs makes use of technologies known from smart cards, like GlobalPlatform. Running a JavaCard VM allows reusing applets from smart cards also for SEs without modification. These similarities are not surprising, as smart cards and SEs are even equipped with the same chip family. NXP's SmartMX3 series is used in smart cards and SEs, for example. Therefore, a lot of the descriptions in Chapter 4 mention smart cards as well as SEs. Many of the tests in Chapter 5 and Chapter 6 are done using smart cards in preparation for a test with SEs. As smart cards are easier to get and interact with, they are ideally suited to prototype SE applications.

2.4 Java Card

Since version 3.0, the Java Card platform is split into two variants, the Classic Edition and the Connected Edition. Java Card Classic Edition is a direct successor of the previous version 2.2.2 and targeted for smart cards and other elements implementing ISO/IEC 7816 or ISO/IEC 14443. The new Java Card Connected Edition offers many new features like multi-threading and network support, and therefore also requires more powerful hardware. The base communication protocol of the Connected Edition is TCP/IP. As all further chapters refer to ISO/IEC 7816 compatible devices, this section will only give a brief overview of the Classic Edition.

The Java Card Classic Edition platform is defined in three complementing specifications. While the idea of Java Card Classic Edition is to bring a programming language similar to Java and with support of Java paradigms to embedded security devices, Java Card CE is sharply different from Java. These differences are visible in all three parts of the specification [13].

2.4.1 Java Card Runtime Environment

The first document of the platform specification describes the Java Card Runtime Environment. Part of the runtime environment are the Java Card Virtual Machine, the Java Card Application Programming Interface, as well as support services. The Java Card VM, described in more detail in the next section, lives throughout the whole lifetime of the SE - and not only until power is removed.

Once started, the virtual machine keeps running forever - in Java Card terminology. This property is achieved by ensuring the VM is in a consistent state at all times. The meaning of this is evident when the card is powered. When power is removed from the card, volatile information may be lost, but data can be persisted on appropriate memory, like EEPROM or flash. Whenever the power supply is restored, the VM can restore its heap objects from the persistent storage and continues running in the same state as before.

Objects created by applets that span over power-sessions are called persistent objects. Any object referenced by persistent objects must be persistent themselves [13, Java Card

Runtime Specification]. Furthermore, any operation on persistent objects may imply updates on non-volatile memory. This is an essential property for the wear on memory cells. Its impact is part of the consideration in the cluster design of Chapter 4.

The Java Card Runtime environment offers different types of objects to reduce the number of required write operations on the card's non-volatile storage: [13, Java Card Runtime Specification].

- **Temporary objects** are used only in the current execution to compute temporary values. They cannot be referenced from persistent objects. Objects are automatically declared as temporary by the Java Card Runtime Environment, and storing references to them is prohibited by the firewall functionality.
- **Array views** are an example of temporary objects and offer read-write access to a subset of an array. When using array views, they behave the same as the original array by mapping the read or write access to the original array.
- **Transient objects** on the Java Card platform are persistent objects that have transient fields. Using transient objects does not only decrease the write operations on non-volatile memory, but they also improve security and speed up the operation. This is because data is not persisted, and access to non-volatile memory is typically slower than RAM access. The content of transient objects is cleared on certain events.

A Java Card applet's lifetime differs from the execution lifetime of the Java Card VM. An applet may begin its lifetime at any time during the execution of the Java Card VM when the `Applet.register` method is called. The `register` method must be called from within a static method `install`, which is called to create an instance of the applet. Newly installed applets are in a suspended state and must be selected using the `SELECT FILE` command APDU. The applet must accept the select request by returning `true` in its `select` method. After the successful selection of the applet, the content of the `SELECT FILE` command APDU is passed to the `process` method. The `process` method handles all APDU commands (except for `MANAGE CHANNEL`), processes them and generates the response APDU. If another applet is selected, the `deselect` method of the currently selected applet is called. When removing an applet, the `uninstall` method allows cleanup and prepares for deletion. A call to `uninstall` does not automatically end the lifetime of an applet but is called any time the Java Card RE wants to delete the applet instance [13, Java Card Runtime Specification].

Implementations of the Java Card Runtime Environment shall implement context and applet isolation according to the specification. An important mechanism of the recommended isolation is the applet firewall. The Java Card applet firewall is enforced by the Java Card Virtual Machine at runtime and prevents access to data of other applets. Protection mechanisms offered by the Java language are enforced by the Java Card VM within the applet. The Java Card applet firewall allocates a separate space for data per

`.cap` file. These spaces are called contexts and shared between applets belonging to the same `.cap` file. A special context is the Java Card RE context, which is isolated as any applet context but has privileges allowing access to contexts of other applets. If access to methods within another context is desired this can be allowed explicitly. During the cross-context call of methods, data is protected by a context switch [13, Java Card Runtime Specification].

Transactions follow a principle similar to ACID known from databases, which states that any performed transaction shall ensure atomicity, consistency, isolation, and durability. Consistency means that data is mapped from a consistent state to a consistent state. The transaction logic can ensure this. Isolation from concurrent transactions becomes trivial if multi-threading is not available. The durability of the transaction result is guaranteed by using persistent objects. The property of atomicity requires an "all-or-nothing" execution, meaning that the transaction must either be performed entirely or not change any state at all. This last property seems to be the most tricky to guarantee on a card, which can lose power at any time [13, Java Card Runtime Specification].

Therefore the Java Card RE provides the required features to guarantee atomic transactions. Any update to a persistent field is guaranteed to be atomic. If power is lost while performing an update, it is guaranteed that the previous value is restored on power restoration. If multiple operations need to be executed atomically, they can be joined within a Java Card transaction [13, Java Card Runtime Specification].

2.4.2 Java Card Virtual Machine

In the second part of the platform specification, the Java Card Virtual Machine is defined. The Java Card VM implements a hardware abstraction layer bringing the slogan "write once, run anywhere" known from Java to smart cards. Java Card programs are converted into `.cap` files, which offer binary compatibility across all cards running the same Java Card VM version. Thereby developers can write Java Card applets without even knowing the exact target hardware.

The process of bringing a Java Card program onto a card can be outlined as follows: As with standard Java programs also Java Card programs are first written as one or more Java classes (`.java`) and further compiled into `.class` files containing byte code. The class files are combined with export files (`.exp`) containing information about other required packages by the converter, which outputs the `.cap` file. The final `.cap` file is installed onto the card, and the contained byte code can be directly executed in the Java Card VM.

As mentioned before, the Java Card platform aims to be similar to the Java platform, but it implements only a subset of its features. The Java Card VM is designed to run on constrained devices with little RAM, storage, and processing capabilities. Some features like dynamic class loading, multi-threading support, annotations, and multi-dimensional arrays would claim too many resources or would need unavailable hardware features. Also, floating-point data types and character variables are not supported by the Java

Card VM. Exceptions, dynamic object creation, and virtual methods are supported [13, Java Card Virtual Machine Specification].

2.4.3 Java Card API

The Java Card Application Programming Interface (API) is a subset of the Java programming language. It is structured in four core packages [13, Java Card Application Programming Interface]:

- `java.io` is a sharply restricted subset of `java.io` of the Java programming language, offering only the `IOException` class for an identical hierarchy of exceptions.
- `java.lang` includes the base class `Object`, which is the root in the Java Card class hierarchy. Additionally, `java.lang` contains the base class of all errors and exceptions `Throwable` as well as some other exception classes.
- `javacard.framework` contains numerous classes to work with Java Card technologies, like `AID`, `PIN`, and `APDU`.
- `javacard.security` implements commonly used cryptographic functionality like hashing, encryption, and signing. It supports AES, DES, Diffie-Hellman, DSA, EC, and RSA, among others.

Several extension packages, most of them at `javacardx.*` provide additional standardized functionality that a Java Card platform implementation can provide. They include more advanced APDU handling, enhanced support for cryptography and biometry, support for X.509 certificates, and more. Support for extension packages is optional. Not all extension packages may be available in a Java Card platform implementation.

The Java Card API explicitly mentions that parameter values that lead to automatically detected runtime exceptions, like `NullPointerException`, shall not be checked. Adding explicit checks slows down performance and may hide program errors behind normal return values [13, Java Card Application Programming Interface].

2.5 GlobalPlatform

GlobalPlatform defines an open-standards architecture for smart card management. It provides the smart card issuer with control over the application management as well as the general security concept for multi-application card management systems. The GlobalPlatform card specification is specially designed for smart cards but is useful for any multi-application system based on Java Card or MULTOS, including SEs [14, p.39]. A GlobalPlatform device (GPD) is the abstract trusted execution environment defined by GlobalPlatform. It generalizes the concept of smart card management for other secure hardware devices [7, p.88].

The GlobalPlatform card architecture enforces security between different applications by restricting them to separate security domains. Applications can only access the trusted GlobalPlatform API and runtime environment API with functionality restricted to the security domain type. The mandatory Issuer Security Domain represents the card administrator. Additionally, Supplementary Security Domains and Controlling Authority Domains can be enabled optionally. Supplementary Security Domains are used if non-Administrator instances deploy an application to the card [14, p.41].

The GlobalPlatform Environment (OPEN) provides an API for application code loading, card content administration, and memory management. Enforcing security principles of card content is done by OPEN as well as APDU command dispatching. Command dispatching involves application selection, which is also handled by OPEN. Logical channels allow selecting multiple applications at once.

2.5.1 Card Life Cycle

Each smart card and SE has typical lifecycle stages, which are illustrated in Figure 2.3. Not every device has to go through all stages. Some stages may be skipped intentionally; others may not be reached because the device is destroyed beforehand.

The GlobalPlatform card specification maps these organizational states into technical terms. Clearly, "Chip manufacture" does not need a saved state as the chip is not even available yet. `OP_READY` indicates the card and runtime environment are ready to receive APDU commands. During `OP_READY`, the card can be initialized, and applications can be loaded onto the card. Transitioning to `INITIALIZED` is an irreversible action. The meaning of `INITIALIZED` is not standardized but can be used to indicate that initialization has been completed [14, p.51].

Further transitioning to `SECURED` indicates the card is ready for daily use, and all initialization steps are completed. Transitioning from `INITIALIZED` to `SECURED` is an irreversible action.

The state `CARD_LOCKED` is used to disable card functionality like the selection of applications. A locked card may only allow the selection of the final application without the ability to load new applications. A card in the state `CARD_LOCKED` can be put into the state `SECURED`. Therefore the state transition is reversible.

The `TERMINATED` state indicates the end of life of a card. Setting a card to `TERMINATED` disables it permanently. This action is irreversible. `TERMINATED` can be reached from any other state. Terminating the card life cycle can be done by administrators but is typically used by privileged applications. An application or OPEN may logically destroy the card in case of detected security threats [14, p.51ff].

2.5.2 GlobalPlatform Secure Channels

A secure channel allows protected communication between the card and an application outside the card. The secure channel provides secure messaging support for authentication,

Figure 2.3: Organizational view of smart card lifecycle stages [7, p.22]

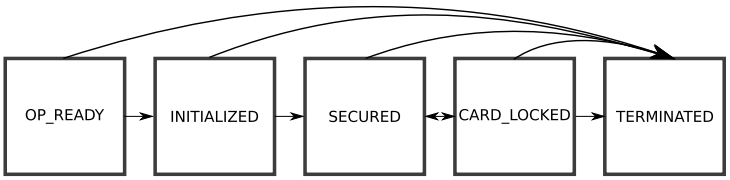

Figure 2.4: GlobalPlatform lifecycle stages [14]

integrity, and confidentiality of APDU command and response messages. During the secure channel initiation, key parameters are exchanged, and the authentication of the off-card application is performed. After the initiation, the Secure Channel Operation phase starts. This phase contains the protected data exchange. Teardown of the Secure Channel is called Termination [14, p.135].

A secure channel can explicitly be created through the appropriate APDU command or using the API on the card. Explicit secure channels allow selecting the security requirements, keys, and time of channel initiation manually. Secure channels are created implicitly by the protocol handler when an APDU command containing cryptographic protection is received. The handler chooses the security level and keys automatically. Keys can also be configured before the implicit channel initiation [14, p.136].

The mandatory step of authenticating the external entity during the Secure Channel initiation can be done using symmetric or asymmetric cryptography. Secure Channel protocols using symmetric keys, like SCP03, require the secret key of the card to be known by the external entity to authenticate. Any entity wanting to establish a symmetric Secure Channel is required to know the secret key. Due to the properties of symmetric keys, different entities using the same symmetric key cannot be distinguished. It is assumed that any successful authentication using a symmetric key is done by the Application Provider [14, p.138].

Asymmetric Secure Channel protocols like SCP11 require a public-private key pair together with a certificate issued by an authority trusted by the Security Domain. Any party owning such a certificate can authenticate successfully. As different entities own different key pairs and certificates, they can be identified individually. Depending on the authenticated entity, the security level is set to `AUTHENTICATED` or `ANY_AUTHENTICATED` [14, p.138].

2.6 JavaCard Open Platform

JavaCard Open Platform (JCOP) is a proprietary software stack developed by NXP Semiconductors. New versions of the software stack are published together with new hardware microcontrollers. The newest revision is JCOP 4 based on P71 controllers, but other versions like 2.4.2 R2 based on the P5 controller are also used for the purpose of this thesis. The software stack can be split into four main components [15]:

- The microcontroller firmware includes the Crypto Library for cryptographic operations. It is used to test and boot the microcontroller as well as for accessing the memory and cryptographic co-processor.
- An implementation of the JavaCard specification includes the JCVM, the JCRE, and the JCAPI.
- The GlobalPlatform Framework manages applets and card content.
- The Secure Box executes native software securely.

Cards based on the JCOP OS include only the last three components and may use another chip card controller. The JCOP 4 stack provides many security features listed in detail in [15] and is evaluated against EAL6 augmented (EAL6+). Additionally, the software stack can include optional functionality, like JCOPX extensions. The JCOPX API provides enhanced and ready-to-use implementations of functionality commonly used by JavaCard applications. The HKS applet described in Section 4.7 makes use of JCOPX features and is, therefore, only compatible with JCOP-based cards.

2.7 PKCS #11

PKCS #11 provides a standardized ANSI C API, named Cryptoki and pronounced like "Crypto-key," for cryptographic devices. The standard abstracts the device characteristics into a uniform interface, allowing applications to become portable and compatible even with unknown device types. Keys, certificates, and other basic cryptographic elements are handled through an object-based approach. Exposing these objects is mostly done through abstract handles, i.e., IDs referencing the objects. The use of handles allows uniform access to different types of objects, without the need for copying large data structures or exposing sensitive information. Further information about the objects can be requested through function calls, but these requests may be denied by the PKCS #11 library to protect data confidentiality.

PKCS #11 is part of the Public-Key Cryptography Standards, originally published by RSA Security. Over the years, RSA Security has handed over the development to OASIS to develop PKCS #11 as an open standard in the OASIS PKCS 11 Technical Committee. The currently released version is 2.40. This version has been used as a baseline for this

section and the implementation of the library in Subsection 5.1.1. A public review draft for version 3.0 is available, but currently support in software is poor [16].

Each function defined in the PKCS #11 specification makes heavy use of custom data types, prefixed with `CK_`, and constants of different prefixes. Constants defining cryptographic mechanisms, i.e., algorithms, are prefixed with `CKM_`, attributes start with `CKA_`, object classes begin with `CKO_`, and others exist. Two of the most important data types are `CK_SLOT_INFO` and `CK_TOKEN_INFO`, carrying information about slots and tokens [16]. While slot and token are universal terms available for any PKCS #11 implementation, they can be imagined as slot being a smart card reader and token a smart card holding the cryptographic information. For implementing the cluster, it is important not to expose information about the physical building blocks, i.e., SEs or smart cards, as user applications shall never have to deal with load balancing or similar. It is the PKCS #11 library's job to hide any details about the available hardware components and provide a uniform interface to all applications.

Typical cryptographic applications make use of several functions provided by the PKCS #11 library. This section outlines some of them. Before calling any other function, a program may use `C_GetFunctionList` to obtain pointers to the library provided functions. `C_Initialize` is then used to set up data structures required by the library. While some information like token or slot info is always available, interaction with cryptographic objects and functionality requires a session. `C_OpenSession` can create it.

It is required to authenticate as a trusted entity using the `C_Login` function to modify cryptographic objects and access sensitive information. Now the application can generate a key using `C_GenerateKey` or key pair using `C_GenerateKeyPair` directly inside the cryptographic device. If the created key is marked as `CKA_NEVER_EXTRACTABLE`, it will be created with an attribute such that it can never leave the cryptographic device. Any attempt to export it will fail.

Cryptographic operations like `C_Encrypt`, `C_Decrypt`, `C_Sign`, or `C_Verify` use the generated keys. Each of these functions has a corresponding Init, Update, and Final function to support multi-step computation. Additionally, PKCS #11 provides support for key wrapping, key derivation, and random number generation. Support for these functions makes sense if the corresponding operation can be executed directly inside the cryptographic hardware. Applications requiring true random numbers can access true random number generators through the standardized and vendor-independent interface. When the PKCS #11 library is not used anymore, the consuming application shall release the acquired resources through functions like `C_CloseSession` and `C_Finalize` [16].

All functions required by the PKCS #11 standard have a return type of `CK_RV`, which is currently defined to `unsigned long int`. The return value indicates the result of the function. It is expected that functions return `CKR_OK`. All other return values indicate different kinds of errors. Functions not available for certain cryptographic devices shall return `CKR_FUNCTION_NOT_SUPPORTED` for example [16].

Each object has certain attributes. Which attributes are available is defined by the class attribute `CKA_CLASS` that is available for all objects [16]. What attributes shall be available is clearly defined in the specification. However, special cases exist, and not all PKCS #11 library consumers interpret the standard equivalently. The problems that occurred during the implementation of the SE cluster library are discussed in Subsection 5.1.5.

CHAPTER 3

State of the Art

Many peer-reviewed papers and books describing concepts related to this thesis and defining conceptions elemental to secure hardware devices have already been published. The following chapter gives an overview of the most important works and summarizes the state of the art in the field of SE clusters.

The terms "smart card" and "hardware security module" are excellently described in [7]. The work also contains essential information on the idea and properties of smart cards, HSMs, SEs, cryptographic algorithms, and security analysis. It motivates the use of secure hardware devices in embedded computing. [17] thoroughly discusses the results of a survey of crypto-processors available in 2006. Although the paper is quite dated, it provides a good overview of different cryptographic processors and attacks on them. [18] shows the differences between commercially available HSMs in 2010.

An important motivation for the development of an SE cluster is to reduce costs as conventional HSMs are expensive. Researchers in Salerno, Italy, proposed another cost reduction mechanism for HSMs in [2]. Their idea was to reduce the required storage space, as it is a considerable cost-factor in some deployments. Based on on-the-fly key computation, the HSM only needs to store initial parameters and an index per key, while conventional approaches store the full keys. If a larger number of HSMs is needed due to performance requirements, the proposed approach is not helpful. It even decreases the performance due to the need for re-computing the key material before it can be used. [19] shows an approach based on non-volatile Field Programmable Gate Arrays (FPGAs) to reduce costs in HSMs without the need for re-generating keys. [20] proposes security features used by the FPGA-HSM and features that are useful for the cluster proposed in this thesis.

Another advantage of the cluster proposed in this thesis is the scalability required by IoT-applications. When deploying multiple IoT-devices that have different requirements

on cryptographic performance, it is desirable to build upon a single, scalable architecture. Other requirements for IoT-HSMs, in particular for automotive use, are described in [5].

[1] outlines the requirements for a trusted HSM in critical infrastructures. The paper states that HSMs must follow a "security by design" approach in software and hardware components. It further proposes an HSM for critical infrastructures based on the ARM TrustZone. While the described solution itself offers no scalability, future research could address it. Because ARM processors are widespread, become faster in short cycles, and offer a promising approach to cryptography based on their TrustZone, HSMs based on trusted computing technology are competing with the SE cluster. The fundamentals of building an HSM with trusted computing are given more elaborately in [21].

The development of an SE grid or cluster was a years-long process that started about two decades ago. [22] introduces the first concept of outsourcing computation to smart cards. It bases on smart cards that are accessed over interfaces known from distributed computing scenarios, such as Java RMI. The setup is built from RISC 32-bit smart cards that support downsized implementations of IP and HTTP, named as "next-generation" back in 2001 when the paper was published.

Enhancing the idea of [22] further, the authors of [23] introduced a network-attached Java Card grid. It is the first paper mentioning the combination of multiple smart cards for achieving a greater goal. The greater goal in this paper is the scalable and distributed computation of functions, illustrated by computing the Mandelbrot set over multiple cards. The grid, as used in [23], does not yet have a separated, application-independent abstraction layer as this thesis will introduce. However, the grid already comes with a scheduling algorithm for concurrent operations. Scheduling is done based on the length of a per-card operations queue. [23] includes a proof-of-concept hardware setup, very similar to the one described in Section 5.2. The researchers of [23] have further noted that the memory type of smart cards can make a huge difference in their longevity. While EEPROM based cards handle around 10^5 write cycles, FRAM cards support at least 10^{12}.

A useful application for the smart card grid was found in [24] in 2010. The existing concept of distributed computing on smart card grids is used to terminate the EAP-TLS connection for RADIUS. While the cards compute the cryptographic functions for the authentication, they also deal with the overhead of the EAP-TLS protocol and TCP. The proposed solution shows that using multiple smart cards increases the throughput, but latency doubles when the grid is accessed over a network interface.

In 2013, Pascal Urien, who at the time was already involved in the grid development of [24], proposed a "secure cloud of secure elements" together with Selwyn Piramuthu in [25]. [25] starts with an overview of the historical development of SEs, beginning from first attempts in the seventies up to current NFC-enabled SEs. The paper also discusses the concepts necessary for relaying NFC and security measures preventing it. A grid of SEs is accessed over the network from NFC-enabled mobile phones, which expose the element over their NFC interface to interact with other devices. Based on the measured timings,

the paper suggests that latency-sensitive applications require a caching functionality due to the overhead of the network connection. [25] only gives few information on the internal organization and scheduling of the grid. Because the grid's relay protocol exposes a `SEND-APDU` command and a `LOCATE` command exposing the serial number of SEs, it is clear that the grid does not perform transparent load distribution. Applications using the NFC Secure Proxy are aware of the internal structure of the grid and have to implement load distribution themselves. The SE cluster described in Chapter 4 of this thesis fully encapsulates all internal structure of the cluster, giving better compatibility with existing applications.

The grid of SEs, or as called in the paper cloud of SEs, introduced by Urien in [25], is further refined in his 2013 paper [6]. Using the ideas of the previous concept, the new paper describes a more detailed setup and access protocol based on USB-attached card readers or soldered elements. The protocol named RACS [26], short for Remote APDU Call Secure, gives a unified way of sending APDUs over the network to a grid. A commercial service for remote accessed SEs is already available at `https://www.implementa.com/`, but it does not use the RACS protocol. Furthermore, [6] mentions that such grid devices could be used to compete with HSMs by using a PKCS #11 API. What parts of SEs an API exposes must be designed carefully, as it might introduce Denial of Service attacks or worse, as shown in [27]. [6] does not mention clustering, load-balancing functionality, synchronization principles, or any encapsulation of the internal structure inside the cluster.

While load-balancing has not been mentioned together with SEs yet, the principle is well studied in the literature. Many surveys and comparisons exist. The latest papers focus on cloud and parallel computing like [28], [29], [30], [31], and [32]. Even though the fields differ sharply at first glance, aspects like multi-level load-balancing, health-monitoring of nodes, and time estimations, are suited to be copied for large secure element deployments. In addition to the current load, the task distribution software should take unexpected behavior of the computing nodes into account, as mentioned in [33]. Load-balancing is not only crucial for fast response times but also has a significant impact on the longevity of the cluster. To extend the lifetime of an SE, mechanisms for wear-leveling on flash memory, as proposed in [34], shall be used. Scheduling write cycles on the flash memory is not part of this thesis.

An approach similar to synchronizing key material from one element to another is backup and restore. [35] describes auditing and import/export mechanisms of HSMs based on the OpenHSM project by providing a list of required roles and algorithms. The paper additionally hints that HSMs are not sufficient to provide security for all scenarios, but appropriate processes must accompany them.

[36] evaluates the performance differences between Host Card Emulation (HCE) and hardware security elements. Chapter 6 tackles a similar problem, where the performance of the smart card cluster is compared to the SE cluster.

CHAPTER 4

Secure Element Cluster

A new secure hardware device based on clustered SEs is described in the following chapter. After highlighting the overall architecture concept, the trust relationships and cluster management strategies are defined. Cluster management includes key synchronization, load distribution, and self-healing properties in case of failures. The chapter concludes with a security evaluation of the cluster concept.

4.1 Cluster Architecture

The main building parts of the cluster are secure cryptographic chips of any kind. Therefore, the following sections describe the cluster architecture independently of the chip type and form factor. It can be implemented using smart cards, tokens, or SEs. These building parts are connected and operated by the Hardware Key Safe (HKS) applet.

The HKS applet is designed for the JavaCard 3.0 specification and can run on any JCOP device supporting the required JavaCard version. The development of the applet was done by Yagoba GmbH in the course of the ESPRESSO research project. A more detailed breakdown of which component existed before this thesis and what has been developed in terms of the thesis is given in Section 4.9.

Provisioning the HKS applet can be done with a Worker or Manager configuration. HKS Workers hold key material and perform cryptographic operations. HKS Managers are needed to synchronize key material across Workers. Additionally, it is possible to define Managers of Managers, allowing to create an arbitrary large hierarchy and to separate key material into domains.

Figure 4.1 gives an example of an applet hierarchy. It contains two separate Manager domains which are connected over a third Manager. The two domains can synchronize within each other and across domain borders if necessary. Separating Workers into logical

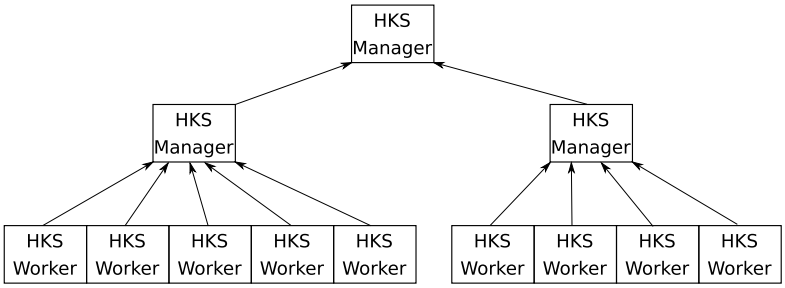

Figure 4.1: Example: HKS applet hierarchy with two levels of Managers

domains allows distributing key material only to a subset of Workers and reduces the impact of a failing Manager.

The cluster is designed to be accessed through an additional software or hardware layer, hiding the HKS Workers and Managers from users. Introducing a management layer gives the cluster its scalability properties described later in this chapter. By hiding the HKS applets, the cluster appears to applications as a single device and thereby improves compatibility with existing applications drastically. Without the abstraction, each application would need to implement its own load balancing and synchronization strategies. Hiding the SEs under a uniform interface distinguishes the cluster approach from existing SE grids.

4.2 Trust Architecture

Trust is a fundamental part of the cluster architecture as one of the main points of using secure hardware is the concept of keys living only inside those secure hardware components. Trust is required when synchronizing key material to other devices.

Therefore, the most straightforward cluster configuration consists of independent HKS Workers which do not synchronize any keys. In this scenario, keys are generated directly inside the chip where they will be used, and no synchronization is necessary. No trust is required between the applets if keys are never synchronized. However, while using different keys may work for signing, it is mandatory to use the same keys across all Workers for encryption or decryption. Additionally, when using an SE cluster, it is typically desired for operations to be transparent on the cluster. Using different keys on different Workers breaks the transparency, and the resulting signature is a direct trace to the used SE.

The typical cluster configuration consists of HKS Workers and one or more layers of HKS Managers. When provisioning the applet, each instance of Workers and Managers generates its own EC key pair. The key pair is used to encrypt messages for the addressed applet instance. After the key pair has been created, a globally trusted issuer key signs the public key, and the applet stores this signature. Configuring the trusted issuer key

is part of the provisioning process, which as for any other applet, typically is done in a physically secure environment. Without the protected environment, not only the trust configuration but also the applet itself could be manipulated.

Each applet instance can have precisely one higher-level Manager. The configured Manager may change over time. It is crucial that the Manager can be changed after the initial deployment as the hardware of the Manager may fail. If the configured Manager would be immutable, all subordinate Workers could not synchronize key material anymore. With configurable Managers, a simple reconfiguration restores full functionality. When setting up the Manager, the public key and its signature are provided to the lower-level instance, which can verify the signature using the preconfigured public key of the trusted issuer. Configuring the lower-level applet instances on an HKS Manager is analogous. Exchanging the signatures creates a bidirectional trust between an applet instance and its Manager.

The root of trust is the global issuer key configured on each applet instance. This key is similar to the root certificate in traditional PKIs. The issuer guarantees that it only signs key pairs generated within hardware protected HKS applets. By trusting the issuer, one can verify that key material is never exposed also during synchronization.

4.3 Synchronization of Key Material

As mentioned before, it is necessary to synchronize key material among HKS Workers. Most importantly, it is the only way of using the same key on multiple devices and is, therefore, a fundamental building part of the cluster. Additionally, key synchronization is necessary when physical elements need to be replaced, or the cluster is upscaled horizontally.

Key synchronization makes use of the bidirectional trust between an HKS Worker and its Manager. The key is encrypted using the public key of the recipient, i.e., the Manager, and can then be exported without danger of key theft. Only the configured HKS Manager can decrypt the ciphertext using its private key and gain access to the exported key. As HKS Managers do not provide for cryptographic operations themselves, they only rewrap the key for another recipient. The key may be addressed to another Manager at a higher or lower level or a subordinate Worker. If the key is rewrapped for another Manager, the procedure continues as before. If the key is rewrapped for a Worker, it can now unwrap the key and store it in the protected hardware environment. The key synchronization from one Worker to another is completed. The bidirectional trust provided by signatures of the issuer key guarantees that the key material is protected during the whole synchronization procedure.

Each applet instance knows its direct neighbors, but Workers cannot build the path to the destination Workers themselves. The cluster management logic must address export, rewrap, and import operations to the correct element. This constraint follows from the restriction that SEs may not communicate with each other directly, but must

be addressed by some host. It additionally allows for a central state management inside the cluster management logic.

4.4 Load Distribution in the Cluster

This section outlines possible strategies for scheduling cryptographic operations on different elements and key material distribution. No single plan is preferred over another but must be selected case-by-case.

4.4.1 Cryptographic Operation Scheduling

To increase the number of cryptographic operations per second, multiple HKS Workers can run the activities in parallel. Parallelization requires having the same key available on multiple Workers but has no other restrictions on the applet instances. The idea is to route queued operations to the next available Worker but also respect the wear on the underlying element. Three approaches are outlined in the following:

- **Least-Wear-First (LWF)** selects the element with the least wear among the available elements. An element is available if it is initialized correctly and currently does not perform any operation. LWF maximizes the lifetime of a component by distributing activities evenly across all elements. Clusters scheduling transactions by using LWF provide the best performance over the whole cluster lifetime without requiring hardware replacement. Scheduling load evenly over similar elements makes the failure of multiple parts in close proximity of time more likely.
- **Most-Wear-First (MWF)** selects the element with the most wear among the available elements. MWF maximizes the time between the failure of multiple elements and minimizes the number of used elements. MWF uses the least amount of devices to reach the required performance, leaving the others in hot standby without generating wear. Clusters that schedule operations using MWF show degrading performance over their lifetime if failed hardware is not replaced.
- **MWF with LWF** combines both approaches. Some elements are grouped in an LWF partition, and the remaining components are kept in hot standby. Operations are scheduled inside the LWF partition only, and wear is only generated on elements in the LWF partition. Failing parts in the LWF partition are immediately replaced by a hot standby element, adding a new element to the LWF partition. MWF with LWF shows a steady performance of cryptographic operations while preparing for failure and gives time for replacing hardware components. MWF with LWF requires more elements to provide the same performance as MWF or LWF.

Figure 4.2 illustrates the differences between the approaches. All plots are to be understood as sketches for typical behavior without the claim of exact measurements. It is assumed that each scenario starts with 100 available elements. The blue line indicates

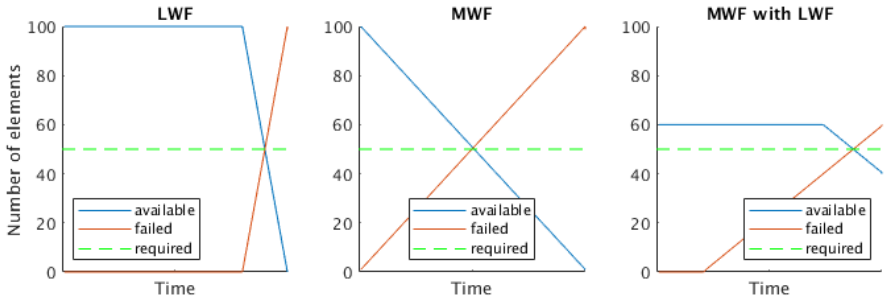

Figure 4.2: Comparison of available elements for LWF, MWF, and MWF with LWF

the number of elements available for cryptographic operations. The red line shows the number of failed elements. This number should rise slowly to give enough time for a replacement. The dashed green line is a baseline to represent the number of elements needed due to request peaks or storage requirements.

It is visible that LWF provides the best performance as all elements can be used for cryptographic operations, and the parts live longer than in MWF because the load is scheduled evenly. However, at the end of their lifetime, the elements fail almost simultaneously, urging hardware replacement with short notice. MWF makes the failing hardware visible by degrading in performance. It gives plenty of time for a replacement. Due to the concentrated load on single elements, they fail relatively fast, degrading the performance continuously. The performance target cannot be met anymore, even if plenty of unused hardware components are still available. MWF with LWF provides a tradeoff between LWF and MWF. By artificially restricting the number of accessible elements, the failed parts can be compensated for automatically. The elements fail slower than in pure MWF scheduling because the load is distributed over multiple elements in the LWF partition. Monitoring of the MWF with LWF cluster allows detecting the number of failed elements and gives a chance for early replacement. If the standby pool is dimensioned appropriately, the failed elements can be replaced without any impact on performance. Having hot-standby components is a well-proven approach in almost any technical scenario, e.g., RAID configurations, network interfaces, and routers.

4.4.2 Key Storage Distribution

Distributing key generation and storage load is equally important as it has effects on operation scheduling. Maximizing the operation throughput requires that key material is available on all Workers such that all Workers can be used for parallelizing operations. If more keys than a single Worker can handle are needed, it is necessary to split the key material into different partitions and reduce the number of available Workers for a given key.

The most straightforward approach is to break the key set into two distinct partitions of equal size and with roughly the same operation load. While it is an obvious solution

to the problem of insufficient key storage space, some instances may prohibit doing so. For example, if the operation load is not evenly distributed among the keys, it may be impossible to split the key set into two parts of the same size and same operation load. This especially is a problem if all keys are not used the same number of times.

A more advanced approach decides for each key how many copies are necessary to serve the operation load and where to place them. Ideally, the keys are distributed such that all performance requirements are met, the available storage space is sufficient for the number of keys, and all elements are facing the same load. This second approach requires a more complex key management logic but allows for more efficient use of resources, of both storage space and processing power. How the keys are distributed across elements may be fixed for each key, i.e., the mapping of the keys to devices is hardcoded in the configuration. The devices can also be selected by some algorithm based on the expected number of cryptographic operations per key. An even more advanced approach refines the key distribution during runtime, based on the actual measured load requirements. It should tend to overprovision resources to cope with performance peaks. By using a learning mechanism, key provisioning can happen according to a repeating cycle, e.g., a day-night cycle.

4.5 Scalability of the Cluster

Theoretically, the cluster can be scaled to an arbitrary number of SEs. Each HKS Worker provides only a limited amount of cryptographic operations per second and limited storage space for keys, but the number of Workers is not limited. If more cryptographic operations per second are required, or the storage space needs to be extended, the cluster can be enlarged by a new SE hosting another HKS Worker. Adding an HKS Worker is similar to the initial setup of the cluster. The HKS Worker must be initialized, and key material must be synchronized to the new Worker. In many scenarios, only a few Managers are required to serve a large number of Workers as key material changes only rarely. The number of Managers may need to be increased and separated into more layers if keys are generated more often or a more sophisticated load distribution mechanism is used.

If this is the case, it is vital to plan the architecture to reduce the load on Managers. Consider the following example illustrated in Figure 4.3: Worker 1 needs to synchronize key material to Worker 5 regularly. If they are directly below the same Manager, only a single rewrap operation is required to transfer the key. If Worker 1 is subordinate to Manager 2, Worker 5 is under Manager 3, and Managers 2 and 3 are connected by a superordinate Manager 1, then a total of 3 rewrap operations are required for a single key transfer. This has drastic impacts on larger deployments. The second case is illustrated in Figure 4.4.

If a large number of Workers and Managers are used over the same connection, e.g., the same bus, this connection can become the bottleneck and limit the overall performance. However, as the cluster design does not require any specific connection type, this can

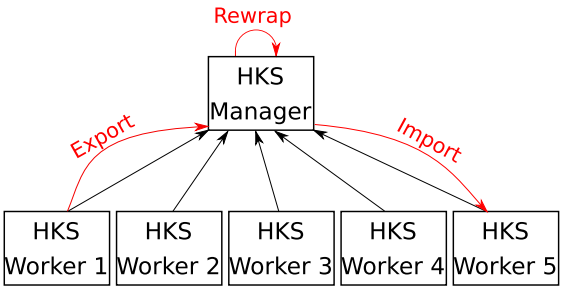

Figure 4.3: Example: Synchronization with a single rewrap operation

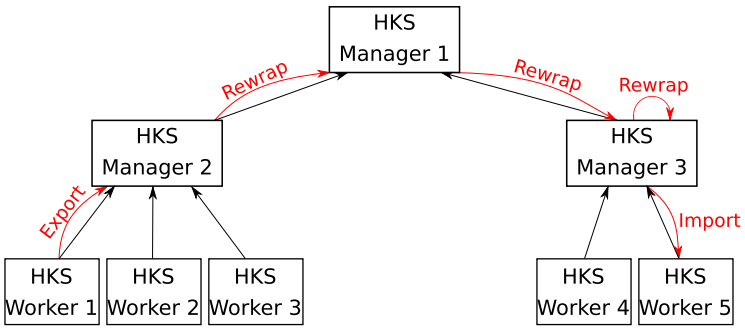

Figure 4.4: Example: Synchronization with three rewrap operations

always be avoided by using different connections and even different connection types. While some Workers are deployed on smart cards and access over a USB-attached reader, other Workers are deployed on SEs placed on a PCI-e card, for example. In even larger deployments, the Workers or Managers can be attached to different physical machines and connected over a network.

In general, it is desired to operate an additional Worker instance along with the Manager instance on the same chip, but prioritizing Manager tasks. This enables to use available resources for cryptographic operations when no Manager tasks are required. Currently, the HKS applet does not support multi-selection making a switch between Worker and Manager more time-consuming. If the Manager hardware is used for additional purposes, e.g., deploying a Worker instance, it must be considered that the device will fail earlier.

4.6 Self-Healing Properties of the Cluster

The cluster management logic is responsible for implementing self-healing properties. Failing elements need to be detected automatically and replaced with an equally configured part. Failed Workers are unregistered, and a new Worker is registered under the same Manager. The key material is synchronized from alive Workers. This requires that at least two copies of each key exist at all times. During the failure and resynchronization

process, the performance of the overall cluster is degraded. Please refer to Subsection 4.4.1 for information on how to limit the performance degradation of failed elements.

Failed Managers can be replaced in an even easier way but they may also have a minor impact on performance. If the failed Manager has a parent, it is unregistered there, and a new Manager instance replaces it. Previously registered children need to be announced to the new Manager, requiring actions on all subordinate Workers and Managers in addition to the registration on the new Manager. A failed Manager results in the inability to synchronize key material from and to its subordinates. If the Manager has no Worker child, the cryptographic performance of the cluster is not impacted. If a Worker child exists, a single APDU per Worker child is necessary to replace its Manager.

4.7 Hardware Key Safe Applet

The fundamental building blocks of the cluster are single SEs, managed by the HKS applet. As mentioned before, the applet can be instantiated as a Worker or Manager. The differences and roles in key management are described in Section 4.2 and Section 4.3. Before the applet can be used, it must be loaded by following the GlobalPlatform defined routine of loading, installation, and instantiation. The communication during the installation of the applet is secured using GlobalPlatform Secure Channels. After the applet is instantiated, it is in the `PERSO` lifecycle state. In this first of two states, the applet can be personalized. Personalization includes setting the applet ID and PIN, signing the applet specific key with the trusted issuer key, and defining a Global Platform Secure Channel security level. Once the last personalization step is done, indicated by a special `STORE DATA` APDU, the applet switches to the `OPERATE` lifecycle state. This state disables all personalization and debugging commands. Applet instances that have been transferred to the `OPERATE` state cannot leave it anymore. Being stuck in the `OPERATE` state forever ensures the security of key material and cryptographic operations.

The HKS applet supports numerous commands, each with standard and extended length APDUs. Commonly used C-APDUs include the following:

- `STORE DATA` is a personalization command for various kinds of settings. It can set the issuer signature, the issuer's public key, the security level, the global PIN, applet and key number limitations, among others.
- `IDENTIFY` returns information about the identity of the applet, including its ID, version, applet type, public key, and public key signature.
- `REGISTER` establishes a secure link from a Worker (or lower-level Manager) applet to a higher-level Manager applet. This command stores the Manager's public key on the Worker (or lower-level Manager). Exports and imports can only be done by this Manager. The secure link can be changed by running the `REGISTER` command again.

- `REGISTER WORKER` is the counterpart to `REGISTER`. `REGISTER WORKER` stores the Worker's (or lower-level Manager) public key on the higher-level Manager. It allows future key exports for a certain Worker (or lower-level Manager).

- `DEREGISTER WORKER` is used to remove a Worker (or lower-level Manager) from a higher-level Manager.

- `ALLOCATE KEY` reserves key storage space for a given handle. It also assigns the key type and size to a handle as well as performing possibly necessary initializations.

- `GENERATE KEY` generates a new key on a given handle. The handle must be allocated by `ALLOCATE KEY`.

- `GET PUBLIC KEY` returns the public key for a given handle if it is an asymmetric key pair.

- `EXPORT KEY` encrypts the key material of a given handle with the higher-level Manager's public key and exports it. This command is used during key material synchronization.

- `IMPORT KEY` decrypts key material encrypted by the higher-level Manager and imports it.

- `REWRAP KEY` is a Manager command used to decrypt the exported key material of a lower-level Worker or Manager and re-encrypt it for its higher-level Manager, a lower-level Manager, or a Worker.

- `DESIGNATE KEY` sets the allowed cryptographic operations for a key. Newly allocated key handles have no allowed operations and must have their permissions set explicitly by `DESIGNATE KEY`.

- `SIGN` can sign bytes with SHA-256 ECDSA, plain ECDSA, or SHA-1 RSA PKCS PSS. The list of algorithms may be extended in future versions.

4.8 Security Evaluation and Consideration

Even though the hardware and software design is not finalized for production use cases yet, it is already possible to consider security threats to the general concept and its prototype implementation. Because the HKS applet is deployed on certified hardware components, like smart cards and SEs, it is reasonable to limit the security evaluation to the HKS applet, connecting hardware components, and the consuming software stack. Any existing vulnerability of the cryptographic hardware will also affect the deployed HKS applet, however.

4.8.1 Key Extraction Mitigation and Synchronization Security

An attack on the physical component is required to extract keys from the SE or smart card directly. If such an attack would exist, it affects all card applications and is not limited to the proposed cluster architecture. Retrieving the keys without attacking the hardware is only possible if the applet exposes it. Exposure can occur on purpose, on accident, and during faulty synchronization attempts. The current applet version supports plaintext key export indeed, but only for debugging during the personalization phase. This phase typically has no keys generated yet, and the debugging command set will be removed for a production version of the applet.

Accidental exposure is relatively easy to prevent in Java Card applets, due to the extensive use of hardware-assisted security features, cryptographic libraries, and restricted language features. Reviewing the source code and especially the provided C-APDUs can effectively prevent accidental exposure of sensitive material.

During synchronization, the key material is encrypted using an algorithm unknown to the author of this thesis at the time of writing. Due to the available information on the sender and receiver ends, it is assumed to be based entirely on elliptic curve keys, like ECIES. All used keys are based on NIST P-256, also known as secp256r1 or prime256v1. This curve is widely used and supported, but the security researchers Daniel J. Bernstein and Tanja Lange have marked it as unsafe in their research [37]. If an attack on the curve used by the applet that also affects the cluster implementation is published or feasible, the algorithm can be changed without affecting the general cluster concept. Nevertheless, before the cluster is used in production environments, the supported algorithms need to be reviewed to the current standards.

A more feasible attack on the implemented synchronization feature is based on untrusted exports. When exporting or re-wrapping key material, it is encrypted using the public key of a Worker or Manager instance. Changing the target public key is possible during the whole lifetime of the HKS applet. If an untrusted public key could be used, the exported key material can be decrypted by an attacker. Two different security protections mitigate this attack. The first involves basic authentication to the applet. Before cryptographic or management operations can be performed, authentication using the global PIN or key PIN is required. An attacker stealing the smart card or getting access to the SE cannot perform any operation or extract key material. While this mitigation measure protects against attackers that do not know the PIN, extraction of the key material would still be possible for the administrator or user of the SE.

The trusted issuer's signature prevents this. Any public key used by HKS applets for synchronization or trust purposes must be signed with the trusted issuer private key. The trusted issuer public key is configured on each HKS applet instance during the `PERSO` phase, which is one reason why personalization must be performed in a physically secure environment. Once the applet is transitioned to the `OPERATE` phase, the issuer configuration cannot be changed. With its signature, the issuer guarantees that it witnessed the secure deployment of the applet, that the applet has not been modified,

and the applet key was indeed generated inside the SE or smart card. Because the issuer knows the applet code and verifies that it has not been altered, the issuer can testify the security of the public key. For any other applet instance using this public key for trust or synchronization purposes there is no way to extract key material because the key is secured and can only be used by trusted applets. Therefore, even an attacker with access to the secret PIN can only configure other trusted HKS applet instances as a synchronization target. For this reason, it is never possible to extract the raw key material. At best, an attacker can create a copy of the key in another HKS instance but requires knowledge of the PIN and access to a second HKS instance signed with the same issuer key. However, the issuer key must be stored in a secure hardware environment and is ideally protected by a multi-person key ceremony. If the issuer and the HKS applet administrator are separate entities, then nobody has key and PIN information to extract any key material.

4.8.2 Software Stack Security

Due to the previously discovered fact that keys cannot be extracted from the SEs or smart cards, no vulnerability in the software stack or hardware connection can lead to key exposure. Additionally, because the hardware building blocks already enforce some security properties and are designed to be operated in insecure environments, the risk of an attack is limited.

However, it is certainly possible that a cached or insecurely stored PIN can be compromised. With the authentication data, an attacker can use the key material inside the cluster for signature, encryption, and decryption. Special care is necessary when implementing the software stack for the cluster to prevent those attacks. Even with entirely secure cluster orchestration software, the key material must be usable by third-party applications, like a Certificate Authority (CA). Vulnerabilities in the CA can be exploited to use its key material for malicious purposes.

The prototype implementation has its cluster orchestration information stored in a file, e.g., information on the storage location of a key and the expected remaining lifetime of an element. This is not only a problem because it can leak sensitive information to an attacker but is also problematic in case the hardware cluster is transferred from one system to another. In a final version of the cluster, all orchestration logic and information should be contained in a single hardware element, e.g., a PCI-e card. Removing the PCI-e card from one system and inserting it into another also transfers all orchestration information. Furthermore, it is harder for attackers to access the data if it never leaves the PCI-e card.

4.9 Breakdown of Cluster Component Development

The introduction to this chapter already mentioned that the cluster development was started in the ESPRESSO research project in 2018. The project was initiated by the IAIK of Graz University of Technology, with its project partners PrimeSign GmbH, Yagoba GmbH, and IoT40 Systems GmbH. While being only a part of the whole research project, the development of the SE cluster is a fundamental goal and requirement for the success of the project. This section gives a clear distinction of which parts of the cluster already existed and what has been introduced for this thesis.

Credit goes to the ESPRESSO research team for the overall idea of the cluster and the HKS applet, which were developed before the start of this thesis. For this thesis, the existing cluster concept has been summarized in this chapter and enhanced with new ideas independently of the research project. The methods of wear-leveling in the context of the SE cluster did not exist before this thesis. The sections of load distribution, scalability, the clusters' self-healing properties, and the security evaluation have been described without existing documentation. Connecting multiple instances of the HKS applet in an abstracting cluster by encapsulating the orchestration logic in a PKCS #11 library was done by the author of this thesis.

In the past, simple tests with multiple smart cards had been performed but had no conclusive outcome. Therefore, the smart card setup presented in Chapter 5 and Chapter 6 was newly developed for the purpose of this thesis. The first prototype that includes SEs was built by Yagoba GmbH. This prototype has been connected to the PKCS #11 library in a joint effort between Yagoba GmbH and the author of this thesis. All measurements, performance evaluations, and cluster sizing estimations given in Chapter 6 were done without external help. The research project and its involved entities did not influence the thesis and its outcomes, apart from providing the existing hardware and software components.

CHAPTER 5

Proof-of-Concept Cluster Implementations

The following chapter describes two different PoC implementations of the cluster. The first is based on smart cards and can be recreated easily by any interested person. The second implementation uses SEs and requires hardware that cannot be obtained as easily. Both scenarios have their advantages and proved useful as the analysis in this thesis shows. A common software library based on PKCS #11 abstracts the hardware and allows integration into existing applications.

5.1 Software Architecture

The overall architecture of the PoCs is as described in Chapter 4. The focus for the PoC was set on implementing the previously described cluster ideas in software and attaching applications to the HKS stack. Communication between the user application, e.g., a CA, and the HKS applets happens in the following way: The user application needs to support the standardized PKCS #11 interface, commonly used for cryptographic operations, either directly or through a wrapper. When using the PKCS #11 interface, the application calls functions directly from the SE cluster library.

All logic for cluster orchestration and key management is contained in a single shared object, named `libsecluster-pkcs11.so`. The library is primarily designed for Linux but should compile and run just as well on other operating systems, maybe requiring minor tweaks. While the library always exports all functions required by the PKCS #11 specification, during the Proof-of-Concept phase, some functions return `CKR_FUNCTION_NOT_SUPPORTED`.

The communication with physical elements, smart cards, or SEs, is encapsulated in separate modules inside the library. This design allows switching between different

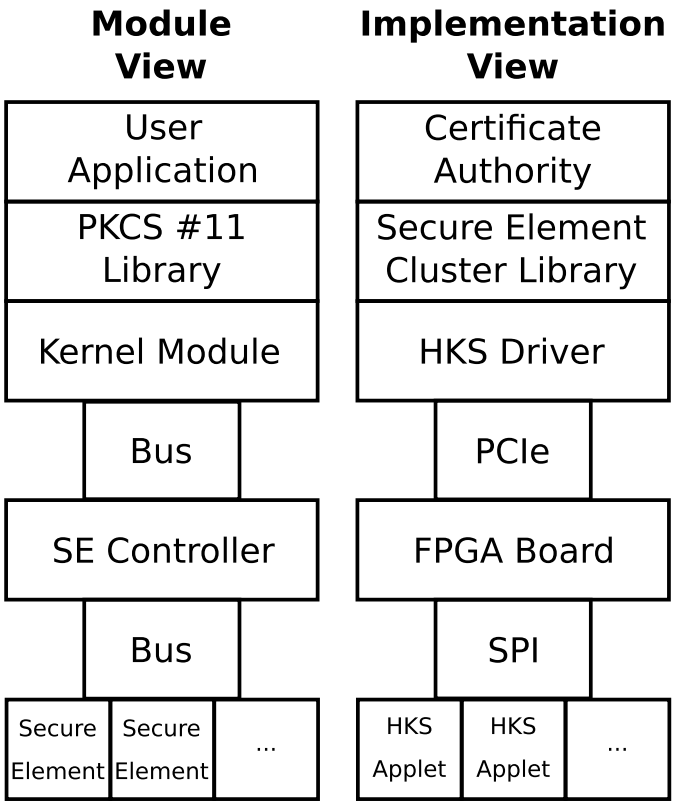

Figure 5.1: Architectural overview of software and hardware components

hardware seamlessly and extending the implementation for different hardware in the future. State keeping for different elements is also separated in a module, allowing to change the software in the future. The HKS applet is required as a base for the cluster implementation but was not modified for this thesis.

Figure 5.1 gives an overview of the software architecture. The next section gives a more detailed elaboration on the single items of the figure.

5.1.1 PKCS #11 Interface

The SE cluster library is based on empty-pkcs11, published by Pkcs11Interop on GitHub[1]. Using this template allows having a standard-validated C framework that includes all required PKCS #11 functions with types, comments, and a build setup for multiple architectures. The PoC implementation focuses on the most important functions required for the analysis of the prototype. Section 2.7 already describes the PKCS #11 standard, therefore, this section will outline the implementation specifics of the SE cluster library.

[1] https://github.com/Pkcs11Interop/empty-pkcs11

As with any PKCS #11 library, `C_Initialize` must be called before the start of any operation. It performs initialization tasks to set up the cluster state. Information about the devices is read from the database, including the number of previously performed sign and key generation operations and the available key material. Currently, the HKS applet does not keep track of this information itself, requiring the orchestrating software to store it for load and wear distribution. Additionally, `C_Initialize` scans for available applets using the PC/SC and FPGA sublayer. The primary and only identification used for matching physical devices to database entries is the HKS applet ID reported by the applet itself. In future implementations, the issuer signature reported by the HKS applet during the identification process will be verified to increase security. After this function returns, the library and the devices are initialized and ready for cryptographic operations.

`C_Finalize` closes all open sessions and frees resources allocated during operation. Additionally, information about performed signatures and key generations are written back to the database. Updating the database only during `C_Finalize` is a tradeoff between performance and data consistency. This design was chosen to have as little impact on the performance tests as possible. However, crashing applications or applications not adhering to the standard may cause data loss. For simplicity, the database that was used for the purpose of this thesis was implemented as a CSV file, which does not guarantee any transactional properties. A production-grade implementation of the cluster will move the state keeping to a lower layer, i.e., directly inside the FPGA, to make the cluster device portable. Therefore a highly sophisticated database implementation inside the PKCS #11 library is not required.

`C_GetSlotList`, `C_GetSlotInfo`, and `C_GetTokenInfo` provide the information required in the standard. The number of physical devices is completely transparent to the caller, as only a single slot and token is reported.

`C_GetMechanismList` and `C_GetMechanismInfo` provide information about the supported algorithms. Due to limitations in the HKS applet, the supported algorithms are plain ECDSA, ECDSA_SHA256, and SHA1_RSA_PKCS_PSS only.

Session handling is implemented using the functions `C_OpenSession`, `C_CloseSession`, `C_CloseAllSessions` and `C_GetSessionInfo`. The PoC implementation supports multithreading only when using multiple sessions, i.e., a session can only perform a single operation at a time. Starting multiple operations in separate threads but within the same session may lead to race conditions and undefined behavior.

`C_Login` and `C_Logout` set the authentication state information, but do not perform a real authentication procedure yet. The test setup is usable without any PIN-based authentication. Implementing it would be easy because a `VERIFY GLOBAL PIN` APDU is available, but it does not lead to additional findings in the PoC phase.

`C_FindObjectInit`, `C_FindObject`, and `C_FindObjectFinal` are implemented using a small subset of attributes. By restricting the implementation to the most common attributes and ignoring filter criteria for unknown attribute values, the implementation already works well enough with most applications. Covering all attributes defined by the

PKCS #11 standard adds much overhead without real benefit in the development phase. Before launching the library in a production environment, the implementation must be completed to cover the full specification.

Signature operations can be started through `C_SignInit` by using the object handles retrieved by `C_FindObject`. Calls to `C_Sign` and `C_SignFinal` complete the process. Multi-step signatures with the `C_SignUpdate` function are currently not supported because the HKS applet does not support it natively. Collecting the data to be signed within the library and sending it down to the applet in a single command would be possible, as long as it stays below the maximum length of extended length APDUs. The applications tested for this thesis did not make use of `C_SignUpdate`, and therefore, it was not implemented.

Generating new asymmetric key pairs is possible with `C_GenerateKeyPair`. The function can generate RSA and EC key pairs, depending on the parameters. Although the function `C_GenerateKeyPair` was not used directly in the durability test, its code was used to collect the data.

The `C_GetAttributeValue` function was added as one of the last during the development phase. Even though the function did not add benefit to the specifically designed pkcs11-helper tool, it is vital in case the PKCS #11 library is used with third-party applications like pkcs11-tool and p11tool. `C_GetAttributeValue` is the only way to retrieve information about object handles from the library. Therefore it is used excessively to filter objects. Implementing a complete `C_GetAttributeValue` function is time-consuming as it requires data from all over the code. Some data are available statically, but others must be retrieved dynamically from the tokens. The `C_GetAttributeValue` function covers the whole dataset known by the library.

A small example demonstrates the use of `C_GetAttributeValue`: Assume a program wants to access a certificate with the label "TEST". It calls the `C_FindObject*` functions with the appropriate filter template and receives a list of object handles. `C_GetAttributeValue` is now used to retrieve further information about the object, e.g., the certificate category `CKA_CERTIFICATE_CATEGORY` and certificate type `CKA_CERTIFICATE_TYPE`. The type determines whether the application understands the certificate if only X.509 certificates are supported. The certificate category information allows distinguishing the end-entity certificate from the CA certificate if certificate chains are stored under the same label. While all this could be already incorporated in a more or less cumbersome way during the `C_FindObjectInit` call, `C_GetAttributeValue` is mandatory to retrieve the actual certificate data under `CKA_VALUE`.

5.1.2 PC/SC Smart Card Communication Layer

Communication with smart cards and readers is handled via the PC/SC abstraction layer. The PC/SC communication layer builds upon the Windows smart card API (WinSCard), which is ported to Linux and Mac OS X by pcsc-lite. While there are small differences

between WinSCard and pcsc-lite as listed in the pcsc-lite API reference[2], the function type signatures are almost identical.

The PC/SC communication layer creates the bridge between the cluster orchestration logic and physical PC/SC compatible devices. This is achieved by only exposing four functions.

- `pcsc_available_elements` lists the available card readers. This function allocates memory and initializes it with information about the card readers. Connection establishment requires the returned structure.
- `pcsc_connect` establishes the connection to a reader. All connection and session information is encapsulated in a `pcsc_connection_t` structure. The structure shall not be interpreted by the caller but must be passed to subsequent function calls as a blob.
- `pcsc_send_apdu` is the primary function of interacting with applets. The function requires a valid connection and a C-APDU and returns an R-APDU. `pcsc_send_apdu` can be used for any applet and is not specific for the HKS applet. The caller of the function builds the C-APDUs and interprets the R-APDUs.
- `pcsc_disconnect` closes the connection to a reader and releases resources allocated by `pcsc_connect`.

5.1.3 FPGA Communication Layer

The FPGA and the SEs are mounted on a PCI-e card, as shown in Figure 5.3. In comparison to the PC/SC setup, a few additional layers are needed to communicate with the SEs from the cluster library. The following section describes the communication flow bottom-up.

SEs are constrained in resources and support only simple communication protocols like SWP and Serial Peripheral Interface (SPI). In our scenario, SPI is used. Orchestration of the SEs and translating the SPI to a standard computer interface, like PCI-e, is done by the FPGA. The FPGA handles the SEs as individual devices by assigning each SE a dedicated memory region. Another FPGA-based module, called SE Arbiter, handles asynchronous communication from the SEs when an operation has finished. The card is connected over the PCI-e bus and therefore appears to the host operating system as a standard PCI-e device, hiding any underlying structure.

Access from the operating system to the FPGA memory is possible by mapping the device into a region of the host's memory. The memory of the FPGA is separated into individual areas per SE, with each area being divided into two subareas for read and write access. Communicating with a specific SE, i.e., sending APDUs, is done by writing to a particular area of memory. Once the APDU is processed, the system receives an interrupt,

[2] https://pcsclite.apdu.fr/api/group___API.html#differences

and the response can be read from its specific memory region. Interrupt handling and direct memory access requires elevated privileges. Therefore, the access functionality is encapsulated in a Linux kernel module. Accessing the card from other operating systems is currently not possible, but could be implemented with some effort.

The kernel module exposes a character device accessible from userspace, behaving similarly to the mapped memory of the FPGA. Write access transmits the SE ID and APDU. Read access blocks until the response is ready. Specifying the SE ID during read access is done using the read buffer. Normal file operations, like open, read, write, and close, can be used to communicate with the SEs.

The SE communication layer of the SE cluster library uses this character device to access the SEs from userspace. Neither the user application nor the library needs elevated privileges to communicate with the cryptographic hardware if the device has appropriate access permissions set.

5.1.4 The pkcs11-test Helper Program

The pkcs11-test program is specifically built to invoke functions provided by the SE cluster library manually. It supports performance tests, durability tests with signing operations, and durability tests with key pair generations. Furthermore, the test runs can be configured with command-line arguments to use either RSA or EC keys, as well as a specific number of threads.

The program has changed drastically during development, and not all test runs can be reproduced in the latest version. The first versions of the program had hardcoded test-runs, which required a recompilation to modify the test cases. Once the basic library function calls were working, the functionality has been incorporated into more general functions.

The significance of the pkcs11-test tool decreased when the library offered all functions required by standard programs, like pkcs11-tool or p11tool.

5.1.5 Pitfalls of PKCS #11

This section gives a brief overview of the problems encountered during the implementation of the PKCS #11 library. The biggest challenge when developing the library was that PKCS #11 is an extensive specification, referencing to other even larger standards.

Data encoding was time-consuming in general, as the data format retrieved from the hardware may be different to what is expected by the PKCS #11 specification. The example of ECDSA signatures illustrates this. While many PKCS #11 attributes like `CKA_EC_POINT` and `CKA_EC_PARAMS` must be DER encoded, the signature must not be. ECDSA signatures retrieved from the HKS applet are DER-encoded. OpenSSL accepts DER-encoded signatures returned from `C_Sign` but will wrap them in another layer of DER encoding. The created ASN.1 structure is perfectly valid, but will not validate when processed with OpenSSL. OpenSSL and the PKCS #11 specification

demand that ECDSA signatures returned by `C_Sign` consist just of R and S, without any further encoding. Therefore, the `C_Sign` function must extract the R and S values of the DER-encoded response from the HKS applet.

Other pitfalls that were quite time-consuming in terms of troubleshooting are unsupported mechanisms or algorithms. Depending on the application it is possible to do the hashing of the input in either software or hardware. There is a limited set of only three mechanisms which are currently supported in the HKS applet, namely SHA256withECDSA, NONEwithECDSA, and SHA1withRSA/PSS. Therefore, multiple applications cannot use the SE cluster software at this point in time. A NONEwithRSA mechanism would add compatibility to many existing applications. Finding out which mechanisms are supported and used in the software and hardware that performs the hashing, and which library versions are required, took a considerable effort.

Compatibility is not only related to available mechanisms. Attributes provide many data values about the managed cryptographic keys. Depending on the application, e.g., p11tool, pkcs11-tool, and keytool, the amount and kind of attributes required to use a key are entirely different. Some applications retrieve as much information as possible, need a vast number of different attributes to work, and then perform the object filtering themselves. Others can operate with only the information marked as mandatory by the PKCS #11 standard but will provide additional information if available. Determining the requirements for a specific application involves tracing the library calls.

Some attributes provide static values that do not change during the lifetime of the library. Other values depend on the available hardware or may change at any time. Building a common data model for all attributes is hard and quickly leads to poor code quality. Even with a perfect data model, the function `C_GetAttributeValue` involves many lines of code due to the number of mandatory attributes and thereby becomes hard to maintain.

5.2 PoC Setup based on Smart Cards

The first PoC setup is designed to get the initial evaluation results of the cluster concept. Therefore it is based on smart cards, easy to build and reproduce, cheap, and still provides every functionality to emulate the final cluster setup.

The HKS applet is deployed on a JCOP v2.4.2 R2 smart card, the NXP J3D081. J3D081 supports RSA keys of up to 2048 bit and ECC GF(p) up to 320 bit. A detailed description of the J3D081 can be found in Subsection 5.2.1. While the applet only uses elliptic curve keys of 256 bit, generating and using the applet-supported RSA keys of 4096 bit is not possible due to hardware limitations. As smart cards are primarily designed to be used with user interaction, cryptographic operations shall be quick. Therefore RSA-4096 is still used seldom and is often not even implemented in the hardware, especially with older card models as the J3D081. In total, 17 smart cards of type J3D081 were available for tests. Some of them had been used in the past and were used for previous stages of

development and testing. Ten cards had never been used before and were selected for the durability test described in Chapter 6.

Other Java-based smart cards were tested, but the applet could not be deployed successfully due to version and card incompatibilities. Yagoba GmbH, the developer of the HKS applet, also provides a tool named YagoCardCreateTool to provision smart cards with the HKS applet. The provisioning tool loads the applet onto the card and personalizes the applet. Using the YagoCardCreateTool with TicTok 2.0, TicTok 3.0, or J2A040 cards failed with different error codes but all during the applet loading phase. The errors were likely caused by missing or mismatching dependency versions. J3D081 is a JCOP v2.4.2 R2 card with GlobalPlatform 2.2, which all others are not. The TicTok cards are shipped without some needed JCOP dependencies, and the J2A040 only comes with JCOP 2.4.1 R3.

Further testing revealed that it is indeed a version and dependency mismatch of the used card types. Other applets can be deployed successfully to these card types. Secured cards, i.e., cards that do not allow applet deployment due to their lifecycle phase, could be ruled out. Even cards coming directly from the chip factory, which were not even pre-personalized, did not accept the applet.

The smart cards used in this PoC setup are all read by identical readers, the IDBridge CT30, to guarantee comparable measurements. Tests with smart cards are carried out from a desktop PC running ArchLinux and a Raspberry Pi 3B+ running Raspbian. Using a 64-bit Intel processor and a 32-bit ARMv7 processor allows testing on different architectures. Figure 5.2 illustrates an example smart card setup that includes a USB 3.0 hub. This hardware setup was used during the tests on the desktop PC. The Manager and Worker setups were changed according to the test case. The setup with the Raspberry Pi is identical, but instead of using a USB hub to connect multiple readers, the readers must be connected directly to the Raspberry's USB ports. An explanation of why a USB hub cannot be used with the Raspberry is given in Subsection 6.2.2.

5.2.1 J3D081 and P5CD081

All performance and durability tests were carried out on J3D081 smart cards from NXP. J3D081 is based on a P5CD081 controller equipped with 80 KiB of EEPROM memory. NXP product names follow a strict naming convention, which the following list explains.

The controller name `P5CD081` is composed of [38, p.2]:

- `P5` defining the product family. P5 indicates a SmartMX controller, P60 stands for SmartMX2, and P71 refers to SmartMX3 controllers.
- `C` representing the category, which in this case, is a PKI controller with 3DES and AES co-processors.
- `D` identifying it as a dual interface controller, providing an ISO/IEC 7816 and ISO/IEC 14443 interface.

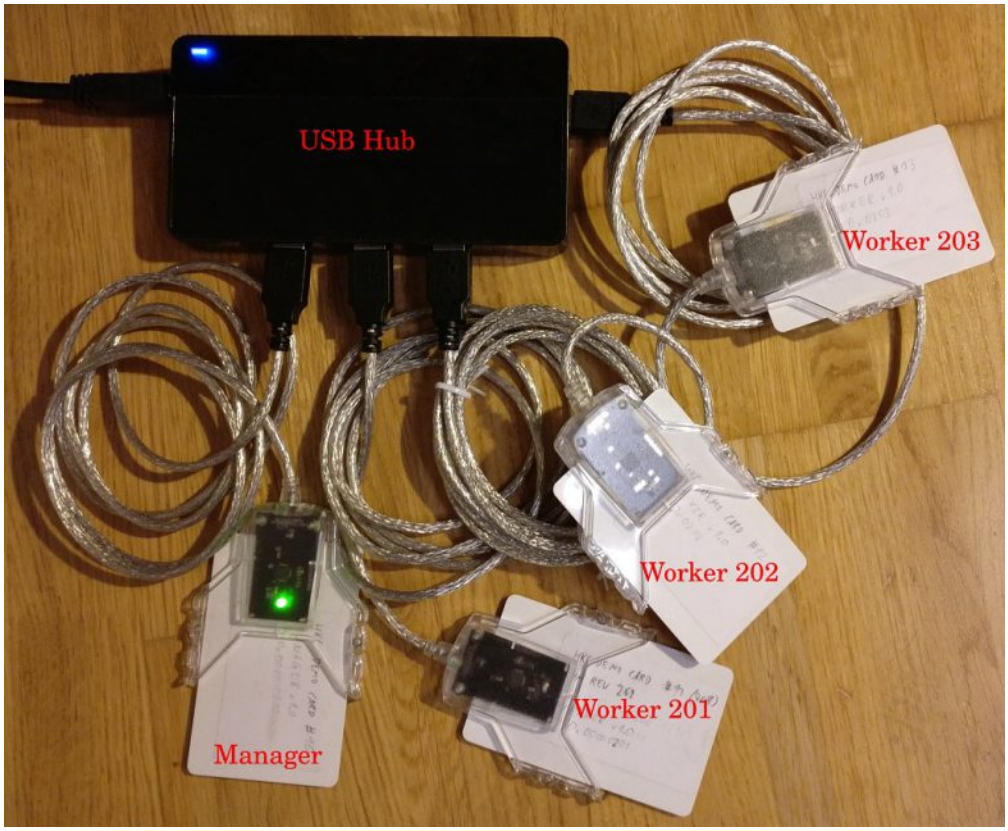

Figure 5.2: Photo of the smart card setup attached to a USB hub

- `081` defining the amount of memory but increases with further product options. The controller only provides 80 KiB of memory, but 081 was used to indicate the new P5Cx081 family.

According to the datasheet, the used non-volatile memory allows for at least 500 000 write cycles and holds data for at least 25 years at +55°C ambient temperature. The number of write cycles will be stressed particularly in Chapter 6.

JCOP cards also follow a naming convention [39, p.15]:

- `J` is a constant indicating a JCOP product.
- `3` defines CD hardware type, corresponding to CD in the controller name.
- `D` represents the JCOP version v2.4.2 R2.
- `081` refers to the amount of memory, corresponding to 081 in the controller name.

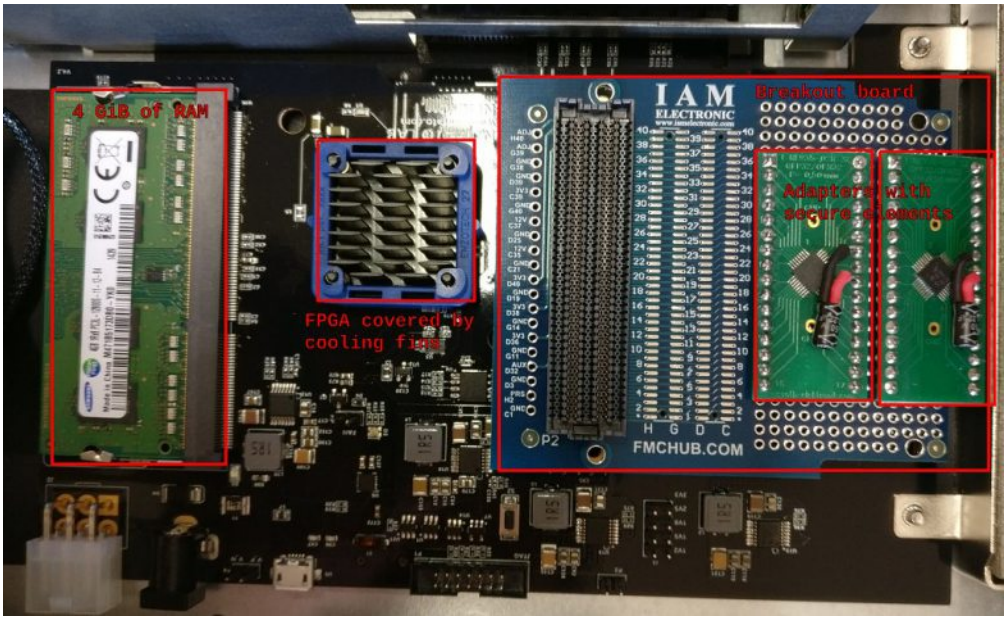

Figure 5.3: Photo of the PCI-e FPGA board installed in a server

5.3 Advanced PoC Setup based on Secure Elements

In the more advanced testing phase, the applet was deployed to P73 SEs from NXP connected over a Nereid Kintex 7 PCI-e board. The PCI-e card is designed for use in servers and workstation computers and equipped with an XC7K160T-FBG676 FPGA. A more detailed description of the FPGA communication stack is given in Subsection 5.1.3. These P73 SEs are based on the P71 series and are designed for automotive applications. They provide high reliability also in the field. Moreover, the SEs are based on a recent hardware and software platform, increasing chip performance and longevity. JCOP 4 and the P73 chip also allow using RSA with 4096-bit keys, which is not possible on the J3D081 cards.

The PCI-e board has the passively cooled FPGA directly mounted to it. A low pin count (or high pin count in future testing steps) breakout board is connected to the PCI-e card over a standard connector. The SEs are soldered to a QFP32/QFN32 adapter, which connects to the breakout board. A picture of the device taken during the first installment in a server is shown in Figure 5.3. Due to space limitations in the PCI-e slot of 1U servers, the number of SEs is very limited in the testing phase. The hand-soldered SEs on the adapter take up a significant amount of space on the breakout board and the PCI-e slot. The full adapter size allows for only two SEs when the card is installed in a server. Sawing off unused adapter pins increases this number to four. Future prototypes will reduce the size by using machine-soldered components. A final version will be even smaller in size if the SEs are connected directly to the board without any breakout boards or additional connectors.

5.4 Using Applications with the PoCs

This section is dedicated to giving a simple example to demonstrate that the cryptographic library with the standardized PKCS #11 interface can be used out-of-the-box with widespread applications. The idea is to set up a CA with hardware-backed key storage that works for both PoC scenarios.

For keeping the example illustrative, easy to reproduce, and simple to set up, the open-source software OpenSSL was chosen. OpenSSL is a tool-suite for many crypto-related tasks and the de-facto standard for handling certificates. While most, especially commercially operated, CAs provide additional features like a certificate database and certificate management protocols (in the simplest case) a CA can exist with just a key and certificate. In the following example, the key is assumed to have already been generated inside the hardware using the label TEST-0002. The standard configuration is enhanced with a few small sections, as shown in Listing 5.1, to use the SE cluster library with OpenSSL. The last part of the configuration specifies the path to the SE cluster library. After the configuration change is completed, the PKCS #11 engine of OpenSSL is ready to use. Listing 5.2 shows how the CA certificate is generated and a software-stored entity certificate can be generated. Both generated certificates are signed with ECDSA-SHA256 over the NIST P-256 curve. The full text-representation of the CA certificate and the entity certificate are given in the Appendix in Listing C.1 and Listing C.2.

By stripping the "-x509" parameter of the first command, one can generate a certificate signing request for a key accessed through PKCS #11. With this procedure, it is possible to have all keys belonging to the PKI stored inside the SE cluster.

Listing 5.1: Extension of the OpenSSL configuration for the PKCS #11 engine

```
openssl_conf = openssl_init

[openssl_init]
engines = engine_section

[engine_section]
pkcs11 = pkcs11_section

[pkcs11_section]
engine_id = pkcs11
MODULE_PATH = /src/secluster-pkcs11/build/linux/libsecluster-pkcs11-
    x86_64.so
```

Listing 5.2: OpenSSL commands to generate a certificate using PKCS #11

```
# Create the new CA certificate using a key inside the SE Cluster
# Remove the "-x509" to generate a CSR for an existing CA
openssl req -new -x509 -subj '/CN=Secure Element Cluster CA/O=TU Wien
    /ST=Vienna/C=AT/' -sha256 -config pkcs11.cnf -engine pkcs11 -
    keyform engine -key 'pkcs11:object=TEST-0002;type=private' -out ca
    .crt

# Print the CA certificate (result is shown in listing below)
openssl x509 -in ca.crt -noout -text

# Generate a new key pair and certificate request for some end entity
openssl req -nodes -new -newkey ec:<(openssl ecparam -name prime256v1
    ) -sha256 -out entity.csr -subj '/CN=Some Entity/O=TU Wien/L=
    Vienna/C=AT/'

# Issue a valid certificate using the CA key in the Secure Element
    Cluster
OPENSSL_CONF=pkcs11.cnf openssl x509 -req -CAkeyform engine -engine
    pkcs11 -in entity.csr -CA ca.crt -CAkey 'pkcs11:object=TEST-0002;
    type=private' -set_serial 1 -sha256 -out entity.crt

# Print the entity certificate (result is shown in listing below)
openssl x509 -in entity.crt -noout -text
```

In addition to OpenSSL, the SE cluster library can be used with PKCS #11 specific tools like p11tool, pkcs11-tool, and any generic software based on the PKCS #11 standard.

As the library is at the time of publication in a proof-of-concept state and does not support every feature mentioned in the PKCS #11 specification, some tools may not behave as expected and need adaption in the source code. Furthermore, not all software products implement the PKCS #11 standard correctly, as some make incorrect assumptions requiring special treatment in the PKCS #11 library.

CHAPTER 6

Evaluations of the Cluster Setup and Recommendations

Performance and durability tests were carried out to evaluate the benefits and scalability of the SE cluster. The following sections describe the testing setups and methods used. Performance results were obtained for the smart card and SE PoC implementation. As the SE PoC is a unique and costly prototype, it could not be used for durability testing because the test is destructive. Finally, the chapter presents a cluster sizing recommendation based on the evaluation results and gives dimensioning examples for known use cases.

6.1 Performance Testing of the Cluster Setup

An important step in evaluating the usefulness and practicability of the cluster setup is testing its performance and scalability capabilities. In the first development stage, the performance of a cluster built from smart cards was evaluated. At the end of the project, the same test was repeated for the SE cluster.

6.1.1 Performance Analysis of Smart Cards

A hardware setup, as described in Chapter 5, was chosen to analyze the performance of a smart card cluster. By varying the number of J3D081 smart cards connected to a Desktop PC using IDBridge CT-30 readers, multiple cluster sizes were simulated. As expected, no load-spikes could be observed on the PC hardware during the test. Therefore, the observed performance limits of the cluster solely depend on smart card performance.

The performance tests were done in several variations, based on a combinatorial combination of the following configurations:

- Tests were carried out with one, three, and six smart cards.
- Tests were performed with NIST P-256, RSA-1024, and RSA-2048 key pairs.
- Tests were performed with different number of threads, depending on the number of smart cards.
- Each test case was repeated three times to average the results.
- Each test run included exactly 108 signature operations, which could be divided for one, three, four, six, and twelve threads equally well.

A single key was created and then copied over to the other smart cards to eliminate any performance variations due to different keys. This was possible for NIST P-256 and RSA-1024 keys, but for unknown reasons, RSA-2048 keys could not be imported. Therefore, multiple RSA-2048 key pairs had to be generated. Nevertheless, it is expected that the used key has no impact on performance, as otherwise a timing side-channel would leak information about the used key. No correlation between timing differences and whether a key was synchronized could be observed. RSA-4096 could not be tested because the used smart card controller P5CD081 does not support it.

Table 6.1 shows an excerpt of the performance values on J3D081. The exact measurements are given in the Appendix. Tests on NIST P-256 keys were performed using `CKM_ECDSA_SHA256`, while experiments with RSA-1024 and RSA-2048 keys used `CKM_SHA1_RSA_PKCS_PSS`.

The efficiency column gives a measurement of the speed of a single card in the setup as a percentage of the speed in the single card measurement. For example, in the NIST P-256 test cases, the single card and single thread measurement yielded 8.66 signature operations per second. When ignoring any scheduling, synchronization, and other overhead, an ideal setup of three cards should provide 25.98 signatures per second. However, only about 25.2 signatures per second could be measured. Therefore 0.78 signatures per second are wasted due to management overhead, yielding an efficiency of 97%. It is easy to see from the measurements that efficiency increases with longer operation duration, i.e., RSA-2048 signatures are a lot slower than signatures using the elliptic curve key. Therefore, more time is spent during the signature in the card, making the overhead in software orchestration negligible.

As long as enough threads are available to utilize all available smart cards, the number of threads has no impact on the cluster performance. Because each signature operation needs to be prepared in software before it can be executed on the hardware platform, the idea was that an excess of threads could improve the performance as the signature operation can already be prepared when waiting for an available device. If the computer hardware is powerful enough - and most computers are, compared to the performance of smart cards - the signature preparation time is negligible. However, if the number of threads is in a reasonable limit, the excess of threads does not decrease the overall performance due to additional overhead either.

Key type	Threads	Cards	Duration (μs)	Sign./sec ($\Theta_{\sigma_{max}}$)	Efficiency (η_σ)
P-256	1	1	12,465,489	8.66	1.000
P-256	6	1	12,470,358	8.66	1.000
P-256	1	3	12,784,112	8.45	0.325
P-256	3	3	4,282,462	25.22	0.970
P-256	6	3	4,285,676	25.20	0.970
P-256	1	6	12,707,262	8.50	0.163
P-256	6	6	2,140,937	50.45	0.970
P-256	12	6	2,140,398	50.46	0.971
RSA-1024	1	1	22,150,540	4.88	1.000
RSA-1024	6	1	22,150,022	4.88	1.000
RSA-1024	1	3	22,565,141	4.79	0.327
RSA-1024	3	3	7,548,832	14.31	0.978
RSA-1024	6	3	7,548,459	14.31	0.978
RSA-1024	1	6	22,465,456	4.81	0.164
RSA-1024	6	6	3,773,917	28.62	0.978
RSA-1024	12	6	3,775,647	28.60	0.978
RSA-2048	1	1	78,450,802	1.38	1.000
RSA-2048	6	1	78,440,459	1.38	1.000
RSA-2048	1	3	78,911,255	1.37	0.331
RSA-2048	3	3	26,370,253	4.10	0.992
RSA-2048	6	3	26,369,237	4.10	0.992
RSA-2048	1	6	78,808,467	1.37	0.166
RSA-2048	6	6	13,179,868	8.19	0.992
RSA-2048	12	6	13,183,329	8.19	0.992

Table 6.1: Performance measurements on J3D081 smart cards

The detailed results in the Appendix further include columns about the number of operations per card and second, the speedup factor, and the duration of each single test run. The operations per card and second decrease slightly if more than one card is used and correlates perfectly to the efficiency value. The speedup factor states how many times faster this test run was than the baseline. The baseline is the average of the test runs using the same algorithm with exactly one card and thread. Because the multithreading setups are not 100% efficient, the speedup factor is slightly smaller than the number of cards. However, the speedup is surprisingly close to the number of cards. The duration of each test run is measured in microseconds. The measurement started after the hardware was initialized and all threads were set up, and ended after the last thread completed its last signature operation. This gives the closest approximation of signature performance possible.

6.1.2 Performance Analysis of Secure Elements

The test described in Subsection 6.1.1 was repeated using the FPGA PoC setup. Instead of up to six crypto-processors available for testing as in the previous configuration, the board was equipped with only four SEs due to physical size limitations. Therefore, the tests were carried out using one, three, and four of these elements. Table 6.2 shows the performance results. The column description is identical to the one given in Subsection 6.1.1. A more detailed result listing which includes all single test runs can be found in the Appendix.

Table 6.2 clearly shows that the P73 SEs are much faster than the J3D081 smart cards, especially when comparing RSA signature throughput. This signature throughput value is based on the average duration and the number of signatures. The detailed timing information of the three test runs given in Table A.4 shows how the average duration is computed. Notably, the performance was influenced more clearly by the number of threads, than it was in the smart card setup. These performance variations also have a direct influence on efficiency. Overall the efficiency was lower than in the smart card tests, i.e., adding more SEs did not increase the performance as much as expected. Multiple reasons caused a decrease in efficiency.

Firstly, the FPGA software and driver stack are still under heavy development. The main goal of the SE prototype was to demonstrate the functionality, rather than being as fast as possible. Currently, the driver contains debugging outputs and idle times for synchronization.

Secondly, the communication between the kernel module and the FPGA is encoded in T=1 frames. These frames are not only relatively small, requiring the split of larger signatures into multiple frames, but a minimum delay between those frames is necessary. The synchronization for those frames is not yet fine-tuned enough, causing an unnecessarily long waiting period at some times and losing frame data at other times. When a response is lost, the APDU has to be repeated. From the measurements taken, it can be assumed that the waiting times slow down ECDSA signatures, making them just as fast as RSA-1024. RSA-2048, on the other hand, must be split into two frames and has a high number of failed responses.

Thirdly, although the event notification is interrupt-based, a busy-wait strategy is used to synchronize with the T=1 frames in some cases. Repeatedly checking if the response is ready yet causes a high CPU load. Furthermore, this slows down the signatures, especially when waiting for multiple elements at the same time.

Fourthly, the SPI communication between the FPGA and the SEs currently operates only at half of the possible clock speed. Increasing the data transfer speed also increases the overall APDU throughput.

Key type	Threads	SEs	Duration (μs)	Sign./sec ($\Theta_{\sigma_{max}}$)	Efficiency (η_σ)
P-256	1	1	8,736,079	12.36	1.000
P-256	6	1	8,767,022	12.32	0.996
P-256	1	3	7,989,657	13.52	0.364
P-256	3	3	3,630,740	29.75	0.802
P-256	6	3	3,451,439	31.29	0.844
P-256	1	4	7,976,281	13.54	0.274
P-256	4	4	3,158,766	34.19	0.691
P-256	8	4	2,400,802	44.98	0.910
RSA-1024	1	1	8,672,496	12.45	1.000
RSA-1024	6	1	8,756,582	12.33	0.990
RSA-1024	1	3	8,686,727	12.43	0.333
RSA-1024	3	3	3,608,981	29.93	0.801
RSA-1024	6	3	3,788,586	28.51	0.763
RSA-1024	1	4	8,628,603	12.52	0.251
RSA-1024	4	4	3,442,260	31.37	0.630
RSA-1024	8	4	2,575,746	41.93	0.842
RSA-2048	1	1	14,842,856	7.28	1.000
RSA-2048	6	1	14,909,649	7.24	0.996
RSA-2048	1	3	15,040,595	7.18	0.329
RSA-2048	3	3	5,318,006	20.31	0.930
RSA-2048	6	3	5,143,407	21.00	0.962
RSA-2048	1	4	14,821,513	7.29	0.250
RSA-2048	4	4	5,134,017	21.04	0.723
RSA-2048	8	4	4,081,429	26.46	0.909

Table 6.2: Performance measurements on P73 SEs

6.2 Durability Testing of the Cluster Setup

Tests have been carried out to get a good understanding of the durability and longevity of smart cards, SEs, and the whole cluster. The durability has been tested with regard to signing and key generation operations by using the setup described in Chapter 5. As no FPGA implementation was available for extensive durability testing, the following tests were performed on smart cards only. In general, the controllers used in smart cards and SEs are similar or even identical, and therefore the test results should be valid for the FPGA implementation as well. It is reasonable to assume that for other differences in setup, like the replacement of USB card readers with a PCI-e connected FPGA board, it does not affect the durability of the setup. The card readers and the FPGA do not see wear to a similar extent as the card's non-volatile memory does. This assumption is justified by the findings of Subsection 6.2.2.

6.2.1 Durability of Smart Cards During Extensive Signing

In order to get reproducible results from experimental testing it is important to get a clear understanding of the initial state. When testing the durability of things, it is important to know how long and in what manner they have been stressed before starting the actual test.

The test setup was built from three new J3D081 smart cards. The state of the smart cards has been verified logically and physically before starting the test. All of the smart cards had default GlobalPlatform keys configured, and no applet had been deployed on the cards. This part concludes the logical test. A physical inspection of the card, especially the contact pads, did not show any fingerprints or scratches. If the card had been used before, the card reader would have left slight scratch marks on the contact pads. All smart cards could be obtained free of charge because there are minor imperfections related to their print. Therefore, they could not be sold. Apart from the printing, the cards were in perfect condition, which was verified by the logical and physical inspection.

After selecting three smart cards for the test, they have been initialized with the HKS applet. The applet IDs were set to `0x000001001`, `0x000001002`, and `0x000001003` respectively. The applet was deployed using a Worker configuration to create keys and signatures. Before starting the test run, each card had the global PIN disabled and a dummy EC key pair generated on handle 0. The initialization procedure was completed with a test signature using ECDSA SHA-256. Listing 6.1 gives the complete APDU trace.

Listing 6.1: Applet initialization for durability testing

```
# Connect to 8 byte AID F011223344100100
00A4040008F011223344100100

# Verify Global PIN 123456
8016000003123456

# Disable the Global PIN
801700000403123456

# Allocate space for 1 ECC key pair on handle 0
80200000020000

# Set key label for handle 0 to "TH ECC test"
802900000D0000544820454343207465737 4

# Generate ECC key on handle 0
80210000020000

# Allow ECDSA SHA-256 and plain for handle 0
802B00000400000003

# Sign the message "test" with handle 0 ECDSA SHA-256
8030000006000074657373400
```

The three smart cards were connected to a Raspberry Pi 3B+ using IDBridge CT-30 readers, as mentioned in Chapter 5. The pkcs11-test program was started with four PKCS #11 sessions and program threads, each running signing operations continuously to give the highest load possible to the cards. An additional fourth thread was chosen because each signing operation had overhead in setup. This fourth thread could prepare everything and jump right into signing as soon as another thread released the lock on the card. It might have been only a very minor performance improvement, but it did not influence the test outcome nonetheless.

As the current PKCS #11 implementation only saves the number of executed signing operations on a clean exit, a crash of the test program would have caused a complete loss of the test results, and the cards could no longer be used for durability testing. To prevent this, the implementation was modified to log the status word of every sent APDU immediately. The additional information allowed for detecting failing cards immediately and recovering the number of completed signing operations in case of a crash.

The test ran for a total of 19 days with a few planned restarts to rotate the log files and save the signing operation counter. No crashes occurred during the whole testing period, and all cards survived. When the test was stopped, the smart cards had performed about 13 million ECDSA SHA-256 signatures each. The exact number of successfully performed signing operations is documented in Table 6.3. All C-APDUs sent to the card were answered with a status code of 0x9000, indicating successful completion.

HKS applet ID	Card type	ECDSA SHA-256 signatures
0x00001001	J3D081	13,099,933
0x00001002	J3D081	12,899,094
0x00001003	J3D081	12,808,801
0x00001007	J3D081	>35,000,000
0x00001008	J3D081	>35,000,000
0x00001009	J3D081	>35,000,000

Table 6.3: Successfully performed ECDSA SHA-256 signatures per card

It was decided to stop the test because the smart cards did not show any degradation in performance or reliability. Additionally, it makes sense that signing does not greatly influence the lifetime of a smart card controller. The lifetime of a controller is mostly limited by the underlying EEPROM or flash memory, and any reasonable implementation of the EC signing operation does not require write access to non-volatile memory. This assumption was confirmed, when testing the durability of smart cards with key generations.

A second test for the longevity of smart cards during signing operations was started after all other tests had been completed. The HKS applet was deployed on fresh cards with IDs `0x000001007`, `0x000001008`, and `0x000001009`. This time the test was not interrupted or restarted to see if the software layer can deal with a large number of

operations as well. The test had already been running for two months by the time this thesis was finished, and each card had generated over 35 million signatures. It seems smart cards can handle an extensive amount of signing operations well, as well as operate reliably also when powered on for several weeks. Furthermore, the software seems to be able to run stable without any incidents. The test was not stopped deliberately at the time this thesis was being finished to get an even better understanding of the reliability and durability of smart cards. The smart cards will keep generating signatures until they fail, more modern hardware supersedes the test setup, or other tests need the devices.

6.2.2 Durability of Smart Cards During Extensive Key Generations

The setup for testing the durability of smart cards through the repeated generation of key pairs is similar to the setup used when testing repeated signature operations. Six J3D081 smart cards were connected to the PC using IDBridge CT-30 readers. The first three smart cards from the previous signing durability test run were complemented with three new cards. This setup allows for identifying any wear caused by the previous test run. This time the readers were not connected to a Raspberry Pi 3B+ because it provides only 4 USB ports and no suitable hub was available. An available 10-port USB hub could not be used with the Raspberry Pi 3B+ due to limits on the USB OTG implementation of the Pi. The Raspberry Pi supports a chain of at most three steps to the end device, i.e., the smart card reader. The 10-port hub consists of three 4-port chips, which are combined using another 4-port chip. Together with the USB root hub of the Pi, this already forms a chain of three, and no USB smart card reader could be detected. Attaching the USB hub to a desktop PC instead of the Pi did not influence the experiment, but the power consumption was higher.

As the test run with signature operations had been stopped before any card had failed, the idea was that this second test gives an idea of the wear caused by the first test. If the signature operations caused considerable wear on the card's memory, they fail faster than the new cards. However, it must be noted that this approach allows detecting the wear of the first test only in certain cases. For example, if the key generation and signature operation write to the same memory block or if both operations create objects in the same memory area. In case both operations perform writes to the memory but use distinct memory areas, the wear of the signature operations remains undetected.

The detailed results are given in Table 6.4 and show how many EC NIST P-256 key pairs could be generated until the smart card controller failed. Surprisingly, each smart card failed within just four days. Also, there is no indication that the cards which had previously generated about 13 million signatures failed earlier than the others. The Pearson correlation coefficient of 0.152 between the generated signatures and key pairs gives evidence of the presumption. However, it is noteworthy that the coefficient gives only an indication of the correlation when applied to small sample sizes.

Each generation of a new key pair included the following subtasks:

- `ALLOCATE KEY` is used to allocate storage space for the key (pair). The key type and size are already specified in this command.
- `GENERATE KEY` is used to generate the key (pair) and store it in the previously allocated memory space.
- `DESIGNATE KEY` sets the allowed cryptographic operations on the key, e.g., allow signing. Without key designation, the generated key cannot be used.
- `DELETE KEY` frees the allocated key handle.
- `RUN GC` is an optional subtask and was only executed when required. It triggers the garbage collection procedure and frees up the maximum amount of persistent memory. A run of the garbage collector was necessary if `ALLOCATE KEY` fails with the status word `0x6F75`.

The subtasks were selected as any real-world key generation requires them. If only `GENERATE KEY` is called repeatedly, the measured numbers are expected to be higher. A particularly bad result was observed for card `0x00001005`, which had only performed a single signature during initialization and already failed after 61,297 EC NIST P-256 key pair generations. Unfortunately, it is not possible to diagnose a failed smart card further because all C-APDUs fail, as shown in Listing B.1. This behavior is identical across all failed smart cards in the test.

As no implementation details about the smart card memory management and HKS applet are known, the following is a reasonable assumption on the memory layout. Each applet can have multiple allocated key handles, which can be chosen from 0 to N-1, where N is a constant picked during applet initialization. It is required to store their state (allocated or not, valid key available or not) and their key designation to manage these key handles. It makes sense to store these data in a central index or directly in the memory area assigned to the key handle. Each of the operations carried-out, `ALLOCATE KEY`, `GENERATE KEY`, `DESIGNATE KEY`, and `DELETE KEY` causes an update on said memory block. This yields a total of 600,000 to 950,000 write operations (without accounting for possible writes due to the garbage collection) on a single memory cell, except for `0x00001005`, which was significantly lower.

The NXP datasheet of the underlying controller P5CD081 guarantees a total number of 500,000 write operations [38]. This target was met by five out of six cards with more or less safety margin. As all six cards were treated identically and stored in the same packaging before use, it is reasonable to assume the test setup did not cause premature failure.

HKS applet ID	Card type	ECDSA SHA-256 signatures	Generated key pairs
0x00001001	J3D081	13,099,933	149,630
0x00001002	J3D081	12,899,094	202,030
0x00001003	J3D081	12,808,801	170,517
0x00001004	J3D081	1	174,967
0x00001005	J3D081	1	61,297
0x00001006	J3D081	1	235,928

Table 6.4: Generated EC NIST P-256 key pairs per card

Variable	Description
$\theta_{\sigma_{max}}$	required signature throughput per second at peaks
$\theta_{\sigma_{avg}}$	expected signatures per second, averaged over the whole lifetime
η_{σ}	efficiency of the cluster during signature operations
$\theta_{\kappa_{max}}$	required key generation throughput per second at peaks
$\theta_{\kappa_{avg}}$	expected key generations per second, averaged over the whole lifetime
η_{κ}	efficiency of the cluster during key generation operations
τ	required lifetime of the cluster in years
α	safety margin to configure oversizing, must be >1
$\Theta_{\sigma_{max}}$	maximum possible signature throughput per second
$\Theta_{\kappa_{max}}$	maximum possible key generation throughput per second
Σ	expected number of signatures until the SE fails
K	expected number of key generations until the SE fails
N_{σ}	number of SEs required due to signature lifetime constraints
N_{κ}	number of SEs required due to key generation lifetime constraints
$N_{\sigma\kappa}$	number of SEs required due to performance constraints
N	total number of SEs to satisfy the requirements

Table 6.5: Variables of the cluster sizing formula in Equation (6.4)

6.3 Recommendation for Sizing of the Cluster

The findings of the previous sections allow for deriving a formula to size a cluster adequately. The formula contains variables describing the requirements on the whole cluster and properties of single SEs. Requirement variables are denoted by lowercase Greek letters. Uppercase Greek letters indicate hardware properties.

Table 6.5 lists and describes the variables used in the cluster sizing formula in Equation (6.4).

Equation (6.4) is composed of subexpressions described in Equations (6.1), (6.2), and (6.3). Equation (6.1) represents the number of required SEs to fulfill the lifetime requirements on signatures. This is the minimal amount of SEs such that the cluster survives the expected number of signatures during its whole lifetime. Analogously, Equation (6.2)

Controller	Algorithm	$\Theta_{\sigma_{max}}$	$\Theta_{\kappa_{max}}$	Σ	K	η_σ
P5CD081	P-256 SHA256	8.66	0.73[1]	>35,000,000	165,728	0.970
P5CD081	RSA-1024 SHA1	4.88	0.32[1]	>35,000,000[2]	165,728[2]	0.978
P5CD081	RSA-2048 SHA1	1.38	0.046[1]	>35,000,000[2]	165,728[2]	0.992
P73	P-256 SHA256	12.36	1.46[2]	7,000,000,000[2]	33,145,600[2]	0.910
P73	RSA-1024 SHA1	12.45	0.64[2]	7,000,000,000[2]	33,145,600[2]	0.842
P73	RSA-2048 SHA1	7.28	2.92[2]	7,000,000,000[2]	33,145,600[2]	0.909

Table 6.6: Known property values of a single chip for cluster sizing

specifies the number of SEs required for key generation operations. Equation (6.3) describes the number of elements required when the performance peaks of signatures and key generations occur at the same time. This is the upper bound of the performance requirement. Equation (6.3) also respects the cluster scaling behavior, i.e., adding more elements is not perfectly efficient. The cluster efficiency factors for multiple algorithms are given in Table 6.1. Finally, Equation (6.4) combines the results and adds a safety margin α, e.g., adding 20% more SEs to counteract premature failures.

As the resulting number of required SEs is heavily dependent on the used chip controller as well as the key and signature algorithms, it is important to assign accurate values to the property variables. Example values gathered during performance and durability testing can be found in Table 6.6.

$$N_\sigma = \left\lceil \frac{\theta_{\sigma_{avg}} * \tau * 365.25 * 24 * 3600}{\Sigma} \right\rceil \tag{6.1}$$

$$N_\kappa = \left\lceil \frac{\theta_{\kappa_{avg}} * \tau * 365.25 * 24 * 3600}{K} \right\rceil \tag{6.2}$$

$$N_{\sigma\kappa} = \left\lceil \frac{\theta_{\sigma_{max}}}{\Theta_{\sigma_{max}} * \eta_\sigma} + \frac{\theta_{\kappa_{max}}}{\Theta_{\kappa_{max}} * \eta_\kappa} \right\rceil \tag{6.3}$$

$$N = \left\lceil max\left(N_\sigma, N_\kappa, N_{\sigma\kappa}\right) * \alpha \right\rceil \tag{6.4}$$

6.4 Dimensioning Examples for Known Use Cases

The formula introduced in the previous section can be used to estimate the number of required smart cards and SEs for known use cases. Example scenarios and their assumed performance requirements are given in Table 6.7. The performance requirements are

[1] averaged from durability test
[2] estimated

Application	$\theta_{\sigma_{max}}$	$\theta_{\sigma_{avg}}$	$\theta_{\kappa_{max}}$	$\theta_{\kappa_{avg}}$	τ	α
Small CA	1	0.000012	0.2	0.0000001	10	1.2
Medium CA	5	0.5	0.2	0.00001	10	1.2
Large CA	15	4	1	0.1	10	1.5
Medium OCSP	150	25	0.017	0.000000032	5	1.4
Large OCSP	1000	167	0.017	0.000000032	5	1.4

Table 6.7: Requirements for typical use cases

based on typical values and experience gained when working with these kinds of systems. An estimation of how many chips are required to satisfy the needs is given in Table 6.8 for smart cards and Table 6.9 for SEs. The real values of Σ and K for RSA-2048 on smart cards are unknown but have been assumed to be equivalent to P-256. For systems performing a massive number of signatures, e.g., OCSP servers, the assumed lifetime of at most 35,000,000 signatures is a significant limitation. In reality, Σ for P-256 is likely to be much higher, and therefore N_σ is much lower. This, as a result, lowers N drastically for P-256. If the P5CD081 controller can create 100,000,000 signatures instead of the assumed 35,000,000, N drops down to 370 for the "Large OCSP" scenario.

Because the P73 SEs are not publicly sold yet, no datasheet about the used memory could be found. A similar SE, the SE050, is designed for use in IoT, and its flash memory sustains 100 million write cycles. Assuming the memory of the P73 is similar, it is 200 times more durable than the EEPROM found in the P5CD081 smart card controller. This, on average, would allow for the generation of 33,145,600 EC keys or 7,000,000,000 ECDSA signatures based on the estimations from the smart card durability test. The estimates can be found in Table 6.6.

It must be noted that a configuration with the calculated number of SEs does not lead to a perfectly suited system for all use cases. Not all required properties are respected with the developed formula or do scale linearly. A perfect witness of poor scaling is the duration of a single signature. If a single signature takes 100 milliseconds, it will still take 100 milliseconds when doubling the number of SEs. In this case, adding more hardware can increase the throughput in a given time, but the individual signature is not sped up. This may not satisfy the latency requirements for some protocols like the Online Certificate Status Protocol (OCSP).

Application	Controller	Key Type	N_σ	N_κ	$N_{\sigma\kappa}$	N
Small CA	P5CD081	P-256	1	1	1	2
Small CA	P5CD081	RSA-2048	1	1	6	8
Medium CA	P5CD081	P-256	5	1	1	6
Medium CA	P5CD081	RSA-2048	5	1	9	11
Large CA	P5CD081	P-256	37	191	4	287
Large CA	P5CD081	RSA-2048	37	191	34	287
Medium OCSP	P5CD081	P-256	113	1	19	159
Medium OCSP	P5CD081	RSA-2048	113	1	112	159
Large OCSP	P5CD081	P-256	753	1	123	1055
Large OCSP	P5CD081	RSA-2048	753	1	742	1055

Table 6.8: Required number of smart cards for known use cases

Application	Controller	Key Type	N_σ	N_κ	$N_{\sigma\kappa}$	N
Small CA	P73	P-256	1	1	1	2
Small CA	P73	RSA-2048	1	1	1	2
Medium CA	P73	P-256	1	1	1	2
Medium CA	P73	RSA-2048	1	1	1	2
Large CA	P73	P-256	1	1	2	3
Large CA	P73	RSA-2048	1	1	3	5
Medium OCSP	P73	P-256	1	1	14	20
Medium OCSP	P73	RSA-2048	1	1	26	37
Large OCSP	P73	P-256	4	1	89	125
Large OCSP	P73	RSA-2048	4	1	170	238

Table 6.9: Required number of secure elements for known use cases

CHAPTER 7

Discussion

It was the goal of this thesis, to propose a cluster setup based on SEs or smart cards as well as analyze it regarding performance and durability. This chapter reflects on the cluster setup in general and the findings of the analysis of the different configurations. Furthermore, it gives an outlook on possible future research topics based on open questions.

7.1 Reflection on the Prototypes

The results of this research show that the SE cluster is an excellent possible alternative to existing solutions based on single SEs or overly powerful HSM. The tests described in Chapter 6 showed that the cluster scales almost linearly with a factor close to 1 in relation to the number of smart cards. It was reasonable to already assume good scalability before performing the tests because the overhead of managing multiple smart cards or SEs is relatively small compared to the time it takes to generate a signature. The transfer of the result via slow connections, as they are used for smart cards, increases the time additionally.

Further, it was expected that computationally complex algorithms, as RSA-2048, scale better than algorithms based on elliptic curves. However, all the tested algorithms showed efficiency values of over 97% when using smart cards. This result is better than expected. The most surprising finding is that scaling from 1 to 3 cards decreases efficiency, but a further increase from 3 to 6 cards has no impact. The performance penalty is found to be unclear, but the result gives hope that the setup can scale up to dozens or hundreds of cryptographic processors.

The scalability of the FPGA setup was worse than anticipated after the smart card tests. However, the issue arises from the unstable implementation and is most likely not related to the architecture. Nevertheless, even the inefficient SE prototype is faster

than the perfectly scaled smart card setup. As Yagoba GmbH built the FPGA-based hardware and the corresponding driver stack, only minor tweaks were done by the author of this thesis. Adaptions include the interrupt handling, that was changed from exclusive interrupts to shared interrupts, but fixing the stability issues was beyond the scope of the thesis. The performance of a single SE is astonishingly good, especially for RSA-2048 signatures. While the measured performance of ECDSA signatures does not show an equally significant improvement, it suggests that there is room for improvement in the prototype implementation. Optimizing the driver software and solving the timing issues during T=1 frame decoding may improve the measured performance considerably.

Another significant advantage of the secure elements is their durability. The datasheet of similar SEs proof this, even though no durability test could be carried out on the SE prototype. While the smart card of the experiment was based on EEPROM supporting up to 500,000 write cycles, modern SEs are equipped with flash memory lasting up to 100,000,000 write cycles. The improved memory type allows for the use in key-generating CAs.

An interesting finding which was not particularly searched for during the performance analysis is that different devices based on the same smart card controller type show different timings also when using the same key. The difference is minimal, in the range of a few milliseconds, but could be measured repeatedly and consistently. In the tests, the same smart card readers, connected to the same USB port, with identical smart cards and transfer speeds, were used. It is unclear where this timing difference comes from, but it does not impact the findings of the thesis further. A similar observation could not be made when testing SEs due to the unoptimized kernel module. Multiple test runs on the same element varied so strongly that such small differences between elements could not be measured.

7.2 Future Work

This thesis and its results show that more research and development on the topic is required to bring the proposed SE cluster into a production-ready state. The tests done in this thesis give an impression on the scaling behavior of the cluster, but tests using more than a handful of cards or elements are necessary. Unfortunately, due to the cost and time limitations, it was not possible to test an SE cluster with hundreds of elements or smart cards. While the tested configurations with up to six smart cards are already perfectly useful in IoT environments and can compete with existing technology well, for challenging HSMs, like the Luna series of Thales, a larger number of cryptographic elements is required.

No longevity test for smart cards that generate RSA keys repeatedly was performed because no further smart cards were available that could be destroyed. All smart cards available for the thesis are required in new tests.

Comparing the durability during EC and RSA key generations will give a better insight

into the internal memory management of smart cards and their life expectancy. Additionally, the tests shall be repeated with modern smart cards to compare with the result of the used J3D081. It is expected that newer card generations deliver better performance and are more durable. The results of modern smart cards can also be compared better with the FPGA setup due to the newer P73 SEs.

Software improvements are necessary on all layers. The HKS applet currently supports only a minimal number of key types and signature algorithms. In order to use the cluster with existing products, e.g., OpenSSL, a raw RSA implementation is required, however only SHA-1 RSA PKCS PSS is available. Support for encryption, decryption, and symmetric keys is desired. The PKCS #11 library already supports multiple HKS Workers and can evenly distribute the load among them, but does not implement a full cluster behavior yet. For a feature-complete implementation, at least HKS Managers must be supported, and key material must be transparently synchronized on creation. Furthermore, the cluster must implement auto-repair to be able to replace failed smart cards easily. Implementation of this functionality requires some effort. This was not strictly necessary for the tests described in this thesis, and may become superfluous with the next point.

All cluster functionality should be moved from the software to a hardware implementation. Abstracting the single SEs already in the FPGA makes the device more portable and secure. Especially state management, i.e., the element.db, shall be done directly in the PCI-e card. Otherwise, removing the card and placing it into another server may produce inconsistent states. Cluster orchestration currently implemented in the PKCS #11 library can be moved inside the FPGA, the PKCS #11 interface for applications remains unchanged. However, before any functionality can be pushed down into lower layers, the existing communication stack must be improved. The HKS kernel module is currently unstable and crashes unexpectedly, resulting in kernel panics. Moreover, the APDU exchange is unreliable and causes performance penalties by requiring retransmissions.

Once all the points mentioned above are implemented, tests in a small production environment are possible. Using the SE cluster in real-world scenarios will uncover new problems and improve the product further.

In summary, it can be said that the proposed SE cluster architecture is a promising concept to bring more reliable and performant cryptographic devices to the IoT, but may also challenge server-grade HSMs in the near future. The scalable architecture gives clear advantages, including economic benefits, especially for use cases where no suitable device is available. Such scenarios include medium-sized CAs or document signing systems that require reliability and signature throughput but not necessarily signature latency.

CHAPTER 8

Conclusion

This thesis introduced a new, scalable approach for storing key material and performing cryptographic operations during changing demands. It is based on clustered secure elements that behave like a single element from an application point of view. By encapsulating cluster management in a software library, the solution offers far better compatibility than existing secure element grids. The number of elements can be adjusted as needed, providing an efficient solution for all scales.

Chapter 1 gave a first introduction to the topic and it motivated the proposed cluster approach. After describing the used technologies like HSMs, smart cards, and SEs in Chapter 2, an overview of the state of the art was given in Chapter 3. It was found that apart from SE grids, no similar design exists yet. The SE grids are incompatible to existing applications as they require a custom implementation of the cluster orchestration for each application.

Chapter 4 described the new cluster architecture. The chapter includes details on the required trust, key material synchronization, load distribution, scalability, and self-healing properties of the proposed cluster. Furthermore, the security of the architecture was evaluated.

Chapter 5 presented two prototype implementations. The first was based on smart cards and the second on SEs. Both device types operate the same HKS applet. It provides a low level interface for cryptographic operations and key management. The ideas presented in Chapter 4 were realized in a PKCS #11 software library, i.e., the cluster orchestration was implemented and made accessible over a standardized and widely used interface. An OpenSSL CA demonstrated the easy inclusion of the library into existing software.

An analysis of the prototypes regarding performance and durability can be found in Chapter 6. Connecting multiple smart cards to form the cluster resulted in almost linear scaling behavior. This success showed that the cluster's scalability allows for larger setups. The scalability performance of the SE prototype was not as good due to the

non-optimized software. However, SEs proved much more durable than smart cards and show better single-chip performance. When the SE kernel module is optimized, the scalability should be as good as it is for smart cards and the performance distinctly superior. Similar findings could be made concerning durability. Smart cards support only a limited number of write cycles, but SEs are designed for up to a hundred million write cycles. This makes the use of SEs efficient also in larger deployments. The analysis was concluded by presenting a formula for optimal cluster sizing. Depending on the properties of the cryptographic processors and the requirements of the use case, the best number of elements can be calculated. The existence of such a formula underlines the scalability features of the cluster design, offering optimized costs while satisfying all requirements.

Finally, Chapter 7 reflected on the findings. The test results of smart cards and SEs were compared and interpreted. The chapter includes a discussion of possible future work, especially research and development, and existing problems that must be solved were highlighted.

The thesis motivated a new hardware security device and proved its capabilities. Improving the prototype implementations for production use will create a superior product competing with smart cards, SEs, and HSMs, especially in mid-sized deployments.

APPENDIX A

Measurement results

ID	Key type	Threads	Cards	$\Theta_{\sigma_{max}}$	$\Theta_{\sigma_{max}}$ per card	Speedup	η_σ
1	P-256	1	1	8.66	8.664	1.000	1.000
2	P-256	6	1	8.66	8.661	1.000	1.000
3	P-256	1	3	8.45	2.816	0.975	0.325
4	P-256	3	3	25.22	8.406	2.911	0.970
5	P-256	6	3	25.20	8.400	2.909	0.970
6	P-256	1	6	8.50	1.417	0.981	0.163
7	P-256	6	6	50.45	8.408	5.822	0.970
8	P-256	12	6	50.46	8.410	5.824	0.971
9	RSA-1024	1	1	4.88	4.876	1.000	1.000
10	RSA-1024	6	1	4.88	4.876	1.000	1.000
11	RSA-1024	1	3	4.79	1.595	0.982	0.327
12	RSA-1024	3	3	14.31	4.769	2.934	0.978
13	RSA-1024	6	3	14.31	4.769	2.934	0.978
14	RSA-1024	1	6	4.81	0.801	0.986	0.164
15	RSA-1024	6	6	28.62	4.770	5.869	0.978
16	RSA-1024	12	6	28.60	4.767	5.867	0.978
17	RSA-2048	1	1	1.38	1.377	1.000	1.000
18	RSA-2048	6	1	1.38	1.377	1.000	1.000
19	RSA-2048	1	3	1.37	0.456	0.994	0.331
20	RSA-2048	3	3	4.10	1.365	2.975	0.992
21	RSA-2048	6	3	4.10	1.365	2.975	0.992
22	RSA-2048	1	6	1.37	0.228	0.995	0.166
23	RSA-2048	6	6	8.19	1.366	5.952	0.992
24	RSA-2048	12	6	8.19	1.365	5.951	0.992

Table A.1: Summary of performance measurements on J3D081 smart cards

ID	Number of sign.	Run 1 (μs)	Run 2 (μs)	Run 3 (μs)	Average (μs)
1	108	12,458,381	12,467,960	12,470,126	12,465,489
2	108	12,466,640	12,473,997	12,470,436	12,470,358
3	108	12,776,614	12,791,419	12,784,302	12,784,112
4	108	4,279,728	4,282,938	4,284,719	4,282,462
5	108	4,283,529	4,285,868	4,287,630	4,285,676
6	108	12,704,439	12,709,757	12,707,590	12,707,262
7	108	2,141,944	2,140,402	2,140,464	2,140,937
8	108	2,140,049	2,139,018	2,142,126	2,140,398
9	108	22,149,031	22,155,174	22,147,415	22,150,540
10	108	22,147,995	22,149,021	22,153,051	22,150,022
11	108	22,561,322	22,563,584	22,570,516	22,565,141
12	108	7,546,215	7,549,416	7,550,864	7,548,832
13	108	7,546,338	7,550,772	7,548,267	7,548,459
14	108	22,463,796	22,466,247	22,466,326	22,465,456
15	108	3,773,970	3,773,953	3,773,827	3,773,917
16	108	3,779,732	3,772,633	3,774,577	3,775,647
17	108	78,456,320	78,454,532	78,441,554	78,450,802
18	108	78,439,681	78,431,411	78,450,285	78,440,459
19	108	78,913,660	78,912,216	78,907,888	78,911,255
20	108	26,366,000	26,372,157	26,372,601	26,370,253
21	108	26,369,476	26,366,680	26,371,556	26,369,237
22	108	78,821,772	78,798,842	78,804,788	78,808,467
23	108	13,180,554	13,177,642	13,181,407	13,179,868
24	108	13,180,898	13,184,782	13,184,308	13,183,329

Table A.2: Timing values of the performance measurements on J3D081 smart cards

ID	Key type	Threads	SEs	$\Theta_{\sigma_{max}}$	$\Theta_{\sigma_{max}}$ per card	Speedup	η_σ
1	P-256	1	1	12.36	12.363	1.000	1.000
2	P-256	6	1	12.32	12.319	0.996	0.996
3	P-256	1	3	13.52	4.506	1.093	0.364
4	P-256	3	3	29.75	9.915	2.406	0.802
5	P-256	6	3	31.29	10.430	2.531	0.844
6	P-256	1	4	13.54	3.385	1.095	0.274
7	P-256	6	4	34.19	8.548	2.766	0.691
8	P-256	12	4	44.98	11.246	3.639	0.910
9	RSA-1024	1	1	12.45	12.453	1.000	1.000
10	RSA-1024	6	1	12.33	12.334	0.990	0.990
11	RSA-1024	1	3	12.43	4.144	0.998	0.333
12	RSA-1024	3	3	29.93	9.975	2.403	0.801
13	RSA-1024	6	3	28.51	9.502	2.289	0.763
14	RSA-1024	1	4	12.52	3.129	1.005	0.251
15	RSA-1024	6	4	31.37	7.844	2.519	0.630
16	RSA-1024	12	4	41.93	10.482	3.367	0.842
17	RSA-2048	1	1	7.28	7.276	1.000	1.000
18	RSA-2048	6	1	7.24	7.244	0.996	0.996
19	RSA-2048	1	3	7.18	2.394	0.987	0.329
20	RSA-2048	3	3	20.31	6.769	2.791	0.930
21	RSA-2048	6	3	21.00	6.999	2.886	0.962
22	RSA-2048	1	4	7.29	1.822	1.001	0.250
23	RSA-2048	6	4	21.04	5.259	2.891	0.723
24	RSA-2048	12	4	26.46	6.615	3.637	0.909

Table A.3: Summary of performance measurements on P73 SEs

ID	Number of sign.	Run 1 (μs)	Run 2 (μs)	Run 3 (μs)	Average (μs)
1	108	8,779,177	8,628,806	8,800,254	8,736,079
2	108	8,788,339	8,739,667	8,773,060	8,767,022
3	108	8,014,946	7,946,472	8,007,552	7,989,657
4	108	3,606,276	3,645,455	3,640,490	3,630,740
5	108	3,341,041	3,421,891	3,591,384	3,451,439
6	108	7,950,743	7,979,222	7,998,879	7,976,281
7	108	3,130,845	3,181,051	3,164,402	3,158,766
8	108	2,388,200	2,469,430	2,344,777	2,400,802
9	108	8,559,500	8,665,504	8,792,485	8,672,496
10	108	8,843,986	8,688,713	8,737,048	8,756,582
11	108	8,693,665	8,696,255	8,670,261	8,686,727
12	108	3,635,027	3,706,931	3,484,986	3,608,981
13	108	3,913,577	3,790,504	3,661,676	3,788,586
14	108	8,677,367	8,595,885	8,612,557	8,628,603
15	108	3,500,141	3,527,025	3,299,614	3,442,260
16	108	2,551,851	2,634,121	2,541,266	2,575,746
17	108	14,810,358	14,911,718	14,806,493	14,842,856
18	108	14,902,192	14,939,855	14,886,899	14,909,649
19	108	15,052,997	15,014,014	15,054,775	15,040,595
20	108	5,229,081	5,282,300	5,442,638	5,318,006
21	108	5,134,653	5,097,295	5,198,272	5,143,407
22	108	14,830,005	14,826,569	14,807,965	14,821,513
23	108	5,129,836	5,252,345	5,019,871	5,134,017
24	108	3,990,870	4,168,129	4,085,287	4,081,429

Table A.4: Timing values of the performance measurements on P73 SEs

APPENDIX B

Behavior of Smart Cards After Durability Testing

Listing B.1: Output of the dead card 0x00001005 after durability testing

```
# pcsc_scan
Mon Jan  6 12:28:00 2020
    Reader 0: Gemalto PC Twin Reader (B5D98847) 00 00
    Event number: 2
    Card state: Card inserted,
    ATR: 3B F9 13 00 00 81 31 FE 45 4A 43 4F 50 32 34 32 52 32 A3

ATR: 3B F9 13 00 00 81 31 FE 45 4A 43 4F 50 32 34 32 52 32 A3
+ TS = 3B --> Direct Convention
+ T0 = F9, Y(1): 1111, K: 9 (historical bytes)
    TA(1) = 13 --> Fi=372, Di=4, 93 cycles/ETU
    43010 bits/s at 4 MHz, fMax for Fi = 5 MHz => 53763 bits/s
    TB(1) = 00 --> VPP is not electrically connected
    TC(1) = 00 --> Extra guard time: 0
    TD(1) = 81 --> Y(i+1) = 1000, Protocol T = 1
-----
    TD(2) = 31 --> Y(i+1) = 0011, Protocol T = 1
-----
    TA(3) = FE --> IFSC: 254
    TB(3) = 45 --> Block Waiting Integer: 4 - Character Waiting
       Integer: 5
+ Historical bytes: 4A 43 4F 50 32 34 32 52 32
    Category indicator byte: 4A (proprietary format)
+ TCK = A3 (correct checksum)

Possibly identified card (using /home/tictacmoe/.cache/smartcard_list
    .txt):
```

```
3B F9 13 00 00 81 31 FE 45 4A 43 4F 50 32 34 32 52 32 A3
3B F9 13 00 00 81 31 FE 45 4A 43 4F 50 32 34 .. .. .. ..
    NXP JCOP v2.4.x (see hist bytes for more info)
    /


# gp -info
GlobalPlatformPro 19.06.16-5-g3067bd5
Running on Linux 5.4.3-arch1-1 amd64, Java 1.8.0_232 by Oracle
   Corporation
Reader: Gemalto PC Twin Reader (B5D98847) 00 00
ATR: 3BF91300008131FE454A434F503234325232A3
More information about your card:
    http://smartcard-atr.appspot.com/parse?ATR=3
       BF91300008131FE454A434F503234325232A3

Could not SELECT default selected: 0x6D00 (Invalid INStruction)


# gp -list
Could not SELECT default selected: 0x6D00 (Invalid INStruction)
```

APPENDIX C

OpenSSL-Generated Certificates

Listing C.1: Textual representation of the CA certificate

```
Certificate:
Data:
    Version: 3 (0x2)
    Serial Number:
        34:97:c1:ee:cc:b5:07:89:dc:11:09:3c:d3:6a:5c:fb:32:84:55:35
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: CN = Secure Element Cluster CA, O = TU Wien, ST = Vienna,
        C = AT
    Validity
        Not Before: Feb 13 11:22:25 2020 GMT
        Not After : Mar 14 11:22:25 2020 GMT
    Subject: CN = Secure Element Cluster CA, O = TU Wien, ST = Vienna
       , C = AT
    Subject Public Key Info:
        Public Key Algorithm: id-ecPublicKey
            Public-Key: (256 bit)
            pub:
                04:8e:87:90:3f:aa:b3:04:e1:d4:fa:e5:5a:22:3a:
                16:05:8a:0e:ad:50:82:cd:7a:eb:12:b9:b7:6d:c9:
                6c:ba:0e:bc:62:ee:6e:f0:9e:46:17:bc:b0:00:e6:
                71:c6:d0:a1:7c:ce:ee:a9:6a:69:ee:2d:48:7c:8c:
                fd:cf:10:41:93
            ASN1 OID: prime256v1
            NIST CURVE: P-256
    X509v3 extensions:
        X509v3 Subject Key Identifier:
            A3:F2:3E:FC:2C:77:E3:AC:74:0A:DB:E7:07:00:22:6A
               :92:06:02:79
        X509v3 Authority Key Identifier:
```

```
            keyid:A3:F2:3E:FC:2C:77:E3:AC:74:0A:DB:E7:07:00:22:6A
               :92:06:02:79

        X509v3 Basic Constraints: critical
            CA:TRUE
Signature Algorithm: ecdsa-with-SHA256
        30:45:02:21:00:8a:36:0e:5b:b6:33:83:87:1f:8d:48:3c:0c:
        94:1d:95:36:4d:8b:8b:bb:2f:54:50:b7:fa:8e:20:5b:56:c3:
        cf:02:20:39:2e:9f:60:16:89:cb:6e:95:26:aa:fa:f8:c1:28:
        4c:83:c7:c4:05:1a:6d:9d:b0:ee:d2:5c:e7:a7:2b:d0:98
```

Listing C.2: Textual representation of the entity certificate

```
Certificate:
Data:
    Version: 1 (0x0)
    Serial Number: 1 (0x1)
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: CN = Secure Element Cluster CA, O = TU Wien, ST = Vienna,
        C = AT
    Validity
        Not Before: Feb 13 12:13:15 2020 GMT
        Not After : Mar 14 12:13:15 2020 GMT
    Subject: CN = Some Entity, O = TU Wien, L = Vienna, C = AT
    Subject Public Key Info:
        Public Key Algorithm: id-ecPublicKey
            Public-Key: (256 bit)
            pub:
                04:34:8a:fe:35:1c:75:16:1d:fd:f9:71:4b:ff:eb:
                6c:e2:f5:0b:25:41:6d:46:3c:cd:be:73:4e:e0:df:
                7e:49:63:85:dc:63:6e:08:3b:da:10:e7:60:05:90:
                f2:5d:c5:8b:65:fc:59:92:09:c3:33:bf:a7:5b:fc:
                83:f1:c5:fd:92
            ASN1 OID: prime256v1
            NIST CURVE: P-256
Signature Algorithm: ecdsa-with-SHA256
        30:44:02:20:74:9f:e2:07:31:b9:1b:ce:ab:3d:fb:ac:df:cc:
        3e:a7:4f:4d:74:f4:1b:89:48:06:71:36:0a:53:af:ad:31:2d:
        02:20:0c:1a:cd:a1:20:f0:73:30:0f:e0:06:54:4b:56:28:8f:
        97:58:6f:90:3d:97:d1:cc:b0:ad:b0:7b:43:64:40:8a
```

List of Figures

List of Tables

Acronyms

APDU Application Protocol Data Unit. 11–14, 16–18, 25, 34, 36, 41–44, 54, 57, 59, 67, 81

API Application Programming Interface. 16–19, 25, 42, 43

ASK Amplitude-Shift Keying. 10, 11

CA Certificate Authority. 37, 39, 42, 49, 62, 63, 66, 67, 69, 79

CC Common Criteria. 6, 7

CSP Critical Security Parameter. 7

FIPS Federal Information Processing Standard. 6, 7, 83

FPGA Field Programmable Gate Array. 23, 41, 43, 44, 48, 54, 55, 65–67, 81

HCE Host Card Emulation. 25

HKS Hardware Key Safe. 19, 27–30, 32–46, 56, 57, 60, 67, 69, 81

HSM Hardware Security Module. 1–3, 5, 6, 8–10, 12, 23–25, 65–67, 69, 70

IoT Internet of Things. 2, 23, 24, 62, 66, 67

JCOP JavaCard Open Platform. 19, 27, 45–48

NIST National Institute of Standards and Technology. 6, 36, 49, 52, 58–60, 83

NSP Network Security Processor. 5

OCSP Online Certificate Status Protocol. 62, 63

OPEN GlobalPlatform Environment. 17

PCI-e PCI Express. 1, 5, 33, 37, 43, 48, 55, 67, 81

PKI Public Key Infrastructure. 6, 29, 49

PP Protection Profile. 7

S2C SigIn-SigOut-Connection. 12

SE Secure Element. 1–3, 5, 12, 13, 16, 17, 20, 21, 23–25, 27–29, 32–40, 43–45, 48–51, 54, 55, 60–62, 65–67, 69, 70, 74, 75, 83

SPI Serial Peripheral Interface. 43, 54

SWP Single Wire Protocol. 12, 43

TRSM Tamper-Resistant Security Module. 5

UICC Universal Integrated Circuit Card. 12

Bibliography

[1] A. P. Fournaris, K. Lampropoulos, and O. Koufopavlou, "Hardware Security for Critical Infrastructures - The CIPSEC Project Approach," in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 356–361, Bochum, DE, July 2017.

[2] R. De Prisco, A. De Santis, and M. Mannetta, "Reducing Costs in HSM-Based Data Centers," in *Green, Pervasive, and Cloud Computing*, pp. 3–14, M. H. A. Au, A. Castiglione, K.-K. R. Choo, F. Palmieri, and K.-C. Li, Eds. Cham, CH: Springer International Publishing, 2017.

[3] M. Hendry, "The Secure Element," in *Near Field Communications Technology and Applications*, pp. 58–72. Cambridge, UK: Cambridge University Press, 2014.

[4] O. Foundation, "OWASP Top 10 Application Security Risks - 2017," OWASP Foundation, Bel Air, MD, Tech. Rep., 2017, Available: https://web.archive.org/web/20191106133452/https://www.owasp.org/index.php/Top_10-2017_Top_10, Accessed: 2019-11-06.

[5] M. Wolf and T. Gendrullis, "Design, Implementation, and Evaluation of a Vehicular Hardware Security Module," in *Information Security and Cryptology - ICISC 2011*, H. Kim, Ed. Berlin, Heidelberg, DE: Springer Berlin Heidelberg, Jan. 2012.

[6] P. Urien, "Cloud of Secure Elements perspectives for mobile and cloud applications security," in *2013 IEEE Conference on Communications and Network Security (CNS)*, pp. 371–372, National Harbor, MD, Oct 2013.

[7] K. Markantonakis, D. K. Mayes, F. Piper, and R. N. Akram, *Secure Smart Embedded Devices, Platforms and Applications*. New York, NY: Springer New York, 2014.

[8] L. Sustek, "Hardware security module," in *Encyclopedia of Cryptography and Security*, H. C. A. van Tilborg and S. Jajodia, Eds., pp. 535–538. Boston, MA: Springer US, 2011.

[9] "FIPS PUB 140-2: Security Requirements for Cryptographic Modules," National Institute of Standards and Technology, Gaithersburg, MD, Standard, Jan. 1994, Available: https://csrc.nist.gov/publications/detail/fips/140/2/final, Accessed: 2020-04-11.

[10] "Information technology - Security techniques - Evaluation criteria for IT security - Part 3: Security assurance components," International Organization for Standardization, Geneva, CH, Standard ISO/IEC 15408-3:2008, Jun. 2011.

[11] "Information Technology - Security Techniques - Physical Security Attacks, Mitigation Techniques and Security Requirements," International Organization for Standardization, Geneva, CH, Standard ISO/IEC 30104:2015, May 2015.

[12] "Identification cards - Integrated circuit cards - Part 4: Organization, security and commands for interchange," International Organization for Standardization, Geneva, CH, Standard ISO/IEC 7816-4:2013, Apr. 2013.

[13] "Java Card Classic Platform Specification 3.1," Oracle Corporation, Redwood City, CA, Specification, Jan. 2019.

[14] "GlobalPlatform Technology Card Specification Version 2.3.1," GlobalPlatform, Inc, Redwood City, CA, Specification, Mar. 2018.

[15] "JCOP 4 P71 - Security Target Lite for JCOP 4 P71 / SE050," NXP B.V., Eindhoven, NL, Specification, Jun. 2019, Available: https://web.archive.org/web/20200214101453/https://www.commoncriteriaportal.org/files/epfiles/%5BST-LITE%5D%20JCOP4_P71_SecurityTargetLite_v3.4.pdf, Accessed: 2020-02-14.

[16] "PKCS #11 Cryptographic Token Interface Base Specification Version 2.40 Plus Errata 01," OASIS Standard Incorporating Approved Errata 01, Specification, May 2016, Available: https://web.archive.org/web/20200209103402/http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html, Accessed: 2020-02-09.

[17] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov, "Cryptographic Processors-A Survey," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 357–369, Feb. 2006.

[18] J. Ivarsson and A. Nilsson, "A Review of Hardware Security Modules Fall 2010," AB Certezza, Stockholm, SE, Tech. Rep., Dec. 2010, Available: https://web.archive.org/web/20200411132336/https://www.opendnssec.org/wp-content/uploads/2011/01/A-Review-of-Hardware-Security-Modules-Fall-2010.pdf, Accessed: 2020-04-11.

[19] D. Parrinha and R. Chaves, "Flexible and low-cost HSM based on non-volatile FPGAs," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–8, Cancun, MX, Dec 2017.

[20] R. Druyer, L. Torres, P. Benoit, P. Bonzom, and P. Le Quere, "A survey on security features in modern FPGAs," in *2015 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–8, Bremen, DE, Jun. 2015.

[21] A. Fournaris and G. Keramidas, "From Hardware Security Tokens to Trusted Computing and Trusted Systems," in *System-Level Design Methodologies for Telecommunication*, N. Sklavos, M. Hübner, D. Goehringer, and P. Kitsos, Eds., pp. 99–117. Springer, Cham, Jan. 2014.

[22] C. Siegelin, L. Castillo, and U. Finger, "Smart Cards: Distributed Computing with $5 Devices," *Parallel Processing Letters*, vol. 11, pp. 57–64, Mar. 2001.

[23] S. Chaumette, P. Grange, A. Karray, D. Sauveron, and P. Vigneras, "Secure distributed computing on a Java Card (TM) grid," in *19th IEEE International Parallel and Distributed Processing Symposium*, pp. 1–8, Denver, CO, Apr. 2005.

[24] P. Urien, E. Marie, and C. Kiennert, "An innovative solution for cloud computing authentication: Grids of eap-tls smart cards," in *2010 Fifth International Conference on Digital Telecommunications*, pp. 22–27, Athens, GR, Jun. 2010.

[25] P. Urien and S. Piramuthu, "Towards a secure Cloud of Secure Elements concepts and experiments with NFC mobiles," in *2013 International Conference on Collaboration Technologies and Systems (CTS)*, pp. 166–173, San Diego, CA, May 2013.

[26] P. Urien, "RACS: Remote APDU call secure creating trust for the internet," in *2015 International Conference on Collaboration Technologies and Systems (CTS)*, pp. 351–357, Atlanta, GA, Jun. 2015.

[27] M. Roland, J. Langer, and J. Scharinger, "Practical Attack Scenarios on Secure Element-Enabled Mobile Devices," in *2012 4th International Workshop on Near Field Communication*, pp. 19–24, Helsinki, FI, March 2012.

[28] S. Sridevi and V. R. Uthariaraj, "A survey of soft computing techniques applied in cloud load balancing," in *2016 Eighth International Conference on Advanced Computing (ICoAC)*, pp. 131–137, Chennai, IN, 2017.

[29] T. Deepa and D. Cheelu, "A comparative study of static and dynamic load balancing algorithms in cloud computing," in *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, pp. 3375–3378, Chennai, IN, 2017.

[30] S. Garg, D. D. V. Gupta, and R. K. Dwivedi, "Enhanced Active Monitoring Load Balancing algorithm for Virtual Machines in cloud computing," in *2016 International Conference System Modeling Advancement in Research Trends (SMART)*, pp. 339–344, Moradabad, IN, 2016.

[31] Y. Fahim, E. Ben Lahmar, E. h. Labriji, and A. Eddaoui, "The load balancing based on the estimated finish time of tasks in cloud computing," in *2014 Second World Conference on Complex Systems (WCCS)*, pp. 594–598, Agadir, MA, 2014.

[32] V. Kumar, A. Grama, and N. Vempaty, "Scalable Load Balancing Techniques for Parallel Computers," *Journal of Parallel and Distributed Computing*, vol. 22, no. 1, p. 60–79, Jul. 1994.

[33] A. Samir and C. Pahl, "Anomaly Detection and Analysis for Clustered Cloud Computing Reliability," in *Proceedings of the Tenth International Conference on Cloud Computing, GRIDs, and Virtualization*, Venice, IT, May 2019.

[34] J. Liao, F. Zhang, L. Li, and G. Xiao, "Adaptive Wear-Leveling in Flash-Based Memory," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 1–4, 2015.

[35] T. Souza, J. Martina, and R. Custódio, "Audit and backup procedures for Hardware Security Modules," in *IDtrust 2008, Proceedings of the 7th Symposium on Identity and Trust on the Internet*, pp. 89–97, Gaithersburg, MD, Jan. 2008.

[36] A. Umar, K. Mayes, and K. Markantonakis, "Performance variation in host-based card emulation compared to a hardware security element," in *2015 First Conference on Mobile and Secure Services (MOBISECSERV)*, pp. 1–6, Gainesville, FL, 2015.

[37] D. J. Bernstein and T. Lange, "SafeCurves: choosing safe curves for elliptic-curve cryptography," https://safecurves.cr.yp.to, Accessed: 2020-04-12.

[38] "P5CD016/021/041 and P5Cx081 family - Product short data sheet," NXP B.V., Eindhoven, NL, Datasheet, Mar. 2010, Available: https://web.archive.org/web/20200106144749/https://www.nxp.com/docs/en/data-sheet/P5CD016_021_041_Cx081_FAM_SDS.pdf, Accessed: 2020-01-06.

[39] "NXP J3D081_M59_DF, and J3D081_M61_DF Secure Smart Card Controller Revision 2 - Evaluation documentation," NXP B.V., Eindhoven, NL, Tech. Rep., Mar. 2013, Available: https://web.archive.org/web/20200106150346/https://www.commoncriteriaportal.org/files/epfiles/0860b_pdf.pdf, Accessed: 2020-01-06.