University of Bergen
Department of Informatics

Using Smart Cards to Mitigate Security Threats on Mobile Devices

---

Author: Henrik Mathisen Sivertsgård
Supervisors: Marcus Aloysius Bezem, Federico Mancini

Master thesis
May 30, 2016

Abstract

This master thesis developed and tested the idea that smart cards are able to help mitigate security threats on mobile devices that are handling sensitive data. Investigating the limitations of smart cards is a fundamental part of the idea and we performed in-depth testing and analysis of what smart cards are capable of. Our study shows that smart cards are limited by their low processing power, low amount of memory and a severely lacking application programming language. These limitations affect smart cards usefulness concerning cryptography and data processing. Additionally, lack of standard support for smart cards in modern mobile operating systems is a limitation we investigate and propose solutions for.

Despite these limitations, smart cards can still be a useful asset as they offer a secure execution environment and they are tamper resistant. Viable use cases include secure key generation, management and storage, digital signing, encryption, strong authentication and the possibility to run small specialized applets securely. More complex use cases are also possible, but require additional external components and infrastructure to be realized. For instance, a smart card could be used as a simple policy enforcement point, given that we had a trusted third party available, and a functioning public key infrastructure in place.

We were able to construct an Android library and a smart card applet for secure communication, but there still remains research on the topic. Not all functionality were implemented due to time constraints and technical issues, but the framework foundations are in place so that extensions can be quickly and easily implemented. Future work may include full scale testing of our framework, additional development and testing on more technological advanced smart cards.

Acknowledgments

Foremost, I would like to thank my supervisors Prof. Marcus Aloysius Bezem and Federico Mancini for their engagement, cooperation and guidance during my work on the thesis. Your support is greatly appreciated. I would also like to thank Forsvarets Forskningsinstitutt for providing software, equipment and resources which enabled me to perform research on this topic.

Henrik Mathisen Sivertsgård
May 30, 2016

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Problem statement and motivation

The original use of mobile phones was to make calls and send text messages. As mobile phones evolved into what we call smartphones, their possible fields of application have increased, and with that the need for better security. In today's society we use smartphones to access our bank accounts, control our cars and houses, identify ourselves, pay in shops, buy tickets and much more. Modern smartphones do implement a lot of security, which is mainly intended for the average private user. The innovation and potential of smartphone technology has attracted the attention of other types of users like military forces, governmental employees, health personnel, and others. These new users handle more sensitive data and perform more critical tasks than most normal users. In order to be able to adopt smartphones in these settings, their integrated security mechanisms may not always be enough, and additional technology and novel solutions are needed.

A specific problem of smartphones, and mobile devices in general, is that they are much easier to lose or get stolen than stationary equipment. Mobile devices are relatively easy to get physical access to as they are often left unattended, even only if for a few minutes. In addition, one of the things that makes mobile devices so great, is their ability to constantly be connected to a network through either Wi-Fi, radio, Bluetooth, NFC or USB. This is a double edged sword as it offers multiple entry points that an attacker can use to get in even without physical access. One assumption one must make, is that a motivated attacker will most likely be able to compromise the device

at some point. In a compromised device we cannot trust anything, not even the operating system as an attacker may have root access. This possibility is not something we can accept when particularly sensitive data is being processed, stored and exchanged on the device. A common solution is to encrypt the data on the device, but if the device is compromised one must assume that the cryptographic keys are also compromised.

The reason why we chose to investigate smart cards as an additional secure element, is that we believe that they can increase the security of smartphones in the exact situation as described above. These cards have an internal secure execution environment which can be used to generate, store and use cryptographic keys, and since they are tamper resistant, they provide protection against physical attacks. In addition, they have their own operating system that can run small customized applets which can be trusted to execute critical task securely even if the mobile device is compromised.

In particular we look at what it is possible to achieve by using "off-the-shelf" mobile devices deployed with an additional secure element like a smart card. In other words, we are not interested in solutions that require a modification of the mobile device, like having to flash a customized OS in order to enable additional modules or security mechanisms.

1.2 Goals and research questions

The overall goal of this thesis is to explore and evaluate the possibilities of how smart cards and mobile devices can co-operate to achieve a higher level of information security. To better understand what limitations there are, we will need to create a framework for easy communication between a mobile device and smart card. This will also include developing test applications for both the mobile device and the smart card. The framework, mobile application and smart card application, needs to be tested with as close to real life use cases as possible.

In order to achieve this goal we will attempt to answer the following research questions:

- "What are the limitations of smart cards in the context of hardware?"
- "What are the limitations of smart cards in the context of software?"

- "What types of security threats are we able to mitigate by using a smart card with an "off-the-shelf" mobile device?"

1.3 Chapter organization

Chapter 1 - Introduction Presents a problem statement, motivation and research question for this master thesis.

Chapter 2 - Background Introduces all concepts needed for understanding how smart cards and relevant technology function. In addition, it provides a basic understanding of the most common threats which a mobile platform is exposed to.

Chapter 3 - Smart Card Framework - Goals and Environment Discusses the high-level goals we have for the framework and describes the tools and technologies we used when developing the framework.

Chapter 4 - Challenges and use cases In this chapter we identify possible use cases for the application of smart cards to mobile security and discuss which challenges must be solved to realize them.

Chapter 5 - Framework design and implementation Gives an overview of how the Android library and smart card applet have been implemented and structured.

Chapter 6 - Testing and use case implementation Shows the testing environment for the smart cards and provides detailed testing results for smart card limitations and use cases.

Chapter 7 - Conclusion Discusses the research questions that we have established and what experiences we encountered during this master thesis.

Chapter 2

Background

In this chapter we will provide the background information necessary to understand how smart cards function and how they increase security on mobile devices. In particular, we will cover smart card technology, relevant mobile device technology (in our case Android OS), basic concepts of cryptography and typical threats in the context of mobile devices.

2.1 Smart card

The term smart card refers to a card with an integrated circuit. In practice, it is a miniature computer with limited computing power, which can process information in a secure isolated environment and exchange data with the device that usually also provides it with the necessary power to operate.

2.1.1 Smart card architecture and specifications

The micro processor is able to perform tasks that involve processing input, give ouput and storing small amounts of data. Smart cards vary in sizes defined by the standards ISO/IEC 7810 [27] and ISO/IEC 7816 [28]. The most common card dimensions are approximately 85x54 mm [38, Ch. 3.1]. The processing power of smart cards are between 25-32 MHz and sport anywhere from 8Kbit to 128Kbit memory (EEPROM) [48].

The micro processor can be powered and communicate in two ways. Contact

smart cards have integrated contact pads. When the smart card is inserted into a card reader the contact pads of the card reader provides power to the micro processor via the contact pads of the smart card. The contact pads also functions as a medium for transferring data. Contactless smart cards uses radio-frequency induction to power the micro processor and to transmit data between an antenna and the card. Most modern smart cards supports both technologies (hybrid cards). Figure 2.1 shows a contact smart card and reader.

An everyday example of a smart card is the modern credit and debit card. Most of these cards are contact cards that require the user to insert the card into a card reader, but newer cards are of the hybrid type, and have the ability to communicate over radio frequencies. Credit and debit cards utilizes the input/output capabilities of the smart card, but they also store information on the card authenticating the users bank information.

Credit: *Thiemo Schuff (CC BY 3.0 DE)*

Figure 2.1: Contact smart card and reader.

In figure 2.2 we can see how the architecture of a smart card (NFC) is built up. In the bottom we have the hardware of the smart card which includes

the ROM, EPROM, RAM and in the case of NFC cards the NFC receiver. On top of the hardware is the operating system of the smart card. The operating system has drivers installed along with API which it provides to the virtual machine. On top of the virtual machine multiple applets are able to run as mentioned above.

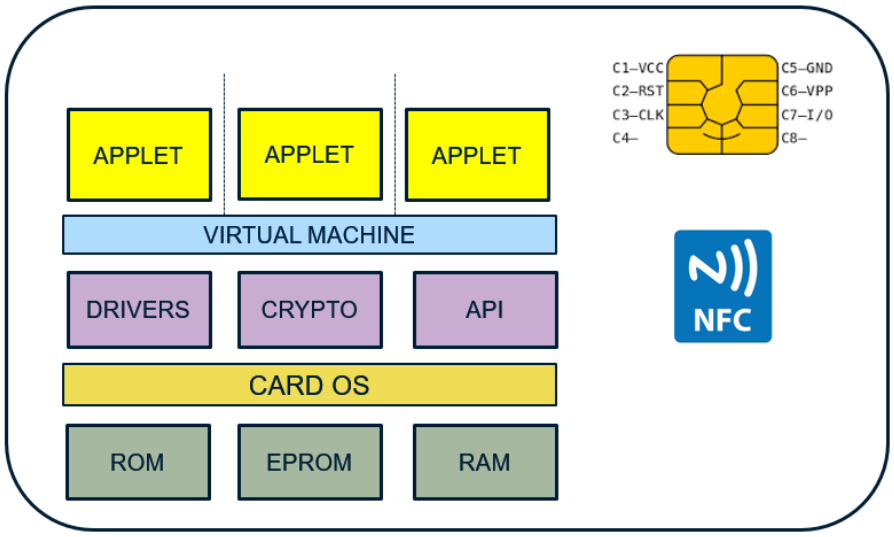

Figure 2.2: Smart card architecture

2.1.2 Communication standard for smart cards

Application Protocol Data Unit (APDU) is a standard that describes how a smart card application should communicate with other applications (off-card) and is defined by ISO7816-4 [26]. There are two types of APDU messages: Command APDU and Response APDU [38, Ch. 8.3, Message Structure: APDUS].

Command APDU is split into header and body. Refer to table 2.1 for instruction explanation and summary. The header is mandatory for all transactions and consists of 4 bytes that is split into CLA, INS, P1 and P2. The body of a Command APDU is split into 3 parts; LC, Payload and LE. LC is 1 byte, payload is maximum 255 bytes and LE is 1 byte.

Newer smart cards supports Extended APDU which allows the payload to be up to a maximum of 65535 bytes. If the payload data is greater than

255 bytes LC must be 3 bytes where the first byte is 0x00 to denote that the APDU is extended and the remaining 2 bytes denotes the length. If extended APDU is used then LE consists of 2 bytes to account for longer responses.

Name	Number of bytes	Description
CLA	1	Command type class, type of command
INS	1	Instruction code, command to run
P1	1	Free parameter
P2	1	Free parameter
LC	0, 1 or 3	Length of payload
Payload	0 - 65535	Payload data
LE	0, 1 or 2	Expected response length

Table 2.1: Command APDU layout.

Most Command APDUs falls into three abstract categories. The first category is retrieving something stored on the smart card. Often this command does not need any extra data (payload) and is characterized by INS, P1 and P2 deciding what data that is desired, and LE deciding what length the response is. For example: we have a smart card applet and in the instruction 01 01 00 we have stored a 8 byte variable that we want to retrieve (LE is 08). In this case the payload is empty since our theoretical method does not need any payload, but we will need to explicitly state that in the command APDU (LC is 00). The Command APDU would then look like:

CA	INS	P1	P2	LC	Payload	LE
80	01	01	00	00		00

The second category is storing variables on the smart card. The characteristics of the Command APDU differ from the first category by having a LC and payload, but rarely needs any response apart from "success" or "failure". For example our smart card applet let's us store 8 byte in the instruction 01 02 00. LC would then be 08 and the payload would contain the 8 bytes we want to store. We set LE to 02 as we define 9000 as success and 0999 as failure. The Command APDU would then look like:

CA	INS	P1	P2	LC	Payload	LE
80	01	02	00	08	5964574e704a593d	02

The third category is sending data to the smart card and have the smart card process the data. In context of security the best example of this category is cryptographic functions. This category will use all variables of a Command APDU (INS, P1 and P2 usage depends on application complexity). As an example let us say that we want to send a Command APDU that request the ciphertext of a clear text. The instruction is 01 03 00 and our cipher algorithm in this case is very basic and "increments" byte values. LC and LE is 8 byte. The Command APDU would then look like:

CA	**INS**	**P1**	**P2**	**LC**	**Payload**	**LE**
80	01	03	00	08	5964574e704a593d	08

Response APDU is split into body and response trailer. The body consist of the response data and is at maximum 255 bytes or 65535 bytes depending on if extended APDU is used. All Response APDUs must contain a response trailer of two bytes which denotes the processing status (error, success, wrong format, etc.) of the Command APDU. Refer to table 2.2 for definitions and summary.

Name	**Number of bytes**	**Description**
Response	0-65535	Response data
SW1+SW2	2	Command processing status

Table 2.2: Response APDU layout.

If everything is successful when processing a Command APDU with no expected response data the Response APDU may look like:

Response data	**SW1**	**SW2**
	90	00

If we use the example from the third category of Command APDUs that used ciphering as an example the Response APDU would look like:

Response data	**SW1**	**SW2**
6065584f714b603e	90	00

To better understand when to use the Command APDUs and when to use Response APDUs can use the following example: A locked door has a card reader connected to it. A person walks up to the door and presents his

contactless smart card to the reader. The card reader sends a Command APDU to the card asking for the ID. The smart card processes the Command APDU and sends a Response APDU back to the card reader containing the ID of the person. This example is visualized in figure 2.3 and in figure 2.4 we have the same example, but with actual APDU commands.

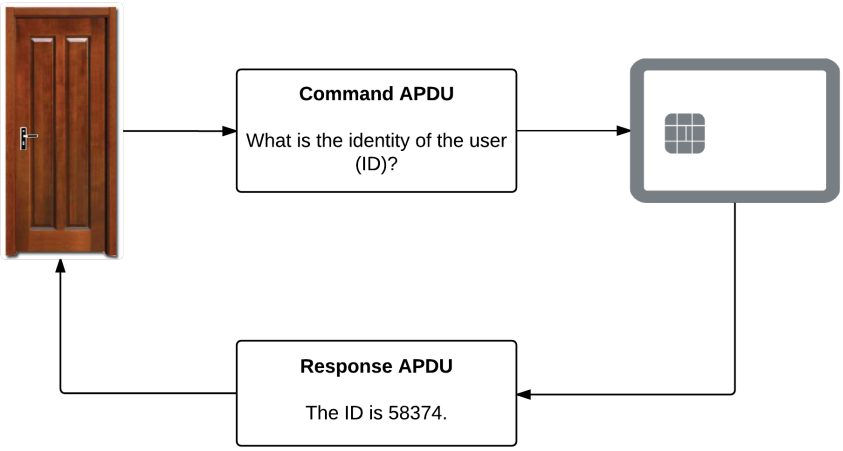

Figure 2.3: Door lock using smart card to unlock.

On most smart cards there is an application manager which listen for a special type of Command APDU: a Select APDU. The Select APDU contains information on what type of smart card application a sender is trying to communicate with and the job of the application manager is to activate the correct application. Before communicating with a smart card the sender should send out a Select APDU, if this step is skipped you run the risk of sending information to the wrong smart card application. A Select APDU sent to a smart card with application ID "0102030405060708090007" is defined as:

CA	**INS**	**P1**	**P2**	**LC**	**Payload**	**LE**
00	A4	04	00	0B	01020304050607080900 07	

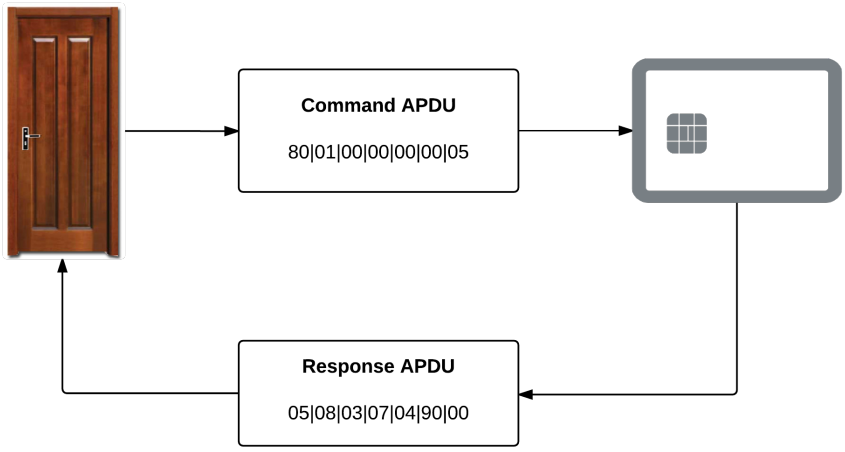

Figure 2.4: Door lock using smart card to unlock with corresponding APDU commands

2.1.3 Java Card

In section 2.1 we described that smart cards are able to store data as well as process input and output. All smart cards have their own operating system that allows developers to write applications that run on the smart cards. Smart cards are not limited to one applications per card, but are able to have multiple applications installed. Traditionally it was not feasible to create programs that ran on different smart cards as the micro processors were manufacturer specific [57]. This created an environment where smart card issuers and their developers were locked to a specific manufacturer.

The company Schlumberger [41], later joined by Sun Microsystems [33], outlined Java Card 1.0. Java Card were to alleviate the problem of manufacturer specific code and to let developers write generic applications. Newer Java Card version includes a development kit that provides a test environment and a converter tool that prepares the Java Card applet/program for installation onto a smart card. The newest Java Card version is currently 3.0.5 [30].

The Java Card language is in practice a very basic version of the standard Java language. Many Java classes and features are not present. For example: int, double, long, java.lang.SecurityManager, threading and object cloning

[3]. The structure of the applets differs from standard Java applets. All Java Card applets must implement:

- `void install (byte [] barray, short bOffset, byte bLength)`
- `void process(APDU apdu)`.

`Install` is invoked when the applet is downloaded onto the smart card and should register and initialize the applet [58]. `Process` is the entry point for all requests to the application and where the applet specific logic is done [59].

Garbage collection in the Java Card language differs from standard Java. In standard Java garbage collection is performed by the Java Virtual Machine (JVM) and runs independently of the application. In Java Card objects are stored in the persistent memory (EEPROM) and writing to this memory is very time-consuming. As a result it was decided that there would be no automatic garbage collection in Java Card and that garbage collection should be invoked by the application itself. Deciding when to perform garbage collection is very difficult as on one side it is resource intensive, while on the other side you may risk running out of memory.

The fact that there is now a standardized smart card platform based on Java Card technology, means that once an applet as been written, it can be easily installed on different types of smart cards. For instance cards in micro SD format as shown in Figure 2.5 and cards as shown in figure 2.1 would be able to run the same application (given that both support Java Card).

Figure 2.6 shows how the general outline of smart card architecture (see figure 2.2) translates to Java Card. The Loader/Installer is responsible for installing the applet onto the smart card so that it can run in the Java Card VM where as the Card Manager is responsible for activating the correct applet and forwarding APDUs.

Figure 2.5: Micro SD card from Gemalto.

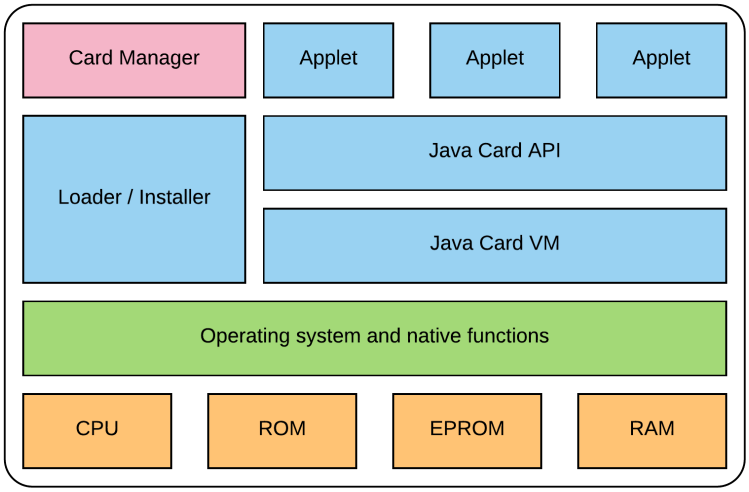

Figure 2.6: Java Card architecture

2.1.4 Other smart card programming languages

Besides Java Card, another widespread smart card operative system is MULTOS [47]. MULTOS was the first open high security operating system that supported multiple applets and supports the programming languages native assembly (MEL), C and Java. Java Card and MULTOS differs from how they handle applets shared space/memory on the smart cards. MULTOS security targets are: Applets can only be loaded by the card issuer, applets are segregated and secure package deployment. Java Card uses a secure channel to distribute applications whereas MULTOS uses secure packaging. The essence is that third parties can deploy new applets to the smart card containing sensitive data. This is possible as the package is encrypted with the public key of MULTOS (card unique) and thus you do not need a secure channel [10, 46].

There exists a .NET version of a smart card OS started by Microsoft which led to other versions developed by Gemalto and CardWerk. We will use Java Card in this thesis as this is supported by our smart cards and is similar to standard Java which is used in our mobile devices (refer to section 2.2).

2.2 Android operating system

Mobile operating system refers to the operating system running on a mobile device (smartphones, GPS devices, tablets, etc.). In this thesis we will focus mainly on smartphones and tablets since their capabilities are within our scope and the fact that they often share the same operating system. For practical reasons and because of time constraints we will restrict ourselves to one mobile operating system and the operating system we chose is Android.

Android is an open source licensed mobile operating system and is based on one of the LTS (long-term support) branches of the Linux kernel. Google Inc. [2] is the current developer of the mobile operating system and their primary focus has been smartphones and tablets. In later years Google has put resources into incorporating Android with TVs, wrist watches and cars. In 2015 Q2 82,8 % of smartphones worldwide was shipped with a version of the Android mobile operating system [49].

The Android mobile operating system supports applications that are written in Java, GO and C/C++. The applications run in their own sandbox with

their own allocated memory space, but applications can also access shared resources given permission to do so by the user.

Another reason for picking Android as our mobile platform is the flexibility of the Android system and devices when it comes to secure element support. iOS devices have no support for micro SD smart cards and Windows Phones does not seem to have any libraries support smart card communication. Blackberry is an alternative, especially their latest iterations which is a custom Android version, but due to their low market share they are not representative in a "bring your own device" situation. We discuss the Blackberry Priv more in depth in section 2.2.2.

2.2.1 Smart card support in Android

In Android version 2.3 (API level 9) NFC support was added which enabled a mobile device to read NFC Data Exchange Format (NDEF) from NFC devices/tags. In Android 2.3.3 (API level 10) the class `IsoDep` was added which enabled mobile devices to send byte data via NFC to devices with NFC support. This means that any device supporting API level 10 or higher and with a NFC adapter can communicate with smart cards via NFC.

As mentioned in section 2.1.3 smart card applications can run on micro SD cards. Mobile devices with a micro SD card slot can in theory communicate with these types of smart cards, but Android has no native support for communicating with them. This essentially means that the hardware is there on some mobile devices, but developers lack the bridge between their application to the hardware.

The Secure Element Evaluation Kit (SEEK) for the Android platform is an open source project, maintained by Giesecke & Devrient GmbH [18], which has a vision to make it easier to use secure elements in Android applications. The framework enables access to a variety of secure elements such as SIM cards, micro SD cards and embedded secure elements. Their final goal is to have their library as an integrated part of the Android operating system such that all new mobile devices comes with hardware-backed security support [43]. Some mobile device manufactures have included this framework, but there exists no documentation stating which devices have it, what channels are open and how to access the secure elements. Often the case is that only the channel to the mobile device SIM card is open meaning you cannot access custom secure elements.

Even though SEEK is open source and gives us access to the source code which would let us modify and tailor it to our needs, the main obstacle to its adoption is that it must be compiled with the kernel. This translates to creating, compiling and installing a custom operating system on our test devices. However, we made it clear from the beginning that our goal is to use off-the-shelf mobile devices. Another reason for not running a custom version of an operating system, is that it requires an organization to maintain and support it. From a security standpoint this is not viable as you will not automatically get security updates and will be running an unofficial version of an operating system. Running a custom operating system may also cause issues for the users as some services may block the user if they detect it (banking applications, proprietary applications, etc.).

As an alternative to flashing the operating system one can use proprietary drivers from smart card manufacturers. Gemalto produces a Java library, IDGo800, which is a cryptographic and communication middleware for their smart cards (more on this in section 5.2.3). Installing/using third-party applications/libraries to communicate with smart cards may not always be the best option, as you make yourself dependent on a third-party. In essence, opting for this solution results in a vendor lock. Security wise you should not blindly trust a third-party and you may not have access to the software if you wish to evaluate it.

2.2.2 Blackberry Priv

The Blackberry Priv by Blackberry is branded as a "secure smartphone" [9]. Blackberry's states that all data on the device is hardware-encrypted, the operating system performs a integrity check every time it boots, and that Blackberry's hardware suppliers are to be trusted. Blackberry Priv runs a modified Android version and comes with native secure element support. What this entails is that the Blackberry Priv may be the "off-the-shelf" solution we seek for both hardware and software (operating system) as we do not need to run and maintain custom frameworks or flash the operating system to communicate with secure elements as discussed in section 2.2.1.

The downside of opting for the Blackberry Priv is that we lock ourselves to a specific vendor which defeats the "bring your own device" concept. Another thing to keep in mind is that this is one single device, meaning it would need to be reviewed as to whether it fits the organization needs in terms of functionality and other characteristics (size, weight, battery,

etc.). The device is also rather expensive compared to devices with similar specifications.

In this thesis we will not seek further investigation on the Blackberry Priv and we will focus on more standardized smartphones, but it is important to know that it exists for future research. Information on our test device can be found in the chapter 6, section 6.1.1.

2.3 Cryptography

Cryptography is a method for protecting confidential data using complex mathematics and computer science. Most cryptographic functions/algorithms relies heavily on the fact that the mathematical problems which they are based on are so complex that they are "unbreakable" without knowledge of the encryption/decryption key.

2.3.1 Public-key cryptography

Public-key cryptography refers to a set of methods for asymmetric cryptography. It is based on the concept that one entity (user, server, etc.) generates a key pair consisting of one public key and one private key. Data encrypted using the public key can only be decrypted by the private key and due to the complexity of the keys it is improbable that the private key can be generated from the public key. The public key, as the name suggests, is publicly available for other entities. This combination allows entities to communicate securely given that they have each others public key and their private key is stored securely. Figure 2.7 shows how a sender can send a message that only the receiver can read. Although it is important to note that this operation is more resource intensive and time consuming compared to symmetric key encryption/decryption.

One of the most common public-key cryptosystems is RSA, which is widely used for secure communication. RSA builds on the principle of factorization of the product of two prime numbers, or rather the difficulty of factorize the product. It is not impossible to factorize the product of two prime numbers and there was a challenge by the RSA Laboratories where one could win prizes for factorized RSA-keys [52], but the most complex RSA that were cracked was 768-bit. Proving that RSA is secure is out of the time scope for this thesis. Other sources conclude that with long enough keys and correct

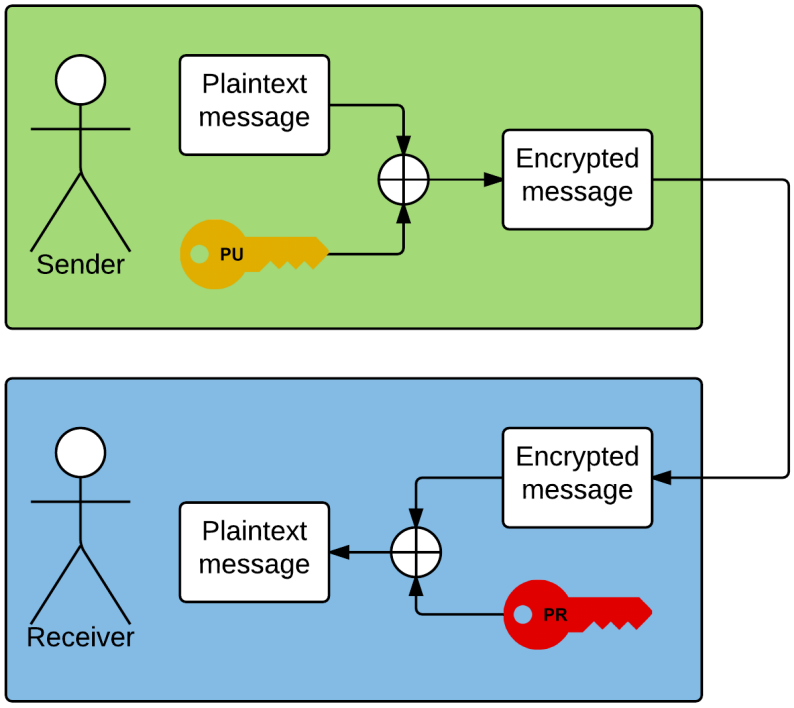

Figure 2.7: Asymmetric key encryption/decryption using public-private key pair.

protocol implementation, the math behind RSA can be considered secure [11, p. 194]. Information on the inner workings of RSA can be found in the book "Understanding Cryptography" by Christof Paar and Jan Pelzl, chapter 7 "The RSA Cryptosystem" [11].

Message authentication can also be done by public-key cryptography. First the message is hashed using a secure hash function, for instance SHA-2 [39], which creates a digest. The digest is then encrypted with the private key and the "digital signature" is then sent with the original message. The receiver can then verify the integrity of the message by computing the hash of the message using the same secure hash function and decrypt the "digital signature" using the senders public key. If they are a match the receiver can with certainty conclude that the message has not been tampered with and originates from the sender.

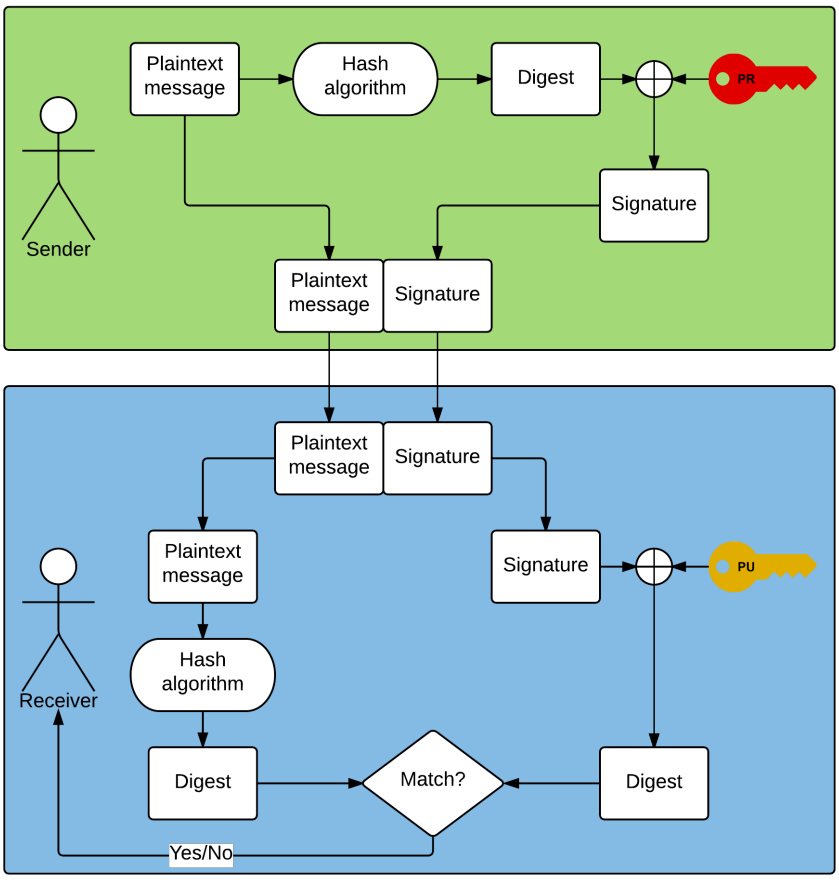

Figure 2.8: Digital signing using public-private key pair.

One of the key issues with public-key cryptography is verifying the identity of the public key owner. Authentication is not a trivial matter and is solved by using certificates to prove that the holder of the key is who they claim to be. The certificates needs to be signed by a trusted third party whom both the sender and receiver of the message trust. The mutual trust relationship is a complex matter and is not a topic we will investigate further in this thesis, but we will assume that it is possible to achieve.

2.3.2 Symmetric-key cryptography

Symmetric-key cryptography uses the same cryptographic key for both encryption and decryption. There are two main areas of application for symmetric-key cryptography; secure storage of data and secure communication. Secure storage of data is the most straight forward of the two. An entity (user, server, etc.) generates a key, encrypts the data using the key, stores the key for future use and decrypts the data using the key whenever the entity require the data. As long as the key is stored securely and the encryption algorithm is secure the data can be stored in an unsecured environment. Secure communication using symmetric-key is similar, but instead of the same entity decrypting the data, the encrypted data is transmitted to a new entity which decrypts it using the same key. This requires the key (known in this case as shared secret) or key generation process to be known by both parties. There are two methods for symmetric-key encryption/decryption. Stream ciphering takes one byte at the time and encrypts/decrypts it whereas block ciphering takes bigger chunks of data and encrypts/decrypts the data. Both methods have their own weaknesses and strengths [11, Ch. 2.1.1].

Stream ciphering is fast and relatively simplistic to implement. This along with the fact that it can encrypt byte by byte makes it very suitable for use when plaintext data comes in unknown length and over time (streams). Areas of application includes voice chat, video feed and http communication. Disadvantages of stream ciphering is that if the algorithm is cracked then it is susceptible to insertions and modifications as well as the fact that a single plaintext symbol is represented as a single ciphertext symbol (limited alteration).

Block cipher is a more complex and requires more overhead. First of all, data must be divided into equal size blocks. The blocks cannot be too small as they would be prone to dictionary attacks and not too big as this would make the encryption/decryption process too resource intensive. In block ciphers the block size must be fixed through the encryption process. This will often result in redundant data. For example, 200-bit plaintext with a 64-bit block size will result in three blocks of 64-bit and a fourth block with only 8 bit of "real" data and 56 bit of redundant bits. Since block ciphering uses the previous block to cipher the current block it is possible to detect tampering and faults, but this also results that data may be lost if a block becomes corrupt.

Advanced Encryption Standard (AES) is one of the most common symmetric-

key cryptography methods. AES does not rely on number factorization (opposed to RSA), but rather substitution and permutation using a key. It can be seen as hashing data and being able to reverse the hashing using the same key. There are currently no known analytic attacks against AES which are less complex than brute-force attacks and consensus is that it is secure as long as long enough keys are used.

2.4 Mobile technology vulnerabilities

Mobile technology vulnerabilities and attack vectors are numerous and before looking into how smart cards can help mitigate an alleviate threats we will need to identify and characterize them.

2.4.1 Physical access

An attacker may gain physical access to the users mobile device through theft or simply that the user misplaced the mobile device. With physical access to the device an attacker would be able to retrieve data from the device. A common defense against this is encrypting the data on the device, but this requires the keys to be stored somewhere securely. If the keys are not stored securely the result is that the data on the device may fall into the wrong hands.

Apart from encrypting the device a more basic form of protection against physical access is using screen locks along with disabling USB debugging etc. What is important to remember is that if an attacker or organization has enough time and resources they may at some point become successful in breaking these security measures. This was recently proven correct by the Federal Bureau of Investigation when they were able to hack into the iPhone 5C used by one of the "San Bernardino shooters" after Apple refused to help bypass the security measures [17].

2.4.2 Remote access

An often overlooked attack vector is badly implemented applications on the mobile device. Inherently a lot of functionality is secure, but due to negligence or bad planning functionality is implemented in a bad way. This

can include memory leaks, weak cryptography, open for code injections or openly exposing private data to third parties. We classify these vulnerabilities as "remote" vulnerabilities as an attacker rarely needs physical access to exploit them. In the "Top 10 Mobile Risks 2014" from OWASP [35] most of the bullets fall under this category. For instance, client side injection allows attackers to remotely execute code on a user device by abusing poor validation of resources.

In rare cases an application with flaws may expose other applications for attacks, but there exists countermeasures to this, for instance that all applications run in their own sandbox. In Android, all applications run in their own environment and applications can only access their own resources and shared resources. The vulnerabilities mentioned above can potentially enable sharing resources that was meant to be private.

2.4.3 External vulnerabilities

External vulnerabilities differs from remote access by that external vulnerabilities are not tied to the device, they focuses on data or information in transit.

Communication is a vital part of modern systems; data is sent between devices and between devices and servers. Sensitive data requires a secure communication channel which cannot be tapped into by a third party. Secure communication on public networks involves agreeing upon encryption keys which the data should be encrypted with before being sent. Encrypting the communication channel will protect against man-in-the-middle attacks, but this requires both parties to authenticate themselves as encrypting the data won't help if you are sending the data directly to the attacker.

As mentioned above, a secure communication channel is useless if you are sending the data directly to the attacker. A vital part of communication security is being able to authenticate the parties in a communication transaction. If the attacker is able to impersonate another party by installing fake certificates on the mobile device or by tricking the user into communicating with the attacker the consequences can be of great significance. All external parties should be treated as hostile or untrusted parties until proven otherwise.

2.4.4 The result of infected or compromised devices

The type of virus or malware on an infected device can vary, some are harmless and serve more as an annoyance or trying to trick the user into visiting bogus websites, but some are more malicious and will access private files and information. From a security stand-point it is a disaster if a virus or malware is able to read and modify data which is otherwise confidential.

Often the user will not know that their mobile device is infected and some viruses or malware are very hard to detect by anti-virus. The "2015 Cheetah Mobile Security Report" [1] reports that the number of viruses on Android devices exceeds over 9,5 million and that the problem is growing. That there exists over 9,5 million viruses for Android shows that Android is a sought after platform to compromise.

When designing and developing applications one should take into consideration that the mobile device may be infected or compromised as well as the possibility for the device to at some point become infected or compromised.

Chapter 3

Smart Card Framework - Goals and Environment

As mentioned in section 1.2 we want to create an Android framework allowing easy development of applications that utilizes smart card capabilities to enhance their security. The framework should implement basic communication protocols for different smart cards and basic out-of-the-box security functions. Both of these should be extendable so that developers can implement their own smart card based security. We want to achieve this not only for testing, but to be able to hand over a ready to use framework for any parties interested so that further development and testing may continue. In this chapter we will discuss the basis for the framework and the environment we will use for creating it. The framework should lay the very foundation needed for using the mobile device and smart card in a secure fashion. The basic things we want to cover are:

- Secure communication
- Key management
- Basic encryption

If we manage to create a framework covering these three points we believe that we have a great starting point for further testing and development.

3.1 Design goals

The framework should be functional and require little work to integrate into an application. Therefor the basic design goals for the framework will be:

- Easy to use.
- Little to no understanding of Java Card and smart cards required.
- Extendable.

Even though most users of our framework will have a basic understanding of smart cards we believe that abstracting some central concepts will make the framework easier to use. One of the concepts we abstract is APDU. As a user/developer of the framework you can choose not to work with APDUs and use pre-implemented methods.

As we cannot possibly predict all types of uses for the framework we will also include a method for sending custom commands to the smart cards. This ensures that developers don't feel limited in how they can use the framework as well as catering to advanced users. More on the implemented methods in section 5.2.

Along with the Android framework we will also provide a simple Java Card applet that corresponds to the functionality we implement in the Android framework. This applet should follow the same principles as the Android framework, but will require some understanding of smart cards by the developer.

3.2 Development tools and technology

3.2.1 Smart card

The first part of the complete framework is the application on the smart card. This part of the framework will perform the tasks that we can place on the smart card.

Java Card version
The cards we have support Java Card 2.2.2 and this is the version we will

target. A natural question is "Why don't we target Java Card 3 and above?". Smart cards used for banking or handling other highly confidential data needs to be evaluated under the Common Criteria [4, Ch. 26.3.2] standards. Potentially an application may handle confidential data and as a result we want smart cards with a Evaluation Assurance Level (EAL) 4 or above. Achieving EAL4 or above is an expensive and long process and relatively few products have this certification.

Level	**Description**
1	Functionally Tested
2	Structurally Tested
3	Methodically Tested and Checked
4	Methodically Designed, Tested and Reviewed
5	Semiformally Designed and Tested
6	Semiformally Verified Design and Tested
7	Formally Verified Design and Tested

Table 3.1: Evaluation Assurance Level

Table 3.1 shows the difference between EAL levels. Comparing the EAL levels is a rather hard task (other than looking at their name and what they test) as there is no guarantee that what has been tested corresponds to the real world [4, Ch. 26.3.3].

When we decided on Java Card 2.2.2. we had to consider if we wanted a newer Java Card version with more functionality or if we wanted to comply with government directives (EAL requirements). The obvious choice was the latter as we have to comply with government directives.

The micro SD card we have access to are certified with EAL5, but only supports Java Card 2.2.2. [22]. This is also the case with the contact+contactless cards we have access to [21]. The Java Card operating system supports "Java Card v2.2.2 (3.0.1 for the Elliptic Curves algorithms)", which means that some of the cryptographic functionality of Java Card 3.0.1 is present. Refer to table 3.2 for available cryptographic algorithms.

Type	Supported algorithms
Symmetric-key cryptography	3DES (ECB, CBC), AES (128, 192, 256 bits)
Public-key cryptography	RSA (up to 2048 (on-card generated), up to 4096 (off-card generated))
Hashing	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512
Elliptic curves	ECC(up to p-521)

Table 3.2: Available cryptographic algorithms in IDCore 3010 and IDCore 8030.

Development environment

In order to develop applications for the smart cards we will be using Eclipse 3.2 with Java development kit version 1.6.45. In order to develop smart card applications more easily we will use the Eclipse-JCDE plugin [14] which provides a virtual runtime environment along with build tools. Even though Eclipse 3.2 is severely outdated it provides the tools necessary to do the job.

In figure 3.1 we can see the tools for generating the deployable smart card application package (.cap file). The screenshot also shows how the editor looks like any other Eclipse version. Even though this version of Eclipse includes tools for sending and receiving APDUs to the application (testing), we have decided not to use these tools as they proved themselves to be unstable and not representative of real world use. This is mostly due to the fact that the application is deployed to an emulator and does not have *any* hardware limitations of a physical smart card.

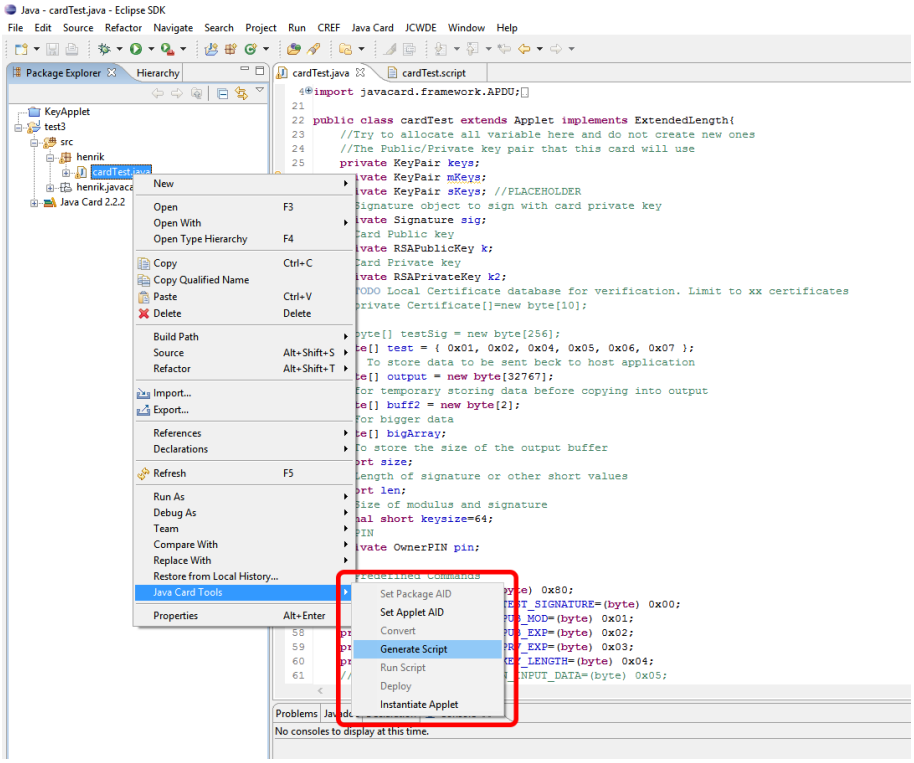

Figure 3.1: Screenshot of Eclipse Java Card tools.

Deployment

We will be using GlobalPlatformPro (GP) [19] to deploy and manage applets on the physical smart cards. GP is a command line tool and is compatible with our hybrid Gemalto card with reader as well as the micro SD card. In figure 3.2 we can see which part of the assembly line GP is responsible for. GP takes the generated .cap file and deploys it to the smart card via the card reader using the computer drivers.

There are three essential steps when deploying an application to a smart card:

- Delete the smart card application along with the stored data.
- Delete the smart card package.
- Install the new smart card application.

To do this we will utilize a simple batch script which consisting of three

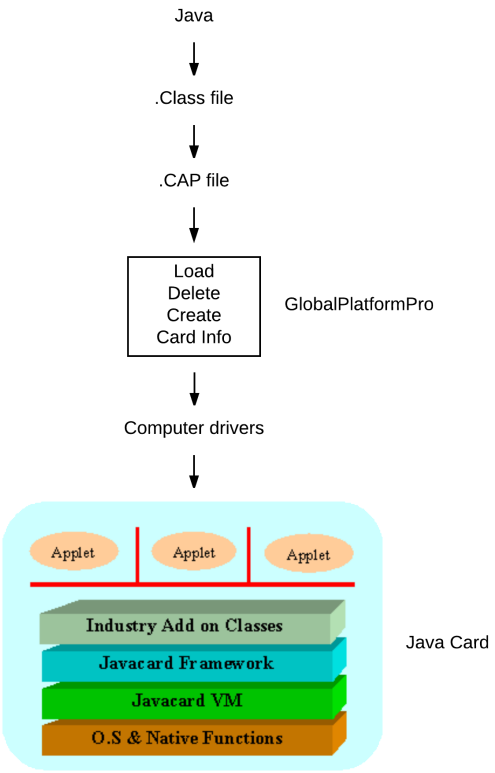

Figure 3.2: Deployment line using GlobalPlatformPro.

lines of code (listing 3.1). To gain access to the cards we need to provide a key, set by the manufacturer. This requirement is an additional security measure to verify that developers are supposed to have write access to the smart cards. Lastly we supply the AID we wish to use for our application. It is important to note that the AID must be unique and the installation will not succeed if the AID is in use.

Listing 3.1: Install and deploy script for GlobalPlatformPro.

```
1    gp.exe -visa2 -key %KEY% -delete %AID%
2    gp.exe -visa2 -key %KEY% -delete %PACKAGEID%
3    gp.exe -visa2 -key %KEY% -install %PATH% -d
```

Test environment

To test the smart card application that is deployed on the physical cards (without going through an Android application) we will be using PyApduTool [37]. pyApduTool is a tool for sending APDUs to a smart card through a card reader or memory card reader and lets us observe how the card behave when receiving and transmitting data.

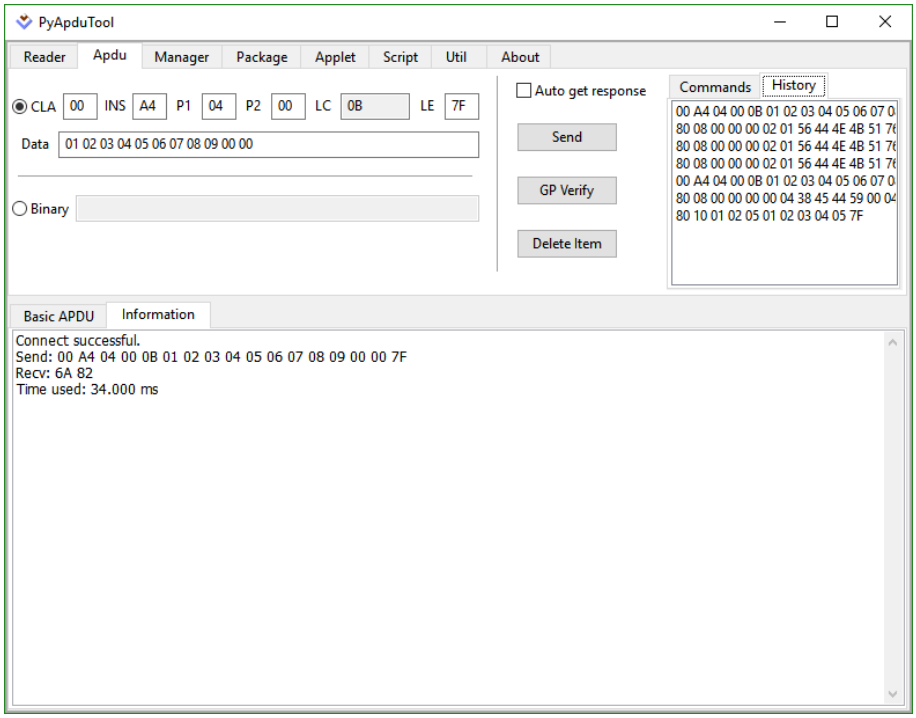

Figure 3.3: Select APDU sent to smart card via PyApduTool

PyApduTool does not support extended APDU and this limits us to a high degree when testing our smart card application. Testing through PyApduTool does not test mobile device behavior such as: out of memory, too much traffic or NFC limitations. Another key point is that it is difficult to create test tools that works all smart cards as manufacturers make small adjustments in their version of Java Card. The results was that we often encountered weird errors with seemingly no clear cause. After the initial basic testing, PyApduTool became obsolete and we had to test via an Android application.

As a result we are very limited when it comes to testing the performance

of the smart cards. The only way of measuring resource usage on the smart cards are with time stamps and by looking at the elapsed time do an evaluation on the performance. This makes it difficult to identify bottlenecks on the smart cards.

3.2.2 Android application

The second part of the framework is an Android library. The library will serve as an intermediate between the Android application and the smart card application.

Android version
When we started working on the Android library, our mobile devices were running Android 4.4 (API level 19). By the end of this thesis Android 6.0 became more popular and we were able to migrate our framework to Android 6.0. The minimum SDK required for the library is API level 19 (Android 4.4) and the target SDK is API level 23 (Android 6.0). Google frequently provides data on what Android versions their user base uses. As of May 2, 2016, Android version 4.4 to 6.0 covered approximately 78% of the users [6]. Our opinion is that covering $\tilde{7}8\%$ of the user base is a realistic and sufficient goal.

IDE
Android Studio [7] is the official IDE for Android application development. Android Studio is based on IntelliJ IDEA [24] and provides many automated tools for building, deploying and publishing Android applications. Android Studio ships with Android Debug Bridge (ADB). ADB is an interface for communicating with virtual Android instances or physical Android devices. ADB gives developers the ability to log output from applications as well as monitor memory, GPU and CPU usage of the mobile device.

We utilize the log ability of ADB to great extent as it is more efficient in our case to look at byte values rather than constructing complex graphical user interface elements for our use cases. In essence, the log output from ADB is our graphical user interface when testing the various functions of the framework.

Test environment

To test the application we will be using the built in ADB in Android studio as well as doing empirical tests on the Android device. Figure 3.4 and 3.5 shows runtime examples of the Android device. These monitor tools gives us a clear indication if we are doing an operation the Android device cannot handle or if we are trying to perform operations that are too resource intensive.

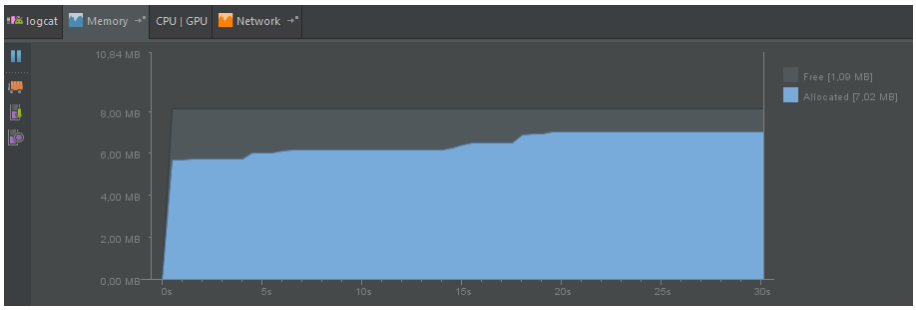

Figure 3.4: Android Debug Bridge memory monitor connected to a running Android device.

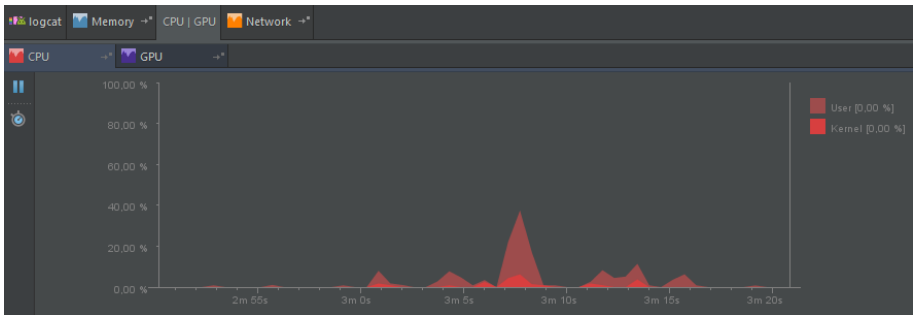

Figure 3.5: Android Debug Bridge CPU monitor connected to a running Android device.

Even though ADB provides us with good resource usage tools we wish to perform more informal tests using timers to get a feel for how long an operation takes. We can use the built in Android class `System` and the method `nanoTime()` to get an accurate start and stop time for an operation and calculate elapsed time. This combined with visually inspecting the running application can help us get an indication of how responsive the application is when executing tasks.

3.3 Development flow

Recall the tools and technologies from the previous sections. Figure 3.6 shows the development process. The process is linear from top to bottom, but it is important to note that we will need to develop for both the Android application and smart card applet in parallel. By parallel we mean that in order to test some new functionality we will need to implement functionality on both platforms.

The figure clearly shows that everything concerning the Android side of the framework is handled by Android Studio along with the Android SDK except for the Gemalto library for micro SD support. Google has put a lot of resources into streamlining the Android development process whereas we are dependent on independent or proprietary software for smart card applet development.

One important thing to note is that the figure shows that we cannot observe any test results of the smart card. Observation of the smart card's test results must be done via the Android application (refer to section 7.3).

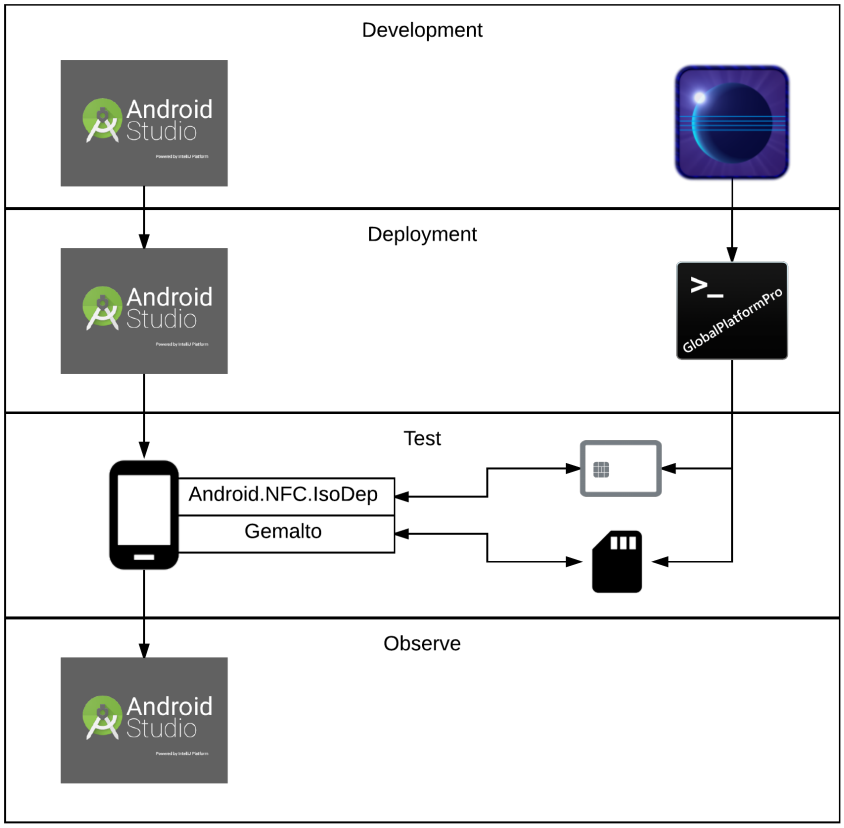

Figure 3.6: Development flow of the smart card framework

Chapter 4

Challenges and use cases

In this chapter we analyze some of the challenges one may face when trying to use smart cards to enhance the security of Android mobile devices. In chapter 2 we covered the attack vectors and vulnerabilities that exists on the mobile device platform and we believe that smart cards can help mitigate them. We have established that smart cards are able to run critical operations in a secure and closed execution environment along with the fact that they are tamper resistant. As a result they are perfectly suited to generate and store cryptographic keys, perform cryptographic operations such as verifying signature or encrypt sensitive data, or enforce strong access control through PIN codes.

On the other hand, smart cards have limited computational power and memory, they are passive elements meaning that they cannot initiate an operation and must rely on external commands, as well as the fact that they are not "aware of the outside world" except for pre-installed information or the information they receive.

Based on these characteristics we have identified two main areas of use where smart cards can be used to mitigate threats on the mobile device platform:

1. Generate and store cryptographic keys to mitigate the threat of data loss because of a stolen or lost device. Mobile devices that store cryptographic keys without using smart cards or secure elements are prone to key theft via operating system exploits or weak user passwords.
2. Use the smart card as a separate trusted operating system that can se-

curely handle the communication with off-card and off-device services and enforce simple security policies. As it is a separate execution environment, this can be achieved even if the mobile device operating system is compromised.

However, in order to ensure that these proposed areas of use function as intended, we will need to overcome some obstacles. First off we will need to establish a trust relationship between the mobile device and smart card in order to verify that they are meant to cooperate and lock/bind them to each other. In section 4.1 we elaborate and investigate this challenge. Secondly we will need to define how smart cards and mobile devices handle the encryption keys needed for a secure solution. With this challenge the best we can do is to have a "best effort approach", meaning that we have to trust that the solutions provided by the mobile platform are properly implemented. Section 4.2 investigates this problem. Lastly, we describe how policy enforcement might function using smart cards in section 4.3.

In the next chapter we will perform tests related to the proposed solutions. Even though we were not able to fully implement and test all aspects, we still chose to outline how they can be solved.

4.1 Binding card and mobile device

4.1.1 Problem description

One of the key challenges when utilizing smart cards with mobile devices is establishing an initial trust relationship. How can the mobile device know that it communicates with a certified smart card (company/department issued) and how can the smart card know that it is interacting with a trusted user on a mobile device? The biggest problem of the binding process is that the smart card must trust the mobile device, as we have no way of knowing if we are binding to a compromised mobile device. If the mobile device is not initially compromised and we are able to use the smart card as a bootstrap for trust, then the direct result is that we can use smart cards as a policy enforcement point (PEP) and secure key storage/generation. To initialize this trust relationship we need to perform a handshake where we verify that all concerning parties can authenticate and authorize each other.

Binding the mobile device and smart card mainly protects against offline attacks where the attacker tries to access resources on the mobile device or

smart card independently of each other. For instance if the attacker tries to use the smart card with an un-paired device to access the stored keys on the smart card.

4.1.2 Goals

By binding the smart card and phone together we wish to ensure that a smart card can only be paired with one mobile device and that we are in full control during the process. If we achieve this, then:

- The keys stored on the smart card cannot be retrieved or be used on a different mobile device.
- Our application on the mobile device cannot be used without the smart card that was paired with our mobile device.
- If the binding is successful we may be able to detect if the mobile device becomes compromised on a later point and react to it (delete keys on card, block communication, etc.).

This will mitigate vulnerabilities such as:

- Lost or stolen device.
- Unsecure communication channels.
- Authentication challenges.

4.1.3 Key concepts

To authenticate the two parties, smart card and mobile device, we will need a third party which they both trust. We introduce a new party, the authority, which acts as a trusted third party. The authority issues the smart cards and employ the users. A direct consequence is that they both trust the authority, otherwise we have no starting point. Since they both trust the authority they can ask the authority to verify the other party as shown in figure 4.1.

One thing that differentiates our challenge from traditional authentication challenges is that the smart card is not able to communicate directly with a trusted third party. All communication from our smart card to off card applications or third parties must go through a mobile device. A technical

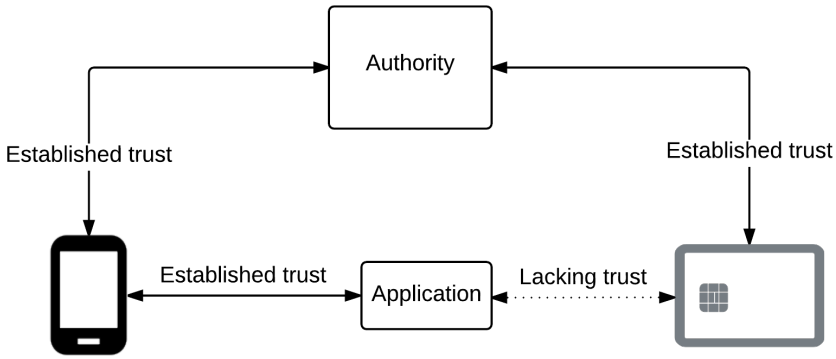

Figure 4.1: Using a third party (Authority) to establish a trust relationship between two parties (Application and smart card) lacking trust.

illustration of this relationship can be seen in figure 4.2 where the authority is represented as a server. This drawback introduces a new problem which we have to consider. How do we know if the mobile device relays information between the server and smart card correctly in the authentication process?

To address that all information flowing from the smart card has to go through a possibly compromised mobile device, we can utilize the fact that the authority we are trying to communicate with is also the smart card issuer. What this means is that we can pre-install the authority certificate and public key on the smart card as well as retrieve the public key of the smart card before handing the smart card to the user/employee. In other words, the authority and the smart card have already exchanged all necessary information to securely authenticate each other before deployment.

The mobile device will also need to authenticate with the authority so that the smart card can trust the mobile device. In this process we will need to make some assumptions. The first problem is that we need to ensure that the mobile device communicates with the right server (authority). By hardcoding the server URL in the mobile device application and making sure that the user installs the right application on his device we can mitigate this threat. To make sure that the user installs the right application we need a secure distribution platform. By using Google Play as distribution platform,

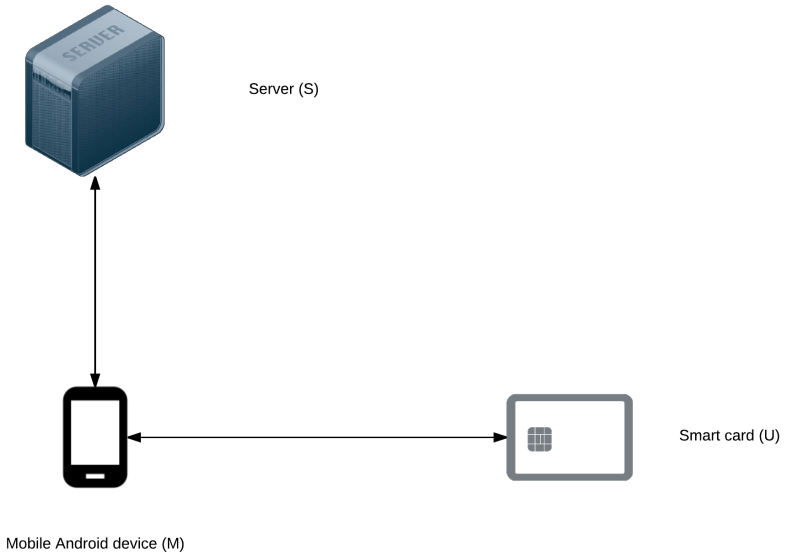

Figure 4.2: Server, mobile device and smart card communication flow.

we can minimize the risk that the user will download a rogue application with the same name [36]. To further mitigate the risk of installing the wrong application the user can disable the ability to side-load applications and avoid using a rooted device.

Another possible attack we should defend against, is man-in-the-middle attacks between the application and the server. If we assume that the user was able to download the correct application we will need to secure the communication channel. To secure the channel we will need to use Transport Layer Security (TLS) [54] which also provides us with protection from replay attacks [32, Ch. 9.2.2]. The only downside by using TLS in our case is that we will need the server certificate to verify the server. Traditionally we will need to either pre-install certificates on the mobile device (makes "bring your own device" more difficult) or register with a certificate authority (depend on third party). In our case the server certificate is already on the smart card and we can install it on the mobile device. If the mobile device is compromised the certificate may not be used at all or replaced by a malicious certificate. This can be done as the smart card has no control

over the mobile device. However, as we describe later, we encrypt some particular data with the public key of the authority on the smart card, and we are thus not prone to man-in-the-middle attacks since the attack cannot decrypt it.

Further we will need the user to authenticate with the authority. We have two options in order to achieve this. First option is that users use a username and password combination directly with the authority, but considering the binding is normally a one time case a more simplified process may be to hand out a one time code along with the smart card. One could also look into distributing one time codes through e-mail or a text message (SMS).

The second option is that the user inputs a PIN to the smart card and if the pin code is correct the smart card can verify that the user is the user he claims to be. This option requires very little overhead and saves a lot of resources in that regard. The problem with this approach is that if the mobile device is already compromised, the PIN code may be stolen before the user is able to bind the mobile device with the smart card. If the PIN is one the user regularly uses an attacker may also be able to get the information from the user via other means (social engineering, key loggers, etc.). A one time PIN (OTP) may be a better option than a PIN although this does add more overhead to the process.

The final requirement for the binding process, is that we have a cryptographic mechanism that in practice binds the mobile device and smart card together. We define the binding process to be done when both parties have the public key of each other and have verified that the keys they have are in fact the keys of the two parties. When this is completed, the mobile device and smart card can mutually authenticate each other before initiating any transaction involving sensitive data.

We have described how the parties can authenticate each other, but we will need to describe this as a unified process (refer to section 4.1.4) and identify attack vectors and weaknesses (refer to section 4.1.7).

4.1.4 Proposed solution

Pre-conditions
The authority issues the smart cards and administrates the server. During the setup of the smart card the public key and certificate of the server must be installed on the smart card. The public key of the smart card must also

be extracted and stored on the server. This creates a base for all future processes.

Verification package
A verification package is a package containing all the information a third party server needs to authenticate the smart card and mobile device. First the mobile device public key and a newly generated AES key is signed by the smart card. Then we encrypt the package with the public key of the third party server. Figure 4.3 visualizes this structure.

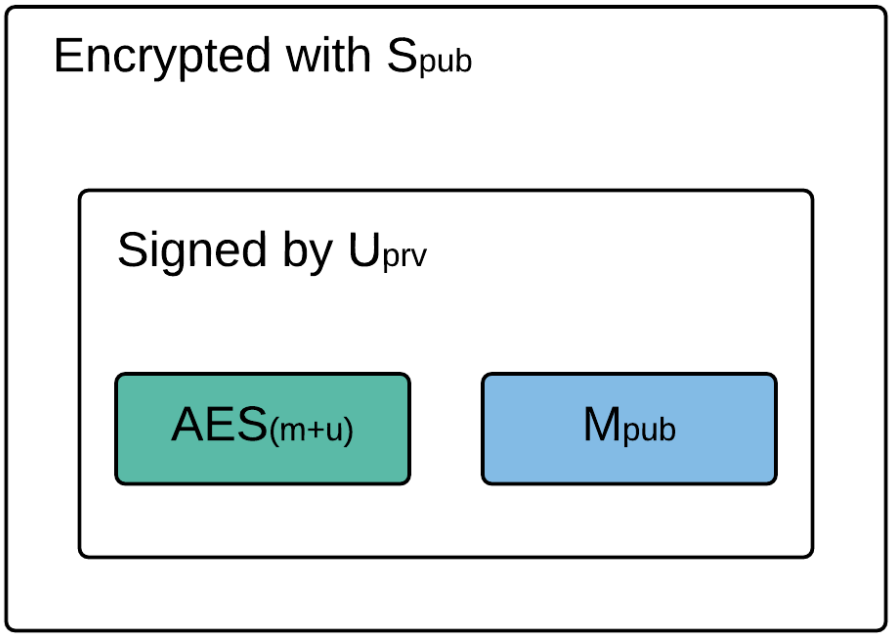

Figure 4.3: Verification package structure.

We propose the following protocol for binding mobile devices and smart cards:

Abbreviations
U - Users smart card
M - Mobile device
S - Server, representation of authority
H_0 - Verification package

$\{\text{Entity}\}_{\text{pub}}$ - Public key of an entity (U, M, S)
$\{\text{Entity}\}_{\text{prv}}$ - Private key of an entity (U, M, S)
$\{\text{AES}\}_{\text{Entity+Entity}}$ - AES key of two entities (U, M, S)

1. Install the (correct) Android application the on mobile device (M) and insert smart card (U).
2. M generates RSA key-pair and stores it securely on the device.
3. M sends M_{pub} to U and requests verification package (H_0) from (U).
4. U asks for a PIN/OTP.
5. M provides PIN/OTP.
6. U generates H_0 (refer to figure 4.3) and sends it to M.
7. M connects (URL is either hardcoded into application or from certificate) to the server (S) via TLS and sends (H_0) to S.
8. S decrypts (H_0) using S_{prv} and verifies the signature of U.
9. If everything is ok then S saves $AES_{(M+U)}$ for safekeeping, signs M_{pub} and sends the signed M_{pub} to M.
10. M forwards the signed M_{pub} to U.
11. U verifies that M_{pub} was signed by S and if successful U sends U_{pub} to M.

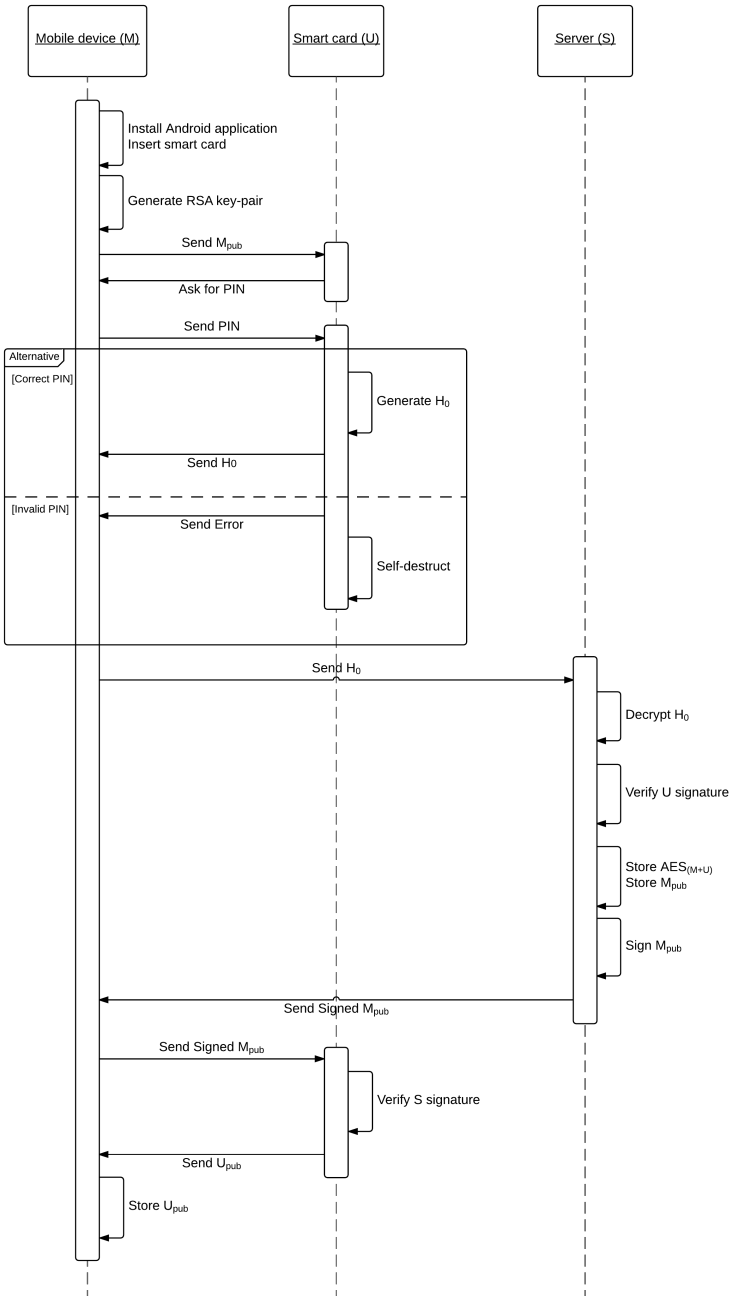

Figure 4.4: Sequence diagram for binding mobile device with smart card.

The end result of the transactions is that the smart card and mobile device have shared their public keys through the trusted third party and can thus communicate securely. The server will also have a record of the transactions and the parties. If we chose to do so we can also save a backup of the first symmetric key on the server.

Potentially we could add more steps to the process to further heighten the security. In step 7 we could add that a user may need to answer a challenge such as provide a one time password (OTP). This would add more overhead and require more resources administrating.

A direct consequence of a successful binding is that the smart card is now locked to the mobile device. Inserting the smart card into a different device will simply not work due to the fact that the smart card has the public key of the original device and they have already agreed upon a symmetric key. To further enhance this trait, we require the mobile device to identify itself if sensitive services are requested. This can be done by sending a challenge from the smart card to the mobile device which requires the private key of the original bound device to solve (e.g. challenge is encrypted with M_{pub}). In addition since the smart card is able to securely communicate with the authority the smart card can lock itself up and require a signed package from the authority. This package could contain information regarding deletion of keys, force a new binding, etc. We discuss this possibility further in section 4.3.

4.1.5 Protocol analysis

In this section we will justify and evaluate the parts of the solution that have security implications. Steps that are present for the solution to function, but with no security implications, will be skipped.

1. Install the (correct) Android application the on mobile device (M) and insert smart card (U).
Installing the correct application is a vital part of the protocol. A malicious application can spoof all communication between the smart card and the mobile device, although it will not be able to compromise the communication between the smart card and server because of their pre-shared public keys. All user and application data that at some point is handled by the malicious application would be considered compromised. As mentioned earlier we can

use Google Play as our distribution platform which will minimize the risks involved.

2. M generates RSA key-pair and stores it securely on the device

For the smart card to bind itself to a mobile device we need a unique identifier or key that no other device is able to replicate or spoof. An RSA key-pair provides this functionality as the mobile can use the private key to sign data and the smart card can encrypt data with the mobile device public key. Unless another mobile device is able to extract the private key we are in the clear security wise. More on this and other solutions in section 4.2.

4. U asks for PIN/OTP

We chose to add a PIN code step to the binding process to add another layer of security. The PIN code ensures that a person is verified by the employer to perform the binding process. Using PIN codes is not a guaranteed measure against someone unauthorized trying to carry out the binding process. The important steps to make sure the PIN code process is secure are:

- The binding process should be carried out as soon as possible after obtaining the PIN code to avoid someone leaking or losing the PIN code.
- Add a limited number of tries for inputting the PIN code on the smart code to mitigate brute-force attacks.
- In connection to the previous point; the PIN code length should correspond to the number of tries.

5. M provides PIN/OTP

At some point the user will need to supply the mobile device the PIN or OTP in order for the mobile device to send it to the smart card. Theoretically an attacker can have compromised the device and intercept the PIN/OTP to use it with another device. This scenario is unlikely as the attacker will also need to get the physical smart card before the binding process is completed. Potentially an attacker can perform a denial of service attack in this step and never let the binding process complete. In section 4.1.7 we discuss the possibility to use Google SafetyNet to detect malware on the device.

6. U generates the verification package.
The smart card is a secure environment and should be in charge of generating the verification package. We include the AES key for safekeeping on the server incase the user loses the smart card. We sign the package using the private key of the smart card so that the server can verify that it is a legit smart card since the server has the public key. This step is necessary as anyone would be able to send a verification package to the server as the server public key is public. Lastly the smart card encrypts the verification package using the server's public key. The end product, the verification package, is secure in the sense that only the server can read the data and the server can authenticate the sender using the signature even if the application or TLS connection have been compromised.

7. M connects to the server (S) and sends the verification package to S.
The verification package is encrypted with the server's public key. The direct result is that even if the package is lost or leaked no third party would be able to read the content. We will use TLS for the connection regardless as it may be necessary to add additional functionality such as username-password login to verify the user. To establish a TLS connection one would access to the server certificate either via a third party certificate provider or via a pre-installed certificate. TLS will also serve as a counter to replay-attacks and man-in-the-middle attacks.

9. The server signs the public key of M.
If the signature of the smart card is in order we can proceed with generating the response package. The server signs the public key of the mobile device. This is done because of the need to confirm that the verification package was indeed sent to the server. By letting the server sign a response which is then forwarded to the smart card lets the smart card verify that the server approved the user/mobile device.

11. U verifies that $\mathbf{M_{pub}}$ was signed by S and if successful U sends $\mathbf{U_{pub}}$ to M.
Even though public keys usually are publicly known we choose to keep the public key of the smart card semi-public or on a "need to know basis". Using this technique we do not inherently make the solution secure, but it does add another hurdle a potential attack will need to overcome. In

theory, the more steps an attacker will need to do; the higher the chance for detecting him. By rotating the keys the effectiveness of this measure increases substantially.

4.1.6 Cryptography evaluation

This solution relies heavily on correct use of protocols such as TLS (communication), cryptography such as RSA and AES, and correct key generation. Section 2.3.1 and section 2.3.2 describes RSA and AES and why they are secure. Assuming we use them correctly we can conclude that this part of our solution is secure.

In the solution we mitigate man-in-the-middle attacks using TLS for secure communication. TLS can utilize both RSA and AES and if we use strong keys we deem it secure (assuming TLS 1.2) from a mathematical perspective. If TLS is implemented correctly and does not allow for common attacks (Heartbleed, DROWN [51], etc.) it is classified as "probably secure" or "secure until proven otherwise".

Correct key generation is discussed in section 4.2 and the conclusion is that it is possible to securely generate keys. All cryptographic parts of the solution is considered secure if done correctly and we can thus conclude that the cryptography included in the solution is secure.

4.1.7 Potential attack vectors

Rogue technical party
The three technical parties involved are the server, the mobile device and the smart card. As discussed previously "the authority" issues the smart cards and administrates the server. Since the public keys and certificates are exchanged before the smart card is distributed the server is able to detect if there are a rogue/fake smart card trying to bind to a mobile device. If the mobile device tries to connect to the wrong server (man-in-the-middle attack, wrong URL, etc.) and the server tries to pose as a legit server, it will not be able to complete the binding process due to needing the matching private key for the public server key on the smart card.

Thus the only attack vector on technical parties is where the mobile device is rogue. By rogue in the context of the mobile device we mean compromised as in rooted or malware/spyware. In our solution we have no way of knowing

if the user is binding a rogue mobile device. The end result is that we have securely bound the mobile device and smart card, but the mobile device cannot be trusted.

To address this we need to do two things. First and foremost we need to educate the user on mobile security and how they should not install applications from untrusted sources etc. Secondly we can run tests on the mobile device to try and detect if the mobile device is rooted or has malware/spyware. This can prove to be hard as it is very difficult, if not impossible, to detect malware/spyware which operate with root access. Google has been working on a security framework, SafetyNet, which goal is to detect if a device has been tampered with or is infected [40]. In order to decide if this is sufficient we would have to do more research on SafetyNet specifically.

Rogue user or administrator
Potentially we can have a rogue user which deliberately installs malware/spyware on their mobile device to compromise our system. We will disregard this case as if we have a rogue user we have bigger problems than a compromised mobile device. It is also important to note that any information the mobile device receives the user is also likely to know regardless, and can thus release this information independently of the mobile device.

The bigger problem would be a rogue administrator. The administrator would have access to the initial setup of the smart cards and may extract the private key of the server. Even though this has more impact than a rogue user we are very limited on what we can to do protect against it. We can make it near impossible to extract private keys, logging and require more than one administrator, but it will still be possible to cause harm. Although the same principle applies here: if you have a rogue administrator you have bigger problems than smart card binding.

4.1.8 Additions

In step 9 and 10 of the proposed solution we can add a payload to the signed M_{pub}. One of the uses for this payload may be to send information on how the smart card should handle communication, key generation, encryption & decryption as well as administration. More on this in section 4.3.

4.2 Mobile device keys

4.2.1 Problem description

Even though one of the features of the smart card is to store and manage keys, we are still dependent on the mobile device being able to securely store at least one set of keys. This is directly tied to the binding process of the mobile device and the smart card which were discussed in section 4.1. The problem lies in the fact that we assume that the key pair generated by the mobile devices cannot be extracted and installed on another device and can only be used by our application. The question is: how we can be sure that this is the case? We have no way of proving that the keys are generated on the mobile device. And even if we were, are they stored securely? We can also envision that there might emerge other use cases at later stages which require keys on the mobile device.

4.2.2 Goals

Our primary goals for mobile device keys are:

- Make sure that the device keys are actually on the device.
- Store keys securely, ensuring that they cannot be exported or used by other applications or devices.

4.2.3 Key concepts

Android Keystore system
The Android Keystore system is a system that lets users and developers store and access cryptographic keys and certificates on the mobile device. The main goal of the system is to protect the keys against unauthorized use and extraction. This is done by defining which applications that should have access to the keys stored. E.g. application A generates and stores a key and defines that the key is available to application A and B. If application C tries to access the key the Android Keystore system blocks the action.

The applications do not have direct access to the keys. If an application wants to perform a cryptographic operation it feeds the data to the operating system which performs the cryptographic operation. If an application is

compromised an attacker gains access to the keys via the application, but the attacker is not able to extract the keys as the keys are never present in the application.

In early iterations the keys were stored in a software-protected file meaning that only the Android Keystore had access to the data. This system had a flaw in which any users or applications with root access could access the Keystore file. The solution to this is using secure hardware such as "Secure Element" (more or less a smart card) and "Trusted Execution Environment (TTE)" [56]. If hardware-backed storage is enabled (as seen in figure 4.5), it is not possible to extract keys even if the operating system is compromised.

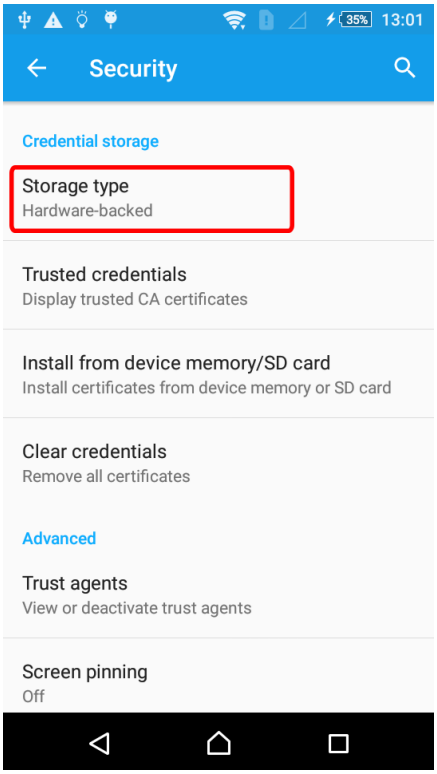

Figure 4.5: Screenshot of Android settings showing hardware-backed storage "enabled".

An application can check if the mobile device uses secure hardware for key storage using the Android class `KeyInfo`. The `KeyInfo` class contains all available information about a key and a single call to the method

`isInsideSecureHardware()` will determine the storage status. Listing 4.1 provides a sample implementation of this functionality. `KeyInfo` was added in API 23 and requires Android version 6.0 or newer.

Listing 4.1: Obtaining storage status of keys using KeyInfo.

```
1  public boolean checkStatus(PrivateKey key){
2      KeyFactory factory = KeyFactory.getInstance(
3          key.getAlgorithm(), "AndroidKeyStore");
4      KeyInfo keyInfo;
5      try {
6          keyInfo = factory.getKeySpec(key, KeyInfo.class);
7          return keyInfo.isInsideSecureHardware();
8      } catch (InvalidKeySpecException e) {
9          // Not an Android KeyStore key.
10     }
11     return false;
12 }
```

4.2.4 Generate keys on mobile device

To generate keys on the mobile device an application must initialize the classes `KeyGenerator` or `KeyPairGenerator`. `KeyGenerator` is used for generating symmetric secret keys and the most notable supported algorithms are AES(up to 256-bit), HmacSHA256 and HmacSHA512. As the name suggest `KeyPairGenerator` is used for generating key-pairs. Pre API level 23 it was possible to generate DSA key-pairs, but the support was removed in favour for more secure algorithms. The two main supported algorithms are RSA (up to 4096-bit) and Elliptic Curve algorithms (P-224, P-256, p-384 and P-521).

Generating long keys may put some strain on the mobile device and should either be done in an asynchronous thread or during a setup process on first time launch of the application. However this should not be a deciding factor of whether or not the mobile device should generate its own keys as it is a one time process. Listing 4.2 shows how an application can use `KeyPairGenerator` to generate a 4096-bit RSA key-pair.

Listing 4.2: Generating RSA key-pair on Android device using KeyPairGenerator

```
1 public KeyPair generateKeyPair(){
2     KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
3     kpg.initialize(4096);
4     return kpg.generateKeyPair();
5 }
```

4.2.5 Generate keys on server

The Android Keystore system accepts *PKCS#12* archive files which it then stores on the mobile device. The PKCS#12 file can contain certificates, keys and key pairs [34]. After the archive file is manually installed on the device, applications that are authorized are free to use the keys via the Android Keystore system. The consequence of this is that we do not have to rely on the mobile device to generate the secure keys.

The most interesting characteristic of this solution is that a company or a similar entity can have a dedicated server generating these archive files. The server is able to run more security software than a mobile device and can be treated as a secure environment for key generation.

The PKCS#12 archive file can be protected by a password and should definitely be protected by one to be even considered secure. This does add more overhead to the process of distributing the PKCS#12 files.

The biggest obstacle when using server generated keys is distribution. You do not want to hand the PKCS#12 file to the user on a USB stick along with a password on a notepad as you cannot be sure that the USB is destroyed or wiped properly along with the password. The solution to this is hosting the PKCS#12 file on a website. The only way to access the PKCS#12 file is through a portal which requires two-factor authentication (or similar) which limits user to only access the PKCS#12 file they are supposed to reach. The file is then downloaded onto the mobile device and the users can install the file from their device memory (see figure 4.5).

This solution does not remove the need for the users to know the password for the PKCS#12 file and the file will still reside in the device memory. The only solution to this is to have strict security policies where the memory is wiped afterwards as well as removing the accessibility of the PKCS#12 file

on the server (from a users perspective).

One alternative when it comes to distributing the password for the PKCS#12 file is using a password generated from a seed (salt) located on the smart card. By using this alternative, only user with access to the smart card is able to install the PKCS#12 file on their mobile device. As an added security mechanism the smart card can lock or delete the seed/password afterwards and only be reset by an administrator. This would give us a certain degree of control over which devices that can have the keys installed.

4.2.6 Evaluation and comparison

In the problem statement we pointed out the binding process being dependent on the security of the keys generated and stored on the mobile device. We know that we are able to store the keys securely on the mobile device as long as hardware-backed storage is enabled and is being used. It is also possible to check if a key is stored on hardware in an Android application. In other words we have a solution to the storage issue: Use a mobile device which supports hardware-backed storage and validate it in the Android application using `KeyInfo`. The paper "Analysis of Secure Key Storage Solutions on Android" by Tim Cooijmans, Joeri de Ruiter and Erik Poll discusses hardware-backed storage further [53].

Security-wise the key generation on the mobile device and on a server is very similar. If the key generation on the mobile device is done correctly with properly generated secure initialization vectors the process is considered secure. The only realistic attack vector on key generation is that the mobile device is running a custom operating system which has it's own implementation of how key generation is done. To address this scenario it may be necessary to use Google's security framework, SafetyNet, as mentioned in section 4.1.7 when we discussed attack vectors on the binding process.

The biggest benefit of generating the mobile device keys on a remote and secure server is that we can strengthen the insecure elements of the binding process. In the proposed solution of the binding process (section 4.1.4) we have to trust that the mobile device is a device that is supposed to perform the binding process. For example if an attacker is able to get his hands on a smart card before it is bound to a mobile device, he may try to start the binding process with his own device. We counteract this with requiring PIN code and the possibility of adding user authentication with the server.

If the mobile device keys are pre-generated by a server and installed by a user on the mobile device, we can use these keys to verify that the device is authenticated (as other devices does not have the keys). A consequence of this is that we may be able to simplify some of the steps in the binding process as we can trust the mobile device.

The drawbacks of pre-generating the keys, apart from more overhead, is the distribution process, which can introduce new attack vectors. In section 4.2.5 we discussed how we could use a web server to distribute the keys. What is also worth mentioning is that this solution requires a web server to be maintained and protected. Such a system also places a lot of trust in the user's hands as he is responsible for deleting the PKCS#12 file as well as not disclosing the password.

An other distribution solution is to have a trusted administrator install the keys on the mobile device. For some organizations this could be cumbersome as now all users will need to visit the "headquarters". This is a direct hindrance to the "Bring your own device"-idea and can potentially introduce extra costs when it comes to human resourcing. Another key point to keep in mind is what should the protocol be if the organization wishes to update all keys? This would put a lot of stress on the distribution department.

However, we assume already that there is a trusted server involved in the binding process, so the PKCS#12 file could simply be distributed together with the response package described in section 4.1.4. This does imply that we have solved the distribution of the server certificate issue which the PKCS#12 can solve as it can contain the server certificate.

There is no definitive "best solution" to the key generation problem. It all boils down to the question: "Can you afford the infrastructure needed for server generated keys?" If the answer to that question is yes then there is a lot to gain by using server generated keys as we have discussed above. Opting for the cheaper solution, generate keys on mobile device, does not equal an non-secure solution, but we do sacrifice some control of the process.

4.3 Security policy enforcement

4.3.1 Definition

A security policy defines what measures a system needs to follow in order for the system, organization, group, etc. to be secure. Security policies are not necessarily a technological restriction/rule and can exist as a socially enforced rule. Examples of security policies are:

- All employees needs to have a background check.
- All company doors will be locked after 4 pm.
- Password must be changed once a month.
- Sensitive data must be encrypted with AES-256.

4.3.2 Problem description

Enforcing security policies via a mobile device is not a new concept and there exist multiple third party solutions for enforcing them. One example is Microsoft Exchange ActiveSync which enforces policies such as minimum password length, disable camera and application blocking [8, 20]. ActiveSync utilizes the fact that a user must connect through Microsoft Exchange Server in order to access company resources and verifies if the mobile device has enforced the policies through the established connection [16].

The problem in these types of solutions lies in the fact that you cannot trust the mobile device to actually enforce the policies. How does the server know if the mobile device is telling the truth about policy enforcement? What if the mobile device says it requires the user to enter a password, but does not enforce it?

4.3.3 Goals

By using smart cards in the context of policy enforcement we wish to achieve the following:

- Policies are enforced.

- Not possible to spoof policy enforcement by the mobile device.
- Policies cannot be tampered with.

Optionally we want to achieve the following:

- Policies can be updated.

4.3.4 Shift responsibility to a trusted party

One way of making sure that policies are enforced is to give the responsibility of the action to a trusted party. Imagine that company A have a policy which requires their employees to update their password on their personal computers once a month. If the user profiles only exists locally on the computers, company A has no way of checking if the users update their passwords and run the risk of the users saying they have updated the password without doing so. To address this issue, the user profiles are moved to company A's server and the personal computers now do a lookup for the user profiles on the server. In this case, A has control over the user profiles and can check if the passwords are updated.

The limiting factor to a solution like this is that the trusted party might not be able to perform the job. A trusted server might not be able to enforce locking doors after 4pm just as a smart card is not able to perform all jobs we want it to do. A big part of the job is to identify which part of a security policy action we can move to the smart card to enforce the security policy.

4.3.5 Proposed solution

Relevant policies for a smart card
The smart card is limited in what kind of policies it is able to enforce. It cannot enforce policies regarding forcing the mobile device into doing things only the mobile device controls. The smart card can provide vital data needed for performing an action, but it is up to the mobile device to actually perform the action. This effectively rules out policies such as, "The mobile device must encrypt all files stored locally.", since we can only provide the keys to do so, but the mobile device has to perform the action. We have to assume that the mobile device, more specifically the running application, wants to perform the action.

The core abilities of a smart card are: generate keys, store keys, encrypt and decrypt small amounts of data, sign data and store variables. We can make the application rely on the keys generated and stored in the smart card and as a result we have full control over the keys. The smart card can as a result enforce policies such as key rotation, key size and key availability (unlock key with PIN). A more general use case is to have the smart card refuse to provide services or information to the application if certain conditions are not met or if one suspect that the device might be compromised.

Installing policies

One disadvantage of using smart cards is that once an applet has been installed it is a static application. It is not possible to add new code to a running applet. What we can do is to program all policies we may wish to utilize and disable them. With this technique we can dynamically turn on and off policies, given that we are able to communicate with the card. The first premise is to install all possible security policies we may need on the smart card before shipping the smart card to the user. The second premise is to translate all policies to a set of parameters which can be combined logically to form rules. If both premises are met we are able to manipulate the parameters and effectively changing the rules.

Enable and update policies

To enable policies or set parameters for the policies we require a protocol to exchange information with the smart card from a server. One of challenges we have is "How do we ensure that the mobile device relays policy updates to the smart card?". One solution is to have the smart card lock itself and require a verification package from a trusted server after a number of operations. It is important to note that the smart card have no concept of time and therefore the smart card cannot use time to control lock cycles. A way to ensure that the server is the only party that can provide the verification package, is to utilize keys that are already on the smart card. For example the public key of the server or the symmetric key exchanged a the beginning of the binding protocol. The simplest way of doing this is utilizing signed policy requests and policy response as shown in figure 4.6 and figure 4.7. The smart card sends a policy request to the server and the server answers with a policy response.

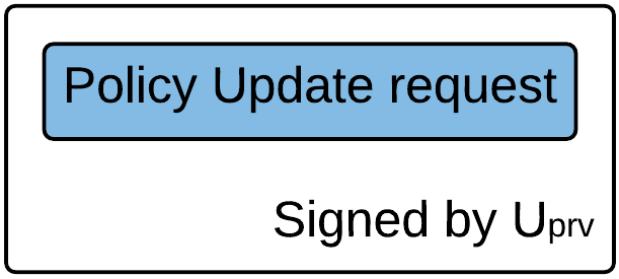

Figure 4.6: Smart card policy request.

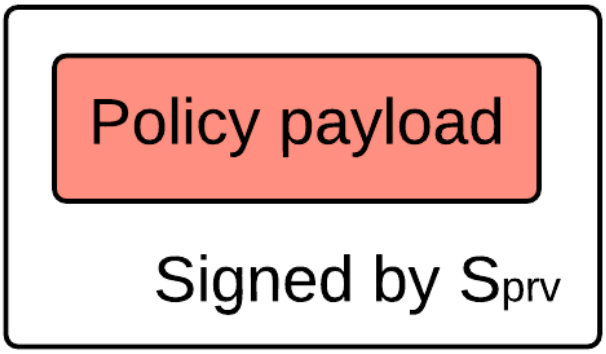

Figure 4.7: Smart card policy response.

Policy format
When designing the format of the policy package there are two things to keep in mind. It should be human-readable for easy modification and not deviate too far from a machine-readable format as the smart card will need to interpret it. Another requirement to take into consideration is that data being sent to the smart card, must be converted to hex.

We can utilize JSON to make human-readable policies and as an added bonus JSON is readable for most programming languages. Listing 4.3 shows how this can be structured.

Listing 4.3: Human-readable policies in JSON.

```
{
  "policies": [
    {
      "id": "19",
      "description": "PIN attempts",
      "enabled": "true",
      "attempts": "3"
    },
    {
      "id": "21",
      "description": "Keylength (AES)",
      "enabled": "true",
      "keyLength": "256"
    }
  ]
}
```

Recall section 2.1.2 where we described how a Command APDU must be structured. In the header we can use INS as a flag for "policy update instruction", for instance `09`. Next we have to look at what we technically want to achieve:

1. Set boolean variable to true, depending on policy ID and "enabled" flag.
2. Set other variable values.

In the body we will use the payload data field to send information on the policy updates. The incoming data will be in byte format. Mapping bytes to variables could normally be solved using `HashMap` and map entries with `<byte, Object>`; the first parameter is the incoming byte command and second parameter being the variable we want to manipulate. `HashMap` is not a part of the Java Card API and we need to do our own hardcoded manual mapping. The result is a rigid structure for policies.

The individual policies data structure needs to be carefully crafted and specialized, but we should also focus on making it as dynamic as possible. This is best described using an example. To demonstrate how it can be done, we will use listing 4.3 as example data. The resulting APDU is figure 4.8 and it clearly shows that we have to hardcode how many fields a policy

will need.

CLA	INS	P1	P2	LC	"Number of policies"
80	09	00	00	08	02

"PolicyLength	"Policy ID"	"Enabled"	"Attempts"
03	13	01	03

"PolicyLength	"Policy ID"	"Enabled"	"KeyLength"	"KeyLength"
04	21	01	01	00

LE
01

Figure 4.8: Example policy APDU with two policies

When designing the smart card code for interpreting the policy APDU we need a registry of some sort for keeping track of how much of the payload each policy uses. Listing 4.4 uses figure 4.8 as incoming APDU.

Listing 4.4: Pseudo code for interpreting policy APDU with Java Card.

```
1  public class cardApplication extends Applet implements
       ExtendedLength{
2      ...
3
4      //Policies
5      final short offset = 5;
6      short counter;
7
8      //Policy 13
9      boolean enabled13;
10     short attempts;
11
12     //Policy 21
13     boolean enabled21;
14     short keyLength;
15
16     public void process(APDU apdu) {
17         ...
18         byte[] buff = apdu.getBuffer();
19
20         switch(buff[ISO7816.OFFSET_INS]){
21             case 0x09:
22                 counter = 6;
23                 for(short i = 0; i < buff[offset]; i++){
```

```
24                      if(buff[counter+1] == 13){
25                          enabled13 = (buff[counter + 2] != 0);
26                          attempts = buff[counter + 3];
27                      }
28                      else if(buff[counter+1] == 21){
29                          enabled21 = (buff[counter + 2] != 0);
30                          keyLength =
31                              (short)((buff[counter+3]<<8)
32                               | (buff[counter+4]))
33                      }
34                      counter += (1 + buff[counter]);
35
36                  }
37                  break;
38
39
40              ...
41
42          }
43          Send(apdu);
44      }
45
46      private void send(APDU apdu) {
47          //Package outgoing buffer
48          //Send response APDU
49      }
50  }
```

4.3.6 Solution evaluation

Section 4.3.5 described how a smart card can enforce policies and how to manage policies. The solution is complex, intricate, requires a lot of overhead and is very rigid. Despite these drawbacks, using a smart card for policy enforcement could be well worth it. If an organization need a system where they are in full control and that is tamper proof, a smart card solution is viable option.

4.3.7 Potential attack vectors

Replay attack
As we are not encrypting the policy response from the server we run the risk of an attacker replaying older policy responses to the smart card. This can be mitigated by adding a version number or counter to the policy response. All the smart card needs to do with this solution is keeping track of which version it is currently on, and refuse policy responses with older version numbers.

Policy disclosure
This proposed solution does not encrypt the policy response to save resources on the server and on the smart card. A direct consequence of this is that anyone receiving the policy response is able to read what policies are currently being enforced. For example, sharing how many password attempts or minimum password length may not be something an organization wishes to do. If this is a requirement for the organization the policy response should also be encrypted with the public key of the smart card.

Chapter 5

Framework design and implementation

This chapter gives an overview of the implementation of the Java Card applet and Android framework we described in chapter 3. We include a discussion of the high level design of the classes and interfaces the framework consists of, the libraries used, what functionalities have been implemented and what is left for future work.

5.1 Java Card applet

The Java Card side of the complete framework includes functionality for performing basic data exchange with the Android application, basic cryptography and operations needed to implement the binding protocol. The goal of the smart card application was to create an autonomous and easy to extend platform for future tests. This resulted in an application split into three parts: initialization, data processing and finalization.

Initialization
As described in section 2.1.3 all Java Card applications must implement the method `Install`. This method is invoked only once when the smart card applet is installed. `Install` invokes the constructor of the smart card and this is where all variables that need initialization are initialized. For instance if the smart card application needs to generate keys or random numbers this

is where it is done as the constructor will be invoked only once. All buffers that needs to be used should also be initialized here to avoid allocating memory every time the application is activated. This method should also be used to generate and store the unique RSA key pair that will identify the smart card in all future transaction. Other cryptographic functions such as signing, initialization vectors and PIN holders should also be instantiated here.

Data processing
In the mandatory `Process` method (refer to section 2.1.3) all data processing takes place. First a built in method in the Java Card API, `selectingApplet()`, is invoked. This method checks if the incoming APDU is a SELECT APDU and acts accordingly. If the incoming APDU is not a SELECT APDU the incoming APDU is copied to a new buffer for easier data manipulation. Next, we use a switch statement switching over the second byte, INS, to determine which instruction we want to perform. After processing the data and performing the work we want to do (sign data, encrypt, etc.) we copy our response to the outgoing buffer. For this section we have defined a series of basic operations that are needed to perform our tests and creates a basis for more advanced functionality.

Finalization
At the end of the `Process` invocation we invoke the `send` method which takes the data in the outgoing buffer, package it for sending and send it as a response APDU. It is important to handle the outgoing buffer in the most efficient and cleanest way possible, in order to avoid memory errors or other faults.

The result
What we end up with is a test platform where we are only concerned with declaring variables, initializing variables and writing code for the specific test case. Listing 5.1 shows pseudocode for the Java Card application with the extendable areas highlighted. The complete listing of the code is located in appendix A.

Listing 5.1: Pseudo code for javacard test application.

```
public class cardApplication extends Applet implements
    ↪ ExtendedLength{

    //Variable declarations

    private cardApplication() {
        //Variable initialization
    }

    public void process(APDU apdu) {
        //Process incoming APDU
        if (selectingApplet()) {
            return;
        }
        buff = apdu.getBuffer();

        switch(buff[ISO7816.OFFSET_INS]){
            case 0x00:
            case 0x01:
            ...
            case 0xff:
            default:

        }
        Send(apdu);
    }

    private void send(APDU apdu) {
        //Package outgoing buffer
        //Send response APDU
    }
}
```

As seen in listing 5.1 we allow for 256 cases/uses of the smart card, but if we include the use of P1 and P2 from section 2.1.2 there are in theory $256^3 = 16777216$ possible cases. This does not include the pre-implemented methods which are explained in section 5.2. Their counterparts in the smart card application have the following byte values:

- byte SEND_U_PUB_MOD = (byte) 0x01;
- byte SEND_U_PUB_EXP = (byte) 0x02;

- byte SIGN = (byte) 0x03;
- byte BINDING = (byte) 0x05;
- byte RSACRYPTO = (byte) 0x06;
- byte AESCRYPTO = (byte) 0x09;

`RSACRYPTO` and `AESCRYPTO` uses P1 to differentiates between encrypting and decrypting. `0x01` for encryption and `0x02` for decryption. We will not go into detail on how the individual cases are built up as their functionality should be self-explanatory. Refer to Appendix A for Java Card code.

5.1.1 Extending the Java Card application

When adding more functionality to the Java Card application there are a few things to keep in mind. First of all one should follow the recipe shown in listing 5.1 to minimize clutter and to follow the principles of Java Card programming. Secondly it is important to keep in mind that Java Card does not have standard garbage collection (refer to section 2.1.3). A direct consequence is that any extension or extra functionality added to the smart card application may lead to "Out of Memory" errors.

Installation time of the smart card application may also be affected by extra functionality. Key generation on the smart card is a relatively expensive process, and one may find that it is not worth adding installation time to the whole application for one function. One approach is to split functionality in multiple applications. We will discuss this approach later in chapter 7, section 7.5.

5.2 Android framework

We used the same approach on the Android framework as on the smart card application; an autonomous and easy to extend platform for tests. This resulted in a new library, "smartcardlibrary", which sole purpose is to transmit APDUs as easily as possible along with some essential functionality.

5.2.1 Achieving framework goals

In section 3.1 we described the goals of the framework. We wanted a framework that was easy to use, preferably requiring little to no understanding of smart cards, and at the same time be extendable. To achieve this, we abstract as much as possible of the smart card aspect, and create controllers that developers can utilize.

Figure 5.1 is a package diagram of the Android framework and the easiest way of using the framework is to only focus on the `Controller` package. More specifically one will only need to understand `CommunicationController` to be able to communicate with smart cards (either via pre-implemented methods or custom APDUs, this is described in 5.2.4). To accommodate the need for more advanced functionality, or rather have the framework be extendable, developers have, if they choose, direct access to the NFC and mSD controllers in the same package.

The package diagram, figure 5.1, shows how the packages in the library are structured.

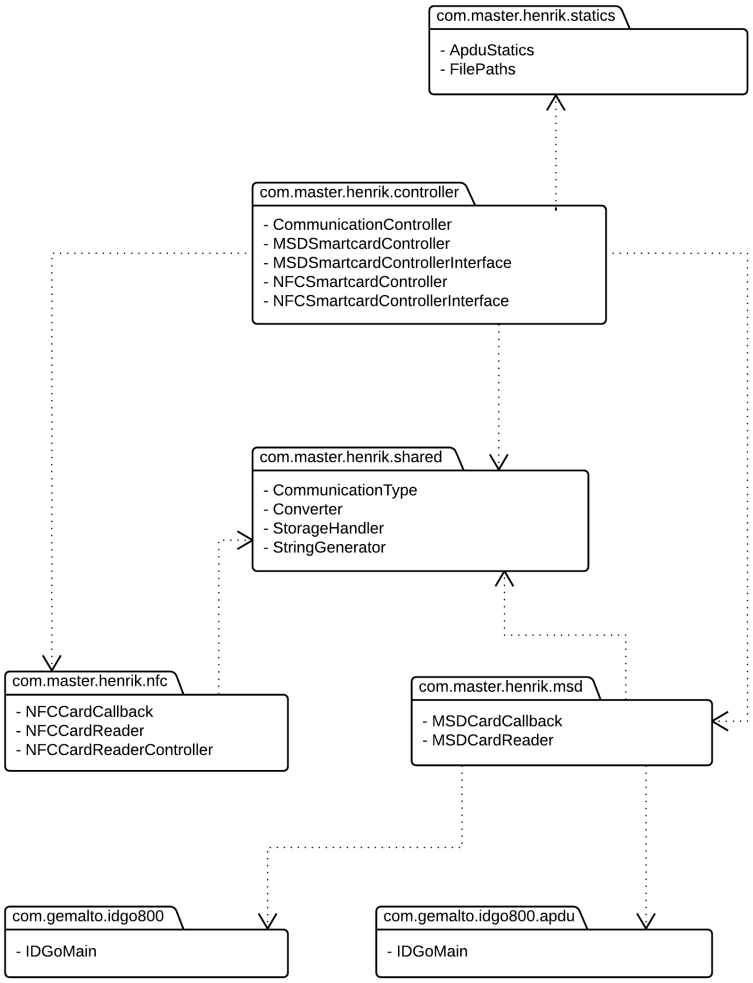

Figure 5.1: Library package diagram.

5.2.2 Responsibility areas

In light of chapter 4 it is important to understand how the smart card library fit into the bigger picture and what it is responsible of. In figure 5.2 we can see that there are 3 main "layers"; the Android operating system, the Android application and the smart card library. Even though we define these layers as separate entities, we are technically incorrect when stating so,

e.g., the smart card library is technically a part of the Android application and thus they are the same layer. From a more abstract perspective the separation of layers is more correct and can help give a better understanding of the architecture.

The first layer is the Android operating system. The Android operating system is in charge of persistent storage (if needed), key storage and key generation. We discussed key storage and key generation in section 4.2 and how it should be handled. It is also worth noting that the Android operating system is responsible for running the Android application.

The second layer is the Android application. This layer is responsible for serving the user interface to the user. This may include PIN input, displaying sensitive data, etc. If the Android application handles sensitive data it is vital that it disposes the information correctly after use.

The last layer is the smart card library. The smart card library handles the incoming and outgoing communication with the smart cards. The smart card library utilizes a temporary cache to achieve asynchronous communication with smart cards and as with the Android application, it is necessary to clean out the temporary cache if sensitive data has been passing through the library.

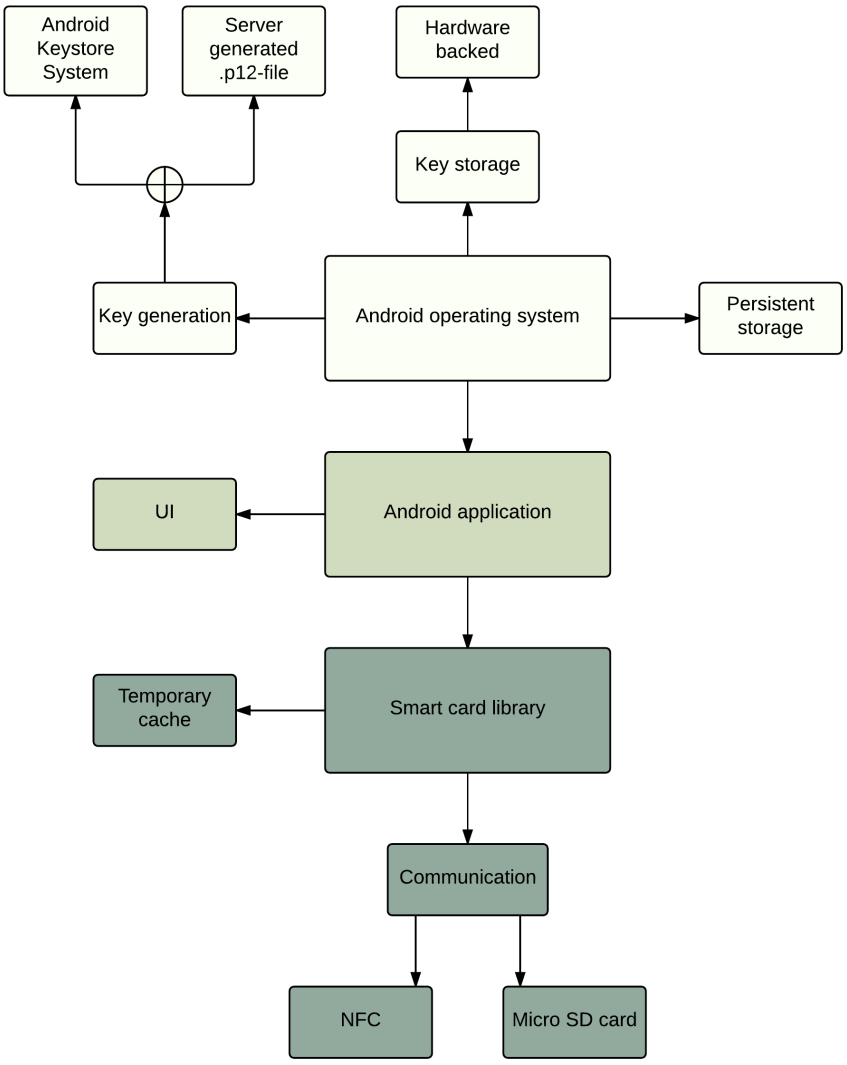

Figure 5.2: Diagram showing which responsibilities the layers in an Android application have.

5.2.3 3rd party libraries

Gemalto provides a java library, IDGo800, for communicating and utilizing built-in methods with their smart cards.

> "IDGo 800 for Mobiles is a cryptographic middleware that supports the Gemalto IDPrime cards and Secure Elements on Mobile platforms: Contact and contactless smart cards, MicroSD cards, UICC-SIM cards, embedded Secure Elements (eSE) and Trusted Execution Environment (TEE)."
>
> (Gemalto.com [23])

The part of IDGo800 SDK we are interested in is very small and enables us to send custom APDUs to micro SD smart cards.

We will be using the "android.nfc" package in order to communicate with NFC smart cards. This package is included in the standard Android SDK which in turns means that all Android devices with a NFC reader and minimum API level 9 [5] can use our library.

5.2.4 Framework functionality

The first functionality we will describe of the implementation is "extendable" or in other words, being able to send custom APDUs. Explaining how this is implemented will give a better understanding of how the framework is built up and makes it easier to understand the pre-implemented methods.

Custom APDUs

To send custom APDUs to a smart card,`CommunicationController` must be instantiated and the application must know the application identifier of the smart card application. Further the current activity must implement `NfCSmartcardControllerInterface` or `MSDSmartcardControllerInterface` (depending on smart card type) in order to be notified when the transaction is complete. Before continuing one will need to call the methods `setupNFCController` or `setupmSDController` depending on the smart card. Listing 5.2 shows an example implementation on how an activity may utilize the library for sending custom commands to a NFC smart card.

Listing 5.2: Java code example showing how to send and receive commands to a NFC smart card.

```
1
2  public class PayloadActivity extends AppCompatActivity
3      implements NFCSmartcardControllerInterface {
4      CommunicationController cc = new CommunicationController();
5      ...
6
7      private void initNFCCommunication(){
8
9
10         String AID = "0102030405060708090007";
11         String hexMessage = "95404F3FB1";
12         String INS = "06";
13         String p1 = "00";
14         String p2 = "00";
15         cc.setupNFCController(this, this);
16         cc.initNFCCommunication(AID, INS, p1, p2, hexMessage);
17     }
18
19     @Override
20     public void nfcCallback(final String completionStatus){
21         if(!completionStatus.equals("OK")){
22             return;
23         }
24         StorageHandler stHandler =
25             new StorageHandler(getApplicationContext());
26         String response =
27             stHandler.readFromFileAppDir(
28                 FilePaths.tempStorageFileName
29             );
30     }
31 }
```

In order for the library to perform an asynchronous transaction the library will temporary save the responses from the cards to a file only accessible by the running application. To retrieve the data the current activity should use the included `StorageHandler` class as used in listing 5.2. The library also provides the class, `Converter`, for converting between Strings, hex and byte arrays.

Pre-implemented methods

Recall the areas we want to cover from the beginning of the chapter. The functionality we have implemented so far are:

- Bind smart card to mobile device.
- Encrypt/decrypt data using RSA key on card.
- Encrypt/decrypt data using AES key on card.
- Get public key of the smart card.
- Sign data using the public key of the smart card.

To use these functionalities one would only need to create an Android `Activity`, invoke either `setupNFCController(...)` or `setupmSDController(...)` (depending on smart card), and utilize the desired methods. In figure 5.3 we can see how `CommunicationController` is designed to be the abstraction layer between Android activities and smart cards.

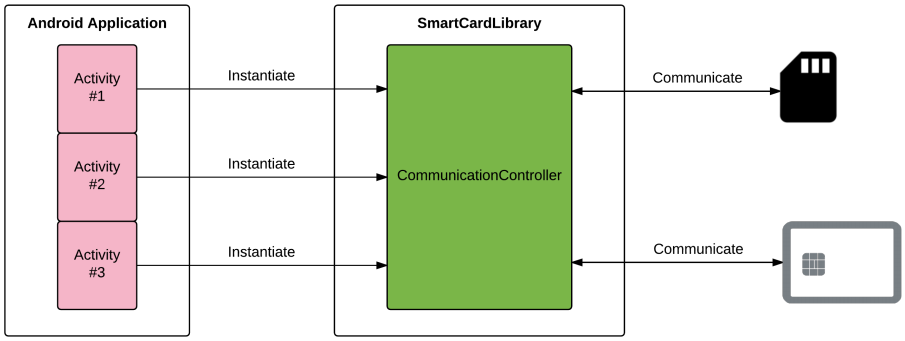

Figure 5.3: Abstraction layer between Android activities and smart cards.

The methods available are:

- public void disableNFC(...)
- public void signData(...)
- public void cryptoRSA(...)
- public void cryptoAES(...)
- public void getCardPubMod(...)
- public void getCardPubExp(...)

- public void bindingStepOne(...)
- public void bindingStepTwo(...)
- public void bindingStepThree(...)

The methods and their functionality should be self-explanatory except for the binding process. The binding process is designed in three steps. First step is to ask the smart card if it requires a PIN-code and how many attempts are left. Second step requires a PIN-code and if this is correct the smart card application will move on to step three. The last step is sending the public key of the mobile device and getting the verification package from section 4.1.4. More discussion on this matter in section 4.1.

Listing 5.3 shows how an activity can use the `CommunicationController` to sign a simple message. The `signData(...)` method takes 3 parameters: CommunicationType, AID and the hex message to be signed. In the method `nfcCallback(...)` the developer are free to do whatever they want. Typically it is a good idea to check what the completeionStatus string is before trying to fetch the response data. Read appendix B for all method signatures.

Listing 5.3: Java code example showing how to send sign a message using a NFC smart card.

```
1
2  public class SigningActivity extends AppCompatActivity
3      implements NFCSmartcardControllerInterface {
4      CommunicationController cc = new CommunicationController();
5      ...
6
7      private void initNFCCommunication(){
8
9
10         String AID = "0102030405060708090007";
11         String message = "This message must be signed.";
12         String hexMessage = Converter.StringToHex(message);
13         cc.setupNFCController(this, this);
14         cc.signData(CommunicationType.NFC, AID, hexMessage)
15     }
16
17     @Override
18     public void nfcCallback(final String completionStatus){
19         if(!completionStatus.equals("OK")){
20             return;
```

```
21          }
22          StorageHandler stHandler = new StorageHandler(
23            getApplicationContext()
24            );
25          String response = stHandler.readFromFileAppDir(
26            FilePaths.tempStorageFileName
27            );
28      }
29  }
```

Available classes

Figure 5.4 provides a simplified and technical overview of how the Android side of the library is built up. The classes seen in the simplified class diagram 5.4 shows the three controller classes, `CommunicationController`, `NFCSmartCardController` and `MSDSmartCardController`, we have implemented along with their method signatures. The enum class `CommunicationType` is used for indicating which type of communication one wishes to invoke when using the methods of `CommunicationController`.

The two controllers, `NFCSmartCardController` and `MSDSmartCardController`, can be used directly. If this is done, one cannot use the pre-implemented methods in the Android library. The corresponding methods on the smart card are still available, but with this approach one will need to construct the APDUs manually and conform to their message structure. This approach is still a viable way of using the framework, as both controllers provide a method for sending data to the smart card using the parameters: card AID, INS, P1, P2 and Payload. The method ensures that the APDU being sent to the smart cards conform to the ISO standards of APDUs, but the developer run the risk of invoking functionality that does not exist or with wrong parameters.

Depending on which type of communication type one wishes to use, one will need to implement the corresponding interface in the Android Activity as we discussed earlier. This can be seen in action in listing 5.3.

The complete class diagram for the Android library can be found in Appendix C, figure C.1. This figure contains all implemented classes, interfaces and enums. If there is a need to implement custom controllers for smart card communication it is possible to use the classes which communicates directly with the Gemalto library or the Android.nfc package. A potential

future need might be to add communication with a new type of smart card. Classes such `Converter`, `StorageHandler`, `ApduStatics` and `FilePaths` provide necessary functionality for APDU construction, message parsing and asynchronous message handling. The package diagram in figure 5.1 shows how the specific smart card controllers include these classes.

We have deliberately not made any of the classes protected, meaning that all classes can be instantiate anywhere. This allows new frameworks or new functionality to build on our work without having to edit the library. Of course, this opens up the possibility for uses which are not possible to execute. For example sending APDUs to a smart card without opening the channel. We expect developers that chooses this path to have an understanding of how smart card communication must be built up.

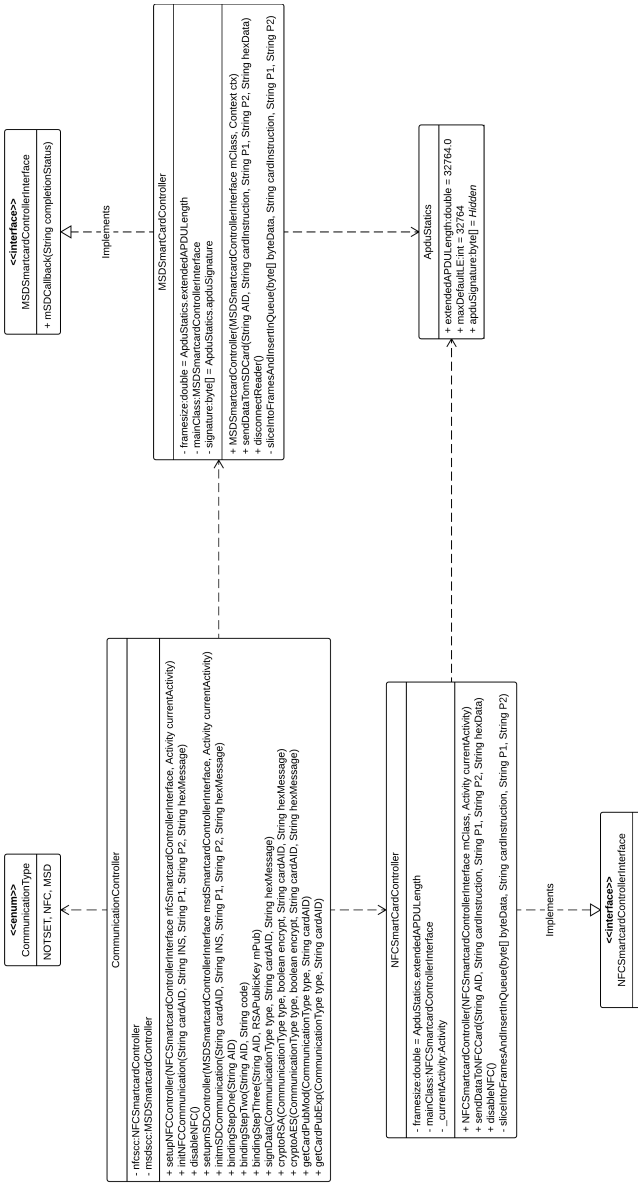

Figure 5.4: Simplified class diagram for Android Library.

Chapter 6

Testing and use case implementation

In this chapter we describe the tests we designed and ran to evaluate the cryptographic capabilities and performance of the smart cards, communication and compatibility between smart cards and Android, and possible technical problems that may present themselves when using smart cards. The implementation and tests took a lot of resources and time as it was difficult to debug problems that occurred on the smart cards and in the third party libraries we used. The remaining time was spent on implementing and testing the binding protocol we described in section 4.1.

6.1 Setup

In order to do research on how smart cards can have an impact on mobile data security and to perform an evaluation on how effective they are we need to have a proper test environment. We define "proper test environment" as an environment as close to reality as possible.

6.1.1 Equipment

Test device
The device we will be using for deploying the applications and performing tests on is considered to be a mid-range device. The device is a Sony Xperia

M2 Aqua smartphone running Android 5.1.1 with the following relevant specifications:

- Chipset: Qualcomm MSM8926-2 Snapdragon 400
- CPU: Quad-core 1.2 GHz Cortex-A7
- RAM: 1 GB

More information on the specifications of the phone can be found on GSMArena.com [50].

Java smart card
We will be using two types of smart cards. The first type is a micro SD memory card (IDCore 8030 MicroSD card) as shown in figure 2.5 produced by Gemalto. The reason for using this card for testing is that Gemalto delivers ready-to-use cards along with a framework for communicating with them. The cards we will be using have nothing pre-installed on them and we can freely deploy custom applications to the card. In order to use the provided framework we need a key provided by Gemalto which has a validity period of 120 days.

The second type of card we will be using is a contactless smart card with no pre-installed software which is also provided by Gemalto [21] along with a standard card reader. In this case we are not reliant on the framework provided by Gemalto as Android has built in support for NFC communication in the standard SDK.

Both types of card are able to run the same application and thus makes it very convenient when comparing their performance against each other.

6.1.2 Limitations and problems

We encountered problems with the library provided by Gemalto. The library refused to function properly and returned error codes that we had no basis for understanding. It was not possible to debug the library as the library had been run through an obfuscator [13, Ch. 5], meaning that we could not inspect the code to find the error. After a lot of back and forth between us and Gemalto it became evident that they had provided us with the wrong license key for the framework.

When we started testing the implementation it became evident that the micro SD smart card we had did not support extended APDU. As a result we are not able to perform tests that involve micro SD cards and extended APDU. Limitations like these took up a lot of resources since we had no way of knowing if they were actual limitations or if we were implementing our solutions the wrong way. We needed to deplete all possible solutions and alternatives before we could conclude with that they were limitations.

6.2 Tests

6.2.1 Data Transfer Speed

Description and motivation
Transfer speed is a very vital for part of the smart card interaction. If the smart card application or the transportation layer is incapable of handling large amounts of the data we will need to take that into account when examining the usability of smart cards. In order to test and eliminate as many variables as possible the smart card is programmed to receive data, copy the incoming data to the buffer and send the exact same data in return. Figure 6.1 describes this process using an NFC card as a platform for the Java Card Applet.

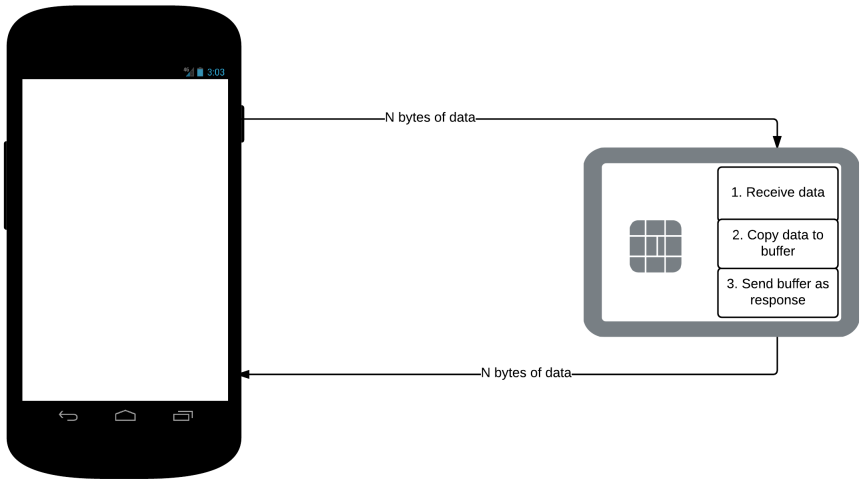

Figure 6.1: Data flow of data transfer speed test for NFC.

The design of the tests was an iterative process. Transfer speed test was one of the first tests we ran on the smart cards and we did not know beforehand how much data a smart card would be able to handle. As a result we ended up with 3 different configurations.

T1 configuration consists of the Android application sending 255 byte of data to the smart card application, receive the response and write the response to an internal file. This process is repeated until all of the data is processed.

T2 configuration consist of the Android application sending data to the smart card application using frames of size 255 byte until all data is sent. Simultaneously the responses are written to an internal file using FileOutputStream (provided by the standard Java library).

T3 configuration consist of the Android application sending data to the smart card application using frames of size 32768 byte until all data is sent. Simultaneously the responses are written to an internal file using FileOutputStream (provided by the standard Java library).

With some pre-testing it quickly became apparent that T1 configuration is vastly inferior to T2 and T3. A non-asynchronous (not using streams) Android application is not representative of the "real-world" and we decided on not pursuing further test results using this configuration when we moved over to Micro SD card testing.

NFC results

Data size (byte)	**T1**	**T2**	**T3**
10000	3,8s	4,1s	3,6s
100000	41,3s	35,0s	24,7s
1000000	602,1s	361,3s	235,1s

Table 6.1: Table of NFC transfer speed test.

The tests results(table 6.1) show a significant improvement from T1 to T3. The exception is when we are sending small amounts of data to the smart card applet which suggest that the difference is miniscule. However, when we upped the data size to 100.000 byte, T2 and T3 was 15% and 40%, respectively, faster than T1. When sending 1.000.000 bytes of data T1

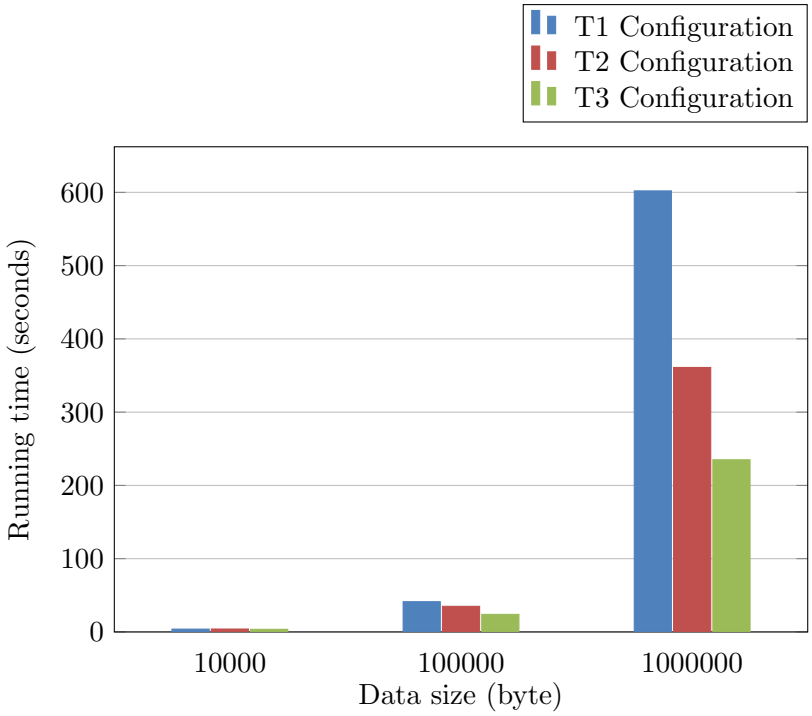

Figure 6.2: Graphical representation of table 6.1.

used 602,1s. T2 had an improvement of around 40% whereas T3 had an improvement of around 60%. The test results clearly show that T3 is the optimal configuration of the 3.

Micro SD results
We were not able to test T3 configuration as explained in section 6.1.2.

Data size (byte)	T2	T3
10000	3,18s	N/A s
100000	16,14s	N/A s
1000000	142s	N/A s

Table 6.2: Table of micro SD transfer speed test.

As we were not able to test T3 on micro SD cards we cannot compare the configurations with each other. What we can do is compare the T2 results for micro SD to the T2 results from the NFC section. The difference is not that great with small amounts of data, but when we go up to 1.000.000 bytes of data micro SD is 60% faster than the NFC card (361,3s to 142s). This points in the direction of micro SD cards achieving better results than NFC cards.

Conclusion
From table 6.1 and figure 6.2 we can learn that we are able to optimize the data transfer and processing speed between the Android application and the NFC card. It is also clear that when we are transmitting low amounts of data there is virtually no difference between the configurations; T1, T2 and T3. The differences are more prominent when the data amounts increase. Even though we achieved an improvement of approximately 60 % from T1 to T3 when sending 1 MB of data, the process is still time consuming.

If we compare the test results for T2 configuration on the NFC card and micro SD card we can clearly see an improvement on the micro SD card. The micro SD card had a 60 % better running time over the NFC card when sending 1 MB of data. Although we were not able to test configuration T3 on the micro SD card, results point in the direction of micro SD cards having better performance than NFC cards. We are not able to confirm this and thus cannot be treated as a fact.

Even though we are able to optimize and improve data transfer speeds, we are still very far from transferring and processing large amounts of data quickly. We have to take this into account when evaluating areas of use for the smart card. Transfer and processing speed rules out many areas concerning large amounts of data, such as full data encryption.

6.2.2 Symmetric-key cryptography

We want to discover the encryption abilities on the smart card and decide if it is feasible to let the smart card handle the encryption of confidential data. From the smart card documentation we know that we are able to use AES to encrypt data, but we do not know how long it will take to encrypt the data. We will need to perform a run time tests with different amounts of data in order to determine the performance of the smart card. This test will help us establish guidelines for how much data it is reasonable to encrypt directly on the smart card without affecting the user experience of an Android application.

Test setup

The framework we are using is designed around extended APDU and the test platform we have designed in Java Card utilizes extended APDU. As a result we are not able to use the micro SD cards (as described in section 6.1.2) and we will be using the NFC smart cards.

The encryption algorithm we will use is AES cipher algorithm with block chaining (CBC). The version of Java Card that we are using along with our smart cards limits us to using 128 bits keys and no padding. More specifically the only working AES algorithm from javacard is `ALG_AES_BLOCK_128_CBC_NOPAD`, even though the Java Card documentation for `Cipher` supports more algorithms [31]. Others have encountered the same discrepancy [44] suggesting that only three of the twelve supported algorithms works, but there exists no official information on the issue.

`ALG_AES_BLOCK_128_CBC_NOPAD` is as the name suggest an algorithm with no padding. The block size the AES algorithm expects is 16 byte and as a result we will need to pad the data ourselves on the mobile device.

Results

The test results show that the encryption process of AES on smart card is very close to being linear. Even though the relationship between time and bytes processed is close to linear the encryption process is slow. The processing power of the AES setup translates to roughly 500 bytes of data per second.

Data size (byte)	Elapsed time
16	0,15s
10000	20,18s
32000	61,81s
100000	183,78s
1000000	1835,49s

Table 6.3: Table of AES encryption speed test.

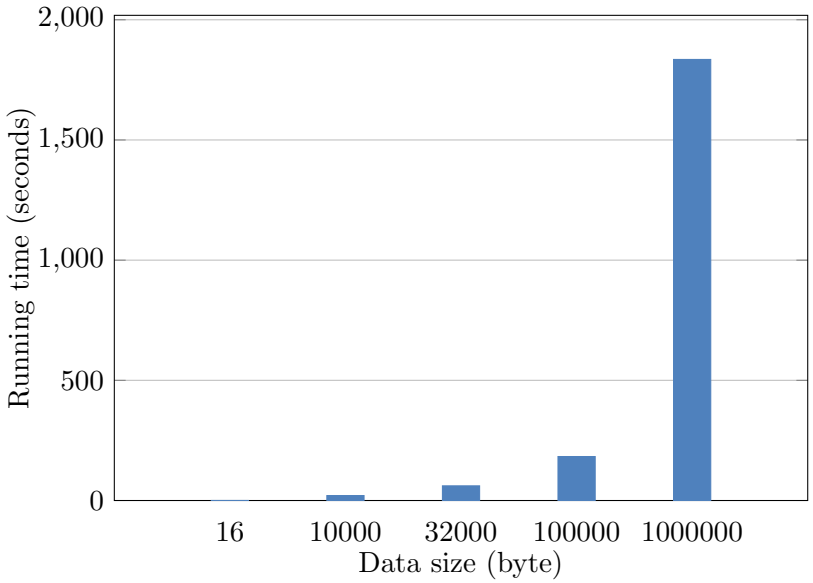

Figure 6.3: Graphical representation of table 6.3.

Conclusion

From the test results we can learn that encrypting data on the NFC smart card takes a lot of time. Encrypting 1MB of data uses approximately 30 minutes, which from a real world perspective is an unacceptable amount of time. Using the NFC smart card for full encryption of user data is in other words not achievable and we will need to look for other options for encryption.

Encrypting 16 bytes of data uses 0,15 seconds. A use for the encryption capabilities on the smart card may be to encrypt small amounts of data such as GPS coordinates. GPS coordinates can be represented by only 16 bytes (depending on accuracy). One could also imagine that we will only

need to encrypt parts of a document, and as long as we keep the data size small we can let the smart card do the encryption.

6.2.3 Public-key cryptography

Similarly to the symmetric-key encryption test we want to investigate the feasibility of public-key encryption of sensitive data. The smart cards and Java Card version we have supports RSA cryptography, but we will need to run performance tests to confirm that it functions properly and finish within a timely manner.

Another part of public-key cryptography we want to investigate is digital signing. Being able to sign and verify data is an essential part of many security mechanism and a vital part of what we wish to achieve by using smart cards. If tests show that digital signing is unfeasible on smart cards we will need to re-evaluate their use in a mobile device ecosystem.

Test setup
Just as with symmetric-key cryptography, we will be using the Android framework we have created, which uses extended APDUs, and as a result we cannot perform these tests on our micro SD cards.

For digital signing we will be using keys of length 512-bit and 2014-bit. Early tests shows that key sizes greater than 1024-bit crashes our NFC cards (refer to section 6.2.5). This is problematic as we wish to use longer keys, minimum 2048-bit and preferably up to 4096-bit, as this is what is recommended by the industry [55]. Even though 1024-bit keys does not represent our goals, it could point us in the right direction when it comes to feasibility. We use the `Signature` class for signing with the algorithm `ALG_RSA_SHA_PKCS1` meaning that we do not need to pad our data.

For RSA encryption we were able to test with both 512-bit keys and 2048-bit keys. We will be using the `Cipher` class to perform the encryption, but one important thing to note is that the plaintext sent in to the cipher function is limited by the key size. 512-bit and 2048-bit keys are only able to process 52 byte and 244 byte of data respectively. This is because of the architecture of RSA and iff we wish to encrypt more data we will need to split the data in the Android application.

Results

Data size (byte)	Elapsed time (512)	Elapsed time (1024)
10000	0,74s	1,72s
32000	5,32s	5,98
100000	15,49s	16,26
1000000	141,79s	146,23

Table 6.4: Table of digital signing (RSA) speed test.

The test results from digital signing in table 6.4 shows that there are large differences between small amounts of data and large amounts of data, but the two key sizes perform very similar. The byte per second value for 10.000 byte is close to 13.000, whereas the byte per second value for 32.000 byte is half of that with around 6000 byte per second. 100.000 byte has as expected around the same byte per second value as 32.000. This is caused by the fact that 100.000 byte is the same as 3 32.000 byte operations due to the limitations of extended APDUs.

Data size (byte)	Key size	Elapsed time
52	512-bit	0,62s
144	2048-bit	1,33s

Table 6.5: Table of RSA encryption speed test.

Encrypting with RSA is an expensive operation and is clearly shown by table 6.5. We are at maximum able to encrypt 52 bytes of data with a key size of 512-bit and 144 byte of data with 2048-bit keys at the time. Compared to AES encryption the byte per second value is terrible. 512-bit keys have byte per second value of around 70 byte and 2048-bit keys have a byte per second value of around 108 byte. AES encryption from table 6.3, is able to reach a byte per second value of around 500 byte.

Conclusion

We have learned two things from testing public-key cryptography on smart cards. Digital signing on the smart card seem to be viable, at least with a key size of 1024-bit. We are able to sign 10.000 byte of data in right under 2 seconds and from a “real world” perspective 10.000 byte is able to represent a lot of information. However, we are not able to use digital signing for

signing large files such as images or videos. This problem can be solved by signing hashes of the files instead of signing the files directly.

Encrypting data using RSA is not viable according to our test results. The intention of RSA was never to be able to encrypt large amounts of data so the test results are of no surprise to us. If we need to protect large amounts of data, such as if we chose to encrypt the policy response data from section 4.3.5, we can use RSA encryption to encrypt a AES key which in turn is used to encrypt the actual data. This is referred to as hybrid encryption and is a common way of solving the issue where we have more data than we are able to encrypt with RSA.

6.2.4 Binding card and mobile device

In section 4.1 we describe and motivate a solution where the we want to bind a mobile device to a smart card. We find it important to do an empirical test of the solution due to the fact that we have time constraints regarding running time. The parts of the binding process we want to test is:

- Mobile device is able to transfer public key to smart card.
- Smart card can store the public key from the mobile device.
- Smart card can generate the *verifaction package* from section 4.1.4.

Implementation
In light of the parts of the binding process we want to test the outline of the implementation are:

1. Mobile devices asks smart card if its authenticated with PIN.
2. Smart card responds with yes/no and amount of PIN tries remaining.
3. Mobile device prompts user for PIN and sends it to the smart card.
4. Smart card verifies PIN or the process skips back to step 2.
5. Mobile device sends it public key to the smart card.
6. Smart card generates the verification package (figure 4.3).
7. Smart card sends the verification package to the mobile device.

Due to the nature of smart cards we will need to hard code this protocol into the android application and the smart card application. We will use the P1 byte in the APDU to define which step of the process we are in, respectively:

- `0x01` for step 1 and 2.
- `0x02` for step 3 and 4.
- `0x03` for step 5,6 and 7.

To manage PIN verification and management we will use `OwnerPIN` class from the Java Card framework [12]. After the PIN is set on the card during installation we can utilize the methods: `isValidated()` for checking if the right PIN is already provided, `check(byte[] pin, short offset, byte length)` for checking if the provided PIN is correct and `getTriesRemaining()` for getting remaining tries.

All public keys are represented as byte arrays when they are transmitted between the mobile device and the smart card. The format for the byte array is |ModulusLength|Modulus|ExponentLength|Exponent| to allow for extension of key length and dynamic importing. After the APDU is received the public key is stored in a `RSAPublicKey` object for storage.

When we construct the verification package on the smart card we have to convert all the keys on the smart card over to the same format as the mobile device's public key. After we have transformed all relevant keys (see figure 4.3) to byte arrays we can finally put them together, sign the package and encrypt it with the public key of the server.

Configurations

Configuration 1 uses 512-bit keys as the public key between the mobile device and smart card. Although we wish to use keys that are greater than 2048-bit we feel that this is a good starting point.

- NFC card
- 512-bit mobile device public key
- 512-bit smart card RSA key pair
- 2048-bit server public key
- 128-bit AES key

Configuration 2 uses 2048-bit keys as the public key between the mobile device and smart card to simulate a real-world example.

- NFC card
- 2048-bit mobile device public key
- 2048-bit smart card RSA key pair
- 2048-bit server public key
- 128-bit AES key

Installation test
We want to find out if initializing the keys we need for generating the verification package affects the installation of the smart cards. We will deploy and install the smart card application using GlobalPlatfromPro and time how long it takes. Our test limit is set to 100 tests. Table 6.6 shows that the average and maximum times increases significantly from configuration 1 to configuration 2.

Configuration	**Average**	**Maximum**	**Minimum**
1	24,28s	59s	9s
2	39,42	95s	13s

Table 6.6: Time required to install the application on the smart card.

Run test
In this test we will generate the verification package. We will skip the parts of the process involving user input (PIN code). Table 6.7 shows that configuration used 1,2 seconds to generate a verification package.

Configuration	**Elapsed time**
1	1,22s
2	N/A

Table 6.7: Time required to generate the verification package on the smart card application.

6.2.5 Limitations

After implementing the solution some limitations and problems became apparent. It is important to consider these when determining the effectiveness of the solution.

Signing not working with 2048-bit key size
We encountered a bug when switching to "configuration 2". The smart card application started actually crashing, as not in responding with error codes, but actually crashing and losing power. We managed to narrow it down to the exact line of code the application crashes on (see listing 6.1).

At first we suspected that the packet was too big or some mismatch in the parameters. The function still crashed with a smaller packet and all parameters are of the correct length/type/value. It is also worth noting that the 2048-bit key are properly initialized and working. Identifying the problem is hard considering the only clues we have are:

- Signing crashes on different input data with 2048-bit key.
- No error codes - Hard crash.

After searching the Internet for others with the same problem it became apparent that others have had troubles with signing. Some report that the running time of their smart cards increase drastically when using to 2048-bit keys [60]. We are not able to find any official sources on these issues, but there seems to be an underlying problem or bug in Java Card or in the hardware.

Listing 6.1: Java Card failed signing.

```
1  public class cardApplication extends Applet implements
       ExtendedLength{
2
3      private RSAPublicKey k;
4      private Signature sig;
5
6      ...
7
8      private cardApplication() {
9          keys = new KeyPair(
10             KeyPair.ALG_RSA,
11             KeyBuilder.LENGTH_RSA_2048);
12         k = (RSAPublicKey) keys.getPublic();
```

```
13          sig = Signature.getInstance(
14              Signature.ALG_RSA_SHA_PKCS1,
15               false);
16
17          ...
18      }
19
20      public void process(APDU apdu) {
21          ...
22
23          signatureSize = sig.sign(
24              packet,
25              (short) 0,
26              (short) packetSize,
27              h0Unencrypted,
28              (short) 0);
29
30          ...
31      }
32
33      ...
34  }
```

Out of memory

We are dealing with many different byte arrays in our solution, and due to the design of smart cards we will need to allocate memory for these byte arrays when the smart card is initialized. The size of these byte arrays are dependent on the key sizes we use. Allocating 5 byte arrays of length 500 (which is plenty for 2048-bit keys) is independently fine, but if the smart card application allocates a lot of resource alongside this solution we need to be careful of running out of memory.

It is important to include hidden memory sinks such as the `RSAPublicKeys` and the encryption done by the `Cipher` class. We can just as easily run out of memory by changing our key lengths as when allocating byte arrays.

Code rigidity

Because of the design of smart cards our implementation is heavily hard-coded, and as a results is very rigid to change. For instance if the speci-

fication of the verification package change it will require excessive work to reflect the changes. One will have to decide make a decision whether or not this will happen often enough to counter the potential benefits of using smart cards.

6.2.6 Conclusion

We have shown that the smart card area of responsibility in the binding process is possible to perform. Test results show that the installation process can be rather long (ranging from 9 seconds to 95 seconds), but that the verification package generation is effective (~1 second).

The installation process is a one time process and the keys we generate during it are needed for other cryptographic operations on the card. We find it safe to assume that this process is a cost we can afford. The verification package generation process has such a low processing time that we can also assume that we can afford it. Our conclusion is that the binding process is feasible and that the reward, that smart card and mobile device is locked to each other, greatly outscales the costs involved.

Chapter 7

Conclusion

The work that has been done in this thesis can be divided into three parts: preparation, research and a technical part. The preparation part involved researching and studying various publications, technical documentation and internet articles to get an understanding of the technology and the problems at hand.

In the research part we identified key questions and challenges for the topic and proposed different possible solutions. The proposed solutions was analyzed with argumentation from the preparation part in combination with existing solutions.

The technical part involved developing a usable framework for mobile devices and smart cards, as well as testing the proposed solutions from the previous part. The testing involved implementing the proposed solutions and evaluating their feasibility in the "real world". Setting up a working test environment turned out to be more time consuming than initially estimated and as a result we were not able to fully test all of the proposed solutions.

7.1 Research questions

After working through all parts of this thesis we are able to answer the research questions we presented in chapter 1.

- *"What are the limitations of smart cards in the context of hardware?"*

The hardware limitations of smart cards are very dependent on which smart card you are using. Generally smart cards are limited when it comes to computing power and this has a huge effect on how resource intensive operations you are able to perform on the smart card. Secure cryptography are very resource intensive and our test results show that encrypting large amounts of data is unfeasible, both for public-key cryptography (RSA) and symmetric-key cryptography (AES), at least on NFC smart cards. We performed tests regarding transfer speed and they show that the throughput of input/output are limited and that we often run the risk of running out of memory with large amounts of data.

- *"What are the limitations of smart cards applications?"*
  We opted for using Java Card as our programming language. Our version of Java Card does not support advanced datatypes. This combined with the fact that all data being sent to/from the smart card is byte values creates a rigid environment with hardcoded values. Java Card does not support standard garbage collection and thus applications needs to be extra careful when allocating memory.

- *"What types of security threats are we able to mitigate by using a smart card with an "off-the-shelf" mobile device?"*
  Even though smart cards have some areas with limitations we are able to identify use cases where smart cards can be applicable. In use cases involving cryptography a smart card can store the keys securely as well as encrypt/decrypt small amounts of data. Due to the fact that smart cards are tamper proof, meaning that you are not able to extract data (keys), we are confident that smart cards can alleviate threats such as stolen mobile devices and insecure communication channels.

We believe that smart cards can add an extra layer of security for more advanced use cases. In this thesis we described and analyzed a solution where we used smart cards as a basis for policy enforcement. Our understanding is that this type of solution in conjunction with traditional policy enforcement systems will enhance the security.

7.2 Related work

The most closely related paper to our work is "Plug-n-Trust: Practical Trusted Sensing for mHealth" [29], which discusses the possibility to use smart cards to ensure confidentiality of data from medical sensors on a patient. There are a lot of similarities in their problem statements and findings, such as the need for cryptography, attack vectors and establishing trust in a limited environment. Their main area of application is to prepare and send data to a backend service (off card/device) securely. The result is that the paper's goal differs from ours when it comes down to data flow and system architecture which brings new problems to the table that we need to solve. For instance, the paper assumes that the operating system is able to communicate with smart cards natively. This can be done as the required libraries are integrated in a custom kernel, which is outside our scope as we want to look at unmodified off-the-shelf mobile devices.

However, despite other works around smart cards and NFC security, there does not seem to be any publicly accessible research on the challenges involved when securing modern smartphones with smart cards as secure elements. This is strange when there is a rising interest in using smart card to enable the development of secure commercial mobile devices [42, 45]. This may be because of smart card technology is relatively old and thus the interest faded over the years, or that most of the technology is proprietary and as a result difficult to access. Other relevant work is mostly about the technical details about Android support for secure elements [43, 15] and proprietary commercial products for strong authentication [23].

7.3 Experience

During this thesis we have gained insight and experience when working with smart cards. We believe sharing these experiences will increase the efficiency for others working on the field in the future.

Getting started with smart card programming
Getting started with smart card programming can be difficult. After a few years with object-oriented programming most programmers will start to get comfortable using "quality of life" classes such as `ArrayList` and `Enum`. That world gets turned up-side down when moving to Java Card. One is thrown

back to an older Java version and as a programmer you will need to rethink how you solve problems and structure solutions. Most notable is the fact that all incoming data is in the form of a `byte` array that must be mapped to the correct datatypes. Missing functionality such as standard Java garbage collection and standard data types (`int`, `double`, etc.), makes Java Card programming cumbersome and requires time to adapt to.

Debugging smart card applications
Debugging smart card applications differs from standard debugging. Normally when debugging you are able to insert breakpoint, inspect variables and monitor resource usage. The nature of smart cards is to be a secure and closed environment and thus it is hard to monitor how an application behaves. The debugging method we have available is: deploy the application, send data to it and see what the response is. If the response does not match expected output, the best way to debug is creating manual breakpoints. This can be done by adding a line of code returning the current value of variables in order to figure out where the error might be. Sometimes the smart card encounter runtime exceptions and sends a 2 byte response that is mapped to an error message [25].

This type of debugging environment is exhausting and requires a lot of resources. Often we spent time trying to pinpoint an error only to later found out that the error code we got had nothing to with the actual problem. This was especially notable with errors regarding memory usage. One of the best advices concerning smart card debugging is: "Test often with a big array of test data.".

Java Card documentation
The documentation available is very technical. This is not by any means a bad thing, but it do require developers to understand smart cards fully before using the documentation. When comparing Android and Java Card documentation, it is very apparent that Google has put a lot of effort into having an educational approach to the concepts before diving into the technical aspects. In the Java Card documentation there are very few examples of usage, and we spent a lot of time trying to figure out how to properly use classes and methods.

The gap between software and hardware is very apparent in the Java Card documentation. We often encountered functionality that was supposed to

work, but did not work on our smart cards. The result of this was that when we encountered bugs, we did not know if it was a programming mistake or simply not supported by our smart cards. The best example of this was when we tried to use the `Cipher` class with algorithms that proved to not work on our smart cards (section 6.2.2).

Deploying smart card applications
Deploying smart card applications to a smart card is a time consuming task. This became very apparent when working with micro SD smart cards. Deploying a new version to micro SD required us to: remove mSD from mobile device, insert mSD into the computer card reader, run install script, wait on install script to finish, insert mSD into mobile device. Following this procedure once in a while is not a big inconvenience, but in context of debugging it become very tedious to spend 1,5 minutes switching around the mSD card and waiting for the install script.

Literature
A lot of the existing literature on smart cards focuses on the two areas banking and identification. Even though we share some use cases and challenges with these areas we cannot directly apply their solutions to our research. There are two reasons for this. The first reason is that smart cards in banking and identification have a very narrow objective with what the smart card's responsibility is, whereas our smart cards have many different responsibilities (authenticate, authorize, encrypt, key management, etc.). The second reason is that in both banking and identification the smart cards can rely on a third party (server) to verify every interaction, e.g., when paying with debit card (smart card) a server verifies the money transaction. After the initial binding of smart card and mobile device our solution's goal is that the smart card is independent.

The lack of literature on how other companies have solved similar problems to the ones we encountered, may be that many consider secrecy of their "setup" is a good way of keeping their system secure. For instance the problem "How to bind a smart card to a mobile device". Either companies working with smart cards and mobile devices do not perform a binding or they do not wish to disclose how they do it. We suspect the latter, but this is pure speculation.

The consequence of this is that we have to approach the smart card field by

looking at hard facts. ISO standards, RFC standards and documentation directly from the vendors are our main resources. Our experience shows that gathering information directly from the vendor yields a better result than looking at third-party literature as the vendor can provide updated information. See the bibliography for what we consider the best sources for information on the research topic.

7.4 Remarkable results

Apart from the experience gained from working with smart cards and mobile device there is one limitation that we want to classify as extra interesting. When looking at the Java Card documentation, the smart cards are supposed to support the cryptographic functions RSA (2048-bit) and AES-256. This is further confirmed by the documentation by Gemalto (ref. section 3.2.1). Our test results show that we are only able to use up to 1024-bit RSA keys for signing and AES-128 for symmetric-key cryptography. The claimed supported algorithms and actual working algorithms do not correspond, which is worrying.

The consequence of this discrepancy is that organizations may find themselves unable to use smart cards, as the available cryptography does not meet the industry standards. This realization could potentially happen late in the development process when substantial amounts of resources have already been used.

7.5 Future work

The research we have presented in this thesis is a good starting point for developing custom security applications on the Android platform in conjunction with smart cards. The test cases we have looked into point to that micro SD cards have better performance than NFC cards, but our micro SD cards did not support extended APDUs and thus we cannot confirm that micro SD are better than NFC cards. More work on micro SD cards must be performed in order to confirm these suspicions.

We encountered numerous bugs and limitations when working with smart cards which we did not initially predict, and as a result the Android library and the smart card application is not as polished and refined as we had hoped

it would be. This includes adding more pre-implemented functionality, refactoring code to be more readable and optimize code to achieve better performance. Additionally we believe it would be beneficial to look at the possibility of not being dependent on the Gemalto framework (manufacturer specific) for micro SD card communication.

Our evaluations of the proposed solutions are based on protocol analysis and proof of concept. It would be of great interest to perform penetration tests on the outlined solutions to confirm that: *a)* We are able to implement all parts of the solution. *b)* We can show that the solution is methodically tested against known attacks in today's society.

Bibliography

[1] *2015 Cheetah Mobile Security Report*. Last visited: 15.02.2016. 2016. URL: http://www.cmcm.com/article/share/2016-01-13/919.html.

[2] *About Google*. Last visited: 13.01.2016. 2016. URL: http://www.google.com/about/.

[3] *An Introduction to Java Card Technology - Part 1*. Last visited: 19.11.2015. 2003. URL: http://www.oracle.com/technetwork/java/javacard/javacard1-139251.html.

[4] Ross Anderson. *Security Engineering - A guide to building dependable distributed systems, 2nd edition*. Wiley, 2008. ISBN: 978-0-470-06852-6.

[5] *Android NFC, Requesting NFC Access in the Android Manifest*. Last visited: 13.01.2016. 2015. URL: http://developer.android.com/guide/topics/connectivity/nfc/nfc.html#manifest.

[6] *Android Platform Versions - developer.android.com*. Last visited: 08.05.2016. 2016. URL: http://developer.android.com/about/dashboards/index.html#Platform.

[7] *Android Studio Overview*. Last visited: 21.01.2016. 2015. URL: http://developer.android.com/tools/studio/index.html.

[8] *Android support for Microsoft Exchange in pure Google devices*. Last visited: 10.02.2016. 2013. URL: https://static.googleusercontent.com/media/www.google.com/no//help/hc/images/android/MicrosoftExchangePoliciesinAndroid.pdf.

[9] *Blackberry Priv*. Last visited: 16.05.2016. 2015. URL: http://global.blackberry.com/en/smartphones/priv-by-blackberry/overview.html.

[10] Andrew Calafato. *An analysis of the vulnerabilities introduced with Java Card 3 Connected Edition*. Wiley, 2010. ISBN: 0-470-74367-0.

[11] Jan Pelzl Christof Paar. *Understanding Cryptography*. Springer, 2010. ISBN: 978-3-642-04100-6.

[12] *Class OwnerPIN - Javacard documentation*. Last visited: 28.03.2016. 2005. URL: http://www.win.tue.nl/pinpasjc/docs/apis/jc222/javacard/framework/OwnerPIN.html.

[13] Bruce Dang. *Practical reverse engineering x86, x64, ARM, Windows Kernel, reversing tools, and obfuscation*. John Wiley and Sons, 2014. ISBN: 9781118787311.

[14] *EclipseJCDE, Sourceforge.net*. Last visited: 18.01.2016. 2008. URL: http://eclipse-jcde.sourceforge.net/.

[15] Nikolay Elenkov. *Android Security Internals*. No Starch Press, 2014. ISBN: 978-1-59327-581-5.

[16] *Exchange ActiveSync - Overview*. Last visited: 10.02.2016. 2015. URL: https://technet.microsoft.com/en-us/library/aa998357(v=exchg.150).aspx#overview.

[17] *FBI paid under 1 million dollars to unlock San Bernardino iPhone: sources*. Last visited: 18.05.2016. 2016. URL: http://www.reuters.com/article/us-apple-encryption-idUSKCN0XQ032.

[18] *Giesecke and Devrient GmbH*. Last visited: 29.04.2016. URL: https://www.gi-de.com/en/index.jsp.

[19] *GlobalPlatformPro, Github.com*. Last visited: 18.01.2016. 2008. URL: https://github.com/martinpaljak/GlobalPlatformPro.

[20] Michael Van Horenbeeck and Peter De Tender. *Microsoft Exchange 2013 Cookbook*. Packt Publishing, 2013. ISBN: 978-1-78217-062-4.

[21] *IDCore 3010 – Rev B*. Last visited: 20.05.2016. 2015. URL: http://www.gemalto.com/products/top_javacard/download/IDCore3010_RevB_Product_Datasheet_July14.pdf.

[22] *IDCore 8030 MicroSD card, Datasheet*. Last visited: 20.05.2016. 2015. URL: http://www.gemalto.com/products/top_javacard/download/IDCore8030_Datasheet.pdf.

[23] *IDGo 800 Middleware and SDK for Mobile Devices*. Last visited: 21.01.2016. 2016. URL: http://www.gemalto.com/products/idgo_800/index.html.

[24] *IntelliJ IDEA*. Last visited: 21.01.2016. 2016. URL: https://www.jetbrains.com/idea/.

[25] *Interface ISO7816 - JavaCard constants*. Last visited: 28.04.2016. 2005. URL: http://www.win.tue.nl/pinpasjc/docs/apis/jc222/javacard/framework/ISO7816.html.

[26] *ISO 7816 Part 4: Interindustry Commands for Interchange*. Last visited: 19.01.2016. 2015. URL: http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816-4.aspx.

[27] *ISO/IEC 7810:2003, Identification cards – Physical characteristics*. http://www.iso.org/iso/catalogue_detail?csnumber=31432. Last visited: 18.11.2015. 2013.

[28] *ISO/IEC 7816:1-2011, Identification cards – Integrated circuit cards – Part 1: Cards with contacts – Physical characteristics*. http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=54089. Last visited: 18.11.2015. 2011.

[29] Ron Peterson Jacob Sorber MinhoShin and Davic Kotz. *Plug-n-Trust: Practical Trusted Sensing for mHealth*. Pages 309-322. ACM, 2012. ISBN: 978-1-4503-1301-8.

[30] *Java Card Platform, Classic Edition 3.0.5*. Last visited: 19.11.2015. 2015. URL: https://docs.oracle.com/javacard/3.0.5/index.html.

[31] *Javacard documentation - Cipher class*. Last visited: 09.03.2016. 2005. URL: http://www.win.tue.nl/pinpasjc/docs/apis/jc222/javacardx/crypto/Cipher.html.

[32] Rolf Oppliger. *SSL and TLS : Theory and Practice*. Artech House, 2009. ISBN: 1-59693-447-6.

[33] *Oracle and Sun Microsystems*. Last visited: 05.03.2016. 2016. URL: https://www.oracle.com/sun/index.html.

[34] *PKCS #12: Personal Information Exchange Syntax v1.1*. Last visited: 25.04.2016. 2014. URL: https://tools.ietf.org/html/rfc7292.

[35] *Projects/OWASP Mobile Security Project - Top Ten Mobile Risks*. Last visited: 12.05.2016. 2014. URL: https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks.

[36] *Protect against harmful apps, Google Play*. Last visited: 08.02.2016. 2016. URL: https://support.google.com/accounts/answer/2812853?hl=en.

[37] *pyApduTool User Guide*. Last visited: 18.01.2016. 2015. URL: http://javacardos.com/javacardforum/viewtopic.php?t=38.

[38] Wolfgang Rankl. *Smart Card Handbook 4th Edition*. Royal Holloway, University of London, 2013. ISBN: 978-0-470-74367-6.

[39] *RFC4634, US Secure Hash Algorithms (SHA and HMAC-SHA)*. Last visited: 15.01.2016. 2006. URL: https://tools.ietf.org/html/rfc4634.

[40] *SafetyNet: Google's tamper detection.* Last visited: 08.02.2016. 2015. URL: https://koz.io/inside-safetynet/.

[41] *Schlumberger Limited homepage*. Last visited: 05.03.2016. 2016. URL: http://www.slb.com/.

[42] *Sectra Tiger System*. Last visited: 29.05.2016. URL: http://communications.sectra.com/security-solutions/tigers-r.

[43] *Secure Element Evaluation Kit for the Android platform*. Last visited: 29.04.2016. URL: http://seek-for-android.github.io/.

[44] *Securing Java Card applications, Part 2. Limits of Java Card cryptography*. Last visited: 09.03.2016. 2005. URL: http://www.ibm.com/developerworks/wireless/library/wi-satsa2/tmp0002.html#IDAN1V3C.

[45] *Secursmart - SECUSUITE FOR BLACKBERRY 10*. Last visited: 29.05.2016. URL: https://www.secusmart.com/en/for-public-authorities/secusuite-for-blackberryr-10/.

[46] JAN KREMER CONSULTING SERVICES. *MULTOS and JAVACARD*. Last visited: 18.05.2016. URL: http://jkremer.com/White%20Papers/MULTOS%20and%20JAVACARD%20White%20Paper%20JKCS.pdf.

[47] *Smart Card Operating System*. Last visited: 18.05.2016. 2015. URL: http://www.cardwerk.com/smartcards/smartcard_operatingsystems.aspx.

[48] *Smart Card Technology, The micromodule. CardWerk.* Last visited: 04.03.2016. 2015. URL: http://www.cardwerk.com/smartcards/smartcard_technology.aspx.

[49] *Smartphone OS Market Share, 2015 Q2*. Last visited: 13.01.2016. 2015. URL: http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[50] *Sony Xperia M2 Aqua, GSMARENA*. Last visited: 18.01.2016. 2014. URL: http://www.gsmarena.com/sony_xperia_m2_aqua-6582.php.

[51] *The DROWN Attack*. Last visited: 22.04.2016. 2016. URL: https://drownattack.com/.

[52] *The RSA factoring challenge*. Last visited: 22.04.2016. Unknown. URL: http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge-faq.htm#WhatIs.

[53] Joeri de Ruiter Tim Cooijmans and Erik Poll. *Analysis of Secure Key Storage Solutions on Android. In Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '14)*. Pages 11-20. ACM, 2014. ISBN: 978-1-4503-3155-5.

[54] *Transmission Control Protocol, rfc793*. Last visited: 03.02.2016. 1981. URL: http://tools.ietf.org/html/rfc793.

[55] *TWIRL AND RSA KEY SIZE*. Last visited: 22.05.2016. 2003. URL: http://www.emc.com/emc-plus/rsa-labs/historical/twirl-and-rsa-key-size.htm.

[56] Amit Vasudevan. *Trustworthy Execution on Mobile Devices*. Springer, 2013. ISBN: 9781461481898.

[57] *Writing a Java Card Applet - About Java Card Technology*. Last visited: 18.11.2015. 2001. URL: http://www.oracle.com/technetwork/java/embedded/javacard/documentation/intro-139322.html#whatjavac.

[58] *Writing a Java Card Applet - Installing the Applet*. Last visited: 19.11.2015. 2001. URL: http://www.oracle.com/technetwork/java/embedded/javacard/documentation/intro-139322.html#cinst.

[59] *Writing a Java Card Applet - Processing Requests*. Last visited: 19.11.2015. 2001. URL: http://www.oracle.com/technetwork/java/embedded/javacard/documentation/intro-139322.html#proreq.

[60] *Yubico Forum - Slow RSA-2048 encryption/signing*. Last visited: 06.04.2016. 2013. URL: http://forum.yubico.com/viewtopic.php?f=26&t=1207.

Appendix A

Java Card code

Listing A.1: SecureCard.java.

```
1  package henrik;
2
3  import javacard.framework.APDU;
4  import javacard.framework.Applet;
5  import javacard.framework.ISO7816;
6  import javacard.framework.ISOException;
7  import javacard.framework.OwnerPIN;
8  import javacard.framework.Util;
9  import javacard.security.CryptoException;
10 import javacard.security.KeyBuilder;
11 import javacard.security.KeyPair;
12 import javacard.security.RSAPrivateKey;
13 import javacard.security.RSAPublicKey;
14 import javacard.security.Signature;
15 import javacard.security.AESKey;
16 import javacardx.apdu.ExtendedLength;
17 import javacardx.crypto.*;
18 import javacard.security.*;
19 import javacard.framework.JCSystem;
20
21 public class SecureCard extends Applet implements ExtendedLength
       ↪ {
22     //Try to allocate all variable here and do not create new
           ↪ ones
23     //The Public/Private key pair that this card will use
24     private KeyPair keys;
```

```
25      private KeyPair sKeys; //PLACEHOLDER
26      //Signature object to sign with card private key
27      private Signature sig;
28      //Card Public key
29      private RSAPublicKey uPub;
30      //Card Private key
31      private RSAPrivateKey uPrv;
32      //  To store data to be sent beck to host application
33      byte[] output = new byte[32767];
34      //for temporary storing data before copying into output
35      byte[] buff2 = new byte[2];
36      //For bigger data
37      byte[] bigArray;
38      //To store the size of the output buffer
39      short size;
40      //Length of signature or other short values
41      short len;
42      //Size of modulus and signature
43      final short keysize=64;
44
45      //Predefined Commands
46      private final byte SEND_U_PUB_MOD=(byte) 0x01;
47      private final byte SEND_U_PUB_EXP=(byte) 0x02;
48      private final byte SIGN=(byte) 0x03;
49      private final byte BINDING=(byte) 0x05;
50      private final byte RSACRYPTO=(byte) 0x06;
51      private final byte REFLECT=(byte) 0x08;
52      private final byte AESCRYPTO=(byte) 0x09;
53
54
55
56      //Cryptography
57      Cipher cipherRSA;
58      Cipher cipherAES;
59      byte[] cryptoBuffer;// = new byte[32767];
60
61      AESKey aesKey;
62      RandomData randomData;
63      byte[] rnd;
64
65
66      short policy13Offset = 6;
67
68
69      //Binding
```

```
70      byte pinIsPresentFlag;
71      OwnerPIN pincode;
72      final byte PIN_TRY_LIMIT = 0x03;
73      final byte PIN_SIZE = 0x04;
74      final byte INCOMING_PIN_OFFSET = 0x00;
75      byte[] h0Buffer = new byte[15000];
76      RSAPublicKey mPub;
77      RSAPublicKey sPub; //PLACEHOLDER
78
79
80      private SecureCard() {
81          //Instantiate all object the applet will ever need
82          try{
83
84              //Binding
85              pinIsPresentFlag = 0x00;
86              pincode = new OwnerPIN(PIN_TRY_LIMIT, PIN_SIZE);
87
88              byte[] pincombination = {0x01, 0x03, 0x03, 0x07};
89              pincode.update(pincombination, (short) 0, (byte) 0x04
                    ↪ );
90
91
92
93
94              keys = new KeyPair(KeyPair.ALG_RSA, KeyBuilder.
                    ↪ LENGTH_RSA_512);
95              sKeys = new KeyPair(KeyPair.ALG_RSA, KeyBuilder.
                    ↪ LENGTH_RSA_2048);
96              //keys = new KeyPair(KeyPair.ALG_RSA, KeyBuilder.
                    ↪ LENGTH_RSA_2048);
97
98              //Set signature algorithm
99              sig = Signature.getInstance(Signature.
                    ↪ ALG_RSA_SHA_PKCS1, false);
100
101             sKeys.genKeyPair();
102             sPub = (RSAPublicKey) sKeys.getPublic();
103
104             mPub = (RSAPublicKey) KeyBuilder.buildKey(KeyBuilder.
                    ↪ TYPE_RSA_PUBLIC, (short) 512, false);
105
106             //Generate the card keys
107             keys.genKeyPair();
108             //Get the public key
```

```
109             uPub = (RSAPublicKey) keys.getPublic();
110
111             //Get the private key
112             uPrv = (RSAPrivateKey) keys.getPrivate();
113             //Initialize the signature object with card private
                    ↪ key
114             sig.init(uPrv, Signature.MODE_SIGN);
115
116             //Crypto RSA
117             cipherRSA = Cipher.getInstance(Cipher.ALG_RSA_PKCS1,
                    ↪ false);
118
119             //Crypto AES
120
121
122             cipherAES = Cipher.getInstance(Cipher.
                    ↪ ALG_AES_BLOCK_128_CBC_NOPAD, false);
123             aesKey = (AESKey) KeyBuilder.buildKey(KeyBuilder.
                    ↪ TYPE_AES, KeyBuilder.LENGTH_AES_128, false);
124             randomData = RandomData.getInstance(RandomData.
                    ↪ ALG_PSEUDO_RANDOM);
125             rnd = JCSystem.makeTransientByteArray((short)16,
                    ↪ JCSystem.CLEAR_ON_RESET);
126             randomData.generateData(rnd, (short)0, (short)rnd.
                    ↪ length);
127             aesKey.setKey(rnd, (short) 0);
128
129
130
131
132
133             }catch(CryptoException ex){
134                 ISOException.throwIt((short)(ex.getReason()) );
135             }catch(SecurityException ex){
136                 ISOException.throwIt((short)(0x6F10) );
137             }catch(Exception ex){
138                 ISOException.throwIt((short)(0x6F20));
139             }
140
141
142
143     }
144
145     public static void install(byte[] bArray, short bOffset,
            ↪ byte bLength) {
```

```
146         // GP-compliant JavaCard applet registration
147 
148         new SecureCard().register();
149     }
150 
151     public void process(APDU apdu) {
152         // Good practice: Return 9000 on SELECT
153         if (selectingApplet()) {
154             return;
155         }
156 
157         byte[] buff = apdu.getBuffer();
158         short dataOffset = (short) 7; //Hardcoded as it cannot be
                ↪  done dynamically (not working properly)
159 
160 
161         //Switch on the instruction code INS
162         switch (buff[ISO7816.OFFSET_INS]) {
163         case SEND_U_PUB_MOD:
164             //Retrieve the modulus, store it in the output byte
                    ↪ array and set the output length
165             size = uPub.getModulus(output, (short) 0);
166             break;
167         case SEND_U_PUB_EXP:
168             //Retrieve the public exponent, store it in the
                    ↪ output byte array and set the output length
169             size = uPub.getExponent(output, (short) 0);
170             break;
171         case SIGN:
172             short bytesReadSign = apdu.setIncomingAndReceive();
173             size = apdu.getIncomingLength();
174             short echoOffsetSign = (short)0;
175             while(bytesReadSign > 0){
176                 Util.arrayCopyNonAtomic(buff, dataOffset, h0Buffer
                        ↪ , echoOffsetSign, bytesReadSign);
177                 echoOffsetSign += bytesReadSign;
178                 bytesReadSign = apdu.receiveBytes(dataOffset);
179             }
180             size = sig.sign(h0Buffer, (short) 0, bytesReadSign,
                    ↪ output, (short) 0);
181             break;
182         case (byte) BINDING:
183             byte p1 = buff[ISO7816.OFFSET_P1];
184 
185             //First transaction
```

```
186             if(p1 == (byte) 0x01){
187                 output[0] = 0x05; //Type of transaction
188                 if(pincode.isValidated()){
189                     output[1] = 0x01;
190                 }
191                 else{
192                     output[1] = 0x00;
193                 }
194 
195                 output[2] = pincode.getTriesRemaining(); //
                        ↪ PINIsOKFlag
196                 size = (short) 3;
197             }
198 
199             //Second transaction
200             else if(p1 == (byte) 0x02){
201 
202                 //SAFE COPY TO NEW BUFFER
203                 short bytesRead = apdu.setIncomingAndReceive();
204                 size = apdu.getIncomingLength();
205                 short echoOffset = (short)0;
206                 while(bytesRead > 0){
207                     Util.arrayCopyNonAtomic(buff, dataOffset,
                            ↪ h0Buffer, echoOffset, bytesRead);
208                     echoOffset += bytesRead;
209                     bytesRead = apdu.receiveBytes(dataOffset);
210                 }
211 
212                 pincode.check(h0Buffer, (short) 0, PIN_SIZE);
213                 output[0] = 0x05; //Type of transaction
214 
215                 if(pincode.isValidated()){
216                     output[1] = 0x09;
217                     output[2] = 0x00;
218                     size = (short) 3;
219 
220                 }
221                 else{
222                     output[1] = 0x00;
223                     output[2] = pincode.getTriesRemaining();
224                     size = (short) 3;
225                 }
226             }
227 
228             else if(p1 == (byte) 0x03){
```

```
229 //              SAFE COPY TO NEW BUFFER
230                 short bytesRead = apdu.setIncomingAndReceive();
231                 short incomingLength = apdu.getIncomingLength();
232                 short echoOffset = (short)0;
233                 while(bytesRead > 0){
234                     Util.arrayCopyNonAtomic(buff, dataOffset,
                        ↪ h0Buffer, echoOffset, bytesRead);
235                     echoOffset += bytesRead;
236                     bytesRead = apdu.receiveBytes(dataOffset);
237                 }
238
239                 short modLength = Util.makeShort((byte)0x00,
                    ↪ h0Buffer[0]);
240                 short expLenghtPos = (short) ((short) modLength +
                    ↪ (short) 1);
241                 short expLength = Util.makeShort((byte)0x00,
                    ↪ h0Buffer[expLenghtPos]);
242                 short expStartPos = (short) (modLength + 2);
243
244                 boolean mPubIsOK = false;
245
246                 try{
247                     mPub.setModulus(h0Buffer, (short) 1, modLength
                        ↪ );
248                     mPub.setExponent(h0Buffer, expStartPos,
                        ↪ expLength);
249                     mPubIsOK = true;
250                 }
251                 catch(CryptoException ex){
252                     output[0] = (byte) ex.getReason();
253                     output[1] = (byte) 0x02;
254                     size = 2;
255                     break;
256                 }
257                 catch(Exception ex){
258                     output[0] = (byte) 0x08;
259                     output[1] = (byte) 0x08;
260                     size = 2;
261                     break;
262                 }
263
264                 if(mPubIsOK && pincode.isValidated()){
265                     short totalsize = (short) (( (short) ( (short)
                        ↪  mPub.getSize() + (short) uPub.getSize
                        ↪ () + (short) aesKey.getSize()) / (short
```

```
                ↪ ) 8) + 10); //10 in header DANGEROUS
266             byte[] packet = new byte[totalsize];
267             short outputSize = 0;
268
269             //AESKEY
270             aesKey.getKey(packet, (short) 0);
271             short AESKeyLength = (short) (aesKey.getSize()
                ↪ /8);
272             //mPub
273             Util.arrayCopyNonAtomic(h0Buffer, (short) 0,
                ↪ packet, AESKeyLength, (short)
                ↪ incomingLength);
274             short AESmPubLenght = (short) (incomingLength
                ↪ + AESKeyLength);
275             outputSize = AESmPubLenght;
276
277             byte[] tempUPubArr = new byte[incomingLength];
278
279             //uPub - modulus
280             short tempLength = uPub.getModulus(tempUPubArr
                ↪ , (short)0);
281             packet[outputSize] = (byte)tempLength;
282             outputSize += 1;
283             Util.arrayCopyNonAtomic(tempUPubArr, (short)
                ↪ 0, packet, (short) (AESmPubLenght+1),
                ↪ tempLength);
284             outputSize += tempLength;
285
286             //uPub - exponent
287             tempLength = uPub.getExponent(tempUPubArr, (
                ↪ short) 0);
288             packet[outputSize] = (byte)tempLength;
289             outputSize +=1;
290             Util.arrayCopyNonAtomic(tempUPubArr, (short)
                ↪ 0, packet, (short) (outputSize),
                ↪ tempLength);
291             outputSize += tempLength;
292
293             //Signing
294             short signatureSize = sig.sign(packet, (short)
                ↪  0, totalsize, h0Buffer, (short) 0);
295             short h0UnencryptedLength = (short) (
                ↪ signatureSize + outputSize);
296
297             //Create unencrypted package
```

```
298                 byte[] h0Unencrypted = new byte[
                        ↪ h0UnencryptedLength];
299                 Util.arrayCopyNonAtomic(h0Buffer, (short) 0,
                        ↪ h0Unencrypted, (short) 0, signatureSize
                        ↪ );
300                 Util.arrayCopyNonAtomic(packet, (short) 0,
                        ↪ h0Unencrypted, signatureSize, totalsize
                        ↪ );
301
302                 //Encrypt with sPub
303                 cipherRSA.init(sPub, Cipher.MODE_ENCRYPT);
304
305                 try{
306
307                     size = cipherRSA.doFinal(
308                               h0Unencrypted,
309                               (short) 0,
310                               h0UnencryptedLength,
311                               output,
312                               (short)0);
313
314
315                 }
316                 catch(CryptoException ex){
317                     output[0] = 0x09;
318                     output[1] = (byte) ex.getReason();
319                     size = 2;
320                 }
321             }
322             else{
323                 output[0] = 0x09;
324                 output[1] = 0x09;
325             }
326
327         }
328         else if(p1 == (byte) 0x09){
329             pincode.resetAndUnblock();
330             output[0] = 0x05;
331             output[1] = 0x05;
332             size = (short) 2;
333         }
334
335
336
337         break;
```

```
338         case (byte) RSACRYPTO:
339             byte p1RSA = buff[ISO7816.OFFSET_P1];
340             if(p1RSA == (byte) 0x01){
341                 cipherRSA.init(uPub, Cipher.MODE_ENCRYPT);
342             }
343             else if(p1RSA == (byte) 0x02){
344                 cipherRSA.init(uPrv, Cipher.MODE_DECRYPT);
345             }
346 
347             short bytesReadRSA = apdu.setIncomingAndReceive();
348             size = apdu.getIncomingLength();
349             short echoOffsetRSA = (short)0;
350             while(bytesReadRSA > 0){
351                 Util.arrayCopyNonAtomic(buff, dataOffset,
                        ↪ cryptoBuffer, echoOffsetRSA, bytesReadRSA);
352                 echoOffsetRSA += bytesReadRSA;
353                 bytesReadRSA = apdu.receiveBytes(dataOffset);
354             }
355 
356 
357             size = cipherRSA.doFinal(
358                       cryptoBuffer,
359                       (short) 0,
360                       size,
361                       output,
362                       (short)0);
363             break;
364 
365         case (byte) REFLECT:
366 
367             short bytesRead = apdu.setIncomingAndReceive();
368             //size = bytesRead;
369             size = apdu.getIncomingLength();
370             short echoOffset = (short)0;
371             while(bytesRead > 0){
372                 Util.arrayCopyNonAtomic(buff, dataOffset, output,
                        ↪ echoOffset, bytesRead);
373                 echoOffset += bytesRead;
374                 bytesRead = apdu.receiveBytes(dataOffset);
375             }
376             break;
377 
378         case (byte) AESCRYPTO:
379             byte p1AES = buff[ISO7816.OFFSET_P1];
380             if(p1AES == (byte) 0x01){
```

```
381                 cipherAES.init(aesKey, Cipher.MODE_ENCRYPT);
382             }
383             else if(p1AES == (byte) 0x02){
384                 cipherAES.init(aesKey, Cipher.MODE_DECRYPT);
385             }
386 
387 
388             short bytesReadECAES = apdu.setIncomingAndReceive();
389             size = apdu.getIncomingLength();
390             short echoOffsetECAES = (short)0;
391             while(bytesReadECAES > 0){
392                 Util.arrayCopyNonAtomic(buff, dataOffset,
                        ↪ cryptoBuffer, echoOffsetECAES,
                        ↪ bytesReadECAES);
393                 echoOffsetECAES += bytesReadECAES;
394                 bytesReadECAES = apdu.receiveBytes(dataOffset);
395             }
396 
397             try{
398             size = cipherAES.doFinal(
399                    cryptoBuffer,
400                       (short) 0,
401                       (short) size,
402                       output,
403                       (short)0);
404             }
405             catch(CryptoException ex){
406                 size = 2;
407                 output[0] = (byte) ex.getReason();
408                 output[1] = 0x02;
409             }
410 
411             break;
412 
413         default:
414             // good practice: If you don't know the INStruction,
                    ↪ say so:
415             ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
416         }
417 
418         send(apdu);
419     }
420 
421     //Common method that sets the size of the output to the
            ↪ global variable size and sends the content of the
```

```
        ↪ global variable output
422     private void send(APDU apdu) {
423         apdu.setOutgoing();
424         apdu.setOutgoingLength(size);
425         apdu.sendBytesLong(output, (short) 0, size);
426         cleanBuffers();
427     }
428 
429     private void cleanBuffers(){
430         h0Buffer = new byte[32767];
431         output = new byte[32767];
432     }
433 }
```

Appendix B

Android library

We chose to only include the `CommunicationController` in this appendix as including all of the code for the library would exceed the sensible amount of pages in an appendix. We recommend investigating the Android library in an IDE for specifics on how classes are implemented.

Listing B.1: CommunicationController.java.

```
1  package com.master.henrik.controller;
2
3  import android.app.Activity;
4  import android.util.Log;
5
6  import com.master.henrik.shared.CommunicationType;
7  import com.master.henrik.shared.Converter;
8
9  import java.security.interfaces.RSAPublicKey;
10 import java.util.Arrays;
11
12 /**
13  * Created by Henri on 17.04.2016.
14  */
15 public class CommunicationController {
16
17     NFCSmartcardController nfcscc;
18     MSDSmartcardController msdscc;
19     final String TAG = "CommunicationController";
20     CommunicationType type = CommunicationType.NOTSET;
21
```

```
22      /**
23       *
24       * @param nfcSmartcardControllerInterface The interface
             ↪ which the activity implements for asynchronous
             ↪ communication between the smart card and activity..
25       * @param currentActivity
26       */
27      public void setupNFCController(
            ↪ NFCSmartcardControllerInterface
            ↪ nfcSmartcardControllerInterface, Activity
            ↪ currentActivity) {
28          if(nfcscc == null) {
29              nfcscc = new NFCSmartcardController(
                    ↪ nfcSmartcardControllerInterface,
                    ↪ currentActivity);
30          }
31          type = CommunicationType.NFC;
32      }
33
34      public void initNFCCommunication(String cardAID, String INS
            ↪ , String P1, String P2, String hexMessage){
35          Log.i(TAG, "Initiated NFCCommunication.");
36          nfcscc.sendDataToNFCCard(cardAID, INS, P1, P2,
                ↪ hexMessage);
37      }
38
39      public void disableNFC(){
40          nfcscc.disableNFC();
41      }
42
43      /**
44       *
45       * @param msdSmartcardControllerInterface The interface
             ↪ which the activity implements for asynchronous
             ↪ communication between the smart card and activity.
46       * @param currentActivity
47       */
48      public void setupmSDController(
            ↪ MSDSmartcardControllerInterface
            ↪ msdSmartcardControllerInterface, Activity
            ↪ currentActivity) {
49          if(msdscc == null) {
50              msdscc = new MSDSmartcardController(
                    ↪ msdSmartcardControllerInterface,
                    ↪ currentActivity);
```

```
51          }
52          type = CommunicationType.MSD;
53      }
54
55      public void initmSDCommunication(String cardAID, String INS
            ↪ , String P1, String P2, String hexMessage){
56          Log.i(TAG, "Initiated MSDCommunication.");
57          msdscc.sendDataTomSDCard(cardAID, INS, P1, P2,
                ↪ hexMessage);
58      }
59
60      public void bindingStepOne(String AID){
61          if(type.equals(CommunicationType.NFC)){
62              nfcscc.sendDataToNFCCard(AID, "05", "01", "00", "00
                    ↪ ");
63          }
64          else{
65              msdscc.sendDataTomSDCard(AID, "05", "01", "00", "00
                    ↪ ");
66          }
67      }
68
69      public void bindingStepTwo(String AID, String code){
70          if(type.equals(CommunicationType.NFC)){
71              nfcscc.sendDataToNFCCard(AID, "05", "02", "00",
                    ↪ code);
72          }
73          else{
74              msdscc.sendDataTomSDCard(AID, "05", "02", "00",
                    ↪ code);
75          }
76      }
77
78      public void bindingStepThree(String AID, RSAPublicKey mPub)
            ↪  {
79          byte[] publicByteArrModTemp = mPub.getModulus().
                ↪ toByteArray();
80          byte[] publicByteArrMod = Arrays.copyOfRange(
                ↪ publicByteArrModTemp, 1, publicByteArrModTemp.
                ↪ length); //Remove signed short
81          byte[] publicByteArrExp = mPub.getPublicExponent().
                ↪ toByteArray();
82
83          String publicHexKeyMod = Converter.ByteArrayToHexString
                ↪ (publicByteArrMod);
```

```
84          String modLength = Integer.toHexString(publicHexKeyMod.
                ↪ length()/2);
85
86          Log.d(TAG, "Mod: " + publicHexKeyMod);
87          Log.d(TAG, publicHexKeyMod.length() +" : " + modLength)
                ↪ ;
88
89          String publicHexKeyExp = Converter.ByteArrayToHexString
                ↪ (publicByteArrExp);
90          String expLength = "0" + Integer.toHexString(
                ↪ publicHexKeyExp.length()/2);
91          Log.d(TAG, "Exp: " + publicHexKeyExp);
92          Log.d(TAG, publicHexKeyExp.length() + " : " +expLength)
                ↪ ;
93
94          String fullMessage = modLength + publicHexKeyMod +
                ↪ expLength + publicHexKeyExp;
95
96
97          Log.d(TAG, "Fullmsg length: " + fullMessage.length());
98          Log.d(TAG, "Fullmsg: " + fullMessage);
99          nfcscc.sendDataToNFCCard(AID, "05", "03", "00",
                ↪ fullMessage);
100     }
101
102     /**
103      * Sign data using smart card.
104      * @param type
105      * @param cardAID
106      * @param hexMessage
107      */
108     public void signData(CommunicationType type, String cardAID
            ↪ , String hexMessage){
109         if(type.equals(CommunicationType.NFC)) {
110             nfcscc.sendDataToNFCCard(cardAID, "03", "00", "00",
                    ↪  hexMessage);
111         }
112         else{
113             msdscc.sendDataTomSDCard(cardAID, "03", "00", "00",
                    ↪  hexMessage);
114         }
115     }
116
117     /**
118      * Encrypt or decrypt data using RSA. Uses the smart card's
```

```
         ↪ keypair.
119      * @param type
120      * @param encrypt true for encrypt, false for decrypt.
121      * @param cardAID
122      * @param hexMessage
123      */
124     public void cryptoRSA(CommunicationType type, boolean
         ↪ encrypt, String cardAID, String hexMessage){
125         String p1 = "02";
126         if(encrypt) {
127             p1 = "01";
128         }
129
130         if(type.equals(CommunicationType.NFC)) {
131             nfcscc.sendDataToNFCCard(cardAID, "06", p1, "00",
                ↪ hexMessage);
132         }
133         else{
134             msdscc.sendDataTomSDCard(cardAID, "06", p1, "00",
                ↪ hexMessage);
135         }
136     }
137
138     /**
139      * Encrypt or decrypt data using AES. Uses the smart card's
         ↪ symmetric key.
140      * @param type
141      * @param encrypt true for encrypt, false for decrypt.
142      * @param cardAID
143      * @param hexMessage
144      */
145     public void cryptoAES(CommunicationType type, boolean
         ↪ encrypt, String cardAID, String hexMessage){
146         String p1 = "02";
147         if(encrypt) {
148             p1 = "01";
149         }
150
151         if(type.equals(CommunicationType.NFC)) {
152             nfcscc.sendDataToNFCCard(cardAID, "09", p1, "00",
                ↪ hexMessage);
153         }
154         else{
155             msdscc.sendDataTomSDCard(cardAID, "09", p1, "00",
                ↪ hexMessage);
```

```
156             }
157         }
158 
159         /**
160          * Retrive the smart card public key modulus.
161          * @param type
162          * @param cardAID
163          */
164         public void getCardPubMod(CommunicationType type, String
                ↪ cardAID){
165             if(type.equals(CommunicationType.NFC)) {
166                 nfcscc.sendDataToNFCCard(cardAID, "01", "00", "00",
                        ↪  "00");
167             }
168             else{
169                 msdscc.sendDataTomSDCard(cardAID, "01", "00", "00",
                        ↪  "00");
170             }
171         }
172 
173         /**
174          * Retrive the smart card public key exponent.
175          * @param type
176          * @param cardAID
177          */
178         public void getCardPubExp(CommunicationType type, String
                ↪ cardAID){
179             if(type.equals(CommunicationType.NFC)) {
180                 nfcscc.sendDataToNFCCard(cardAID, "02", "00", "00",
                        ↪  "00");
181             }
182             else{
183                 msdscc.sendDataTomSDCard(cardAID, "02", "00", "00",
                        ↪  "00");
184             }
185         }
186     }
```

Appendix C

Diagrams

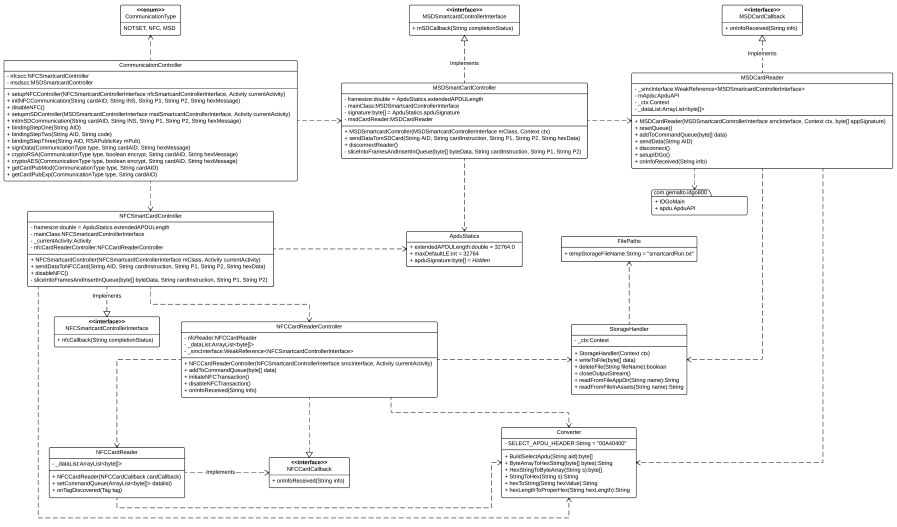

Figure C.1: Class diagram for Android Library.

Appendix D

Framework installation

D.1 Smart card development environment

Setup guide for developing smart card applications using Eclipse 3.2.

1. Unzip "Smart_card.zip"
2. Install Java Development Kit 6u45. Use "jdk-6u45-windows-i586.exe" for 32-bit Windows. Other operating system versions can be found at: `http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase6-419409.html#jdk-6u45-oth-JPR`
3. Copy the contents of "java_card_kit-2_2_2-windows" to a directory of your choice.
4. Copy the contents of "eclipse-SDK-3.2.2-win32" to a directory of your choice.
5. Copy the contents from "eclipse-jcde-0.2
   plugins" into the plugin folder for Eclipse from previous step.
6. Start Eclipse using the batch script "RUN_ME.bat".
7. In the toolbar select "Java Card →Preferences" and make sure the Java Card Development Kit path points to where the kit was installed. Refer to figure D.1.
8. In the toolbar select "JCWDE →Preferences" and make sure the Java

Card Development Kit path points to where the kit was installed. Refer to figure D.1.

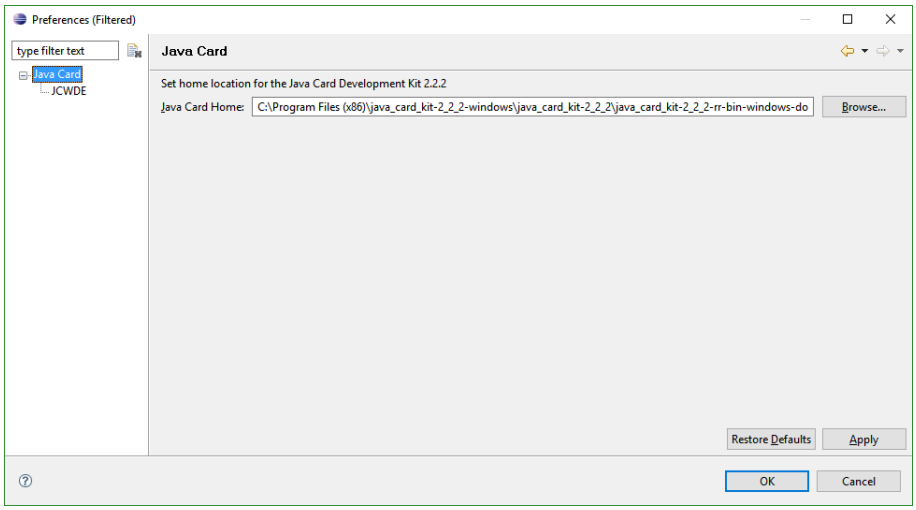

Figure D.1: Configuring Java Card kit 2.2.2 for Eclipse.

D.2 Smart card deployment

Setup guide for deploying smart card applications using GlobalPlatformPro (GP).

1. Copy the contents of "Smart_card_deployment" to a directory of your choice.
2. Configure "runMe.bat" and point to the correct .cap file.
3. Run "runMe.bat" to delete and install your smart card application.

D.3 Smart card testing

Setup guide for testing smart card applications using PyAPDUTool.

1. Copy the contents of "Smart_card_test_tools" to a directory of your choice.
2. Run the tools.

D.4 Android development environment

Setup guide for developing Android applications with our smart card library using Android Studio.

1. Download the newest version of Android Studio from `http://developer.android.com/sdk/index.html`
2. Install Android SDK 6.0 using the Android SDK Manager.
3. Add the libraries "gPKIKeyStore" and "smartcardLibrary" to your Android application. `http://developer.android.com/sdk/installing/create-project.html#ReferencingLibraryModule`
4. Fix any potential path issues.