Mobile data security and cloud storage

Henrik Sivertsgård, HIB: H139734 : UIB: 212779 , henriksive@gmail.com
Supervisors: Marcus Bezem, Federico Mancini
April 30, 2016

Abstract

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Problem statement and motivation

In todays society the standard commercial smartphone is considered secure by the average user. This statement is partially true as the regular user rarely encounters sensitive data apart from personal passwords. As the development on smartphones have skyrocketed new parties have expressed interest in smartphone capabilities. In relation to this there is now a need for the smartphone to handle sensitive data, e.g, health data, military intelligence and governmental information.

Handling senstive data in context of smartphones involves storing the data, processing the data and potentially sharing the data. In these areas there exists obstacles that needs to be dealt with in order to handle the sensitive data securely. Two of the biggest obstacles are being able to encrypt and decrypt data effectively and key management/storage.

In a perfect world the smart phone would be able to overcome these obstacles by itself. Many mobile devices lack the capability to store keys securely and run operations in a secure environment and thus cannot execute all steps needed. A smart card cans provide a solution to these problems, but we have to investigate if the smart cards introduces new attack vectors and constraints.

1.2 Goals

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 includes developing test applications for both the mobile device and the smart card. A framework, mobile application and smart card application allows us to test performance and security, 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 limitiations of smart cards in the context of hardware?"
- "What are the limitiations of smart cards in the context of software?"
- "What are the types of use cases we are able to solve and strengthen the security of using smart cards?"

1.3 Related work

The Secure Element Evaluation Kit (SEEK) for the Android platform is an open source project, maintained by Giesecke & Devrient GmbH [**Giesecke**], which have a vision to make it easier to use secure elements in Android applications. The framework provides 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 [32]. However, their focus are on the Android platform, while this thesis will focus more on the smart card itself and what a smart card brings to the table security wise.

1.4 Chapter introduction

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. Gives a basic understanding for which threats exists for the platform.

Chapter 3 - Smart card framework Discusses the basis for the smart card communication framework, what goals we have for the framework and describes how we are going to work when developing the framework.

Chapter 4 - Challenges Identifies and examines challenges that have emerged when trying to use smart cards in co-operation with mobile devices. Presents and evaluates a solution for the problems.

Chapter 5 - Framework Implementation Gives an overview of how the framework is implemented and structured using code examples and diagrams.

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

Chapter 7 - Discussion and 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 background information necessary to understand how smart cards function and can increase security on mobile devices. The topics covered in this section are smart cards, mobile device operating systems, cryptography and threats in context of mobile devices.

2.1 Smart Card

A smart card refers to a pocket sized card with an integrated circuit, usually in the form of a plastic card with an integrated micro processor. In essence a smart card is a miniature computer with limited computing power.

2.1.1 Smart card architecture

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 [20] and ISO/IEC 7816 [21]. The processing power of smart cards are between 25-32 MHz and sport anywhere from 8Kbit to 128Kbit memory (EEPROM) [34].

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 works 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 cards support both of these technologies. Figure 2.1 is a blank hybrid card.

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, that 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.

Figure 2.1: Hybrid Smart card.

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 [19]. There are two types of APDU messages: Command APDU and Response APDU.

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 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, 2 or 3	Expected response length

Table 2.1: Command APDU layout.

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.

To better understand when to use the command APDUs and when to use response APDUs we will describe the following example: A locked door has a card reader connected to it. A person walks up to the door and holds up 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 by figure 2.2.

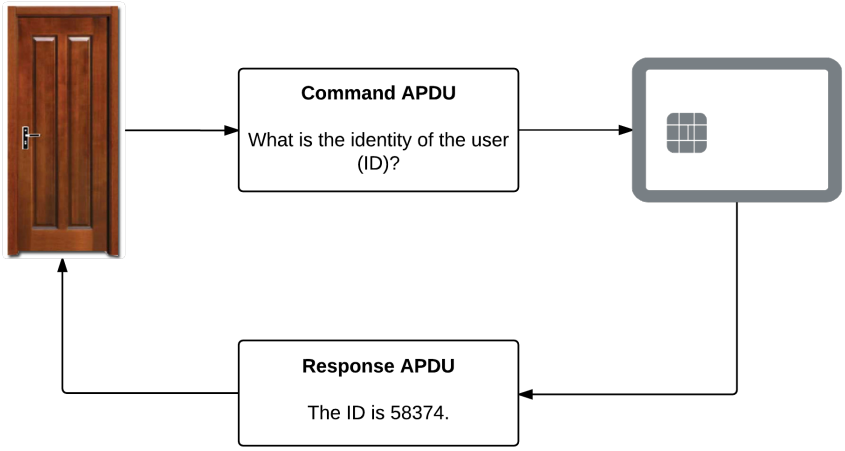

Figure 2.2: Door lock using smart card to unlock.

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.

2.1.3 JavaCard programming language

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 feasable to create programs that ran on different smart card manufacturer cards as the micro processors were manufacturer specific [42]. This created an environment where smart card issuers and their developers were locked to a specific manufacturer.

The company Schlumberger [31], later joined by Sun Microsystems [25], 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 [22].

The Java Card language behaves very similar to standard Java, but there are some substantial differences. Many Java classes and features are not present, e.g., int, double, long, java.lang.SecurityManager, threading and object cloning [3]. The structure of the applets differ 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 [43]. `Process` is the entry point for all requests to the application and where the applet specific logic is done [44].

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 is no automatic garbage collection in Java Card and that garbage collection must be invoked by the application itself. Deciding when to perform garbage collection is very difficult as on one side you don't want to do it often, and on the other side if you run out of memory you are out of luck.

When incorporating the features of a smart card with a mobile device it would be beneficial to not rely on an external contact card or contactless card. One of the key features of the Java Card language is that the software is able to run on technological different smart cards. As such it is possible to deploy the Java Card applet to special micro SD memory cards. This essentially enables the smart card to be integrated with the mobile device, but at the same time be an external runtime environment. Figure 2.3 is a micro SD memory card produced by Gemalto.

Figure 2.3: Micro SD card from Gemalto.

2.1.4 Other smart card programming languages

There exists several other languages for smart card programming. The two most notable are JavaCard (section 2.1.3) and MULTOS [24]. The latter uses the programming language C as the language for writing applications.

2.2 Mobile 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.

2.2.1 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 [35].

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.

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 mathematics are so complex that they are "uncrackable" 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 around 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 2.4 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.

Message authentication can also be done by public-key cryptography. First the message is hashed using a secure hash function, for instance SHA-2 [29], 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 message the receiver can with certainty conclude that the message has not been tampered with and originates from the sender.

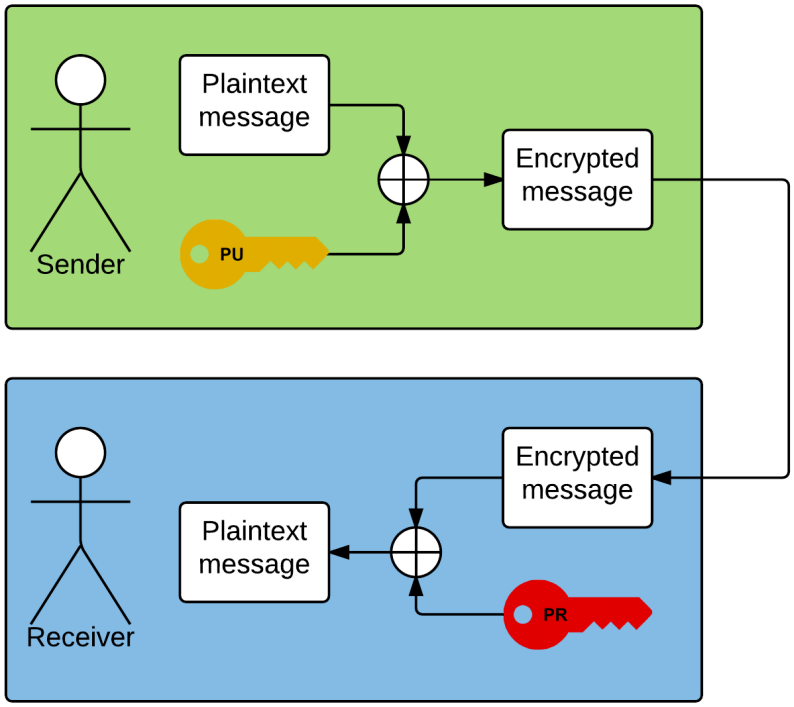

Figure 2.4: Asymmetric key encryption/decryption using public-private key pair.

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 very 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 or key

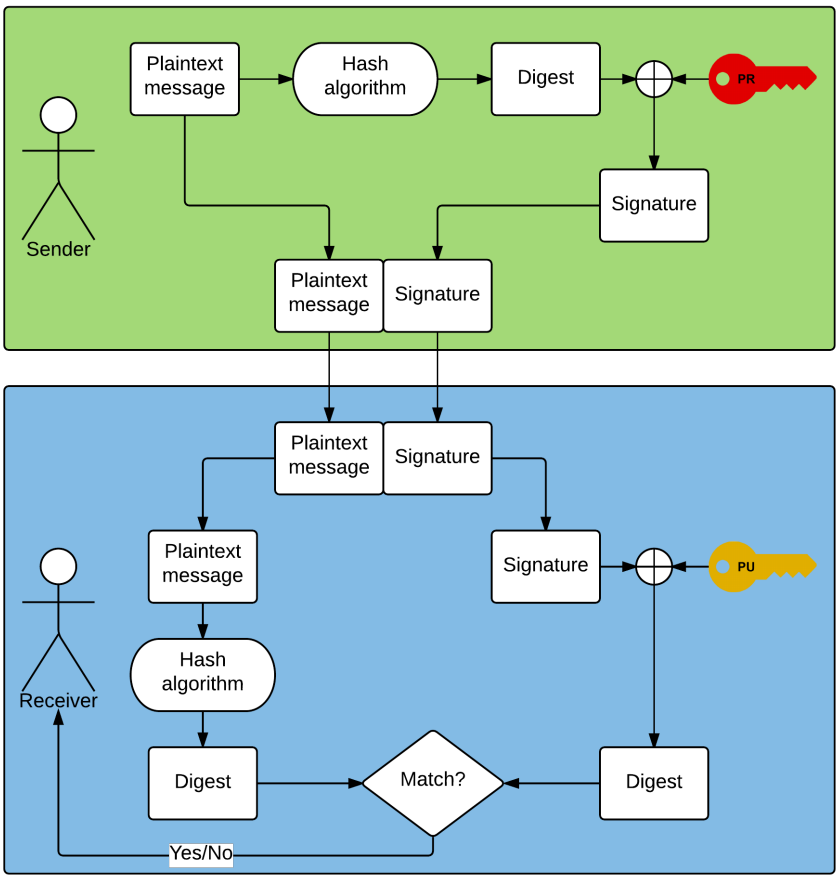

Figure 2.5: Digital signing using public-private key pair.

generation to be a known by both parties, also known as a "shared secret". 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 strenghts [9, 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 data must be divided into equal size blocks. The blocks cannot be too small as then they are prone to dictionary attacks and not too big as it will make the encryption/decryption process too resource intensive. In block cipher 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.

2.4 Mobile Threats

Mobile threats 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 Infected Device

The most obvious threat to mobile devices is when the mobile device itself is infected with virus or malware. The type of virus or malware 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. Taking into concideration that the mobile device we are operating on may be infected is of the utmost importance when designing and developing applications.

2.4.2 Lost or Stolen Device

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 retrive data from the device. A common defence 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 will fall into the wrong hands.

2.4.3 Insecure Communication Channel

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. More on this in section 2.4.4.

2.4.4 Authentication Challenges

As mentioned in section 2.4.3 a secure communication channel is useless if you are sending the data directly to the attacker. A vital part of security is being able to authenticate the parties in a 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 significance.

2.4.5 Insecure Applications

An often overlooked attack vector is badly implemented applications on the mobile device. This may include memory leaks, weak cryptography, open for code injections and plainly exposing private data to third parties. 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.

Chapter 3

Smart card objective

As mentioned in section 1.2 we want to create a framework for easy communication between a mobile device and a smart card. 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 design goals for the framework are:

- Easy to use.
- Require little to no understanding of JavaCard or smart cards.

- 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 make abstract is APDU. As a user/developer of the framework you can choose not to work with APDUs and use pre-created 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 5.2.2.

3.2 Smart card application

The first part of the 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.

3.2.1 Java Card version

The cards we have supports 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. Potentionally we 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 the relatively few products have this certification. The micro SD card we have access to are certified with EAL5, but only supports JavaCard 2.2.2. [14].

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 JavaCard 2.2.2. we had to consider if we wanted a newer JavaCard version with more functionality or if we wanted to comply

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

with government directives (EAL requirement). The obvious choice was the latter.

3.2.2 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 [11] 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. 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.

3.2.3 Deployment

We will be using GlobalPlatformPro (GP) [13] 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.

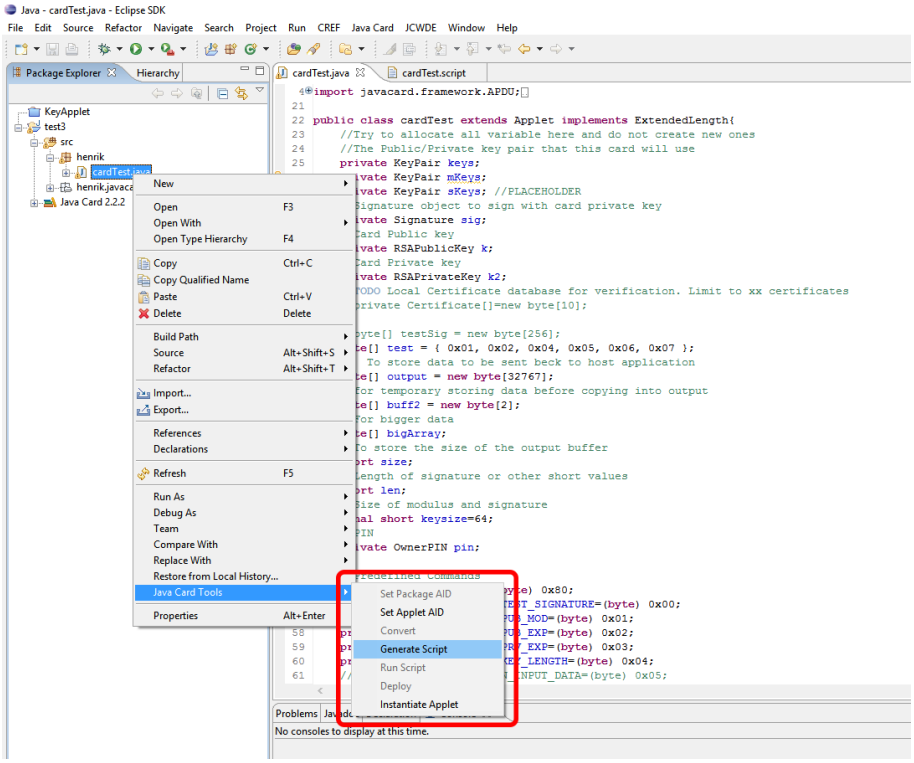

Figure 3.1: Screenshot of Eclipse showing JavaCard tools.

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.

To do this we will utilize a simple batch script which consisting of three 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 added as an added 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
```

3.2.4 Basic testing 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 [28]. 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.

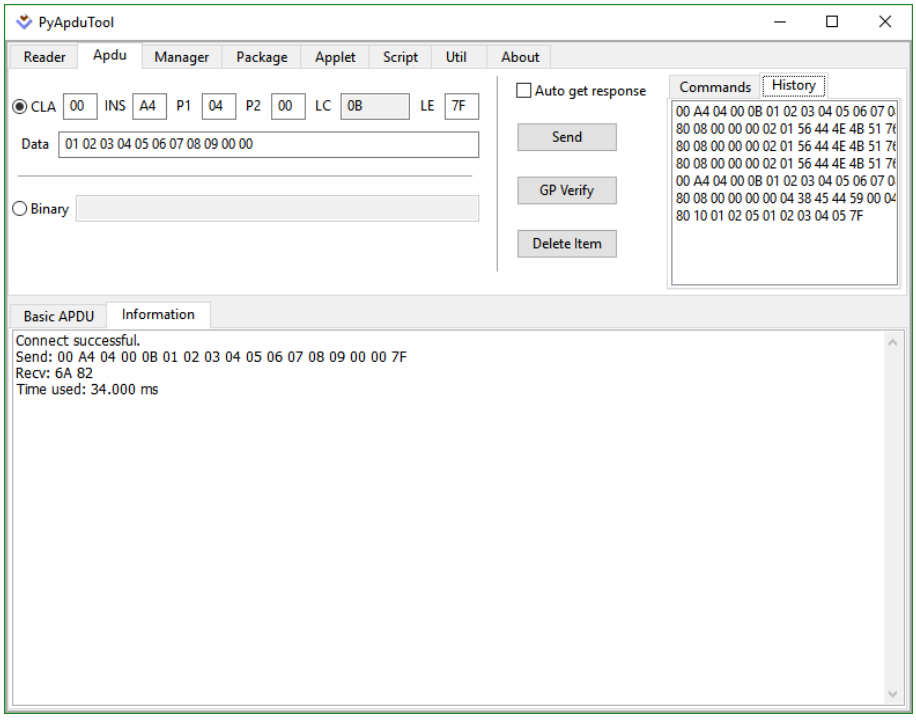

Figure 3.2: Select APDU sent to smart card via PyApduTool

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 behaviour such as out of memory,

too much traffic or NFC limitations. After the initial basic testing PyApduTool became obsolete and we had to test via an Android application.

3.3 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.

3.3.1 IDE

Android Studio [6] is the official IDE for Android application development. Android Studio is based on IntelliJ IDEA [17] and provides many automated tools for building, deploying and publishing Android applications. Android studio comes built in with Android Debug Bridge (ADB) which is an interface for communicating with virtual Android instances or physical Android devices. ADB gives developers the ability to log output from applications aswell as monitor memory, GPU and CPU usage of the mobile device.

3.3.2 Testing 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 3.3 and 3.4 shows runtime examples of the Android device. These monitor tools gives us a clear indication if we are doing an operation the Android device can't handle or if we are trying to perform operations that are too resource intensive.

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.

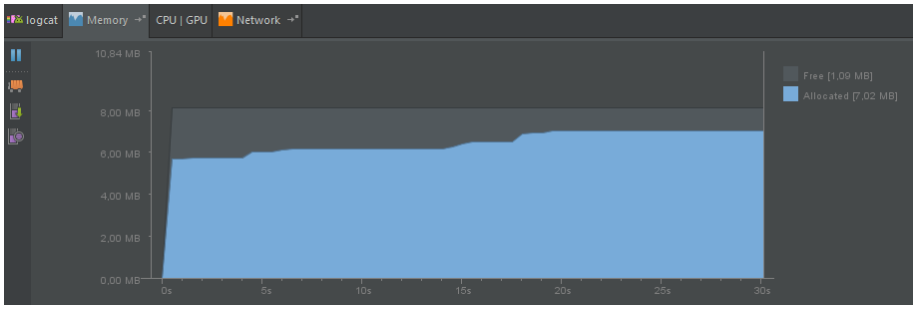

Figure 3.3: Android Debug Bridge memory monitor connected to a running Android device.

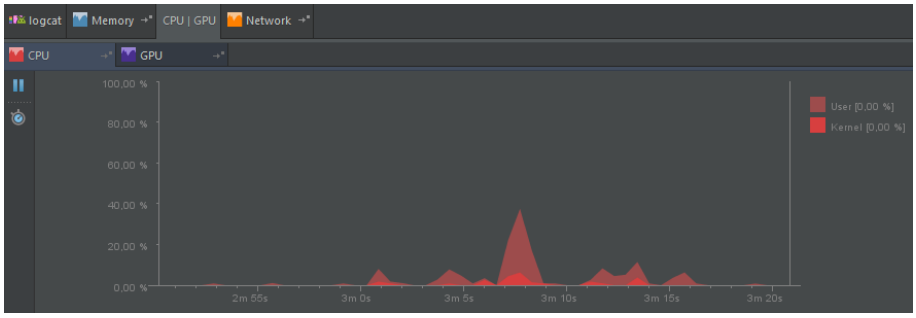

Figure 3.4: Android Debug Bridge CPU monitor connected to a running Android device.

Chapter 4

Challenges and use cases

In this chapter we will examine what challenges there are with regards to mobile devices and smart cards co-operating. The challenges must be viewed from a use case perspective where the benefits does not come from solving the challenges, but rather what we are able to achieve afterwards. For each challenge we will provide problem description, motivation, key concepts, possible solution and describe possible attack vectors.

4.1 Binding card and phone

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.

4.1.2 Goal

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 the direct consequences are:

- The keys stored on the smart card cannot be retrived 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 threats 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 the 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 problem from traditional authentication problems is that the smart card is not able to communicate directly with a trusted third party. All communication with off card applications or third parties must go through a mobile device. A technical 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?

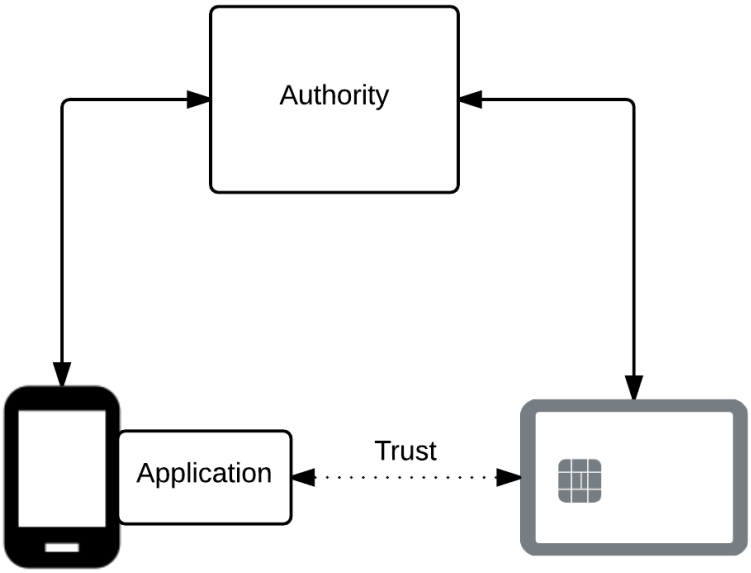

Figure 4.1: Trust based on a third party.

To combat that all information flowing from the smart card has to go through the mobile device, we can utilize the fact that the authority we are trying to authenticate with is also the 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 retrive the public key of the smart card before handing the smart card to the user/employee. The result is that we have already exchanged the keys and certificates we need for authentication before we introduce the mobile device.

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 tries to communicate 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, we can minimize the risk that the user will download a rogue application

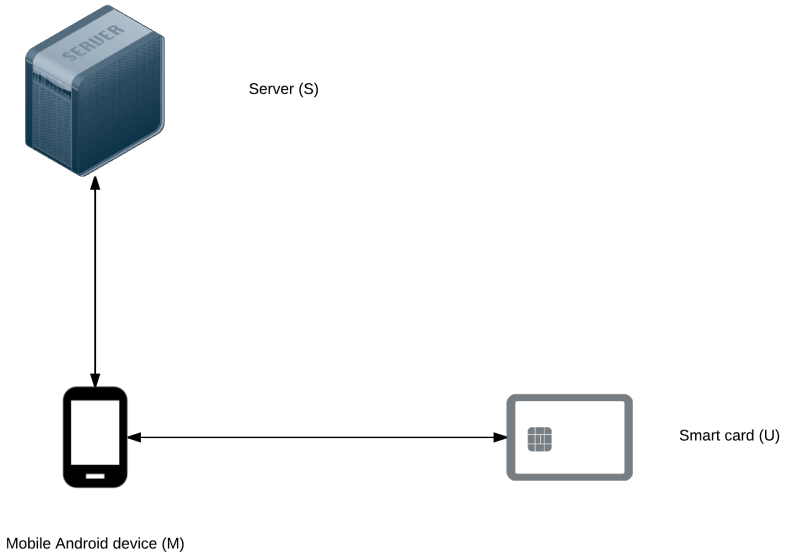

Figure 4.2: Server, mobile device and smart card communication flow.

with the same name [27]. To further mitigate the risk of installing the wrong application the user can disable the ability to side-load applications and not use a rooted device.

The other attack vector on the mobile device and user is man-in-the-middle attacks. 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 TLS [41] which also provides us with protection from replay attacks [8]. 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 the server certificate is already on the smart card and we can install it on the mobile device. An added benefit of this is that if the mobile device is rogue and tries to connect to the wrong server; the data sent to the server will be encrypted by the smart card.

Further we will need the user to authenticate with the authority. We have two options in order to achieve this. First option is that the user use a username and password combination directly with the authority, but

considering the binding is normally a one time case a more simplefied 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 code 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.

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. Firstly the mobile device public key, the smart card public key and generated AES key is signed by the smart card. Then we encrypt this package with the public key of the third party server. Figure 4.3 visualizes this structure.

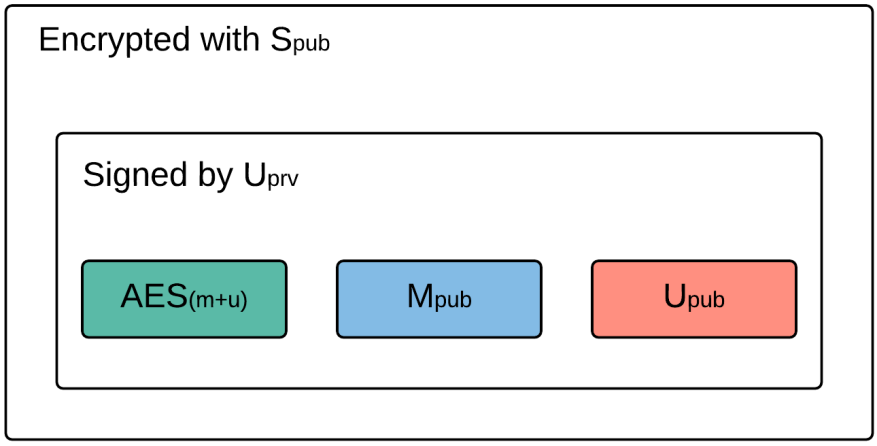

Figure 4.3: Verification package structure.

Abbreviations
U - Users smart card
M - Mobile device
S - Server, representation of authority
H_0 - Verification package
$\{Entity\}_{pub}$ - Public key of an entity (U, M, S)
$\{Entity\}_{prv}$ - Private key of an entity (U, M, S)
$\{AES\}_{Entity+Entity}$ - AES key of two entities (U, M, S)

1. Install Android application on mobile device (M) and insert smart card (U).
2. M generates RSA key-pair and stores.
3. M sends M_{pub} to U and requests verification package (H_0) from (U)).
4. U asks for a PIN code.
5. M provides PIN code.
6. U generates H_0 (refer to figure 4.3) and sends it to M.
7. M connects to the server (S) and sends (H_0) to S.
8. S decrypts (H_0) using S_{prv} and verifies the signature of U.

9. 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.

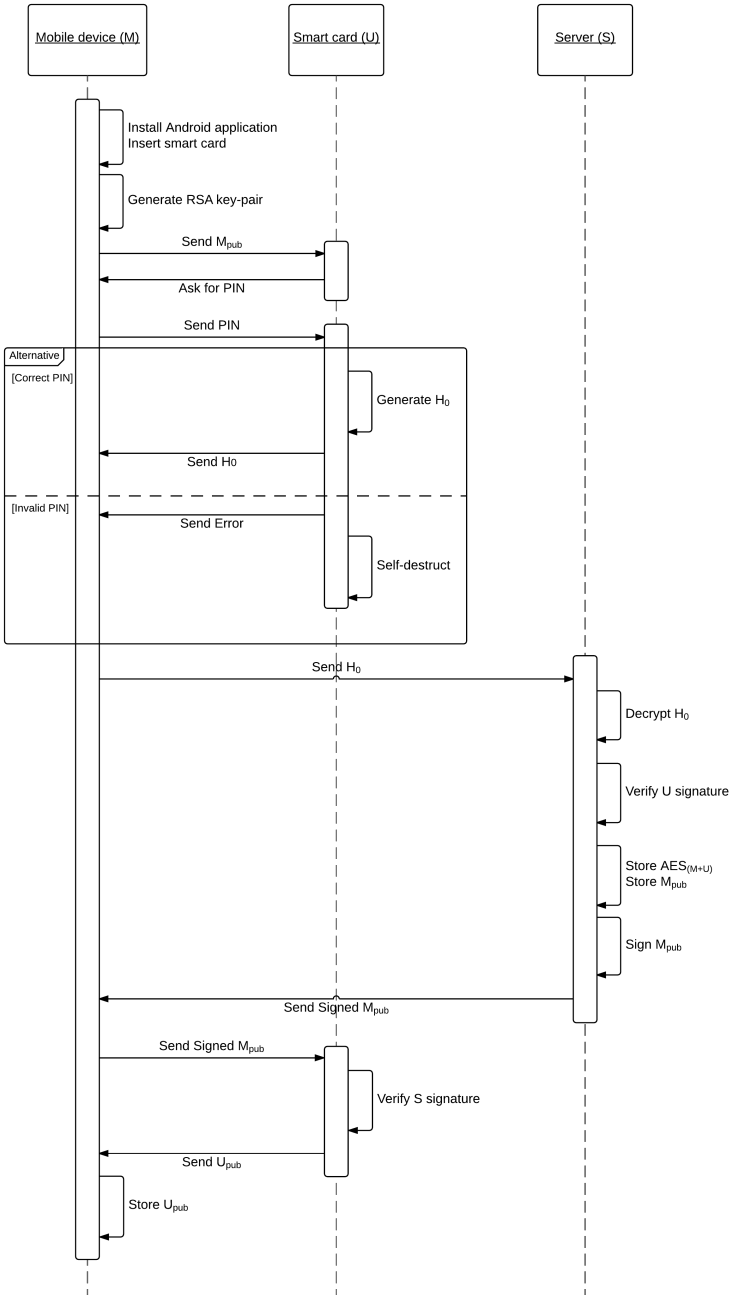

Figure 4.4: Sequence diagram for binding mobile device with smart card.

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 highten 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 can once in a while challenge the mobile device which requires the private key of the original binded device to solve (e.g. challenge is encrypted with M_{pub}).

4.1.5 Architecture evaluation

In this section we will justify and evaluate the parts of the solution that have security implications.

M generates RSA key-pair and stores it.
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.

U asks for PIN code.
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.

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 lose the smart card. We sign the package using the private key of the smart card and 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.

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. TLS will also serve as a counter to replay-attacks and man-in-the-middle attacks.

The server signs the public key of M.
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 it we ensure that the public key that is sent back to the smart card is infact the public key that was in the verification package.

U verifies that M_{pub} was signed by S and if successful U sends 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 does 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.

One of the most common public-key cryptosystems (refer to section 2.3.1) are 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 [39], 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 source conclude that with long enough keys and correct protocol implementation, the math behind RSA can be considered secure [9, 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" [9].

We are not directly utilizing the symmetric-key cryptography system AES, but it is an important aspect of the solution none the less. AES does not rely on number factorization, 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 we can thus conclude that using long keys are to be considered secure [9, p. 116-117].

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 correcly and does not allow for common attacks (DROWN [38], , etc.) it is classified as “probably secure” or “secure until proven otherwise”.

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 binded the mobile device and smart card, but the mobile device cannot be trusted.

To combat this we need to do two things. Firstly 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 [30]. 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 independent from 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 statement

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 store atleast 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 how do we ensure that the keypair are generated and stored securely? By solving this problem we are able to perform the binding process of the mobile device and smart card. We can also envision that there might emerge other use cases at later stages which require keys on the mobile device.

4.2.2 Goal

Our primary goals for mobile device keys are:

- Generate keys on mobile device or receive externally generated keys.

- Store keys securely on the mobile device.

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 availiable to application A and B. If application C tries to access the key the Android Keystore system blocks the action.

The applications does 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)" [16]. 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.

An application can check if the mobile device uses secure hardware for key storage using the Android class `KeyInfo`. The `KeyInfo` class containt 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.

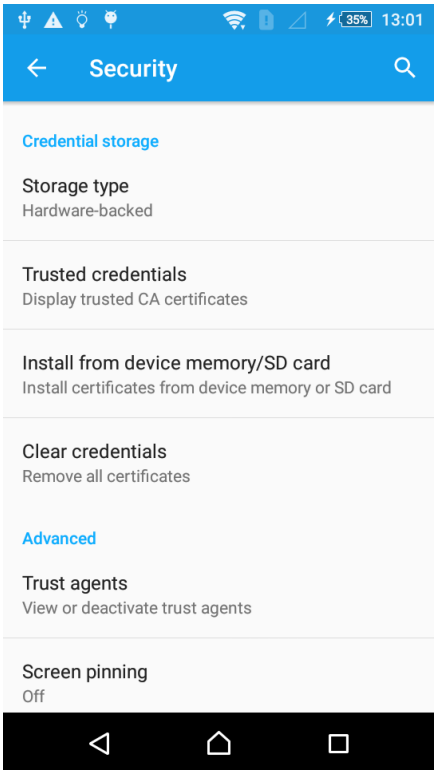

Figure 4.5: Screenshot of Android settings showing hardware-backed storage "enabled".

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 suported 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 Eliptic 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 //p12-fil etc.

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 keypairs [26]. 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 distrbution. 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).

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 white paper "Analysis of Secure Key Storage Solutions on Android" by Tim Cooijmans, Joeri de Ruiter and Erik Poll discusses hardware-backed storage further [40].

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 combat 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 are supposed to perform the binding process. For example if an attacker is able to get his hands on a smart card before it is binded 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 an administrator 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 to 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 are the distribution process which in turn introduces 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.

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 nonsecure 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 exists 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 [7]. 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 [12].

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 do 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 combat this 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 now have 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, wishes to perform the action.

The core abilities of a smart card are:

- Generate keys.
- Store keys.
- Sign data.
- Encrypt & decrypt small amounts of data.
- Store and handle 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).

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.

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 are "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. To ensure that the server is the only party that can provide the verification package, is to send a challenge from the smart card which only the server knows the answer to (shared key/secret). Refer to figure 4.6 and figure 4.7 for challenge and challenge response.

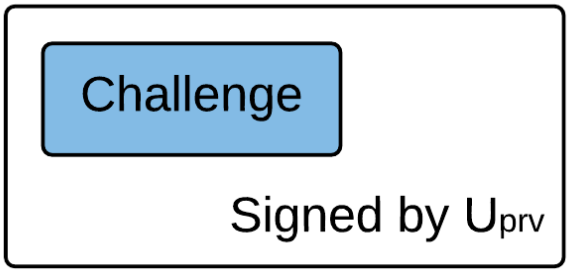

Figure 4.6: Smart card policy challenge.

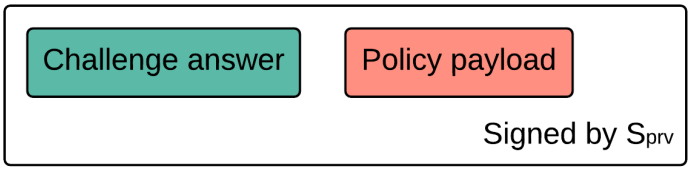

Figure 4.7: Smart card policy challenge response.

Challenge transaction

U - Users smart card
M - Mobile device
S - Server, representation of authority
OH - Original hash
NH - New hash
$\{\text{Entity}\}_{\text{prv}}$ - Private key of an entity (U, M, S)

First we assume the binding of card and phone from section 4.1 was successful. Secondly we assume that the server and smart card has a shared secret: a securely generated 256-bit AES key.

1. U generates a challenge which is a long random hash (OH).
2. U signs OH with U_{prv}.
3. U sends the signed OH package to S via the mobile device.
4. S computes new hash (NH) using the AES key S and U agreed upon.

5. S signs NH and the policy update.
6. S sends the signed NH package (with policy update) to U via the mobile device.
7. U computes the new hash and compares it to the NH it received.
8. U applies policy updates.

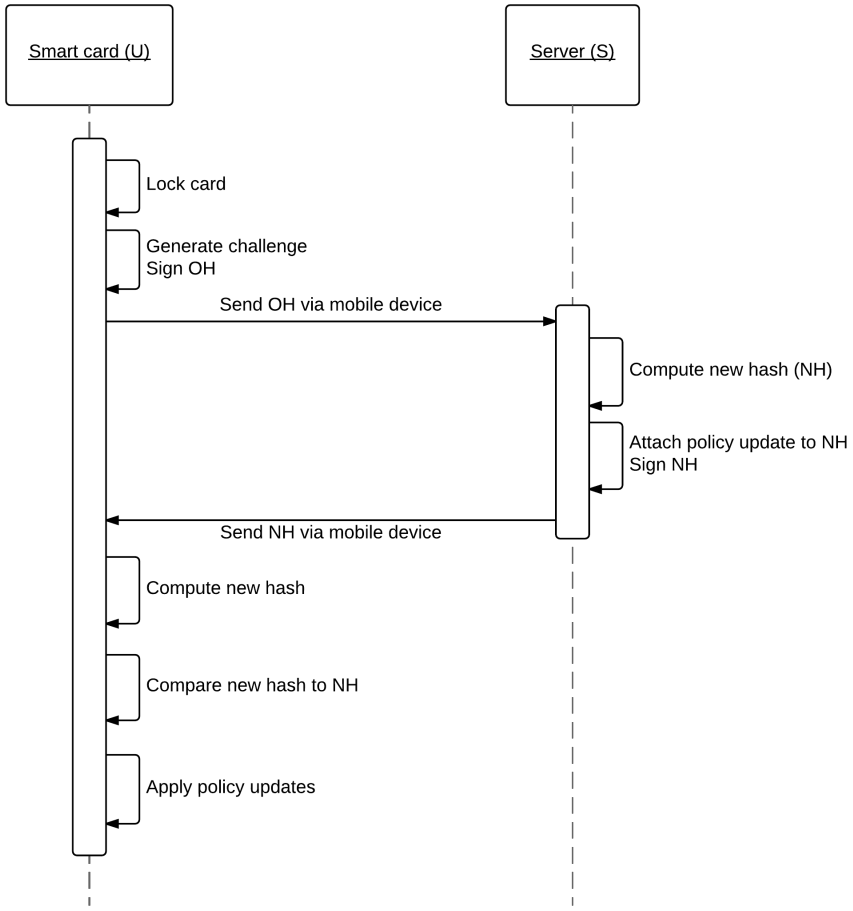

Figure 4.8: Smart card policy challenge.

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 thing 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 smart 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 4.9 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 4.9: 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 4.9 as incoming APDU.

Listing 4.4: Pseudo code for interpreting policy APDU with java smart 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 organisation 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

Known-plaintext attack
The challenge we use is simply two parties creating the same ciphertext from a hash and then comparing them. We then use digital signing to ensure that the challenge-response cannot be intercepted and spoofed. Since we do not encrypt the challenge-response in any way we have to vary of "known-plaintext attacks" [37]. To protect against this type of attack the smart card and server must use keys that are long and algorithms that cannot be brute-forced. The final solution should use atleast 256-bit long AES keys and as an added measure rotate keys.

Chapter 5

Implementation

In this chapter we will give an overview of the framework we have implemented and the implementation details. After reading this chapter the reader should have a good understanding of how the framework is built up and be able to utilize the framework in an Android application.

5.1 JavaCard Application

5.1.1 Application

Goal
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
As described in section 2.1.3 all javacard applications must implement the method `Install`. `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 everytime the application is used.

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 javacard 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.

Finalization

At the end of the `Process` invokation we invoke the `Send` method which takes the data in the outgoing buffer, package it for sending and send it as a response APDU.

The result

What we end up with is a test plattform 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.

Listing 5.1: Pseudo code for javacard test application.

```
1  public class cardApplication extends Applet implements ExtendedLength{
2
3      //Variable declarations
4
5      private cardApplication() {
6          //Variable initialization
7      }
8
9      public void process(APDU apdu) {
10         //Process incoming APDU
11         if (selectingApplet()) {
12             return;
13         }
14         buff = apdu.getBuffer();
15
16         switch(buff[ISO7816.OFFSET_INS]){
17             case 0x00:
```

```
18              case 0x01:
19              ...
20              case 0xff:
21              default:
22
23          }
24          Send(apdu);
25      }
26
27      private void send(APDU apdu) {
28          //Package outgoing buffer
29          //Send response APDU
30      }
31  }
```

As seen in 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 5.2.2. 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 selv-explanatory. Refer to Appendix A for JavaCard code.

5.1.2 Extending the JavaCard application

When adding more functionality to the JavaCard 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 JavaCard programming.

Secondly it is important to keep in mind that JavaCard 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 7.3.

5.2 Android Application

5.2.1 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 [15])

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.

5.2.2 Application

In section 3.1 we described the goals of the framework. The first goal we will describe the implementation of 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.

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

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 asyncronous transaction the library will temporary save the responses from the cards to a file only accessible by the running application. To retrive 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 5.1 we can see how `CommunicationController` is designed to be the abstraction layer between Android activities and smart cards.

The methods available are:

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

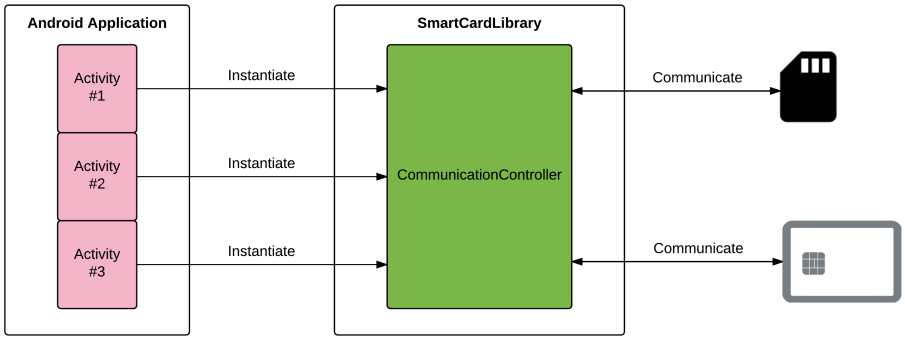

Figure 5.1: Abstraction layer between Android activities and smart cards.

- 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 is before trying to fetch the response data. Refer 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  }
```

Figure 5.2 provides a simplefied and technical overview of how the Android side of the library is built up. This diagram also shows which parameters are required for the methods available in `CommunicationController`. The complete class diagram for the Android library can be found in Appendix C, figure C.1.

The package diagram, figure 5.3, shows how the packages in the library are structured.

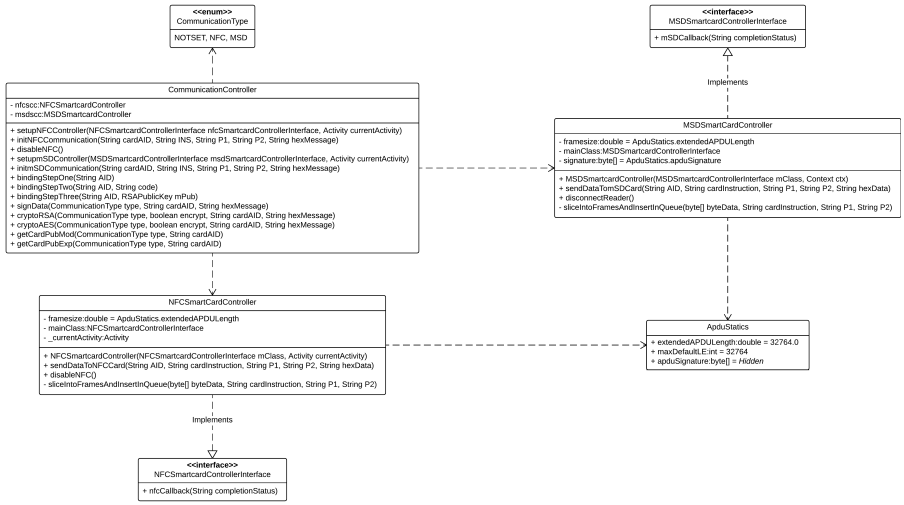

Figure 5.2: Simplefied class diagram for Android Library.

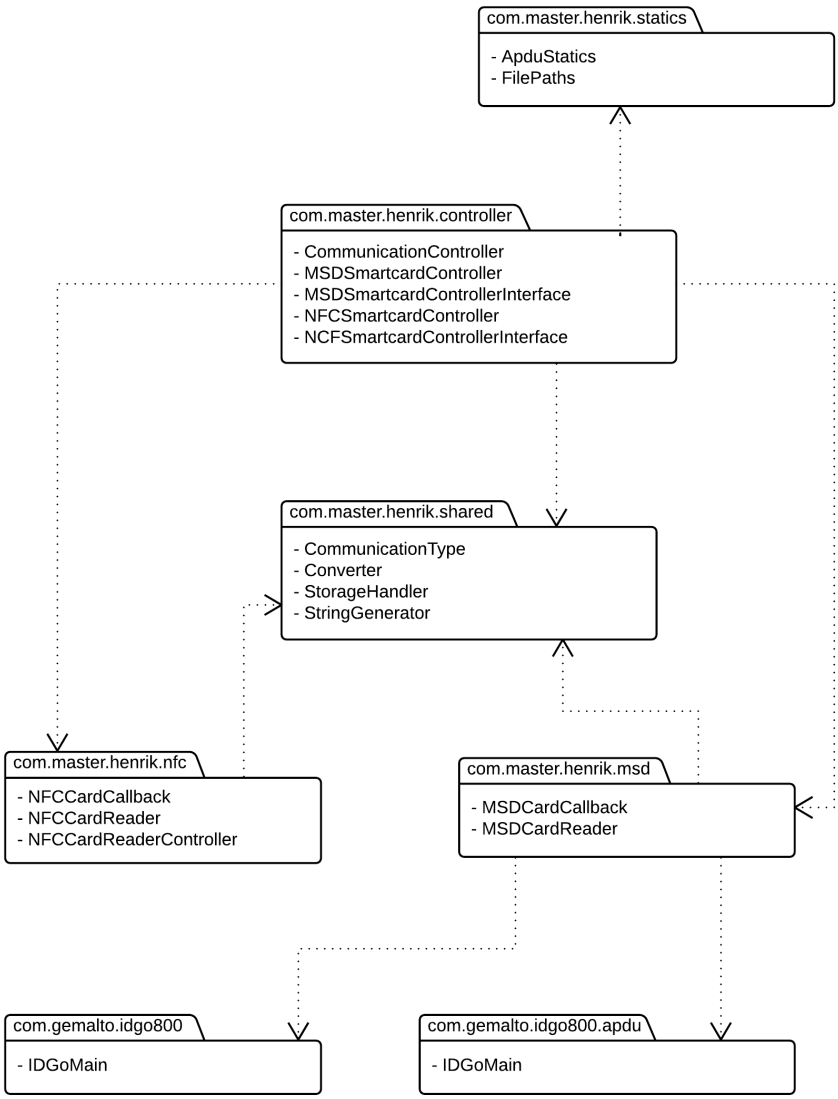

Figure 5.3: Library package diagram.

Chapter 6

Test Cases

In this chapter we will perform tests on the smart card regarding cryptographic performance and proof of concept implementation to evaluate the proposed solutions from the previous chapter. The tests will help us determine if smart cards are a valueable addition to mobile security or if they lack the power to function properly in a mobile device environment.

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 [36].

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 2.3 produced by Gemalto. The reasoning 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 plain hybrid smart card (ref. figure 2.1) with no pre-installed software which is also provided by Gemalto 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 the two card types to each other.

6.1.2 Limitations

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.

6.2 Data Transfer Speed

6.2.1 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 are incaple 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, initialize a buffer, 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.

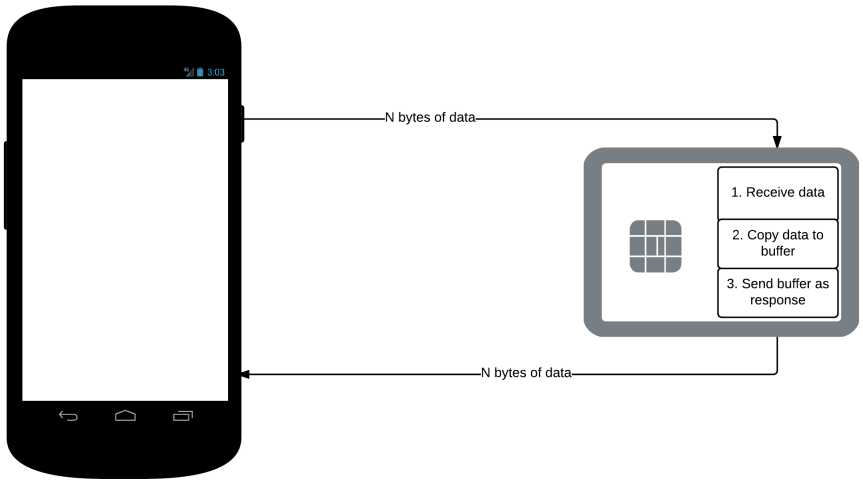

Figure 6.1: Data flow of data transfer speed test for NFC.

6.2.2 Tests and Results

NFC

T1 configuration consists of the Android application sending 255 byte of data to the smart card application, receive the 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).

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.

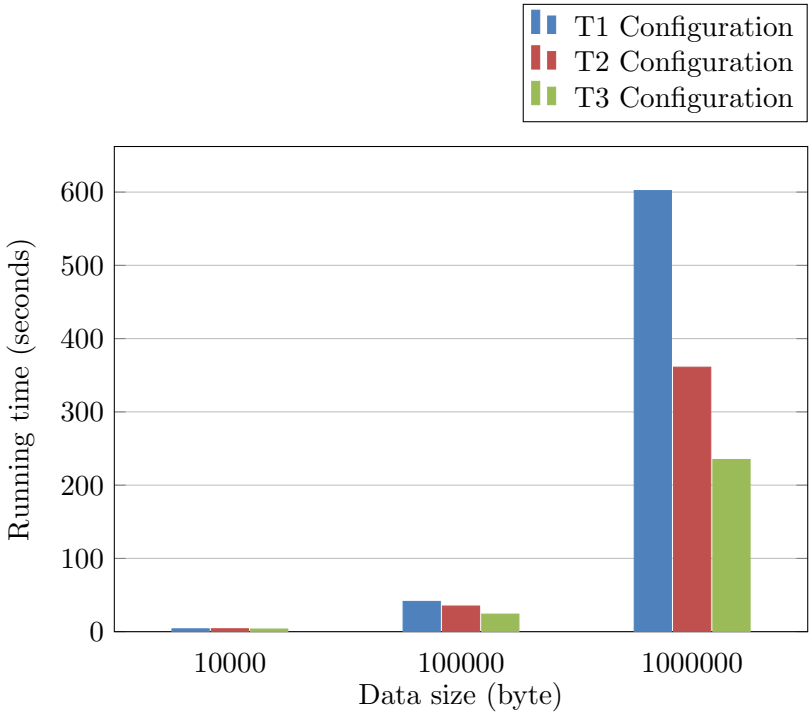

Figure 6.2: Graphical representation of table 6.1.

Micro SD
T1 configuration, from NFC was dropped when testing transfers speed for micro SD as it became clear from previous tests on NFC that T1 is vastly inferior to T2 and T3. A non-asyncronous (not using streams) Android application is not representative of the "real-world" and we decided on not pursuing further test results using this configuration.

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).

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.

6.2.3 Conclusion

From table 6.1 and figure 6.3 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 there are virtually no difference between the configurations; T1, T2 and T3. The differences are more prominent when the data amounts increases. 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.

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.

6.3 Symmetric-key Cryptography

6.3.1 Description and Motivation

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 running time tests with different amounts of data in order to determine the performance of the smart card.

6.3.2 Test setup

The framework we are using is designed around extended APDU and the test platform we have designed in javacard 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 javacard 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 javacard documentation for `Cipher` supports more algorithms [23]. Others have encountered the same discrepancy [33] 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.

6.3.3 Results

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.

6.3.4 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 cordinates. GPS cordinates can be represented by only 16

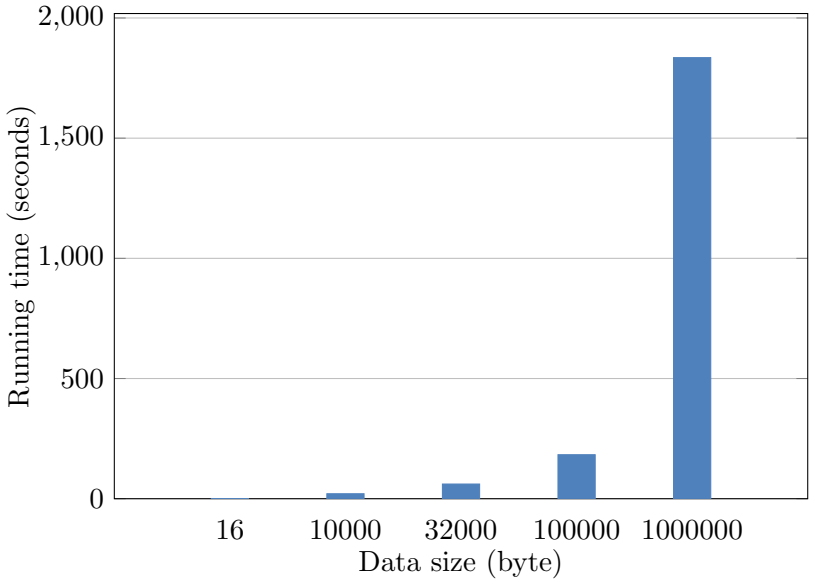

Figure 6.3: Graphical representation of table 6.3.

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.

6.4 Binding card and mobile device

6.4.1 Motivation

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.

6.4.2 Implementation

In light of the parts of the binding process we want to test the outline of the implementation will be:

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 javacard framework [10]. 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 the public key of 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.

6.4.3 Tests and results

Configuration 1
This configuration 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 public key
- 2048-bit server public key
- 128-bit AES key

Configuration 2
This configuration 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 public key
- 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 set is 100 tests.

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

Table 6.4: 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).

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

Table 6.5: Time required to generate the verification package on the smart card application.

6.4.4 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 [45].

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 dependendent 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 hardcoded, and as a results is very rigid to change. For instance if the specification of the verification package change it will require excessive work to reflect the changes. One will have to decide make a desicion wether or not this will happen often enough to counter the potential benefits of using smart cards.

6.4.5 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 feasable and that the reward, that smart card and mobile device is locked to each other, greatly outscales the costs involved.

Chapter 7

Conclusion

7.1 Research questions

- *"What are the limitiations of smart cards in the context of hardware?"*
  There 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). 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 limitiations of smart cards applications?"*
  We opted for using JavaCard as our programming language. Our version of JavaCard 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. JavaCard does not support standard garbage collection and thus applications needs to be extra careful when allocating memory.
- *"What are the types of use cases we are able to solve and strengthen the security of using smart cards?"*
  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 aswell 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 areas that are already solved. In this thesis we described and analyzed a solution where we used smart cards as a basis for policy enforcement. Our comprehension is that this type of solution in conjuction with traditional policy enforcement systems will take security to a new level.

7.2 Experience

After working with smart cards for roughly one year we have made quite a few experiences that are worth sharing.

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, i.e., add a line of code returning the value of variables and try 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 [18].

This type of debugging environment is exhausting and is 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 noteable with errors regarding memory usage. One of the best advices concerning smart card debugging is: "Test often with a big array of test data.".

JavaCard documentation

The documentation available is very technical. This is not by any means a bad thing, but it does require developers to understand smart cards fully before using the documentation. When comparing Android and JavaCard 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 JavaCard 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 JavaCard 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 6.3).

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 computer, 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 tideous to spend 1,5 minutes switching around the mSD card and waiting for the install script.

7.3 Future work

The research we have presented in this thesis are 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 for micro SD card communication [32, *SEEK for Android*].

Our evaluations of the proposed solutions are based on protocol analysis and proof of concept. It would be beneficial 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.

Appendix A

JavaCard 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)
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_A
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 wo
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 l
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
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, bytesRe
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, bytesRe
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, bytesRe
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)
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, (sho
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) (AESmPu
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) (output
291                     outputSize += tempLength;
292 
293                     //Signing
294                     short signatureSize = sig.sign(packet, (short) 0, totalsize, h0Buffer,
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,
300                     Util.arrayCopyNonAtomic(packet, (short) 0, h0Unencrypted, signatureSize
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, byte
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, by
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
422     private void send(APDU apdu) {
423         apdu.setOutgoing();
424         apdu.setOutgoingLength(size);
425         apdu.sendBytesLong(output, (short) 0, size);
426     }
427 }
```

Appendix B

Android Library

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 implement
25      * @param currentActivity
26      */
```

```
27      public void setupNFCController(NFCSmartcardControllerInterface nfcSmartcardControlle
28          if(nfcscc == null) {
29              nfcscc = new NFCSmartcardController(nfcSmartcardControllerInterface, current
30          }
31          type = CommunicationType.NFC;
32      }
33
34      public void initNFCCommunication(String cardAID, String INS, String P1, String P2, S
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 implement
46       * @param currentActivity
47       */
48      public void setupmSDController(MSDSmartcardControllerInterface msdSmartcardControlle
49          if(msdscc == null) {
50              msdscc = new MSDSmartcardController(msdSmartcardControllerInterface, current
51          }
52          type = CommunicationType.MSD;
53      }
54
55      public void initmSDCommunication(String cardAID, String INS, String P1, String P2, S
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, publicByte
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, Strir
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, Strir
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

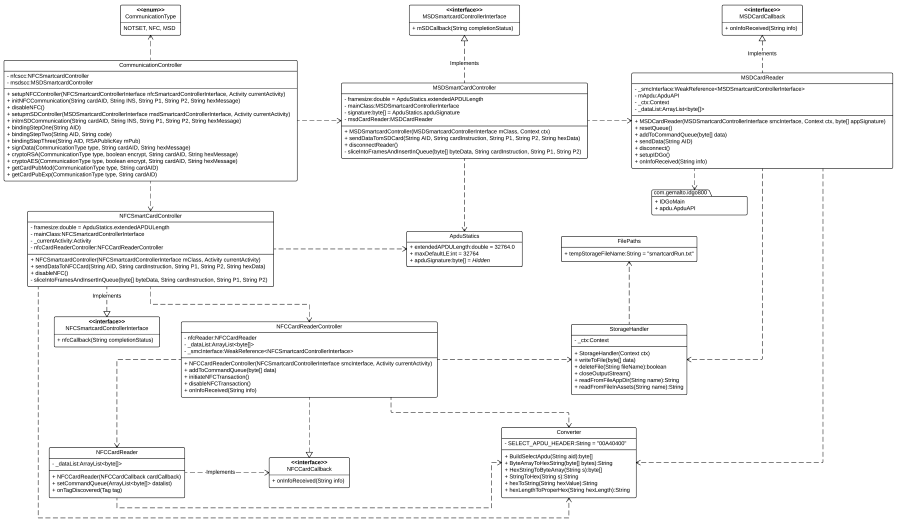

Figure C.1: Class diagram for Android Library.

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

[7] *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.

[8] *Benefits of TLS, OWASP*. Last visited: 03.02.2016. 2016. URL: https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet#Benefits.

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

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

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

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

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

[14] *IDCore 8030 MicroSD card, Datasheet*. Last visited: 16.02.2016. 2015. URL: http://www.gemalto.com/products/top_javacard/download/IDCore8030_Datasheet.pdf.

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

[16] GlobalPlatform inc. *The Trusted Execution Environment*. Last visited: 24.04.2016. 2011. URL: http://www.globalplatform.org/documents/GlobalPlatform_TEE_White_Paper_Feb2011.pdf.

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

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

[19] *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.

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

[21] *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.

[22] *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.

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

[24] *MULTOS Smart Card Operating System, cardwerk.com*. Last visited: 24.02.2016. 2015. URL: http://www.cardwerk.com/smartcards/MULTOS/.

[25] *Orgacle and Sun Microsystems*. Last visited: 05.03.2016. 2016. URL: https://www.oracle.com/sun/index.html?PHPSESSID=8fd76773d26875481e097a4a27c7a6a1

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

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

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

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

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

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

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

[33] *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.

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

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

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

[37] *Tech-FAQ, Known Plaintext Attack*. Last visited: 24.02.2016. 2015. URL: http://www.tech-faq.com/known-plaintext-attack.html.

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

[39] *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.

[40] Erik Poll Tim Cooijmans Joeri de Ruiter. *Analysis of Secure Key Storage Solutions on Android*. Last visited: 26.04.2016. 2014. URL: https://www.cs.ru.nl/E.Poll/papers/AndroidSecureStorage.pdf.

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

[42] *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.

[43] *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.

[44] *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.

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