Bachelor thesis

Implementation and Analysis of OpenPGP Functionality via NFC

Author:
Philipp Jakubeit
0814881
p.jakubeit@student.ru.nl

Supervisor:
Erik Poll
Pim Vullers
Joeri de Ruiter

July 13, 2012

This thesis explains how users of smartphones can send and receive secure e-mails. Securing e-mails is done with Pretty Good Privacy (PGP) a standard for signing and encrypting e-mails. To sign an e-mail, a private key is needed. This private key gets stored on a smartcard, the OpenPGP Java Card, which can be connected to the smartphone using near field communication (NFC). The process of signing is done by touching the smartphone with the smartcard. The results of this research are an implementation of PGP functionality using NFC on a smartphone and a detailed comparison of usability, security and performance with an existing on-phone implementation of PGP. It was possible to create an implementation and to show that the smartcard option is more secure and faster than the on-phone option. The only category in which it does not rank first is usability.

Contents

1 Introduction **1**

2 Background **3**
2.1 NFC . . . 3
2.2 OpenPGP . . . 3
2.3 APG . . . 4
2.4 Android Program . . . 4
2.4.1 Programming Concept . . . 4
2.4.2 Program Structure . . . 5

3 Implementation of NFC functionality **6**
3.1 Design Considerations . . . 7
3.2 Prototype . . . 8
3.3 NPG . . . 9

4 Comparison **12**
4.1 Usability . . . 12
4.2 Security . . . 13
4.3 Performance . . . 15
4.3.1 Time Complexity . . . 15
4.3.2 Time experience . . . 15
4.4 Overall comparison . . . 16

5 Future Work **17**

6 Conclusion and reflection **19**
6.1 Conclusion . . . 19
6.2 Reflection . . . 20

References **21**

A Appendix **22**
A.1 Survey . . . 22
A.1.1 APG . . . 22
A.1.2 NPG . . . 22

B Appendix **23**
B.1 Prototype Code . . . 23
B.1.1 Source . . . 23
B.1.2 Values . . . 25
B.1.3 Layout . . . 25
B.1.4 Manifest . . . 26
B.2 NPG Code . . . 26
B.2.1 Source . . . 26
B.2.2 Values . . . 38
B.2.3 Layout . . . 39
B.2.4 Manifest . . . 39

Abbreviations

APDU	-Application Protocol Data Unit
APG	-Android Privacy Guard
API	-Application Programming Interface
GPG	-GNU Privacy Guard
GNU	-GNU is Not UNIX
GUI	-Graphical User Interface
IBE	-Identity Based Encryption
IETF	-Internet Engineering Task Force
ISO	-International Organization for Standardization
NFC	-Near Field Communication
OS	-Operating System
PC	-Personal Computer
PGP	-Pretty Good Privacy
PKI	-Public Key Infrastructure
RFC	-Request For Comments
RFID	-Radio Frequency Identification
S/MIME	-Secure/Multipurpose Internet Mail Extensions
SD	-Secure Digital
SDK	-Software Development Kit
VM	-Virtual machine

1 Introduction

The majority of Internet users send e-mails in plain text, but there are always threats regarding the confidentiality and the authenticity of an e-mail. These threats can be reduced with the help of e-mail encryption and signatures.

Different algorithms exist to provide these functionalities. In this thesis, Pretty Good Privacy (PGP) will be examined. PGP is the standard solution for securing e-mail. It supports encryption as well as digital signatures.

An important aspect of PGP, as for all asymmetric cryptographic algorithms, is that the private key remains secret. Personal computers (PCs), laptops and smartphones all have an equal threat of malware. Data can be stolen relatively easily from them.

When it comes to the threat of physical theft, it is a different situation. PCs and Laptops have a lower risk than smartphones.

As a smartphone user, I would like to combine both, so that I can use PGP functionalities on my phone. There are already some applications on the market which allow these functionalities. But what about the safety of the private key? It would be useful to have the private key stored somewhere else than on the smartphone, in case of hardware loss. One possibility is an external device which is able to store information such as PGP keys. The OpenPGP card is such a device. The OpenPGP card is an implementation for smartcards which allows PGP functionality. In the case of hardware loss, it could be of great assistance, because the private key is not stored on the device. The OpenPGP card also has an additional advantage with regards to Internet threats, because the private key is inaccessible from the Internet if the card is not connected to the smartphone. When connected, the connection only lasts for a few seconds and is therefore unlikely to be interrupted. Even if OpenPGP card-specific malware exists, it would be impossible to export the private key, simply because the OpenPGP card design does not allow the private key to be read.

An additional benefit could be the hypothesis, that the speed is improved, because of the improved speed of security algorithms in smartcards (such as the OpenPGP Java Card), owing to the hardware implementation of cryptographic algorithms. Smartphones do not have optimization for such algorithms and could therefore be slower in computation.

To provide an example, an app has been developed which runs under Android and is based on Android Privacy Guard (APG), an on-phone solution for PGP functionality. This new app will be compared with APG later.

Since February 2006, a new technology has been implemented into a rising number of phones (Gsmarena, 2012). This technology is near field communication (NFC) and is a standard for contactless communication. It combines existing standards such as the transmission protocol specified by the international organization for standardization (ISO) 14443 (NFC Forum, 2012). This transmission protocol is a standard for communication with contactless smartcards, commonly known as radio frequency identification (RFID).

The OpenPGP card is an OpenPGP standard for smartcards. It supports smartcards that use the ISO 14443 technology (Pietig, 2004), such as the Java Card, where an OpenPGP card implementation exists (de Ruiter, 2012). Therefore an NFC-capable mobile phone can communicate with an OpenPGP card which supports such contactless communication.

The described facts and this hypothesis lead to the following research question:

How can the OpenPGP card be used in combination with NFC?

The research question can be divided into three sub-questions.

- Which changes in the on-phone option are necessary for the NFC extension?
- How difficult is it to adapt an existing app that uses OpenPGP, to use an OpenPGP card?

- Which advantages and disadvantages does the NFC option have compared to the on-phone option, in terms of usability, performance and security?

Different target groups will benefit from this research. Some researchers will have the opportunity to research in this direction.

The developers of APG as well as OpenPGP card distributors will both benefit from this research, because it should generate more interest in their products. Interest will be generated in the OpenPGP card and its implementations. This research will contribute to already existing APG development work and, in addition, will possibly generate more downloads of the APG software.

The user will benefit from this research, because he will have the possibility to turn his wallet (with an OpenPGP Java Card in it) into a PGP-key-safe. This will allow him to split the communication device (smartphone) from the key storage (smartcard). Furthermore, the user will be able to use his NFC-enabled device for a wider variety of activities.

As a result of the growing number of credit card manufacturers distributing NFC devices (see Section 2.1), it is expected that nearly every consumer will be able to afford an NFC-capable device in the near future (Burwell, 2011). The development of micro secure digital (MicroSD) cards which enable NFC functionality developed from DeviceFidelity (2012) or Netcom (2012) also support this idea. Thus there are many people, like myself, who will be affected by this change in technology. We will all have the possibility to use this technology for e-mail encryption while keeping our key private. Both users who who already use PGP and all Internet users in general, could benefit from this research, as most e-mails are sent without encryption or signature. Even though the algorithms have already existed a long time, some new technology and a different usage could perhaps motivate some new users to encrypt and sign their e-mails.

Users who do not trust the security of their smartphone could also use the results of this research to send PGP-encrypted e-mails from their phone. A lack of trust in the security of a smartphone is mainly concerned with the online security threat. Using an OpenPGP smartcard removes the risk of the private key being stolen via the Internet, simply by virtue of the smartcard not being permanently connected to the Internet and not allowing the private key to be read.

Users who already use PGP encryption on their smartphones could also benefit. This research includes a performance comparison between APG and an app which uses the OpenPGP card. The result of this comparison could prove that the OpenPGP card outperforms the smartphone in signing and decryption. This would mean that it would not only be more secure, but also quicker, to use an external key-storage (smartcard).

This paper will be divided into the following sections:

Section 2, Background- gives a background to, and some basic knowledge about, NFC, OpenPGP, APG and Android in general.

Section 3, Implementation of NFC functionality- gives an implementation overview.

Section 4, Comparison- shows the results of the comparison between APG and the app using the smartcard.

Section 5, Future Work- summarizes directions for future research.

Section 6, Conclusion and Reflection- contains an analysis of the results and a reflection on the research.

2 Background

To allow the reader to better follow the rest of this paper, some terms and functionalities are described are described in detail within this section. This section explains the NFC technology, specifically the NFC interface of smartphones and the PGP option for secure e-mails. Furthermore it contains information about APG, which will later be adapted for use with an OpenPGP card. This section ends with a brief overview about some Android-specific features and concepts which are relevant for the implementation.

2.1 NFC

As mentioned in Section 1, NFC is a technology which allows contactless communication with other devices according to several standards, for example ISO 14443 (NFC Forum, 2012). NFC operates on a frequency of 13.56 MHz with a bit rate of 424 kbit/s and a range of less than 0.1 meter (Ortiz Jr, 2006). This makes it slower than comparable technologies such as Bluetooth. NFC only works at closer ranges but the frequency allows it to communicate with RFID supporting hardware, which is a standard for contactless cards and tags. At the moment there is a huge motivation for NFC development the area of contactless payments. Credit card manufacturers such as MasterCard, Visa, American Express, and Discover (Burwell, 2011) support this technology because it represents a profitable opportunity for them. This support, in turn, benefits the phone manufacturers.

2.2 OpenPGP

PGP, secure/multipurpose internet mail extensions (S/MIME) and identity based encryption (IBE) are the most common algorithms for e-mail encryption. Gerck (2012) compared public key infrastructure (PKI), PGP and IBE. His conclusion is that IBE has the lowest security score. In a comparison between PGP and S/MIME, Baranyi (2012) states that S/MIME does not offer a public source code and strong encryption worldwide compared to PGP, which does. PGP supports 4096 bits whereas S/MIME supports only 1024 bits key length in the US version and even less in the export version. These facts give a plausible explanation as to why the decision was made to use PGP.

PGP and its derivatives are frequently used to allow e-mail encryption and signing of e-mails for secure and authenticated communication over the Internet. It was originally developed by Zimmermann and published in 1991 (Zimmermann, 2012). After some legal conflicts the OpenPGP Alliance was founded in 2001 (OpenPGP Alliance , 2012). OpenPGP is the non-proprietary standard version of the licensed PGP version from Network Associates Incorporation. Additionally there is a completely free implementation, GNU Privacy Guard (GPG). All implementations are developed according to the Internet engineering task force (IETF) standard request for comments (RFC) 4880.

PGP is a combination of symmetric and asymmetric encryption paired with hashing. **Symmetric** encryption is based on one secret key which encrypts a message and must be shared between the correspondents. **Asymmetric** encryption uses a private/public key pair to allow encryption. The public key in this scenario can be known by anyone, or should even be known by as many as possible to guarantee its authenticity. Mathematical rules which were applied during the creation of such a pair ensure that messages which are encrypted by the public key could be decrypted by the private key. **Hashing** describes the functionality of reducing or extending an input of variable length to a fixed-sized output, in such a way that the operation of transformation is complex to inverse.

PGP is used for confidentiality and authenticity. Confidentiality consists of the ability to encrypt and to decrypt a message. Authenticity is provided by PGP because of its ability of signing and signature verification. In the case of signing, a hash from the e-mail is generated

which is signed by the private key from the sender. After that, everyone can verify this signature with the senders public key. If the message is hashed again both results can be compared and the authenticity of the e-mail can be determined.

In the case of encryption the e-mail is optionally signed first, then the e-mail is encrypted with a symmetric key, which is generated specifically for this purpose. This symmetric key gets encrypted by the public asymmetric key from the recipient and appended to the e-mail.

If the recipient wants to decrypt the message he has to decrypt the symmetric key with his private asymmetric key before he can use this symmetric key to decrypt the message. Afterwards he can check the appended signature and the e-mail is available for him, signed and in plain text.

2.3 APG

APG is an Android App developed by Thialfihar (APG source, 2012). It is an PGP implementation for Android, which stands out amongst other PGP implementations for Android by virtue of being open source. There are three implementations of PGP for Android on the Android Market (2012). The first, OpenPGP Manager, is not open source. The second, Android PGP, seems to be unsupported. Therefore the choice was APG. Another advantage of APG is its K9 mail integration. K9 is a standard e-mail application for Android phones (Vincent, 2012). It allows the user to use PGP functionalities but requires third-party implementations (e.g. APG) to handle these *intents* (Android specific requests which are described in Section 2.4.1). APG catches these *intents* and performs the requested operation (this *intent* flow can be seen in Figure 1). Furthermore it offers the user the possibility to manage his keys and keyring.

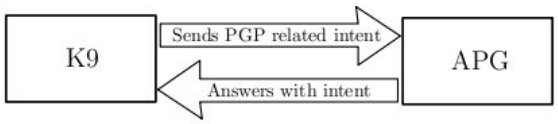

Figure 1: Intent Processing

2.4 Android Program

Android is an operating system (OS) developed by Google. It is designed for smartphones and tablets. It is in fact a Dalvik virtual machine (VM) which runs on a Linux kernel. Android apps are written in Java and are then compiled into byte code and converted to Dalvik compatible files. The big difference is that the Java VM is a stack machine whereas the Dalvik VM is a register machine. This leads to a different programming concept and a different structure. (Google Inc., 2012)

2.4.1 Programming Concept

The concept of Android programming can be described as *activity* programming. Activities can be understood as what the user sees on the display. Therefore the graphical user interface (GUI) is required for programs, except for the so-called 'Services' a sort of daemon processes for Android. A standard application is programmed through input and output interaction with such an *activity*. Therefore the user gets from *activity* to *activity* while using Android. Because everything which is shown to the user is an *activity*, it is easy to navigate backwards through every step. Communication between these *activities* can be implemented as *intent*. Hardware requests are always implemented as *intents*. In this way, every program which listen to a specific *intent* can be chosen by the user. (Mednieks et al., 2011)

2.4.2 Program Structure

The structure of an Android program is different compared to other programming environments. Because of the ability to listen for specific *intents* each program has to have a so called manifest file. This file encodes some basic app information, the *intents* the program listens for and the permissions of the app in XML. The permissions are needed as security management. The user has to decide during installation whether he grants the app the requested permissions or not. The GUI is also specified also within an XML file as well as the values needed for the GUI such as strings, colors or other value information. A source code file which contains the logic of the program also exists. It is helpful to keep this Android specific structures in mind, both in Section 3 of this thesis, and even more so in Appendix B (the program code).

3 Implementation of NFC functionality

The implementation section is split into three sections. The first section outlines the general design considerations, the second explains the creation and use of a prototype and the third describes the final extension of APG. This extension of APG will henceforth be called NPG (**N**FC enabled Android **P**rivacy **G**uard). The prototype is merely a program which is used to learn the behavior of Android and the OpenPGP card. Furthermore it serves as a base for future development.

NPG uses the knowledge acquired through the prototype and combines it with the existing implementation of APG. The result is an extended APG which is able to communicate with an OpenPGP card via the NFC interface.

The first sub question was: *which changes in the on-phone option are necessary for the NFC extension*? Two explanatory diagrams are given to answer this question and help understand the basic concept behind NPG.

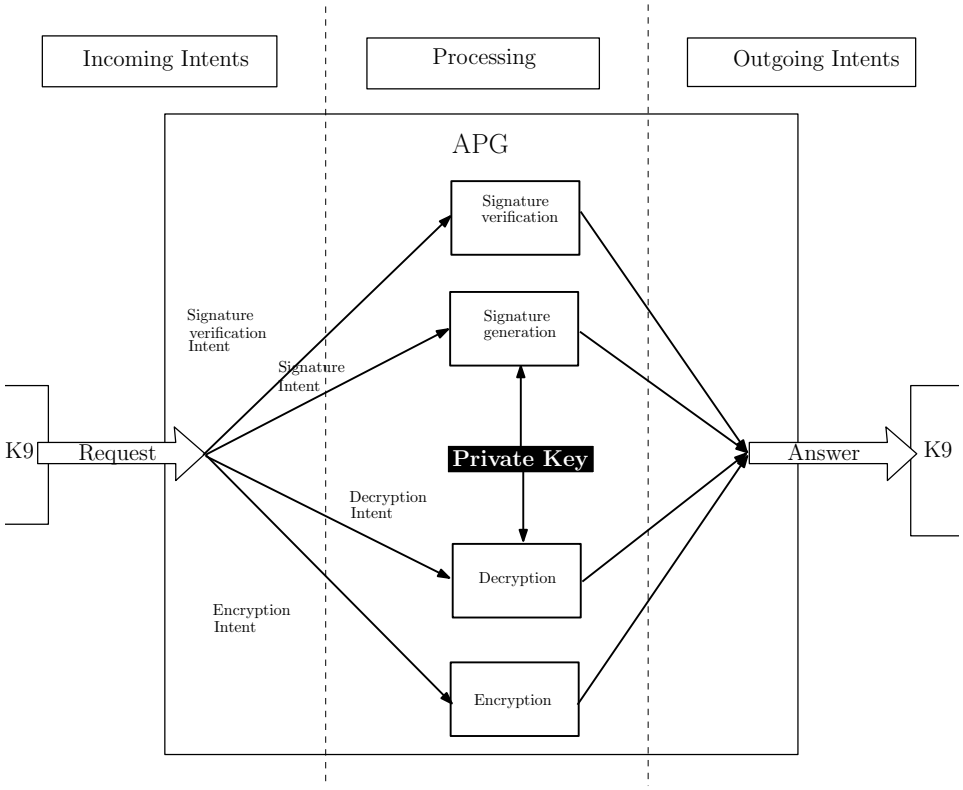

Figure 2: APG Schematic; Incoming and outgoing *intents* are managed by K9 or any other app that can interface with APG.

Figure 2 describes how APG currently works: A request, an incoming *intent*, is sent to APG. This *intent* is send by K9, but could be from any other application which supports the application programming interface (API) of APG. APG is only listening to certain *intents*, namely signature, signature verification, encryption and decryption. Inside of APG the *intent* calls the related methods and delivers its data. The called method computes the requested result and APG returns an answer, an outgoing *intent*, which is sent back to K9.

NPG behaves differently, as shown in Figure 3. The difference is in the call of the methods. NPG also listens to the specific *intents* and processes them if a request occurs, but instead of calling the method within NPG, the user is asked to "Touch the OpenPGP Card" to the

device. This allows a data connection between NPG and the smartcard. The *intents*, signature and decryption, now call methods via the NFC interface on the smartcard. The signature or the decryption is computed on the smartcard and returned via the NFC interface. Then the computed data is returned from NPG as an answer, an outgoing *intent*.

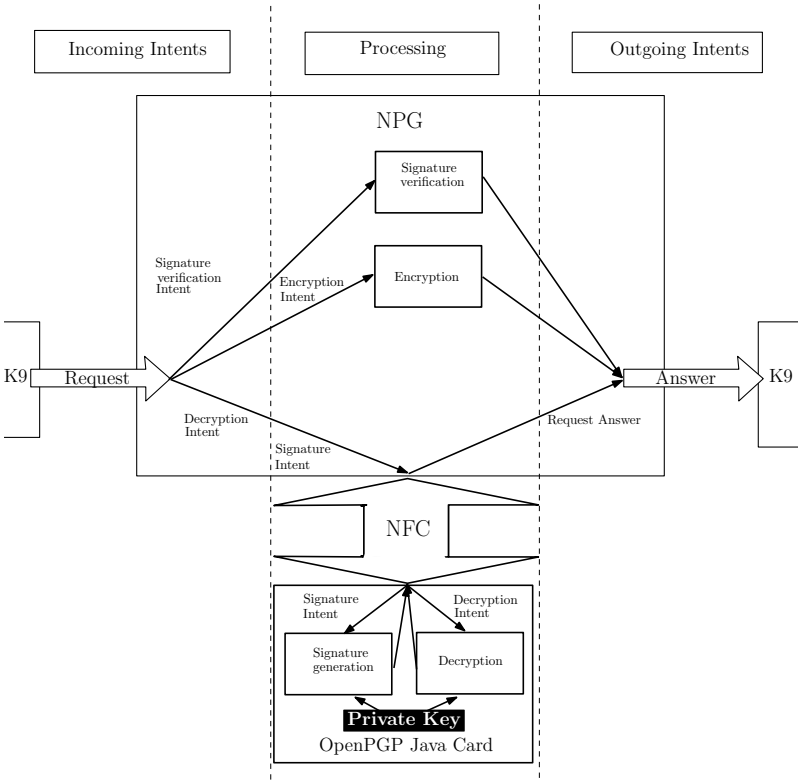

Figure 3: NPG Schematic; Incoming and outgoing *intents* are managed by K9 or any other app that can interface with APG.

A comparison of Figure 2 and Figure 3 shows that NPG tranfers the functionalities of signature generation and decryption to the smartcard. This arises from the decision to move the private key to the smartcard, the main idea behind NPG. The private key is safer on the smartcard, because it cannot be exported.

3.1 Design Considerations

Several global design considerations had to be taken.

- Three different scenarios were thought of:
  1. The first scenario involves the smartphone communicating with the smartcard. The user is asked for a password before he has to bring the smartcard in range.
  2. The second scenario is that the smartphone stores the password and the user only has to bring the smartcard in range.
  3. The third scenario is that the smartcard is not enough for verification. An ID card is used to authenticate the user, which in this case is required for communication with the smartcard.
- K9 was left untouched. The previously mentioned *intent* design pattern from Android software means that it is possible to just change the PGP performing program, APG.

- The interfaces for the user, and thus the *activities*, are mostly the same as in APG. Only some features were added. These consist of a new *activity*, (see Section 3.3), and an added menu entry, which allows global activation of NFC. This has the advantage that the process of choosing a private key and signing the e-mail is almost the same for NPG as for APG. The only difference is that the card has to be used instead of a password. A disadvantage is that the user has to decide beforehand how to sign his mails. Nevertheless this consideration was taken, because the user stays in his usual environment and can even change between K9 and NPG if he needs to switch NFC on or off.

For my implementation one of these scenarios had to be chosen. To make it as user friendly as possible the second scenario has been chosen. The fact that the smartphone knows the password for the smartcard is implemented in NPG.

3.2 Prototype

The prototype is a simple application that allows interaction between an Android smartphone and the OpenPGP card. The prototype was designed in order to gain a better understanding of how NFC and the OpenPGP Java Card work. This can be seen in the schematic representation in Figure 4. The prototype was developed primarily for two reasons. Firstly, it helped to give a better understand of how an Android program can be built. Secondly it shows how an Android program can communicate with the NFC device within the smartphone. This process allows a communication with the OpenPGP Java Card. Therefore all hardware aspects of this research were covered by this prototype. It helped to get to know the different environments.

The complete code, structured according to the Android specific requirements, can be found in Appendix B. The source code from the app "Sticky Note" (The Android Open Source Project, 2012) was used as a base, but it was changed almost completely. The prototype is a program which listens for an NFC tag if the user presses a button. If it recognizes a tag, it reads the tag and tries to connect to it, expecting that the tag is an OpenPGP card. If the tag responds differently, a request is printed to read an OpenPGP card. If the tag responds correctly, the name and public keys are read. After that, the card has to sign the hash of the word "NFC", namely signature(hash("NFC")). Finally, all this information, that is, the user's name, the three public keys (three because the OpenPGP card has a key pair for signature, confidentiality and authorization) and the calculated signature, is printed. An additional benefit is the reusability of the code for the implementation of NPG.

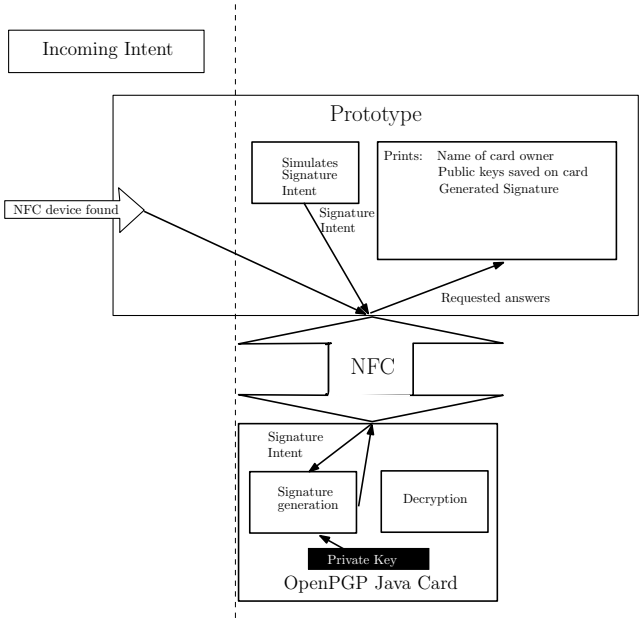

Figure 4: Prototype schematic

3.3 NPG

This section describes some implementation details and answers the second sub-question: *how difficult is it to adapt an existing app that uses OpenPGP, to use an OpenPGP card*?

NFC is activated using a global approach (as described in Section 3.1). The beginning of the implementation involved a new checkbox entry with the label NFC. This allows the user to set the status of NFC and can be seen in Figure 5.

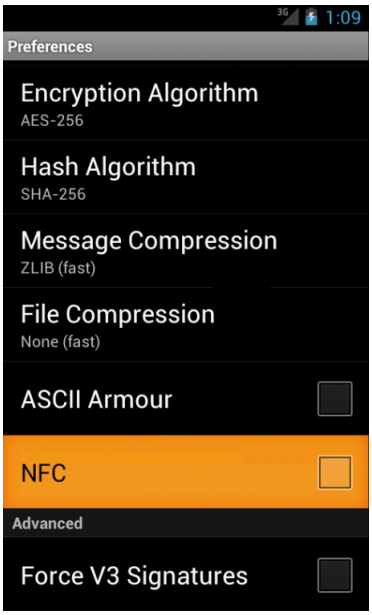

Figure 5: Menu entry; NFC and checkbox

The user sees a new *activity* (see in Figure 6) if he tries to sign an e-mail in K9 and if NFC

is enabled in the NPG menu. The button "NFC" has to be pressed to allow the smartphone to search for an OpenPGP card. This results in a message being shown to the user (Figure 7). If another tag is recognized, an error is prompted. Otherwise, if the card is recognized and stays connected for long enough, the mail is sent.

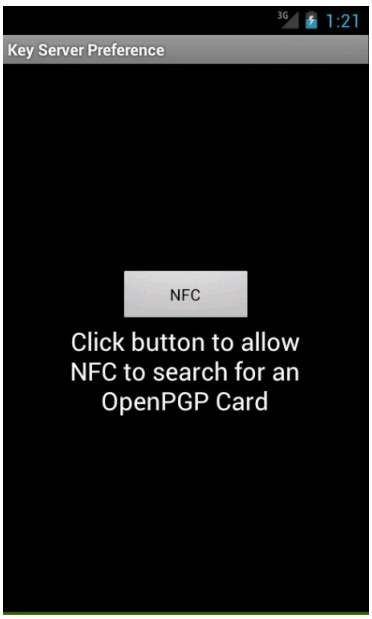

Figure 6: New *activity* with NFC button

It took some time to pinpoint where exactly APG needed to be modified. The manifest file was used as a starting point. This leads the way to the EncryptActivity class, which in turn contains the code required for signing and encryption. Because EncryptActivity often calls the APG class I began to look into it. Unfortunately, this class contains a lot of uncommented methods and functionalities, but I marked all occurrences of the private key. Looking back, it was a necessary task, as it helped me to familiarize myself with the code. However, in some ways it was also a redundant exercise, because I never changed anything in the APG class. I continued to implement functionalities to the EncryptActivity class, but ended with exceptions or program crashes. This motivated me to build my own NFC class which uses parts of the prototype to communicate with the smartcard. After some trouble with parsing values between different *activities*, as a result of my new class being a new *activity*, it turned out to be a good solution.

Figure 7: Message to indicate that the OpenPGP card needs to be touched to the smartphone

With the NPG implementation working correctly, it was time to get the PGP signature, in accordance with RFC 4880. This proved to take longer than expected, so I decided to reuse some appropriate code from the APG class. I also reused and adapted some code from the Spongy Castle classes which were used in APG to manage PGP functionalities (Spongy Castle is an Android equivalent of Bouncy Castle for Java, which is a library containing security related functionalities). These adapted classes have also been added to Appendix B. Some changes had to be made, as a result of the outsourcing of signing to the card, instead of letting Spongy Castle do the calculations. Nonetheless, it was very helpful to use preexisting implementations, which work in accordance with the specifications in RFC 4880, espacially after the number of unsuccessful trials.

4 Comparison

To get an overview of the newly implemented NPG's quality, this section focuses on a comparison between NPG and APG. This answers the third sub-question: *which advantages and disadvantages does the NFC option have compared to the on-phone option, in terms of usability, performance and security?*

This section will be split into three sub-sections, according to the three different comparison criteria: usability, performance and security. A general discussion follows.

4.1 Usability

Participants for the research were split into two groups. One group used APG, the other used NPG. Due to the short time frame and motivation of the participants, it was not possible to compare all three scenarios. Therefore, only one scenario was used. For the useability comparison scenario 2 was chosen. This allowed the participants to do one task. The participants using APG had to type in a password for verification. In the case of NPG, the participants had to bring the smartcard in range.

A sample of students from the Radboud University was picked for the usability test. 50 students were asked to send a signed e-mail from the test phone. In the interest of impartiality, a coin toss was used to decide which participant would be in which group. Those who tossed heads were placed in the NPG group; those who tossed tails in the APG group. The participants then had to send an e-mail and fill out a short survey, consisting of three questions. The questions were as follows (Q3-APG applies only for those in the APG group; Q3-NPG applies only to those in the NPG group):

Q1 Was it difficult to send the signed mail?

Q2 Would you use this sort of e-mail signing on your own mobile?

Q3-APG Do you think it would be easier to hold a card against the mobile instead of typing in the password?

Q3-NPG Do you think it would be easier to use a password in place of the card?

Each question had to be answered with either 'yes' or 'no'. Both surveys can be found in Appendix A.

From the 50 participants, 24 were in the APG group and 26 were in the NPG group. The summed results of the participants who used APG are the following: 0% thought that it was difficult to send a mail with APG. 87.5% would use APG on their own device and 50% would prefer a card.

The summed results for NPG participants are: 7.6% thought that it was difficult to send a mail with NPG. 76.9% would use PGP with smartcard on their own device and 50% would prefer a password.

These results show that more participants have difficulties with the NPG option. It also seems more troublesome for the participants to use a smartcard instead of a password. With regards to the question of whether a different option is preferable compared to those which were tested, both groups disagreed. This is summarized in Table 4.1:

	APG	NPG
Q1	✓	✗
Q2	✓	✗
Q3	✓	✓

Table 1: Usability results

It should also be mentioned that suggestions were made by two participants. One suggested that it would be more safe to type a password and use the card (scenario 1). The other suggested that additional verification of his identity with his ID card would be a good idea (scenario 3). Therefore it would be interesting to make a usability comparison of the other two scenarios.

4.2 Security

The vulnerability of the different versions and possible threats were taken into account when considering security. The risk identification was done by estimating the severity and detectability of security breaches, as well as the probability of this happening. The three different implementation scenarios of NPG were compared with APG for each of the above categories. The security score comprises the sum of these three categories. Each of the categories was scored on a scale from 1 to 5, explained below:

Probability: *1 (likely) - 5 (unlikely)*

Severity: *1 (serious) - 5 (unproblematic)*

Detectability: *1 (difficult) - 5 (simple)*

Thus 15 is the maximum score as well as the optimal security score. Several threats were taken into account:

- The online hack, meaning the loss of the private key via malware.
- Third Party, meaning the physical use of the hardware by a third party, whilst the owner is aware of it.
- Hardware loss, meaning the loss of the equipment.
- Hardware theft, meaning the theft of the equipment.
- Eavesdropping, describing the problem of unnoticed interception in the NFC connection. (Unfair however, as APG does not suffer from this threat)

The following table shows the scores for each category and the final security score. A is used as an abbreviation for APG. S1, S2 and S3 represent the three NPG scenarios described in Section 3.1:

Threat \ Version	Probability				Severity				Detectability				Security Score			
	A	S1	S2	S3	A	S1	S2	S3	A	S1	S2	S3	A	S1	S2	S3
Online Hack	1	5	5	5	1	5	5	5	2	2	2	2	4	12	12	12
Third Party	2	4	3	5	1	1	1	1	1	2	2	2	4	6	7	8
Hardware loss	2	4	4	5	2	2	2	2	3	4	4	5	9	10	10	10
Hardware theft	2	4	4	5	1	1	1	1	3	4	4	5	8	9	9	9
Eavesdropping	5	2	2	4	5	2	2	2	5	1	1	1	15	5	5	7

Total security score	4	5	5	7
Security score without eavesdropping	4	6	7	8

Table 2: Security Scores

The total security score is the lowest security score gained over the five different threats. The threat with the lowest security score is the "weak spot" of each individual version. These scores can be explained as follows:

- The probability of an **online hack** is more likely in the case of APG, because the private key is stored on the smartphone, which is accessible online. In all other scenarios the threat is unlikely, because the private key cannot be extracted from the smartcard whilst online. The same argument holds for the severity of an online hack and teh detection of an onlione threat is still difficult.
- The probability of a **third party** using the smartphone and stealing the private key is more likely for the scenario of APG because the private key is stored within the smartphone. For this reason, it is very unlikely that a third party will be able to steal the private key if a smartcard is used. In scenario 1 it is unlikely, because the third party needs to know the password. It is even more unlikely that a third party will be able to gain access to the smartcard and steal additional identification such as the ID card (scenario 3). The severity of this problem is high for all scenarios, if it occurs. The detectability is difficult in the case of APG, because there is no visual clue, like using the smartcard. In all other cases the detectability is less difficult, because the third party has to use at least one external card.
- The cases of **hardware loss and theft** are often similar. It is more likely that the required hardware is lost or stolen in the case of APG. In all other scenarios, the loss and theft of the required hardware is unlikely, because at least two items have to be stolen - the smartphone and the smartcard. It is even more unlikely in scenario 3, because three items have to be stolen. The severity of the problem is higher for hardware theft than for hardware loss. If all the equipment is lost then the threat is, of course, high, but in case of theft it is more likely that the thief has the intention of obtaining the private key. As far as the detectability is concerned, it is less difficult to detect the loss of three items - the smartphone, the smartcard and the ID card (scenario 3) - then it is to detect the loss of two items (scenarios 1 & 2). It is even more difficult to detect the loss of one item (APG).
- The **eavesdrop** threat to the NFC connection is non-existing in the APG scenario, because there is no NFC connection. It is likely in the scenarios involving only one

NFC connection (scenario 1 & 2). It is less likely in scenario 3, because there are two connections which have to be intercepted. The severity of this threat is high, because an eavesdropping attack could monitor all traffic between the smartphone and the smartcard (e.g. the password). Once again, the threat (and thus the severity of the threat) is non-existing for APG. The same goes for the detectability score for APG. However, the detectability is difficult in all other scenarios, because the attacker has to find a way to capture the connection from a certain distance. This makes it easier for him to stay undetected and means that the user has no idea that sensitive information has been stolen.

The total security scores show that all three NPG scenarios offer a security improvement in comparison with the original APG. Scenarios S1 and S2 score slightly better on security. Scenario 3 (smartcard and ID card) is the only scenario which scores notably better on security. The problem for all NPG scenarios is eavesdropping. All three scenarios score worse for this threat than for any of the other threats. This brings their score closer to that of APG. If the threats are considered in isolation, then it is clear that the only reason APG scores best in eavesdropping is because eavesdropping is a non-existing threat for APG. APG gets the lowest scores for all other threats. Therefore the security scores without eavesdropping are also given and will be used henceforth.

4.3 Performance

The performance was tested in two ways:

1. A comparison of the time complexity of the algorithms
2. A comparison of the time as it is experienced by the user

4.3.1 Time Complexity

The processing speed of PGP on the smartphone (APG) was compared with the processing speed of PGP on the smartcard, in combination with the speed and bandwidth of the NFC connection (NPG). Here, the time was measured in the program itself. Two lines of code measured the time before and after the e-mail was signed. In effect, this means that in the case of APG, the time was counted from the point when the password was entered. The time ceased to be counted once the e-mail was ready to send. In the case of NPG, the time was counted from the point when the smartphone detected the smartcard. The time also ceased to be counted once the mail was ready to send. In both cases, the timings are substracted from each other to calculate the time needed. The timings are only concerned with how long each device takes to compute the signature and append it, in the correct format, to the mail. The time was measured for five different e-mail lengths. The e-mail contained either 5, 26, 260, 520 or 1040 literals.

	APG	NPG
Average time in milliseconds	6.8	4.9

Table 3: Performance results (Hardware)

4.3.2 Time experience

For the comparison of time as experienced by the user, the time was measured by a stopwatch. This stopwatch was operated by a second person in order to guarantee accuracy. The time

was stopped from the point when the "send" button was pressed, until the point when the e-mail was actually sent.

For the APG version an eight-character long password had to be typed in (abc4efg8). It contained two numbers and was eight characters in length in order to imitate a real world situation, where it has been shown that passwords contain seven to eight characters on average(Yan et al., 2004).

The NPG version shows the new *activity* where the user has to press the button (as described in Section 3.3). Then he has to hold the smartcard against the back of the device.

	APG	NPG
Average time in seconds	13.4	7.9

Table 4: Performance results (User experience)

These results show that for NPG, the hardware has a faster computation level and that the time as experienced by the user is shorter. In other words, NPG is faster in both aspects.

4.4 Overall comparison

For the comparison only two versions were taken into account: the original APG and NPG (scenario 2, as described in Section 4.1).

APG scores better than NPG for useability. Section (4.2) shows that APG is worse than NPG when it comes to security. Both versions have their weak spots, but APG clearly has weaker ones. In terms of performance, Table 3 and Table 4 show that NPG outperforms APG in both hardware performance and the performance as experienced by the user. Therefore it can be said that NPG is faster and more secure than APG, but that APG is more user-friendly.

5 Future Work

Several features and functionalities could not be fully implemented in this research. Some required PGP functionalities are missing, as well as some features which, while not completly necessary, would have been useful to have.

The PGP functionalities which this research is lacking are decryption, encryption and signature verification:

- The encryption functionality is based on the public key of the recipient and can be extended with a signature. Therefore it is only a matter of time before this is implemented. In order to implement this, the encryption functionality from APG needs to be taken and combined with the signing functionality of NPG.
- Signature verification is based only on the public key of the sender, meaning that the APG version would act as a sufficient model for the implementation of this functionality. A private key is not needed, which implies that a smartcard is not needed.
- Decryption however, is also based on the private key which is stored on the smartcard. Therefore, another extension has to be developed which can send the encrypted mail to the smartcard, according to its specifications. Afterwards the answer of the smartcard needs to be processed. The implementation of signing has shown that communication with a smartcard is possible. This means that decryption is merely a question of implementing it according to the APIs.

The following three features would have been useful:

1. The ability to generate key pairs on the smartcard. In this research, it was only necessary to have a smartcard which contained keys. The keys of this smartcard were generated on a PC using GPG. Further development could mean that the keys are also generated on the smartcard via the NFC connection.
2. The storing of sub keys is also open for future development.
3. An automated retry. This would help if the communication between smartphone and smartcard gets disturbed.

Furthermore, it would be absolutely essential to have the ability to change the password on the smartcard. For this research it was sufficient in the chosen scenario to use a smartcard with simply one password. However, when considering the real world usage of NPG, a password changing functionality is essential to guarantee the security of the card and thereby the private key.

NPG has some further limitations which are not quite as important as the ability to change the password:

- NPG only allows text as input for e-mails. Alternative versions, which allow data or binary information, could be added in future developments.
- NPG only adds signatures with ASCII armor. (ASCII armor describes a set of text blocks which wrap up the PGP signature. The signature is encoded in another scheme, Radix-64 which is Base64 concatenated with a checksum) An extension could be the generation of a signature without ASCII armor.
- APG standardly produces a version 4 signature, but allows the user to force a version 3 signature for compatibility with older implementations. At the moment, NPG only computes version 4 signatures, so this feature could also be added to NPG.

More globally, NPG could check for NFC availability and warn the user if the smartphone does not support NFC.

The usability and performance of the other scenarios should also be compared in the future. Securing the private key with a smartcard and asking the user for a password (scenario 1) and securing the connection with a smartcard and an additional ID card (scenario 3) should be researched. Even participants who were unaware of the alternative scenarios suggested that they be explored (see Section 4.19. However, both scenarios need to be implemented first.

6 Conclusion and reflection

This section comprises a conclusion and a reflection part. The conclusion draws inferences about the comparison and the implementation itself, and then gives an overall perspective. In the reflection, problems, difficulties and knowledge I gained outside of technical aspects are discussed.

6.1 Conclusion

Some aspects of the implementation clearly took a lot of time and patience. Following the research, it is now possible for the implementation to generate a signature accepted by GPG. Although generating a signature is only one aspect of PGP, this implementation, the abilities of NPG and the abilities of the prototype indicate that it is only a matter of time before a full PGP-supporting version of NPG is built.

The participants tended to prefer APG to NPG. An explanation for the bad usability score of NPG can be found in the range of the NFC chip and the smartcard itself. When the smartcard touches the smartphone, a specific position is needed and the smartcard has to maintain this one position until the signing is complete. If this position is not maintained, the program returns an error and the signing has to be done again. From my experience, I can say that it is definitely possible to learn how to do this in the right way. However, I understand that for the participants this was annoying, as very few people manage to do it correctly on their first try. Therefore, I think that either the technology has to get better (by increasing the range, decreasing the influence of movement, making the connection more stable) or the user needs to get more familiar with the use of NFC. Time could be an important factor here. If in the future the participants have become accustomed to using their smartphone with NFC, it seems highly possible that they would choose NPG over APG.

The time comparison and the security comparison show that NPG is technically superior. It processes the signature faster and the time the user experiences is also shorter.

Furthermore, NPG scores better in the security comparison. Each score (probability, severity and detectability) represents a good estimation, but the sum of these three is not reliable because it is being a simplification without contextual meaning. Nevertheless the security score was helpful in the comparison.

These two results, a working implementation and the comparison, allow us to answer the research question: *how can the OpenPGP smartcard be used in combination with NFC?*

In order to do this, an implementation is needed which uses NFC and handles PGP according to its specification, RFC 4880. One implementation which fulfills those requirements is NPG, since it supports PGP functionality as well as NFC (in this research a smartphone equipped with NFC). This app is capable of generating signatures according to RFC 4880 with the use of an OpenPGP card via the NFC interface. In addition to this, the process of signing is even faster and more secure compared to an app which does not use the NFC interface. The only disadvantage is the restricted user-friendliness, or at least the fact that most users are unaccustomed to NFC.

6.2 Reflection

Looking back on this research, there were several barriers to be broken down. Some lay inside my sphere of influence and some lay outside of it. Two major factors lay outside of my influence.

University: from this side, I got the requested help and support needed to accomplish this thesis in a way I am satisfied with. The only problem was the waiting period for the hardware (the Google Nexus). This messed up my planning somewhat. However I was still able to make up for the lost time caused by this, as I had planned in some extra time in case of unexpected situations. Therefore, allowing extra time in the planning for unexpected situations would be a recommendation for future projects.

APG source code: The second external influence was the APG source code, which had to be extended. It was hard for me to get into the program, because there was almost no documentation at all within the code. To describe the situation in more detail, the APG code consist of an APG package and several other supporting packages. It contains 38 helping files as well as the main APG class, which comprises 2300 rows of code: These rows of code do nearly all the logic within the APG program. The only comments within the code are "to-do's". Therefore, it took time to understand the code well enough to be able to extend it. Another negative point about the APG program is the lack of reusable functionality. In effect, the signature functionality could, for example, be declared once and reused every time needed. In APG the generation of a signature is defined three times, in different contexts.

In terms of what fell under my sphere of influence, I found the following most difficult:

- Getting used to how NFC works
- Hexadecimal communication with the OpenPGP Java Card
- Programming in line with the PGP specification

It took time and I required some help to get used to these three activities. In retrospect, I can say that I now appreciate these obstacles and through overcoming them, have gained a better understanding of the workings of NFC, smartcards and PGP.

References

Android Market (2012). Retrieved July 12, 2012, from `https://market.android.com/search?q=openpgp&c=apps`.

APG source (2012). Retrieved July 12, 2012, from `http://code.google.com/p/android-privacy-guard/`.

Baranyi, L. (2012). Retrieved July 12, 2012, from `http://biphome.spray.se/laszlob/pgp/PGP%20and%20S-mime.doc`.

Burwell, C. (2011). The implementation of contactless payments in the US.

de Ruiter, J. (2012). Retrieved July 12, 2012, from `http://sourceforge.net/projects/javacardopenpgp/`.

DeviceFidelity (2012). Retrieved July 12, 2012, from `http://www.devicefidelity.com/`.

Gerck, E. (2012). Comparison of secure email technologies x. 509/PKI, PGP, and IBE.

Google Inc. (2012). What is android? Retrieved July 12, 2012, from `https://developer.android.com/guide/basics/what-is-android.html`.

Gsmarena (2012). Retrieved July 12, 2012, from `http://www.gsmarena.com/nokia_6131-1434.php`.

Mednieks, Z., Dornin, L., Meike, G., and Nakamura, M. (2011). *Programming Android*. O'Reilly Media, Inc.

Netcom (2012). Retrieved July 12, 2012, from `http://www.engadget.com/2011/06/01/netcom-shows-off-microsd-card-with-integrated-nfc-goodness-vide/`.

NFC Forum (2012). Retrieved July 12, 2012, from `http://www.nfc-forum.org/specs/spec_list`.

OpenPGP Alliance (2012). Retrieved July 12, 2012, from `http://www.openpgp.org/about_openpgp/history.shtml`.

Ortiz Jr, S. (2006). Is near-field communication close to success? *C omputer*, 39(3):18–20.

Pietig, A. (2004). Functional specification of the OpenPGP application on ISO smart card operating systems. Retrieved July 12, 2012, from `http://kerckhoffs.g10code.com/www.g10code.com/docs/openpgp-card-2.0.pdf`.

The Android Open Source Project (2012). Sticky notes. Retrieved July 12, 2012, from `http://nfc.android.com/StickyNotes.zip`.

Vincent, J. (2012). K9 mail. Retrieved July 12, 2012, from `https://code.google.com/p/k9mail/`.

Yan, J., Blackwell, A., Anderson, R., and Grant, A. (2004). Password memorability and security: Empirical results. *Security & Privacy, IEEE*, 2(5):25–31.

Zimmermann, P. (2012). Retrieved July 12, 2012, from `http://www.philzimmermann.com/EN/background/index.html`.

A Appendix

A.1 Survey

A.1.1 APG

	Yes	No
Was it difficult to send the signed mail?	☐	☐
Would you use this sort of email signing on your own mobile?	☐	☐
Do you think it would be easier to use a password in place of the card?	☐	☐

A.1.2 NPG

	Yes	No
Was it difficult to send the signed mail?	☐	☐
Would you use this sort of email signing on your own mobile?	☐	☐
Do you think it would be easier to hold a card against the mobile instead of typing in the password?	☐	☐

B Appendix

B.1 Prototype Code

B.1.1 Source

The file is called NyNFC.java and is in the source folder of the android project.

```
import android.app.Activity;
import android.app.AlertDialog;
import android.app.PendingIntent;
import android.content.DialogInterface;
import android.content.Intent;
import android.content.IntentFilter;
import android.nfc.NfcAdapter;
import android.nfc.R;
import android.nfc.Tag;
import android.nfc.tech.IsoDep;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;

import java.io.IOException;

public class MyNFC extends Activity {
IsoDep isodep;
//APDU's for communication with the OpenPGP card
private String public_signature = "0047810002B60000";
private String public_confidentiality = "0047810002B80000";
private String public_authentication = "0047810002A40000";
private String opening = "00A4040006D27600012401";
private String loggin = "00200082063132333435 36";
private String accepted = "9000";
private String zero = "00";
private String info = "00CA006500";
private String pre_signature = "002A9E9A333031300D060960864801650304020105 00";

//Plain text and hash for signature generation
private String plain_text = "NFC";
private String hash =
"13A717CFAF6CF44760AF64AA969F88A6A632BA73875CAC9628314FF6164C962A";


NfcAdapter mNfcAdapter;
EditText mNote;
AlertDialog ad;

PendingIntent mNfcPendingIntent;
IntentFilter[] mWriteTagFilters;
IntentFilter[] mNdefExchangeFilters;

/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    mNfcAdapter = NfcAdapter.getDefaultAdapter(this);

    setContentView(R.layout.main);
    findViewById(R.id.read_tag).setOnClickListener(mTagReader);
    mNote = ((EditText) findViewById(R.id.note));


    // Handles received NFC intents in this activity.
    mNfcPendingIntent = PendingIntent.getActivity(this, 0,
            new Intent(this,
              getClass()).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP), 0);


    // Intent filters action tag
    IntentFilter tagDetected = new IntentFilter(
      NfcAdapter.ACTION_TAG_DISCOVERED);
    mWriteTagFilters = new IntentFilter[] { tagDetected };
}


/**
 * This method is called when a new intent is fired and if the intent is a
 * discovered Tag it tries to connect to the tag as it is an OpenPGP card.
 * It tries to read the Name, all three public keys and performs signing
 * and decryption otherwise does it prompt an error toast
 * @param intent the intent which is fired
 */
@Override
protected void onNewIntent(Intent intent) {

if (NfcAdapter.ACTION_TAG_DISCOVERED.equals(intent.getAction())) {
  Tag detectedTag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
  isodep = IsoDep.get(detectedTag);

  isodep.setTimeout(10000); //Timeout is set to 10 seconds
                            //to avoid cancellation during signing
  try {
    isodep.connect();
    if (card(opening).equals(accepted)){ //activate connection
      if (card(loggin).equals(accepted)){ // login
        toast("Connected");
```

```
        String name = getName(card(info));
        String sign = getDataField(card(public_signature));
        String confi = getDataField(card(public_confidentiality));
        String auth = getDataField(card(public_authentication));
        String signature = getDataField(card(pre_signature+hash+zero));

        mNote.setText("The card belongs to:\n" + name+
          "\n\nThe public key for signing is:\n" + sign +
          "\n\nThe public key for confidentiality is:\n" + confi +
          "\n\nThe public key for authentication is:\n" + auth +
          "\n\nSIGNING:\nThe plain text is:\n"+plain_text+"\nThe Hash is:\n"+hash+
          "\nThe signature is:\n"+signature);
      }
    }
    else
      mNote.setText("The tag was no OpenPGP Card");
  } catch (IOException e) {
    toast("Error"+e.getMessage());
  }
  ad.cancel();
  }
}



/**
 * The listener for the button which generates the alert dialog
 */
private View.OnClickListener mTagReader = new View.OnClickListener() {
    @Override
    public void onClick(View arg0) {
        enableCommunication();

    AlertDialog.Builder builder =  new AlertDialog.Builder(MyNFC.this)
                .setTitle("Touch OpenPGP Card")
                .setOnCancelListener(new DialogInterface.OnCancelListener() {
                    @Override
                    public void onCancel(DialogInterface dialog) {
                        disableCommunication();
                    }
                });
    ad = builder.create();
    ad.show();
    }
};

/**
 * Enables the communication between the program and a NFC tag
 */
private void enableCommunication() {
    IntentFilter tagDetected = new IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED);
    mWriteTagFilters = new IntentFilter[] {
        tagDetected
    };
    mNfcAdapter.enableForegroundDispatch(this, mNfcPendingIntent,
        mWriteTagFilters, null);
}


/**
 * Disables the communication between the program and a NFC tag
 */
private void disableCommunication() {
    mNfcAdapter.disableForegroundDispatch(this);
}


    /**
     * Brings a text to the screen
     * @param text The String which should be shown to the user
     */
    private void toast(String text) {
        Toast.makeText(this, text, Toast.LENGTH_SHORT).show();
    }

    /**
     * Communicates with the OpenPgpCard via the APDU
     * @param hex the hexadecimal APDU
     * @return The answer from the card
     * @throws IOException throws an exception if something goes wrong
     */
    private String card(String hex) throws IOException{
     return getHex(isodep.transceive(hex2Byte(hex)));
    }

    /**
     * Reduces the raw data from the card by four characters
     * @param output the raw data from the card
     * @return the data field of that data
     */
    private String getDataField(String output){
     return output.substring(0, output.length()-4);
    }

    /**
     * Gets the name of the user out of the raw card output regarding
     * cardholder related data
     * @param name the raw cardholder related data from the card
```

```
     * @return the name given in this data
     */
    private String getName(String name){
     String slength = "";
     int ilength = 0;

     name = name.substring(6);
     slength = name.substring(0, 2);
     ilength = Integer.parseInt(slength,16)*2;
     name = name.substring(2, ilength+2);
     return (new String(hex2Byte(name))).replace('<', ' ');
    }

    /**
     * Converts an integer into a byte array
     * taken from http://snippets.dzone.com/posts/show/93
     * @param value the integer representation
     * @return the byte array representation
     */
    private byte[] int2ByteArray(int value) {
        byte[] b = new byte[4];
        for (int i = 0; i < 4; i++) {
            int offset = (b.length - 1 - i) * 8;
            b[i] = (byte) ((value >>> offset) & 0xFF);
        }
        return b;
    }

    /**
     * Converts a byte array into an integer
     * taken from http://www.java2s.com/Tutorial/Java
     * /0180__File/ConvertBytearraytoInt.htm
     * @param bytes the byte array representation
     * @return the integer representation
     */
    private static int byteArray2Int( byte[] bytes ) {
        int result = 0;
        for (int i=0; i<4; i++) {
          result = ( result << 8 ) - Byte.MIN_VALUE + (int) bytes[i];
        }
        return result;
      }

    /**
     * Converts a hexadecimal string into an byte array
     * taken from http://www.developerfeed.com/javacore
     * /blog/how-convert-hex-string-bytes-and-viceversa-java
     * @param hex the hexadecimal string representation
     * @return the byte array representation
     */
    private byte[] hex2Byte(String hex)
    {
       byte[] bytes = new byte[hex.length() / 2];
       for (int i = 0; i < bytes.length; i++)
       {
          bytes[i] = (byte) Integer
                .parseInt(hex.substring(2 * i, 2 * i + 2), 16);
       }
       return bytes;
    }
    /**
     * Converts a byte array into an
     * taken from http://www.rgagnon.com/javadetails/java-0596.html
     * @param raw the byte array representation
     * @return the  hexadecimal string representation
     */
    public static String getHex( byte [] raw ) {
    String HEXES = "0123456789ABCDEF";
      if ( raw == null ) {
        return null;
      }
      StringBuilder hex = new StringBuilder( 2 * raw.length );
      for ( final byte b : raw ) {
        hex.append(HEXES.charAt((b & 0xF0) >> 4))
           .append(HEXES.charAt((b & 0x0F)));
      }
      return hex.toString();
    }

}
```

B.1.2 Values

The file is called strings.xml and is in the values folder of the android project.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">NFC</string>
  <string name="read_tag">Read</string>
  <string name="default_note_text">
    Please press \'Read\' to scan a OpenPGP Card!
  </string>
</resources>
```

B.1.3 Layout

The file is called main.xml and is in the layout folder of the android project.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
  <LinearLayout android:layout_height="wrap_content"
    android:layout_width="match_parent" android:id="@+id/linearLayout1">
    <Button android:layout_height="wrap_content"
      android:layout_width="wrap_content" android:id="@+id/read_tag"
      android:text="@string/read_tag">
    </Button>
  </LinearLayout>
  <EditText
      android:id="@+id/note"
      android:layout_width="fill_parent"
      android:layout_height="294dp"
      android:gravity="top"
      android:text="@string/default_note_text" >
  </EditText>
</LinearLayout>
```

B.1.4 Manifest

The file is called AndroidManifest.xml and is in the root folder of the android project.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="android.nfc"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-sdk android:minSdkVersion="10" />
    <uses-permission android:name="android.permission.NFC"></uses-permission>

    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name="nfc.MyNFC"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>

        </activity>

    </application>
</manifest>
```

B.2 NPG Code

B.2.1 Source

Added to EncryptActivity

```
private Bundle tieUP(boolean sign){
     Bundle b = new Bundle();
    b.putBoolean("org.thialfihar.android.apg.binary", mDataSource.isBinary());
    b.putBoolean("org.thialfihar.android.apg.sign", mGenerateSignature || sign);
    byte[] temp;
    if ( mDataSource.getData() != null)
    temp = mDataSource.getData();
    else
    temp = mDataSource.getText().getBytes();

    b.putByteArray("org.thialfihar.android.apg.data", temp);
    return b;
    }
```

Changed in EncryptActivity

```
 public void run() {

        String error = null;
        Bundle data = new Bundle();
        Message msg = new Message();

        try {
         InputData in;
            OutputStream out;
            boolean useAsciiArmour = true;

         if (!mPreferences.getNFC()){

            long encryptionKeyIds[] = null;
            long signatureKeyId = 0;
            int compressionId = 0;
            boolean signOnly = false;

            String passPhrase = null;
            if (mMode.getCurrentView().getId() == R.id.modeSymmetric) {
                passPhrase = mPassPhrase.getText().toString();
                if (passPhrase.length() == 0) {
                    passPhrase = null;
                }
            } else {
                encryptionKeyIds = mEncryptionKeyIds;
                signatureKeyId = getSecretKeyId();
                signOnly = (mEncryptionKeyIds == null || mEncryptionKeyIds.length == 0);
```

```
            }

            fillDataSource(signOnly && !mReturnResult);
            fillDataDestination();

            // streams
            in = mDataSource.getInputData(this, true);
            out = mDataDestination.getOutputStream(this);

            if (mEncryptTarget == Id.target.file) {
                useAsciiArmour = mAsciiArmour.isChecked();
                compressionId = ((Choice) mFileCompression.getSelectedItem()).getId();
            } else {
                useAsciiArmour = true;
                compressionId = mPreferences.getDefaultMessageCompression();
            }

            if (mOverrideAsciiArmour) {
                useAsciiArmour = mAsciiArmourDemand;
            }

            if (mGenerateSignature) {
               Apg.generateSignature(this, in, out, useAsciiArmour, mDataSource.isBinary(),
                                     getSecretKeyId(),
                                     Apg.getCachedPassPhrase(getSecretKeyId()),
                                     mPreferences.getDefaultHashAlgorithm(),
                                     mPreferences.getForceV3Signatures(),
                                     this);
            } else if (signOnly) {
             long before = System.currentTimeMillis();
             Apg.signText(this, in, out, getSecretKeyId(),
                             Apg.getCachedPassPhrase(getSecretKeyId()),
                             mPreferences.getDefaultHashAlgorithm(),
                             mPreferences.getForceV3Signatures(),
                             this);
             long after = System.currentTimeMillis();
             long needed = after-before;
            } else {
                Apg.encrypt(this, in, out, useAsciiArmour,
                            encryptionKeyIds, signatureKeyId,
                            Apg.getCachedPassPhrase(signatureKeyId), this,
                            mPreferences.getDefaultEncryptionAlgorithm(),
                            mPreferences.getDefaultHashAlgorithm(),
                            compressionId,
                            mPreferences.getForceV3Signatures(),
                            passPhrase);
            }
        }
         else{
         fillDataDestination();
         out = mDataDestination.getOutputStream(this);
         out.write(signature);

         }


            out.close();
            if (mEncryptTarget != Id.target.file) {

             if(out instanceof ByteArrayOutputStream) {
             if (useAsciiArmour) {
                     String extraData = new String(((ByteArrayOutputStream)out).toByteArray());
                     if (mGenerateSignature) {
                         data.putString(Apg.EXTRA_SIGNATURE_TEXT, extraData);
                     } else {
                     data.putString(Apg.EXTRA_ENCRYPTED_MESSAGE, extraData);
                     }
                 } else {
                     byte extraData[] = ((ByteArrayOutputStream)out).toByteArray();
                     if (mGenerateSignature) {
                         data.putByteArray(Apg.EXTRA_SIGNATURE_DATA, extraData);
                     } else {
                     data.putByteArray(Apg.EXTRA_ENCRYPTED_DATA, extraData);
                     }
                 }
} else if(out instanceof FileOutputStream) {
String fileName = mDataDestination.getStreamFilename();
String uri = "content://" + DataProvider.AUTHORITY + "/data/" + fileName;
data.putString(Apg.EXTRA_RESULT_URI, uri);
} else {
throw new Apg.GeneralException("No output-data found.");
}
            }
        } catch (IOException e) {
            error = "" + e;
        } catch (PGPException e) {
            error = "" + e;
        } catch (NoSuchProviderException e) {
            error = "" + e;
        } catch (NoSuchAlgorithmException e) {
            error = "" + e;
        } catch (SignatureException e) {
            error = "" + e;
        } catch (Apg.GeneralException e) {
            error = "" + e;
        }

        data.putInt(Constants.extras.status, Id.message.done);
```

```
        if (error != null) {
            data.putString(Apg.EXTRA_ERROR, error);
        }

        msg.setData(data);
        sendMessage(msg);

    }

private void initiateEncryption() {
        if (mEncryptTarget == Id.target.file) {
            String currentFilename = mFilename.getText().toString();
            if (mInputFilename == null || !mInputFilename.equals(currentFilename)) {
                guessOutputFilename();
            }

            if (mInputFilename.equals("")) {
                Toast.makeText(this, R.string.noFileSelected, Toast.LENGTH_SHORT).show();
                return;
            }

            if (!mInputFilename.startsWith("content")) {
                File file = new File(mInputFilename);
                if (!file.exists() || !file.isFile()) {
                    Toast.makeText(this, getString(R.string.errorMessage,
                                                   getString(R.string.error_fileNotFound)),
                                   Toast.LENGTH_SHORT).show();
                    return;
                }
            }
        }

        // symmetric encryption
        if (mMode.getCurrentView().getId() == R.id.modeSymmetric) {
            boolean gotPassPhrase = false;
            String passPhrase = mPassPhrase.getText().toString();
            String passPhraseAgain = mPassPhraseAgain.getText().toString();
            if (!passPhrase.equals(passPhraseAgain)) {
                Toast.makeText(this, R.string.passPhrasesDoNotMatch, Toast.LENGTH_SHORT).show();
                return;
            }

            gotPassPhrase = (passPhrase.length() != 0);
            if (!gotPassPhrase) {
                Toast.makeText(this, R.string.passPhraseMustNotBeEmpty, Toast.LENGTH_SHORT).show();
                return;
            }
        } else {
            boolean encryptIt = (mEncryptionKeyIds != null && mEncryptionKeyIds.length > 0);
            // for now require at least one form of encryption for files
            if (!encryptIt && mEncryptTarget == Id.target.file) {
                Toast.makeText(this, R.string.selectEncryptionKey, Toast.LENGTH_SHORT).show();
                return;
            }

            if (!encryptIt && getSecretKeyId() == 0) {
                Toast.makeText(this, R.string.selectEncryptionOrSignatureKey,
                               Toast.LENGTH_SHORT).show();
                return;
            }

            if (getSecretKeyId() != 0 && Apg.getCachedPassPhrase(getSecretKeyId()) == null) {

             if (mPreferences.getNFC()){
             boolean signOnly = false;
             signOnly = (mEncryptionKeyIds == null || mEncryptionKeyIds.length == 0);

             fillDataSource(signOnly && !mReturnResult);

             Intent intent = new Intent(this, NFC.class);
             intent.putExtras(tieUP(signOnly));
             this.startActivityForResult(intent, 42);

         }
             else
             showDialog(Id.dialog.pass_phrase);
                return;
            }
        }
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        switch (requestCode) {
            case Id.request.filename: {
                if (resultCode == RESULT_OK && data != null) {
                    String filename = data.getDataString();
                    if (filename != null) {
                        // Get rid of URI prefix:
                        if (filename.startsWith("file://")) {
                            filename = filename.substring(7);
                        }
                        // replace %20 and so on
                        filename = Uri.decode(filename);

                        mFilename.setText(filename);
                    }
                }
                return;
            }

            case Id.request.output_filename: {
```

```
                if (resultCode == RESULT_OK && data != null) {
                    String filename = data.getDataString();
                    if (filename != null) {
                        // Get rid of URI prefix:
                        if (filename.startsWith("file://")) {
                            filename = filename.substring(7);
                        }
                        // replace %20 and so on
                        filename = Uri.decode(filename);

                        FileDialog.setFilename(filename);
                    }
                }
                return;
            }

            case Id.request.secret_keys: {
                if (resultCode == RESULT_OK) {
                    super.onActivityResult(requestCode, resultCode, data);
                }
                updateView();
                break;
            }

            case Id.request.public_keys: {
                if (resultCode == RESULT_OK) {
                    Bundle bundle = data.getExtras();
                    mEncryptionKeyIds = bundle.getLongArray(Apg.EXTRA_SELECTION);
                }
                updateView();
                break;
            }
            case 42:{
                if(resultCode==RESULT_OK){
                 signature = data.getByteArrayExtra("org.thialfihar.android.apg.signature");
                 //send(signature);
                 encryptStart();
                }
            }

            default: {
                break;
            }
        }

        super.onActivityResult(requestCode, resultCode, data);
    }
```

New Class NFC

```
package org.thialfihar.android.apg;


import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.OutputStream;

import java.math.BigInteger;

import org.jakubeit.nfc.NPG;
import org.jakubeit.nfc.Sig;
import org.thialfihar.android.apg.R;
import android.app.AlertDialog;
import android.app.PendingIntent;
import android.content.DialogInterface;
import android.content.Intent;
import android.content.IntentFilter;
import android.nfc.NfcAdapter;
import android.nfc.Tag;
import android.nfc.tech.IsoDep;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;
/**
 *
 * @author Phil Jakubeit
 * Class which creates main activity for NFC connection in NPG
 *
 */
public class NFC extends BaseActivity{
    private NfcAdapter mNfcAdapter;
    private IntentFilter[] mWriteTagFilters;
    private PendingIntent mNfcPendingIntent;
private IsoDep isodep;
private AlertDialog ad;
private boolean sign,binary;
private DataSource data;
private Intent intent;
private Sig sig;
private NPG npg;
private OutputStream out;

@Override
/**
 * Initial method of this activity
 */
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.nfc);
```

```
        sig = new Sig(this);
        npg = new NPG();
        mNfcAdapter = NfcAdapter.getDefaultAdapter(this);

        // Handles received NFC intents in this activity.
        mNfcPendingIntent = PendingIntent.getActivity(this, 0,
                new Intent(this, getClass()).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP), 0);


        // Intent filters action tag
        IntentFilter tagDetected = new IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED);
        mWriteTagFilters = new IntentFilter[] { tagDetected };
        //sets intent
        intent= getIntent();

        //set extra values to pass to NPG class
        Bundle bundle = intent.getExtras();
        sign = bundle.getBoolean("org.thialfihar.android.apg.sign");
        binary = bundle.getBoolean("org.thialfihar.android.apg.binary");
        byte[] d = bundle.getByteArray("org.thialfihar.android.apg.data");

        //convert data according to RFC 4880 if not binary
        if (binary){
         toast("Binary data is not yet Supported!");
         finish();
        }
        data = new DataSource();

        data.setData(d);


    }

/**
 * Handles the event if the button is pressed, an Alert dialog is shown
 * and the device listens for NFC intents
 * @param arg0
 */
    public void nfcClick(View arg0) {
        // Write to a tag for as long as the dialog is shown.
     enableCommunication();

        AlertDialog.Builder builder =  new AlertDialog.Builder(NFC.this).setTitle("Touch OpenPGP Card")
                .setOnCancelListener(new DialogInterface.OnCancelListener() {
                    @Override
                    public void onCancel(DialogInterface dialog) {
                     disableCommunication();
                    }
                });
        ad = builder.create();
        ad.show();
        }


    /**
     * Handles the NFC intent
     */
public void onNewIntent(Intent intent) {

        if (NfcAdapter.ACTION_TAG_DISCOVERED.equals(intent.getAction())) {
            Tag detectedTag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
            isodep = IsoDep.get(detectedTag);
            String opening = "00A4040006D27600012401";
            String loggin = "0020008206313233343536"; // must be changed for different scenario, contains PW
            String accepted = "9000";

            isodep.setTimeout(10000); //Timeout is set to 10 seconds to avoid cancellation during signing
            try {
             InputData in = data.getInputData(this, true);
             out = new ByteArrayOutputStream();
isodep.connect();
long before = System.currentTimeMillis();//measuring time for comparison
if (card(opening).equals(accepted)){ //activate connection
if (card(loggin).equals(accepted)){ // login
if (sign){
npg.signText(in, out, getID(),Long.parseLong(getKeyId(),16), sig);
}
}
}
long after = System.currentTimeMillis();//measuring time for comparison
} catch (Exception e) {
toast("Connection  Error"+e.getMessage());
}
            try {
     out.close();
     } catch (IOException e) {
     toast("Cannot close Outputstream"+e);
     }
         intent.putExtra("org.thialfihar.android.apg.signature", ((ByteArrayOutputStream)out).toByteArray());
            ad.cancel();
            setResult(RESULT_OK, intent);//signature is passed back to EncryptActivity
            finish();
        }

    }
/**
 * Gets the user ID
 * @return the user id as "name <email>"
 * @throws IOException
```

```
 */
    private String getID() throws IOException{
     String info = "00CA006500";
     String data = "00CA005E00";
     return getName(card(info))+" <"+(new String(hex2Byte(getDataField(card(data)))))+">";

    }

    /**
     * Gets the key ID
     * @return the key id (last 8 bytes of the fingerprint)
     * @throws IOException
     */

    private String getKeyId() throws IOException{
     String fp = getFingerprint();
     return fp.substring(24);
    }

    /**
     * Gets the fingerprint of the signature key
     * @return
     * @throws IOException
     */
    private String getFingerprint() throws IOException{
     String data = "00CA006E00";
     String fingerprint = card(data);
     fingerprint = fingerprint.substring(fingerprint.indexOf("C5")+4,fingerprint.indexOf("C5")+44);
     return fingerprint;

    }

    /**
     * Calls to calculate the signature and returns the MPI value
     * @param hash the hash for signing
     * @return a big integer representing the MPI for the given hash
     * @throws IOException
     */
public BigInteger calculateSignature(byte[] hash) throws IOException{
String zero = "00";
        String pre_signature = "002A9E9A333031300D06096086480165030402010500042O";
        pre_signature = pre_signature+getHex(hash)+zero;
        String raw = card(pre_signature+getHex(hash)+zero);
        String signature = getDataField(raw);
        String second = "";
        second =  getDataField(card("00C0000001"));
signature += second;

        BigInteger bi = new BigInteger(1,hex2Byte(signature));
        return bi;

}

/**
 * Prints a message to the screen
 * @param text the text which should be contained within the toast
 */
private void toast(String text) {
        Toast.makeText(this, text, Toast.LENGTH_SHORT).show();
    }

/**
 * Activates the listening for NFC tags
 */
public void enableCommunication() {
        IntentFilter tagDetected = new IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED);
        mWriteTagFilters = new IntentFilter[] {
            tagDetected
        };
        try{
        mNfcAdapter.enableForegroundDispatch(this, mNfcPendingIntent, mWriteTagFilters, null);
        }catch(IllegalStateException e){
         toast("Enable Error:"+e);
        }
}
/**
 * Deactivates the listening for NFC tags
 */
public void disableCommunication() {
        mNfcAdapter.disableForegroundDispatch(this);
    }

    /**
     * Gets the name of the user out of the raw card output regarding card holder related data
     * @param name the raw card holder related data from the card
     * @return the name given in this data
     */
    private String getName(String name){
     String slength = "";
     int ilength = 0;

     name = name.substring(6);
     slength = name.substring(0, 2);
     ilength = Integer.parseInt(slength,16)*2;
     name = name.substring(2, ilength+2);
     name = (new String(hex2Byte(name))).replace('<', ' ');
     return (name);
}
```

```
/**
     * Reduces the raw data from the card by four characters
     * @param output the raw data from the card
     * @return the data field of that data
     */
    private String getDataField(String output){
     return output.substring(0, output.length()-4);
    }

/**
 * Communicates with the OpenPgpCard via the ADPU
 * @param hex the hexadecimal ADPU
 * @return The answer from the card
 * @throws IOException throws an exception if something goes wrong
 */
private String card(String hex) throws IOException{
return getHex(isodep.transceive(hex2Byte(hex)));
}
/**
 * Converts a hexadecimal string into an byte array                          21
 * taken from http://www.developerfeed.com/javacore
 * /blog/how-convert-hex-string-bytes-and-viceversa-java
 * @param hex the hexadecimal string representation
 * @return the byte array representation
 */
private byte[] hex2Byte(String hex)
{
   byte[] bytes = new byte[hex.length() / 2];
   for (int i = 0; i < bytes.length; i++)
   {
      bytes[i] = (byte) Integer
            .parseInt(hex.substring(2 * i, 2 * i + 2), 16);
   }
   return bytes;
}

/**
 * Converts a byte array into an hex string
 * taken from http://www.rgagnon.com/javadetails/java-0596.html
 * @param raw the byte array representation
 * @return the  hexadecimal string representation
 */

public static String getHex( byte [] raw ) {
String HEXES = "0123456789ABCDEF";
  if ( raw == null ) {
    return null;
  }
  StringBuilder hex = new StringBuilder( 2 * raw.length );
  for ( final byte b : raw ) {
    hex.append(HEXES.charAt((b & 0xF0) >> 4))
       .append(HEXES.charAt((b & 0x0F)));
  }
  return hex.toString();
}



}
```

New class NPG

```
package org.jakubeit.nfc;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.security.NoSuchAlgorithmException;
import java.security.Security;
import java.security.SignatureException;

import org.spongycastle.bcpg.ArmoredOutputStream;
import org.spongycastle.bcpg.BCPGOutputStream;
import org.spongycastle.jce.provider.BouncyCastleProvider;
import org.spongycastle.openpgp.PGPException;
import org.spongycastle.openpgp.PGPPrivateKey;
import org.spongycastle.openpgp.PGPSecretKey;
import org.spongycastle.openpgp.PGPSecretKeyRing;
import org.spongycastle.openpgp.PGPSignature;
import org.spongycastle.openpgp.PGPV3SignatureGenerator;
import org.thialfihar.android.apg.Apg;
import org.thialfihar.android.apg.InputData;
import org.thialfihar.android.apg.ProgressDialogUpdater;
import org.thialfihar.android.apg.R;
import org.thialfihar.android.apg.Apg.GeneralException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager.NameNotFoundException;

/**
 *
 * @author Phil Jakubeit
 * Class manages signing of NPG
 */
public class NPG {
```

```
private static String VERSION = null;
private static final String mApgPackageName = "org.thialfihar.android.apg";

/**
 * Empty constructor
 */
public NPG(){
}


/**
 * Gives the current version
 * @return statically "NPG v1.0"
 */
    public static String getFullVersion() {
        return "NPG v1.0";
    }

    /**
     * Signs a given Text with and places the signature in ASCII armor
     * @param data The text from the mail
     * @param outStream The declared output stream
     * @param userId the ID of the user
     * @param keyid the ID of the used Key
     * @param sig an instance of class Sig
     * @throws Exception
     * @throws SignatureException
     */
public static void signText(
            InputData data, OutputStream outStream, String userId, long keyid,Sig sig)
             throws Exception,
             SignatureException {
Security.addProvider(new BouncyCastleProvider());

ArmoredOutputStream armorOut = new ArmoredOutputStream(outStream);
armorOut.setHeader("Version", getFullVersion());

PGPSignatureGen signatureGenerator = null;


signatureGenerator = new PGPSignatureGen(1, 8, new BouncyCastleProvider(), sig);
signatureGenerator.initSign(PGPSignature.CANONICAL_TEXT_DOCUMENT);

PGPSignatureSubpacketGenerator spGen = new PGPSignatureSubpacketGenerator();

spGen.setSignerUserID(false, userId);
signatureGenerator.setHashedSubpackets(spGen.generate());




armorOut.beginClearText(8);

InputStream inStream = data.getInputStream();
final BufferedReader reader = new BufferedReader(new InputStreamReader(inStream));

final byte[] newline = "\r\n".getBytes("UTF-8");


processLine(reader.readLine(), armorOut, signatureGenerator);


while (true) {
final String line = reader.readLine();

if (line == null) {
armorOut.write(newline);
break;
}

armorOut.write(newline);

signatureGenerator.update(newline);
processLine(line, armorOut, signatureGenerator);

}

armorOut.endClearText();

BCPGOutputStream bOut = new BCPGOutputStream(armorOut);
signatureGenerator.setKeyID(keyid);
signatureGenerator.generate().encode(bOut);

armorOut.close();

}
/**
 * Processes a line for PGP conform output (taken from APG)
 * @param pLine
 * @param pArmoredOutput
 * @param pSignatureGenerator
 * @throws IOException
 * @throws SignatureException
 */
    private static void processLine(final String pLine,
            final ArmoredOutputStream pArmoredOutput,
            final PGPSignatureGen pSignatureGenerator)
             throws IOException, SignatureException {
```

```
if (pLine == null) {
return;
}

final char[] chars = pLine.toCharArray();
int len = chars.length;

while (len > 0) {
if (!Character.isWhitespace(chars[len - 1])) {
break;
}
len--;
}

final byte[] data = pLine.substring(0, len).getBytes("UTF-8");

if (pArmoredOutput != null) {
pArmoredOutput.write(data);
}
pSignatureGenerator.update(data);
}
}
```

New class Sig

```
package org.jakubeit.nfc;


import java.io.IOException;
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.ArrayList;

import org.thialfihar.android.apg.NFC;

/**
 * Class for Signatures
 * @author Phil Jakubeit
 *
 */
public class Sig {
private ArrayList<Byte> data;
private byte[] hash;
private NFC calling;
private String signature;
public Sig (NFC nfc){
data = new ArrayList<Byte>();
calling = nfc;
}

/**
 * Updates a byte to the signature
 * @param b
 */
public void update(byte b){
data.add(b);

}

/**
 * Updates a byte array to teh signature
 * @param b
 */
public void update(byte[] b){
for (int i = 0; i < b.length; i++)
data.add(b[i]);
}

/**
 * Calculates the hash of the current state of the signature
 */
public void hash() {
try {
MessageDigest md = MessageDigest.getInstance("SHA-256");
byte[] temp = new byte[data.size()];
for (int i = 0; i < data.size(); i++)
temp[i] = data.get(i);
md.update(temp);
hash = md.digest();
        } catch (NoSuchAlgorithmException e1) {
//shouldn't occur
}
}

/**
 * Returns the leftmost bytes of the hash
 * @return the first and second byte of teh hash
 */
public byte[] getFirstTwoHashBytes(){
byte[] firsttwo = new byte[2];
firsttwo[0] = hash[0];
firsttwo[1] = hash[1];
return firsttwo;

}

/**
```

```
 * Calls NFC for calculating the signature on the card
 * @return The big integer representing the MPI
 * @throws IOException
 */
public BigInteger sign() throws IOException{
return calling.calculateSignature(hash);
}

}
```

New class PGPSignatureGen

```
package org.jakubeit.nfc;


import org.spongycastle.openpgp.PGPException;
import org.spongycastle.bcpg.*;
import org.spongycastle.bcpg.sig.IssuerKeyID;
import org.spongycastle.bcpg.sig.SignatureCreationTime;
import org.spongycastle.util.Strings;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.math.BigInteger;
import java.security.*;
import java.util.Date;

/**
 *
 * Generator for PGP Signatures. Taken from Spongycastle and adapted
 */
public class PGPSignatureGen
{
    private int             keyAlgorithm;
    private int             hashAlgorithm;
    private Sig sig;
    private int             signatureType;
    private long keyID;

    private byte            lastb;

    SignatureSubpacket[]    unhashed = new SignatureSubpacket[0];
    SignatureSubpacket[]    hashed = new SignatureSubpacket[0];

    /**
     * Create a generator for the passed in keyAlgorithm and hashAlgorithm codes.
     *
     * @param keyAlgorithm keyAlgorithm to use for signing
     * @param hashAlgorithm algorithm to use for digest
     * @param provider provider to use for digest algorithm
     * @throws NoSuchAlgorithmException
     * @throws NoSuchProviderException
     * @throws PGPException
     */
    public PGPSignatureGen(
        int     keyAlgorithm,
        int     hashAlgorithm,
        String  provider)
        throws NoSuchAlgorithmException, NoSuchProviderException, PGPException
    {
        this(keyAlgorithm, provider, hashAlgorithm, provider);
    }

    public PGPSignatureGen(
        int      keyAlgorithm,
        int      hashAlgorithm,
        Provider provider,
        Sig sig)
        throws NoSuchAlgorithmException, PGPException
    {
     this(keyAlgorithm, provider, hashAlgorithm, provider);
     this.sig = sig;
    }

    /**
     * Create a generator for the passed in keyAlgorithm and hashAlgorithm codes.
     *
     * @param keyAlgorithm keyAlgorithm to use for signing
     * @param sigProvider provider to use for signature generation
     * @param hashAlgorithm algorithm to use for digest
     * @param digProvider provider to use for digest algorithm
     * @throws NoSuchAlgorithmException
     * @throws NoSuchProviderException
     * @throws PGPException
     */
    public PGPSignatureGen(
        int     keyAlgorithm,
        String  sigProvider,
        int     hashAlgorithm,
        String  digProvider)
        throws NoSuchAlgorithmException, NoSuchProviderException, PGPException
    {
        this(keyAlgorithm, PGPUtil.getProvider(sigProvider), hashAlgorithm, PGPUtil.getProvider(digProvider));
    }

    public PGPSignatureGen(
        int      keyAlgorithm,
        Provider sigProvider,
```

```
    int      hashAlgorithm,
    Provider digProvider)
    throws NoSuchAlgorithmException, PGPException
{
    this.keyAlgorithm = keyAlgorithm;
    this.hashAlgorithm = hashAlgorithm;

}

/**
 * Initializes the signature
 * @param signatureType
 * @throws PGPException
 */
public void initSign(
        int             signatureType)
        throws PGPException
    {
        this.signatureType = signatureType;

        lastb = 0;
    }

/**
 * updates the signature data
 * @param b
 * @throws SignatureException
 */
public void update(
    byte    b)
    throws SignatureException
{
    if (signatureType == PGPSignature.CANONICAL_TEXT_DOCUMENT)
    {
        if (b == '\r')
        {
            sig.update((byte)'\r');
            sig.update((byte)'\n');
        }
        else if (b == '\n')
        {
            if (lastb != '\r')
            {
                sig.update((byte)'\r');
                sig.update((byte)'\n');
            }
        }
        else
        {
            sig.update(b);
        }

        lastb = b;
    }
    else
    {
        sig.update(b);
    }
}
public void update(
        byte[]    b)
        throws SignatureException
    {
        sig.update(b);
    }



public void setHashedSubpackets(
    PGPSignatureSubpacketVector    hashedPcks)
{
    if (hashedPcks == null)
    {
        hashed = new SignatureSubpacket[0];
        return;
    }

    hashed = hashedPcks.toSubpacketArray();
}

public void setUnhashedSubpackets(
    PGPSignatureSubpacketVector    unhashedPcks)
{
    if (unhashedPcks == null)
    {
        unhashed = new SignatureSubpacket[0];
        return;
    }

    unhashed = unhashedPcks.toSubpacketArray();
}

/**
 * Sets the key ID to the given value, for external keys on a smartcard
 * @param id the key id
 */
public void setKeyID(long id){
 keyID = id;
```

```
}

/**
 * Return a signature object containing the current signature state.
 *
 * @return PGPSignature
 * @throws PGPException
 * @throws SignatureException
 * @throws IOException
 */
public PGPSignature generate()
    throws PGPException, SignatureException, IOException
{
    MPInteger[]             sigValues;
    int                     version = 4;
    ByteArrayOutputStream   sOut = new ByteArrayOutputStream();
    SignatureSubpacket[]    hPkts, unhPkts;

    if (!packetPresent(hashed, SignatureSubpacketTags.CREATION_TIME))
    {
        hPkts = insertSubpacket(hashed, new SignatureCreationTime(false, new Date()));
    }
    else
    {
        hPkts = hashed;
    }

    if (!packetPresent(hashed, SignatureSubpacketTags.ISSUER_KEY_ID) && !packetPresent(unhashed, SignatureSubpacketTags.ISSUER_KEY_ID))
    {
        unhPkts = insertSubpacket(unhashed, new IssuerKeyID(false, keyID));
    }
    else
    {
        unhPkts = unhashed;
    }

    try
    {
        sOut.write((byte)version);
        sOut.write((byte)signatureType);
        sOut.write((byte)keyAlgorithm);
        sOut.write((byte)hashAlgorithm);

        ByteArrayOutputStream    hOut = new ByteArrayOutputStream();

        for (int i = 0; i != hPkts.length; i++)
        {
            hPkts[i].encode(hOut);
        }

        byte[]                           data = hOut.toByteArray();

        sOut.write((byte)(data.length >> 8));
        sOut.write((byte)data.length);
        sOut.write(data);
    }
    catch (IOException e)
    {
        throw new PGPException("exception encoding hashed data.", e);
    }

    byte[]    hData = sOut.toByteArray();

    sOut.write((byte)version);
    sOut.write((byte)0xff);
    sOut.write((byte)(hData.length >> 24));
    sOut.write((byte)(hData.length >> 16));
    sOut.write((byte)(hData.length >> 8));
    sOut.write((byte)(hData.length));

    byte[]    trailer = sOut.toByteArray();

    sig.update(trailer);
    sig.hash(); //hashing is done manually
    sigValues = new MPInteger[1];
    BigInteger bi = sig.sign();
    sigValues[0] = new MPInteger(bi);


    byte[] fingerPrint = sig.getFirstTwoHashBytes();

    return new PGPSignature(new SignaturePacket(signatureType, keyID, keyAlgorithm, hashAlgorithm, hPkts, unhPkts, fingerPrint, sigValues));
}



private boolean packetPresent(
    SignatureSubpacket[] packets,
    int type)
{
    for (int i = 0; i != packets.length; i++)
    {
        if (packets[i].getType() == type)
        {
            return true;
        }
    }
```

```
        return false;
    }

    private SignatureSubpacket[] insertSubpacket(
        SignatureSubpacket[] packets,
        SignatureSubpacket subpacket)
    {
        SignatureSubpacket[] tmp = new SignatureSubpacket[packets.length + 1];

        tmp[0] = subpacket;
        System.arraycopy(packets, 0, tmp, 1, packets.length);

        return tmp;
    }


}
```

B.2.2 Values

```
package org.jakubeit.nfc;


import java.io.IOException;
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.ArrayList;

import org.thialfihar.android.apg.NFC;

/**
 * Class for Signatures
 * @author Phil Jakubeit
 *
 */
public class Sig {
private ArrayList<Byte> data;
private byte[] hash;
private NFC calling;
private String signature;
public Sig (NFC nfc){
data = new ArrayList<Byte>();
calling = nfc;
}

/**
 * Updates a byte to the signature
 * @param b
 */
public void update(byte b){
data.add(b);

}

/**
 * Updates a byte array to teh signature
 * @param b
 */
public void update(byte[] b){
for (int i = 0; i < b.length; i++)
data.add(b[i]);
}

/**
 * Calculates the hash of the current state of the signature
 */
public void hash() {
try {
MessageDigest md = MessageDigest.getInstance("SHA-256");
byte[] temp = new byte[data.size()];
for (int i = 0; i < data.size(); i++)
temp[i] = data.get(i);
md.update(temp);
hash = md.digest();
        } catch (NoSuchAlgorithmException e1) {
//shouldn't occur
}
}

/**
 * Returns the leftmost bytes of the hash
 * @return the first and second byte of teh hash
 */
public byte[] getFirstTwoHashBytes(){
byte[] firsttwo = new byte[2];
firsttwo[0] = hash[0];
firsttwo[1] = hash[1];
return firsttwo;

}

/**
 * Calls NFC for calculating the signature on the card
 * @return The big integer representing the MPI
 * @throws IOException
 */
```

```
public BigInteger sign() throws IOException{
return calling.calculateSignature(hash);
}

}
```

Added to Preferences

```
 public boolean getNFC() {
        return mSharedPreferences.getBoolean(Constants.pref.nfc_useability, false);
    }

    public void setNFC(boolean value) {
        SharedPreferences.Editor editor = mSharedPreferences.edit();
        editor.putBoolean(Constants.pref.nfc_useability, value);
        editor.commit();
    }
```

B.2.3 Layout

Added to values/strings.xml

```
<string name="Nfc">NFC</string>
<string name="label_forceNfc">Click button to allow NFC to search for an OpenPGP Card</string>
<string name="touch_nfc">Touch OpenPGP Card</string>
```

Changed in values/strings.xml

```
<string name="app_name">NPG</string>
```

B.2.4 Manifest

Added to AndroidManifest.xml

```
<uses-permission android:name="android.permission.NFC"/>

<activity
android:name=".NFC"
android:label="@string/title_keyServerPreference"
android:configChanges="keyboardHidden|orientation|keyboard"/>
```

Changed in AndroidManifest.xml

```
<string name="app_name">NPG</string>
```