Radboud University Nijmegen

Master Thesis

Using Mobile Phones for Public Transport Payment

Author:
François Kooman

Supervisors:
Wouter Teepe, Hendrik Tews

Thesis Number:
613

August 2009

Contents

Introduction **1**

1 Communicating With Phones **3**

1.1 NFC 4
1.2 Hardware 4
1.3 NFCIP 5
1.4 Communication 8
1.5 First approach 8
1.6 Second approach 10
1.6.1 Protocols 14
1.6.2 Initiator Send 14
1.6.3 Initiator sendBlock 14
1.6.4 Initiator Receive 15
1.6.5 Initiator receiveBlock 15
1.6.6 Initiator endProcedure 18
1.6.7 Target Receive 18
1.6.8 Target receiveBlock 19
1.6.9 Target Send 19
1.6.10 Target sendBlock 22
1.6.11 Target endProcedure 22
1.6.12 Implementation Notes 22
1.6.13 Testing 25
1.7 Third approach 28
1.8 Conclusion 29

2 Security Analysis **31**

2.1 Capabilities . . . 32

2.1.1 Phone Identification . . . 32

2.1.2 Data Storage . . . 34

2.1.3 Linking a Phone To an Individual . . . 36

2.1.4 Generic Capabilities . . . 36

2.2 Actors . . . 37

2.3 Actors and Capabilities . . . 38

2.3.1 Customer . . . 38

2.3.2 Friend . . . 38

2.3.3 Transport Company . . . 38

2.3.4 PNO . . . 39

2.3.5 Eve and Mallory . . . 39

2.4 Threats . . . 39

2.4.1 Key and Application Extraction . . . 39

2.4.2 Unique Identification of the Phone . . . 39

2.4.3 Inability To Link a Phone to a Customer . . . 40

2.4.4 Relay Attack . . . 40

2.4.5 Collaborating Actors . . . 40

2.4.6 Insufficient Customer Trust . . . 40

2.5 Solutions . . . 40

2.5.1 Prevent Key Extraction . . . 40

2.5.2 Prevent Unique Identification of the Phone . . . 41

2.5.3 Linking a Phone to a Customer . . . 41

2.5.4 Relay Attack . . . 41

2.5.5 Trustworthy System . . . 42

2.6 Protocol Design recommendations . . . 42

2.7 Conclusion . . . 43

3 Mobile Programming **45**
3.1 Java Editions . . . 45
3.2 Java ME . . . 46
3.3 API . . . 46
3.3.1 CLDC . . . 47
3.3.2 MIDP . . . 47
3.3.3 Personal Information Management (PIM) API . . . 48
3.3.4 FileConnection API . . . 49
3.3.5 Record Management Store . . . 50
3.4 Secure Element . . . 51
3.4.1 Contactless Communication API . . . 52
3.4.2 Contactless Communication Extension API . . . 53
3.4.3 Accessing the Secure Element With Card Reader . . . 54
3.4.4 Unlocking the Secure Element . . . 54

4 MIDlet Suite Construction **57**
4.1 Compilation . . . 58
4.2 Preverification . . . 58
4.3 The JAR Manifest . . . 58
4.4 Packaging . . . 59
4.5 Application Descriptor . . . 59
4.5.1 Permissions . . . 60
4.5.2 Push Registry . . . 61
4.5.3 Signing . . . 62
4.5.4 Application Descriptor File . . . 63
4.6 Installation on mobile phone . . . 64
4.7 Development Tools . . . 64
4.7.1 Sun Java ME SDK . . . 64
4.7.2 Nokia SDK . . . 65
4.7.3 NetBeans . . . 65
4.7.4 Eclipse . . . 65
4.7.5 Building MIDlet Suites . . . 65
4.7.6 Emulators . . . 66

5 The OV-Chip 2.0 Software **67**
5.1 Porting Existing Software 68
5.1.1 Setup 68
5.1.2 Analysis of Data Communication 69
5.1.3 Communication 70
5.1.4 Code 72
5.2 Performance Comparison 72
5.3 Alternative Implementation 75
5.3.1 Remote Method Invocation 75
5.3.2 Design 76
5.3.3 Communication 77
5.3.4 Protocol Implementation 78

6 The Protocol **79**
6.1 Attribute Proving 80
6.1.1 Setting up RSA 80
6.1.2 Setting up the Attribute Expression 80
6.1.3 Proving knowledge of the RSA representation 81
6.1.4 Disclosing certain attributes and proving them 81
6.1.5 Selective Disclosure example 82
6.1.6 Setting up RSA and the System Parameters 82
6.1.7 Disclosing An Attribute 83

7 Nokia S40 Security **85**

8 Future Research **87**

9 Conclusion **89**

Bibliography **91**

A Padding Scheme **93**

B Hello World MIDlet **95**

C Obtaining File List From Phone **97**

D Creating a Java Key Store **99**

E OV-Chip 2.0 Porting Code **101**

E.1 DataChannel.java 101
E.2 Host_protocol.java 102
E.3 Card_protocol.java 102
E.4 Installation Arguments 103

F Java Card Example **105**

F.1 Java Card 106
F.2 Java SE 109

G Gjokii Design **113**

List of Figures

1.1 Path from host to radio frames 5
1.2 Path between MIDlet and radio frames 6
1.3 Low level communication between initiator and target hosts 7
1.4 Initiator send 15
1.5 Initiator sendBlock 16
1.6 Initiator receive 17
1.7 Initiator receiveBlock 18
1.8 Initiator endProcedure 19
1.9 Target receive 20
1.10 Target receiveBlock 21
1.11 Target send 22
1.12 Target sendBlock 23
1.13 Target endProcedure 24
1.14 ACS ACR122 as initiator, phones as target 26
1.15 Phones as initiator, ACS ACR122 as target 27

2.1 Information leaking channels on the Nokia 6131 NFC 33
2.2 Overview of a relay attack 37

3.1 Secure Element Communication (using MIDlet or card reader) 52

5.1 Performance Vector Exponentiation with 5 bases on Java Card and Nokia phones 73

List of Tables

1.1 Chaining indicator 11
1.2 Custom chaining example 12
1.3 Example of communication between the initiator and target 13
1.4 Failures from point of view of the target 13
1.5 NFCIP performance and reliability, showing: average transmission time for a run ($\bar{x}$), sample standard deviation (σ) and number of connection resets during the entire test (R). 26
1.6 Example of communication between the initiator and target using a dummy message, compare this to the situation in table 1.3. 28

2.1 Actors and their capabilities 38

4.1 Some MIDlet suite permissions 61

5.1 Performance comparison between BigInteger libraries 75

A.1 Padding scheme 93
A.2 Some corner case paddings 93

Acknowledgments

I would like to thank my supervisors, Wouter Teepe and Hendrik Tews, for their help. Erik Poll, for arranging the (NFC) phones, card readers and code signing certificate. Roel Verdult for helping me with the NFC/RFID communication analysis.

Abstract

We analyzed the mobile phone Java ME platform in detail and looked at how to use it for (privacy friendly) public transport payment with the OV-Chip 2.0 protocol. We describe: communication, programming, security, the public transport payment protocol, porting the existing OV-Chip 2.0 code, performance testing, Java Card (Secure Element) programming, Nokia S40 platform security. Using a mobile phone for public transport payment with the proposed privacy features will not provide any (performance) benefit over the solution using the Java Card platform.

This paragraph was removed from the public version of this document due to security issues we found on the Nokia S40 platform.

Introduction

Traditionally one buys paper tickets for traveling by public transport. The tickets are bought at ticket offices or ticket machines. During travels these tickets can be requested to verify that one is eligible to travel using a particular bus or train.

In the Netherlands there were originally two types of tickets: one for bus, tram, subway (the "strippenkaart") and one specific for the train. The "strippenkaart" was valid in the entire country and the train ticket in all trains, specific for a certain route indicated on the ticket.

Due to the emergence of private parties interested in providing transport and the "privatization" wave making this possible, the "strippenkaart" was not effective any longer in dividing profits between the companies involved. The "strippenkaart" is bought once for a fixed price and can then be used everywhere. It is not linked to a specific company. This is untraceable and so there is no (digital) usage trail.

To solve this issue the idea of providing a nation wide digital chip card for public transport was born, the OV-Chipkaart[1]. In addition to making it possible to split profits the traveler now has the advantage of only having to carry one card at all times.

However, there are some issues with the OV-Chipkaart, two of which are:

1. The traveler does not have the privacy he had with the "strippenkaart"[2].
2. The used mifare smart cards are relatively easy to manipulate and clone [3].

The Digital Security department of the Radboud University decided to design and implement an OV-Chip 2.0 protocol. The idea is to implement a privacy friendly protocol based on Stefan A. Brands' *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy* [1] on a Java Card smart card. This would make it possible to resolve both the privacy and security issues mentioned above.

The protocol relies on being able to perform a fast modulo exponentiation of the form:

$$g_1^{a_1} g_2^{a_2} g_3^{a_3} g_4^{a_4} g_5^{a_5} \pmod{n}$$

[1] See http://www.ov-chipkaart.nl.

[2] It is possible to buy an "anonymous" ticket for single trips, however this makes it impossible to benefit from subscription services which can lead to missing out on substantial discounts on travels and is in general more expensive.

where g_i and n should at the moment be around 1200 bits, and a_i around 160 bits.

Current Java Card smart cards do not have a hardware accelerated `BigInteger` API which would be required to efficiently do this. Performing this in one's own non accelerated `BigInteger` implementation is unrealistic due to the limited processing power available on the Java Card platform. By using the RSA API on Java Card one can use the RSA implementation to help with calculating modulo exponents. This increases the efficiency considerably, but this is still not enough for real world applications [5].

In this thesis we will evaluate the use of a mobile phone as a replacement for the Java Card implementation. Due to a faster processor we expect the performance to be better than with a Java Card smart card.

The main question we try to answer is:

> **"Can a mobile phone be used to efficiently implement the** OV-CHIP 2.0 **protocol?"**

We will look at the implications of this question. This will lead us to these sub questions:

1. How will the phone communicate with the public transport gates, recharge units and conductors?
2. Where will we securely store the private key used by the protocol?
3. What is the performance of the protocol on the phone compared to the (optimized) Java Card implementation?
4. How do we create software to run on the mobile phone?
5. How can we port the existing OV-CHIP 2.0 software to the mobile phone?

We try to answer these questions in the following chapters.

This paragraph was removed from the public version of this document due to security issues we found on the Nokia S40 platform.

Chapter 1

Communicating With Phones

Introduction

Modern mobile phones can have a wide range of communication channels with the outside world. To list a few:

- GSM
  - SMS
- GPRS
- UMTS
- HDSPA
- Wifi
- Bluetooth
- Infrared
- NFC

We are looking to run a security protocol over this communication channel so the requirements for this would be:

- Fast connection setup
- Bidirectional
- Reliable

- Convenient
- Limited range of operation to discourage eavesdropping, man in the middle attacks and denial of service as much as possible
- Minimal, or no configuration required for the traveler
- Accessible by applications that can be installed on the phone

These requirements limit the possible communication channels to NFC and possibly infrared. Infrared has a disadvantage in that it requires "line of sight" for it to work and can work over long(er) distances than NFC. Thus we will focus on NFC here.

1.1 NFC

The underlying hardware communication protocols of NFC are compatible with existing RFID standards. Radio frames used by NFC technology are the same as the frames used by traditional RFID devices. A NFC tag is the same as a RFID tag, with some data structures are standardized to make it possible to store a URL or a contact on a tag. More advanced implementations, as for example embedded in NFC readers or NFC capable phones have the ability to act as a tag themselves or be a NFC reader depending on the application. One specific mode of operation is the so called NFCIP mode which is a protocol designed for data exchange between two NFC capable devices like mobile phones or NFC readers. NFCIP is described in the ECMA-340 standard [2]. Sometimes NFCIP is also called DEP (data exchange protocol) or NFC P2P (Peer 2 Peer).

This document analyzes current NFCIP implementations in a NFC reader and a NFC phone and designs a protocol to accomplish reliable communication between those NFC devices.

1.2 Hardware

The hardware devices for which the reliable communication is implemented are:

- ACS ACR122 reader with firmware ACR122U102 (from now on: NFC reader)
- Nokia 6131 NFC and Nokia 6212 Classic (from now on: NFC phone or just phone)

The NFC reader has a NFC chip designed by NXP (formerly Philips), the PN532. The reader has a USB connector for communicating with a computer as shown in figure 1.1. The phone uses the Nokia S40 software platform which provides Java APIs that can be

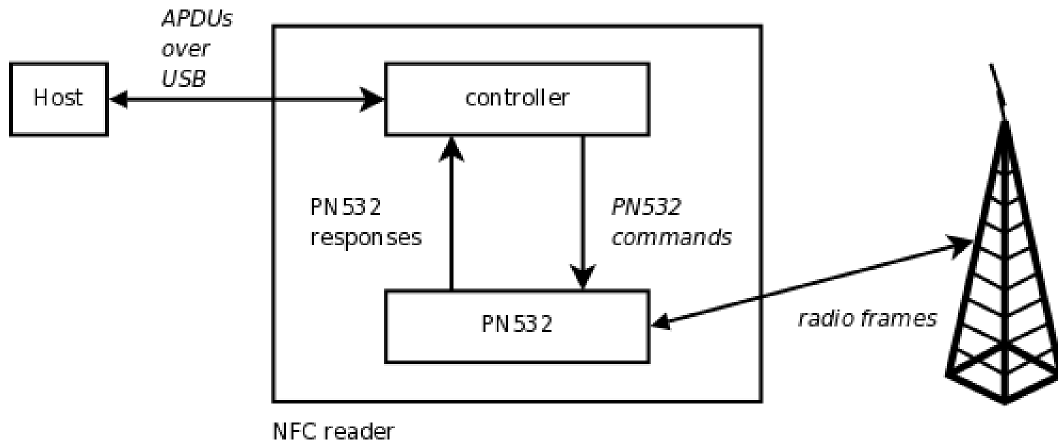

Figure 1.1: Path from host to radio frames

used from mobile Java applications (Java ME MIDlets) running on the phone to talk to the NFC chip. This is shown in figure 1.2.

In case of communicating with the NFC reader the APDUs that are being sent can contain the NFC chip instructions wrapped in 'pseudo APDUs' which gives the host almost full control over the NFC chip (explained below) as opposed to the MIDlet that is restricted by the (simple) Java NFCIP API. This Java API only enables an application to set the mode of operation (initiator or target) and to send and receive data. There is no "low level" control of the NFCIP chip.

In case we talk about the host we talk about the application that is able to send APDUs to the NFC reader. Below we will also describe the situation with two "hosts" to show how the low level communication works. One could also see the phone as a host, but there is no control over the actual commands that are being sent to the NFC chip on the phone, however, the phone will basically use the same commands.

1.3 NFCIP

NFC uses the same underlying technology as RFID as already mentioned before. With NFC there are a few different modes of operation. NFCIP communication can be either active or passive and a device can be either initiator ('reader') or target ('tag'). The initiator is responsible for generating a RF field and starts with sending data. The target waits for an initiator as it needs its RF field before it starts receiving data. Active communication means that both devices (initiator and target) alternately generate an RF field. Passive means that only the initiator generates an RF field.

When the host (the computer that is connected to the NFC reader) is in initiator mode it starts by sending a message to the NFC reader over USB to the USB controller in the NFC

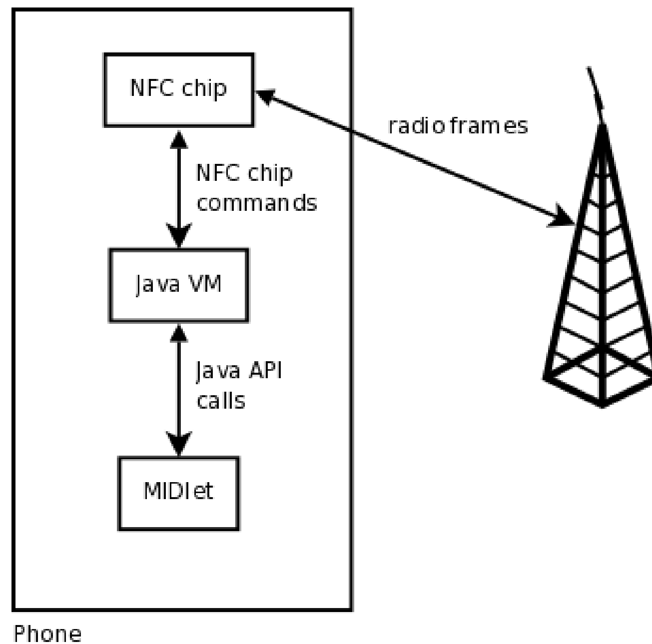

Figure 1.2: Path between MIDlet and radio frames

reader which forwards it to the NFC chip where the data will be prepared for sending to the target using radio frames. At the target side the data needs to be reconstructed from the radio frames and sent to the application.

Pseudo Code 1 Pseudo code for initiator mode

```
BEGIN
SET MODE INITIATOR
// wait for target
DO
  send data
  receive data'
WHILE condition
RELEASE TARGET
END
```

In reality the initiator has actually only one instruction for sending and receiving data (`IN_DATA_EXCHANGE`). The response will be the answer to the send instruction. The target has two instructions (`TG_GET_DATA`, `TG_SET_DATA`). One for receiving the data and one for sending a reply. See pseudo code listing 1 and 2 for a high level overview of the communication. The initiator starts with sending data, the target starts receiving data and then sends a reply back to the initiator. As explained before, the initiator only has one instruction for sending and receiving data so the target has to complete both receiving the data and responding with a reply before the initiator gets a reply to its instruction. This is abstracted

Pseudo Code 2 Pseudo code for target mode

```
BEGIN
SET MODE TARGET
// wait for initiator
DO
  receive data
  send data'
WHILE condition

END
```

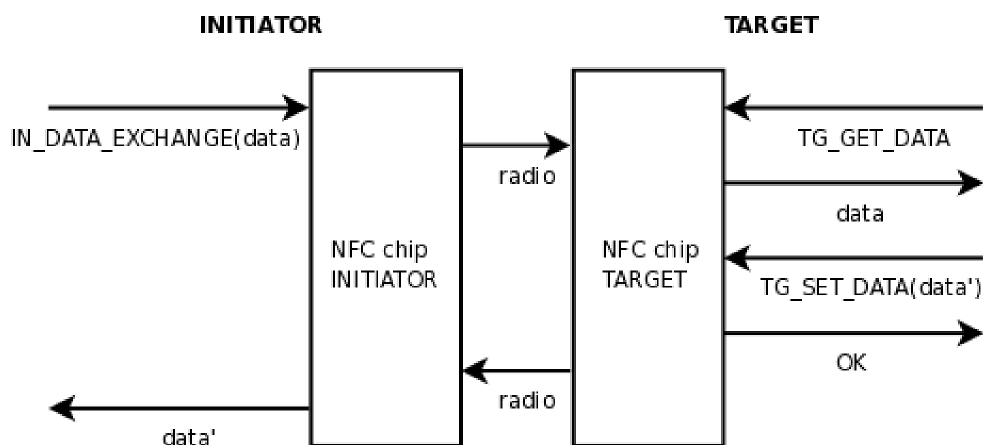

Figure 1.3: Low level communication between initiator and target hosts

away in the phone's NFCIP API and we will create a similar API for communicating with the NFC reader. The receive operation of the initiator is nothing more than passing back the answer to the send instruction by the target. This distinction is important because it shows that the protocol is not symmetric and will require special care when designing a reliable communication API. Figure 1.3 shows the low level communication between the host and the initiator, between initiator and target and between the target and the other host.

It is only possible to send and receive a certain amount of data using one instruction. The NFC chip has limited this to 262 bytes, although the NFC reader will not allow for this as it has restricted this to 252 bytes (in initiator mode) or to 253 bytes (in target mode), so there is no full control of the NFC chip. This is described in more detail in the documentation accompanying this library[1].

[1]See NFCIP Java library documentation at http://nfcip-java.googlecode.com/

1.4 Communication

We try to achieve reliable communication between NFCIP devices. By this we mean:

1. When target and initiator lose contact the transfer of data will be resumed. where it stopped, without any other interaction, when the target moves back in the initiator's range

2. We assume that if (and only if) data arrives, it arrives without error. There is (limited) data checking done on the hardware level using checksums in the radio frames

3. We want to be able to handle data bursts, for example sending a (big) file from the initiator to the target

4. We want to be able to handle small data transfers originating at the initiator and at the target corresponding to the load one expects when running security protocols over them

5. We make sure both the initiator and the target know a (high level) send and receive cycle, possibly consisting of multiple low level send/receive sequences, finishes successfully

6. We focus only on the data layer, not on the higher level protocols, this design only provides for byte array data transfer

In order to send data of arbitrary amounts of data we need to implement a wrapper that takes care of this. There are two ways to approach this problem:

1. Try to be compatible with the NFC phone so it is possible to directly (and only) use the phone API for sending and receiving data;

2. Start from scratch and create a (new) API for both the phone and the NFC readers.

1.5 First approach

We started out with this approach by having the phone be initiator and sending some data to the target (in this case the NFC reader set to target mode) to see what the phone was sending. It turns out the phone requests a passive connection at 106 kbps and uses the DID (Device ID) byte. After this we made the phone send some data, first small blocks, later big blocks. DID adds a byte to every radio frame being transmitted. Radio frames contain a maximum of 60 data bytes, or 59 with DID enabled.

It turns out we receive blocks of maximum size 236 bytes when using `TG_GET_DATA` on the target. So in case the initiator (phone) sends 1000 bytes we receive 4 blocks of size 236 and one last block of size 56 bytes.

We make some observations:

1. When the NFC reader receives data the last block cannot have length between 11 and 22 bytes ($11 <= size <= 22$)
2. When the last block has $length <= 10$ it is added to the previous block so the last block has a size of maximum 250
3. There is no connection between the block size we use for sending data and the block size of receiving data
4. The send method call of the phone is non blocking (only in target mode)
5. The NFC connection is interrupted momentarily (requiring a reset) when the screen turns off for power saving reasons (only in target mode);
6. The Nokia 6131 NFC does not respond to a request for active communication mode by an initiator, only passive mode is supported
7. The Nokia 6212 Classic responds to a request for active communication mode by an initiator, only on 106 kbps, not on 212 or 424 kbps
8. The phone uses built in chaining of data: the 'more data bit' in the status field indicates there is more data coming after the current block

The first one is a problem: the NFC reader enters an unrecoverable state in which it claims the "checksum" of the data is incorrect. To avoid this one has to introduce a padding in the last block to make sure it is at least of length 23. This would have to be implemented on the phone, that is, the phone has to calculate the expected block sizes that will be received by the NFC reader (this is unrelated to the block size that is used for sending as noted before) and add a padding somewhere in between to make sure the target will not receive a last block of $size >= 23$. This problem seems to be a bug in the NFC reader.

As for the third observation: if the initiator sends blocks of size 100 this does not mean the data will be received in blocks of 100, but it will be received in blocks of 240 (or 236 when DID is used), provided there is enough data available.

The fourth observation is also a problem: it means that if after a send method call another call is performed (receive, or close) the transfer that is currently taking place is abruptly terminated.

To solve these issues we came up with some solutions:

1. Make sure that the NFC reader never receives a last blocks of $size < 23$ by creating a padding scheme that extends the amount of data in the last block if necessary
2. We can expect to receive bigger data blocks than the usual size of 236 or 240 bytes
3. We have to successfully predict the receiver's behavior to deal with issue 1 (especially regarding the use of the DID byte)
4. We have to implement some kind of delay to create an artificial blocking mechanism
5. Make sure the transfer is finished before the screen turns off, or deal with connection resets
6. We have to use passive communication mode
7. We have to look at the 'more data bit' of the status byte

Since our goal is to use the phone in target mode all these issues need to be resolved. The first issue is cumbersome and the fourth issue is hard to solve. The first one will involve manipulating the data being sent in such a way to create an effective padding that can be removed by the receiver. We devised a solution for this as described in Appendix A. It introduces significant overhead when running this on the phone, but we have a similar issue in our second approach where the data needs to be split in blocks as well. As for the fourth issue: how long should one wait for the call to finish? There is no feedback as to how to estimate the time one has to wait, there is no indication of the amount of data to expect before it was successfully arrived. This issue was in effect the deal breaker and the reason to come up with our own chaining system (the second approach). Our assumption was that if there is only one 'low level' block each time we can better estimate the time it requires for the operation to complete and we can have a fixed 'blocking' timer, or as it turned out, no blocking at all, but just re-transmits. There is a way to avoid the screen from being turned off but it requires the charger to be connected (and a configuration setting). This is generally not acceptable, but is not a problem during communication testing.

1.6 Second approach

It would be more valuable to use the first approach as it would make it more easy to program the phone, unfortunately that is not possible or efficient anyway as we need to make sure the block size of the last block is not in the range described above to avoid the errors in the NFC reader and we also need to take care of restoring the connection when it breaks, so we need a wrapper on the phone anyway. What follows now is the design of the wrapper that takes care of all the issues by design. We will give some flowcharts indicating exactly what should be done in a particular situation and how to deal with transmission errors both from the side of the initiator and target. Before this we will show some design decisions and give an example.

Suppose the initiator wants to send data to the target of size 1000 bytes. This exceeds the capacity of a single `IN_DATA_EXCHANGE` command, so it needs to be split into multiple parts. We define a maximum block size of 240 bytes. This is a safe limit as it is well below the maximum of 262 bytes of the PN532 chip and below the maximum of the ACR122 (252 bytes) and equal to the receiving block size when using the first approach. We will implement a custom chaining system as opposed to the built in chaining used in the first approach. We assume now here that by using small enough data blocks (i.e.: blocks of size 240, so 239 bytes left for the actual data) we avoid triggering this chaining in the phone. The first byte of each data block (of size 240 bytes) will be the chaining indicator as shown in table 1.1.

Bit	7	6	5	4	3	2	1	0
Description	NU	NU	NU	*dummy block*	*empty block*	*end block*	*block number*	*chaining*

Table 1.1: Chaining indicator

Bit 0: there is another block coming after this one belonging to the same transfer (the same high level send/receive). This bit can be set by both the initiator and the target.

Bit 1: indicates the block number. This bit can be set by both the initiator and the target.

Bit 2: indicate that this block is the last block. This bit can be set by both the initiator and the target.

Bit 3: indicate that this is an empty block. This bit can be set by both the initiator and the target.

Bit 4: indicate that this is a dummy block. This bit can be set by both the initiator and the target.

We use bit 3 to avoid sending ‘empty’ packets so a packet is never supposed to be empty. This is necessary to avoid the problem of the phone method call(s) being non-blocking. When the received data buffer is now empty we know their was a problem and restart the connection and try again.

We use bit 4 to indicate that this is a dummy block. This is used by the `FAKE_INITIATOR` and `FAKE_TARGET` mode, see section 1.7.

The other bits of the chaining indicator are not used.

End blocks and empty blocks do not have a number because the protocol was designed in such a way that it does not matter when the same empty or end block is sent/received multiple times after one another. The number is only required to distinguish actual data

on determining whether or not the data should be concatenated with the data received so far.

There are 239 bytes of the 240 bytes left for the actual data. The initiator splits the data in blocks of 239 bytes, which results in the case of sending 1000 bytes to 4 blocks of 239 and one block of 44 bytes. For the first 4 blocks bit 0 is set, for the last one it is cleared. The block number is alternating set to 0 and 1 to deal with failure 3 from table 1.4.

As a simple example, see table 1.2, we assume here the block size is 3, we want to send 7 bytes from initiator to target. The initiator splits the blocks and sets the corresponding chaining indicator. The actual data to send is for example `0x80 0x81 0x82 0x83 0x84 0x85 0x86`.

Block	**Chaining indicator**	**Data block**
1	000*00001*	`0x01 0x80 0x81`
2	000*00011*	`0x03 0x82 0x83`
3	000*00001*	`0x01 0x84 0x85`
4	000*00010*	`0x02 0x86`

Table 1.2: Custom chaining example

Table 1.3 shows a complete protocol run. In this example we keep the block size 3 and send 7 bytes from the initiator to the target and receive the same amount back from the target (the echo test). This also shows the use of the end block and the empty block. The need for the end block will be explained later.

To make the communication reliable we have to devise a system that is robust against the failure of any of the instructions used. So we need to deal with failure of `IN_DATA_EXCHANGE`, `TG_GET_DATA` and `TG_SET_DATA`. What can exactly go wrong on the target side?

1. `TG_GET_DATA` can fail because the initiator lost the target or the data was corrupted;
2. `TG_SET_DATA` can fail because the initiator lost the target and thus this instruction is not available anymore;
3. `TG_SET_DATA` can succeed from the point of view of the target, but the transmission never arrived at the initiator, there is no way to tell this from the perspective of the target;

These failures of the target all result in the failure of `IN_DATA_EXCHANGE` at the initiator. The initiator however has no way of knowing which of these errors occurred so it can only assume the 'worst' and that is that `TG_GET_DATA` failed and resend the data. The target has to deal with this as it may receive the data from `IN_DATA_EXCHANGE` twice. Hence the block numbering. Table 1.4 shows the possible scenarios.

Initiator		**Data**		**Target**	
`IN_DATA_EXCHANGE`		`0x01 0x80 0x81`	→	`TG_GET_DATA`	
	←	`0x08`		`TG_SET_DATA`	`emptyBlock`
`IN_DATA_EXCHANGE`		`0x03 0x82 0x83`	→	`TG_GET_DATA`	
	←	`0x08`		`TG_SET_DATA`	`emptyBlock`
`IN_DATA_EXCHANGE`		`0x01 0x84 0x85`	→	`TG_GET_DATA`	
	←	`0x08`		`TG_SET_DATA`	`emptyBlock`
`IN_DATA_EXCHANGE`		`0x02 0x86`	→	`TG_GET_DATA`	
	←	`0x01 0x80 0x81`		`TG_SET_DATA`	
`IN_DATA_EXCHANGE`		`0x08`	→	`TG_GET_DATA`	`emptyBlock`
	←	`0x03 0x82 0x83`		`TG_SET_DATA`	
`IN_DATA_EXCHANGE`		`0x08`	→	`TG_GET_DATA`	`emptyBlock`
	←	`0x01 0x84 0x85`		`TG_SET_DATA`	
`IN_DATA_EXCHANGE`		`0x08`	→	`TG_GET_DATA`	`emptyBlock`
	←	`0x02 0x86`		`TG_SET_DATA`	
`IN_DATA_EXCHANGE`		`0x04`	→	`TG_GET_DATA`	`endBlock`
	←	`0x04`		`TG_SET_DATA`	`endBlock`

Table 1.3: Example of communication between the initiator and target

Successful run	**Failure (1)**	**Failure (2)**	**Failure (3)**
GET d1	GET d1	GET d1	GET d1
SET d1'	SET d1'	SET d1'	SET d1'
GET d2	**radio interrupted**	GET d2	GET d2
SET d2'	GET d2 **FAIL**	**radio interrupted**	SET d2'
GET d3	**reset**	SET d2' **FAIL**	**radio interrupted** *initiator did not get d2', but target does not know this*
SET d3'	GET d2	**reset**	GET d3 **FAIL**
	SET d2'	GET d2	**reset**
	GET d3	SET d2'	GET d2 *unexpected, expected d3, got d2 instead*
	SET d3'	GET d3	SET d2'
		SET d3'	GET d3
			SET d3'

Table 1.4: Failures from point of view of the target

From the point of view of the target Failure 1 and 3 are the same (`TG_GET_DATA` fails) if no extra precautions are taken. With failure 3 we need to have a buffer with data that was sent before as we might need it again in the next round (we need to save d2' in this case). This also introduces the need for an `endBlock` as one can expect that the program ends when the target sends the last data (here d3') back and then exits the program. The initiator never received that response so requests it again, but the target is already done and terminated the connection. With an end block the target can now only consider that the communication is over when the end block was received. The target will send back the end block after which the initiator is sure that the target has all the data.

1.6.1 Protocols

Now we show the flowcharts of the protocol. We split the problem into multiple layers. We have the high level send and receive functions and the lower level `sendBlock` and `receiveBlock` functions. The high level send and receive are responsible for splitting the blocks into blocks and merging the blocks again. The `sendBlock` and `receiveBlock` functions are taking care of the low level sending and receiving. We describe here the situation for two NFC readers, although the situation is almost the same in the case of the phone. The only difference is the call to `TG_GET_DATA` and `TG_SET_DATA` and setting the NFC chip mode are replaced by the JSR 257 extension API calls on the phone.

1.6.2 Initiator Send

The initiator starts with splitting the data in equally sized blocks. This is demonstrated in table 1.2. Then for each block the `sendBlock` function is called. See the flow chart in figure 1.4.

1.6.3 Initiator sendBlock

This function looks at the chaining indicator, the first byte, of the block it is supposed to send. In case the chaining bit is set there is more data coming. Then the response to `IN_DATA_EXCHANGE` is the `emptyBlock` which we do not have to store. If the sending is not successful it is tried again indefinitely. A successful transmission of the last block receives already the first block of the response from the target. There may only be one block as a response, or more, but that is taken care of in the initiator receive function. In case the block was sent successfully the control is returned to the high level send function to optionally send the next block. See the flow chart in figure 1.5.

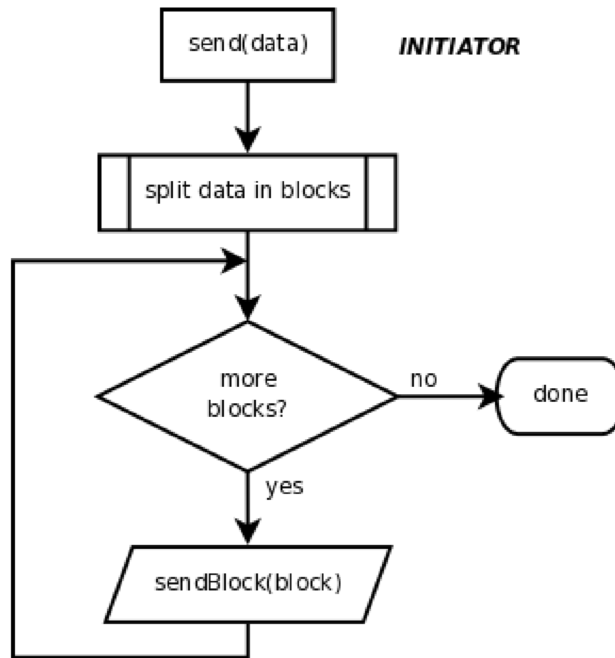

Figure 1.4: Initiator send

1.6.4 Initiator Receive

When the initiator receives data it verifies whether or not the block received is the expected block. The first block to expect has the block number bit cleared (i.e.: the block number 0). When there are more blocks after the first one, the loop is entered in which more data is requested and checked whether it was what we expected. If that is the case we concatenate it to the 'data' variable already containing the first block (and data from previous loops). We raise the expected number by 1, modulo 2, so it alternates between 0 and 1. When the last block was received we call `endProcedure` to take care of the `endBlock`.

We check here whether or not the block is the expected one because this seems the most logical place to check it as opposed to the target receive where it makes more sense to check in the `receiveBlock` function. See the flow chart in figure 1.6.

1.6.5 Initiator receiveBlock

The `receiveBlock` function has no actual low level communication. In initiator mode all communication is handled by the `sendBlock` function. The `sendBlock` function receives the data from the target and puts the (first) response in the "response" variable. This function takes care of calling `sendBlock` with `emptyBlock` as parameter to request more data if there is more data expected. It returns the original "response" variable from the

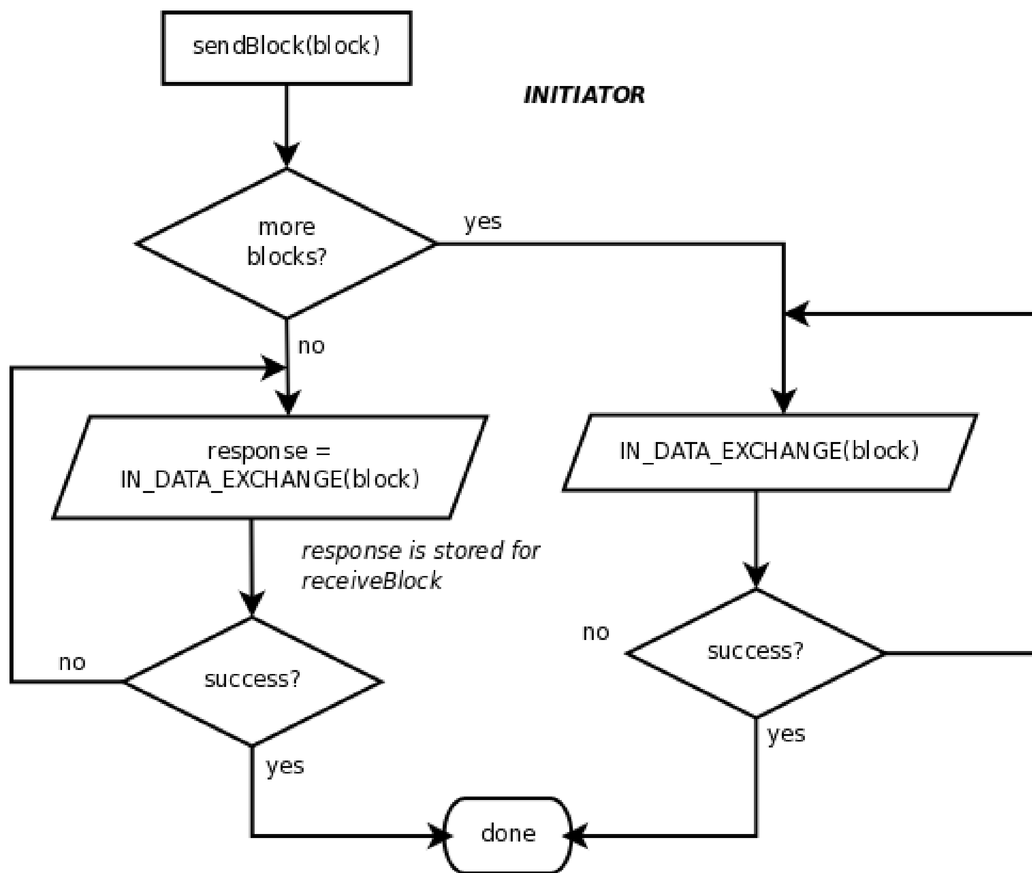

Figure 1.5: Initiator sendBlock

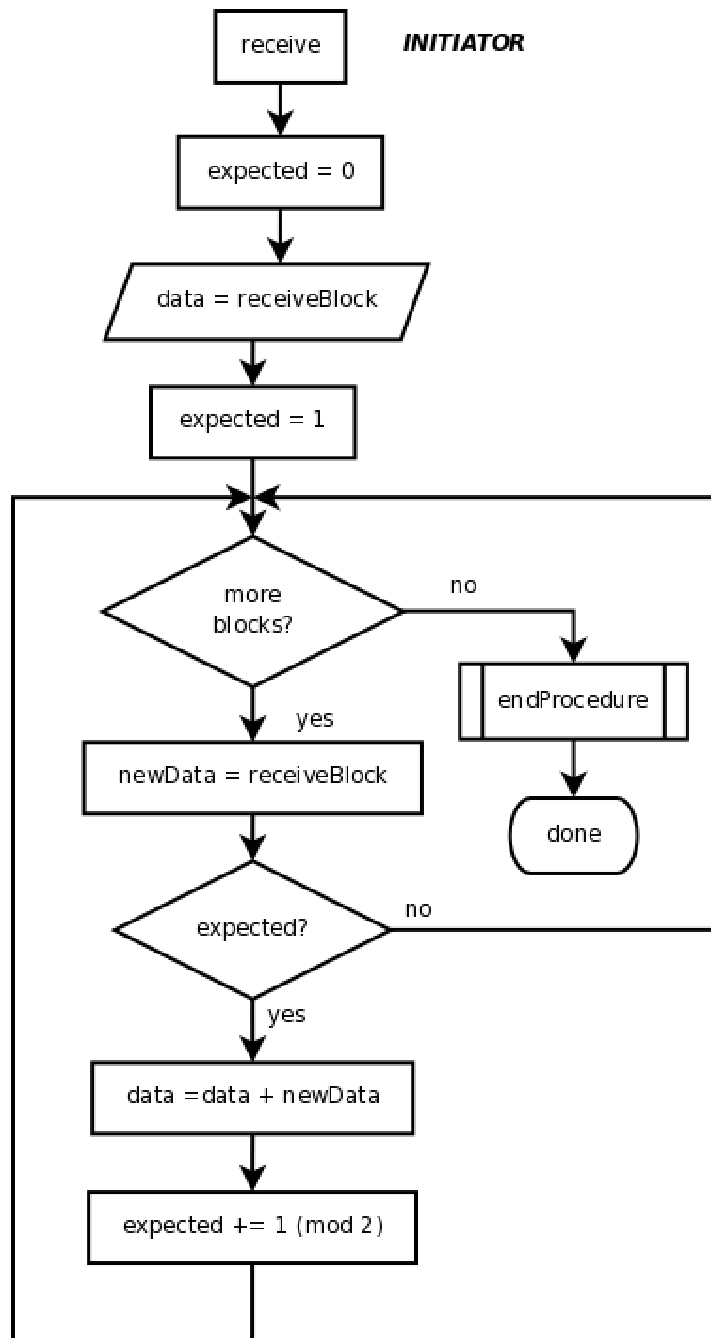

Figure 1.6: Initiator receive

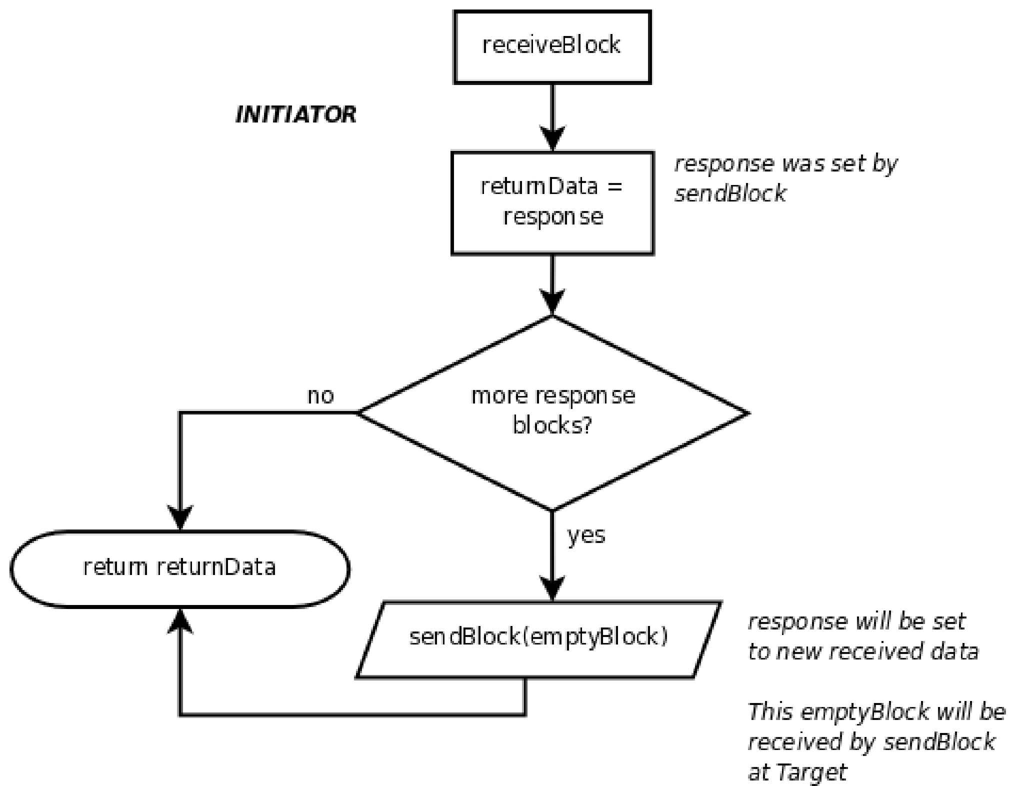

Figure 1.7: Initiator receiveBlock

previous call to `sendBlock` and sets the result of the new request for more data as the new response variable for the next iteration. See the flow chart in figure 1.7.

1.6.6 Initiator endProcedure

The `endProcedure` is very simple. The initiator can always be sure that when the `sendBlock` function succeeded the data was successfully received. So when the initiator sends the `endBlock` and it succeeds it can be sure that the target received it successfully. This terminates the high level send/receive cycle. See the flow chart in figure 1.8.

1.6.7 Target Receive

It is mostly equal to the receive of the initiator except there is no call to `endProcedure` here because data needs to be send back to the initiator first. See the flow chart in figure 1.9.

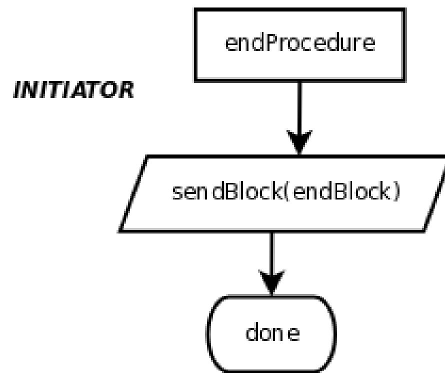

Figure 1.8: Initiator endProcedure

1.6.8 Target receiveBlock

This function has to deal with a possible `endBlock` from the previous high level send/receive cycle. In case the `TG_SET_DATA` of the last low level block succeeded the program can move on to the next high level receive. So in case an `endBlock` is received one sends back an `endBlock` and requests data again.

If an `emptyBlock` is received this was because of the target `sendBlock` receiving a request for more data from the initiator, the contents are not important so returning nothing will suffice.

It is also checked whether or not this is the expected block. Like in the case with the `endBlock` we can receive a block from the previous initiator `sendBlock`. If the block is not expected we send back `oldData` (because the initiator did not receive the previous reply) and request the next block which hopefully is the block we expect.

As the target starts for the first time with receiving data, `oldData` is not set yet (or in the other case: set to the old value from the previous cycle). We assume one cannot end up receiving an unexpected block as the first block because the initiator will make sure that the first blocks arrives. See the flow chart in figure 1.10.

1.6.9 Target Send

The send function for the target is the same as the one for the initiator only here the call to `endProcedure` is included as the target is wants to receive the `endBlock` when it is done sending the last reply. See the flow chart in figure 1.11.

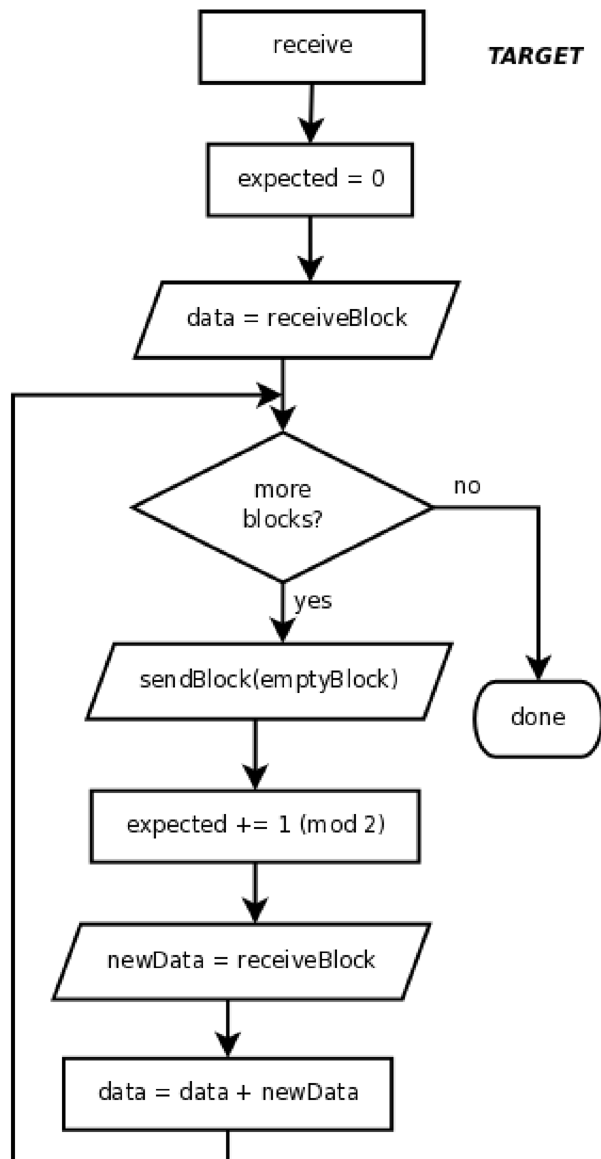

Figure 1.9: Target receive

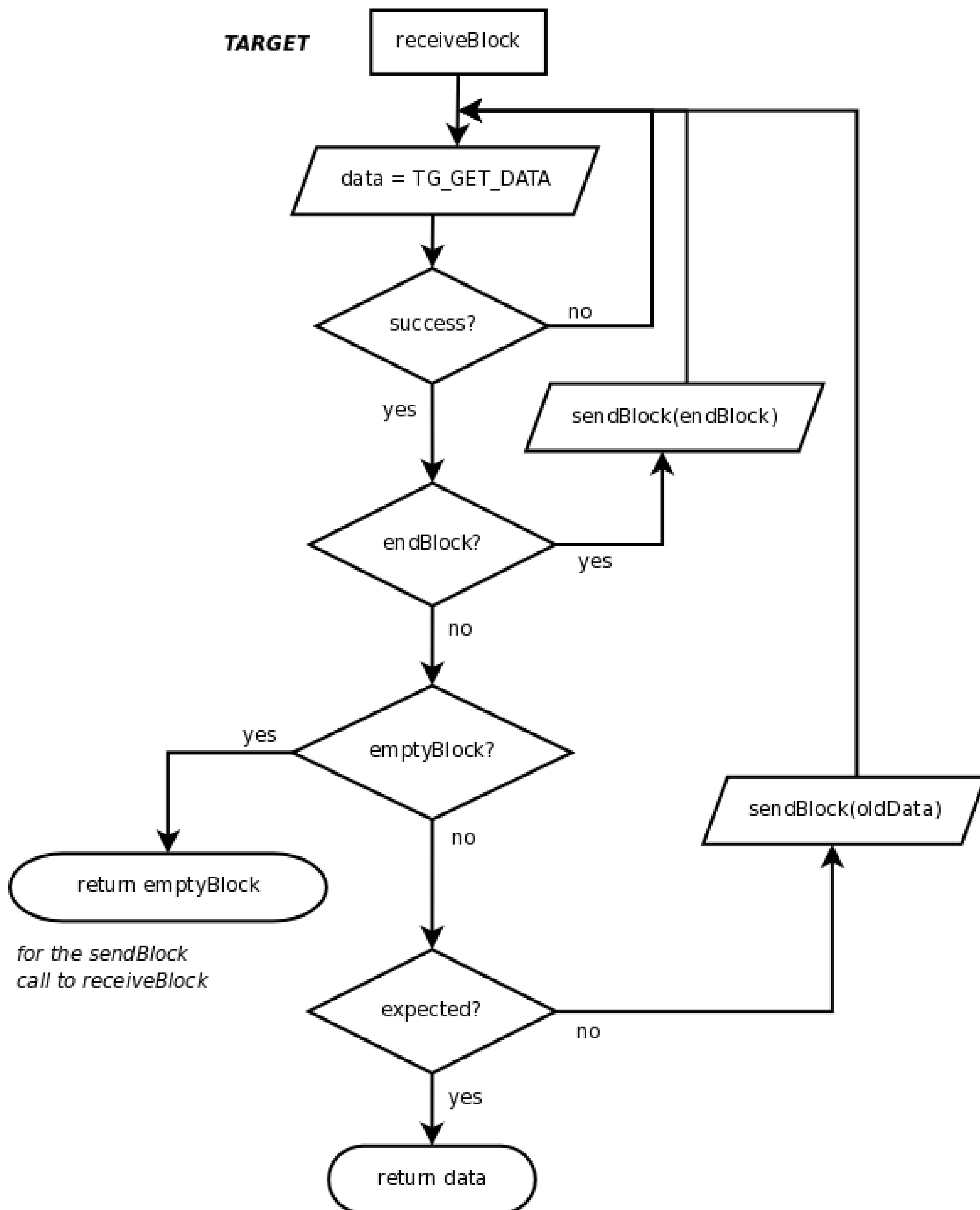

Figure 1.10: Target receiveBlock

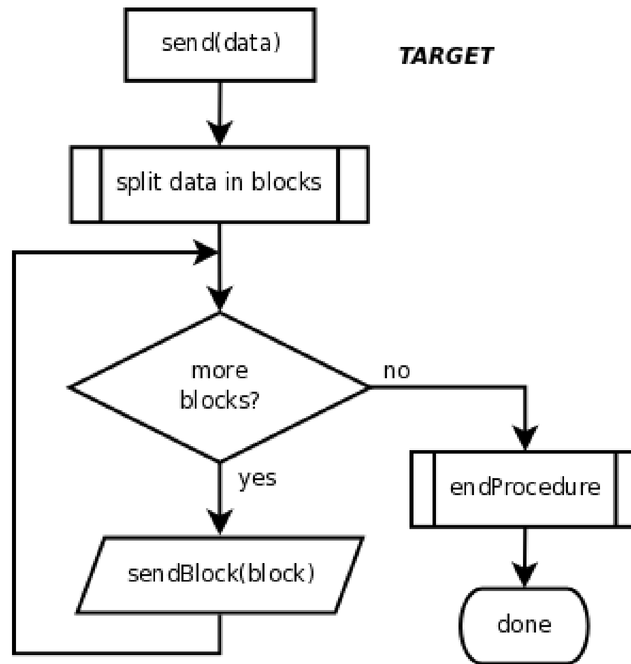

Figure 1.11: Target send

1.6.10 Target sendBlock

When the target sends a block it needs to preserve the data for when the initiator wants to get this block again (when a wrong block number is received). This is necessary to deal with failure 3 in table 1.4. See the flow chart in figure 1.12.

1.6.11 Target endProcedure

This function has to deal with not receiving the `endBlock` from the initiator when the target already expects this. In case something else, maybe the last data block, is received `oldData` needs to be sent back to the initiator again. See the flow chart in figure 1.13.

1.6.12 Implementation Notes

In order to create a successful implementation one should consider the following:

- A connection reset is not shown in the flowcharts. One should reset the connection (i.e.: set the mode again to initiator or target) and then continue when a low level send or receive is not successful

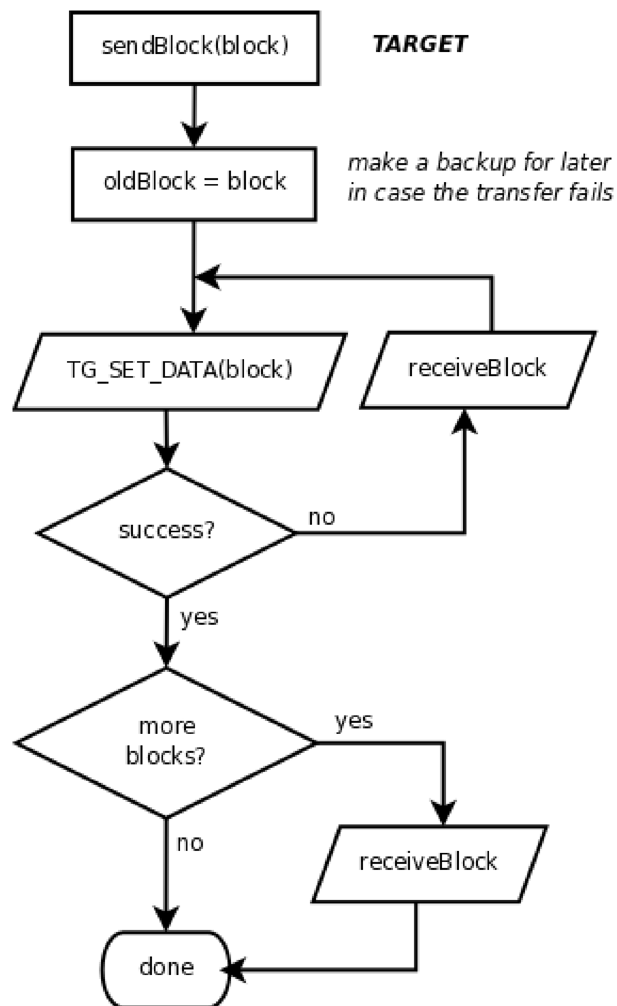

Figure 1.12: Target sendBlock

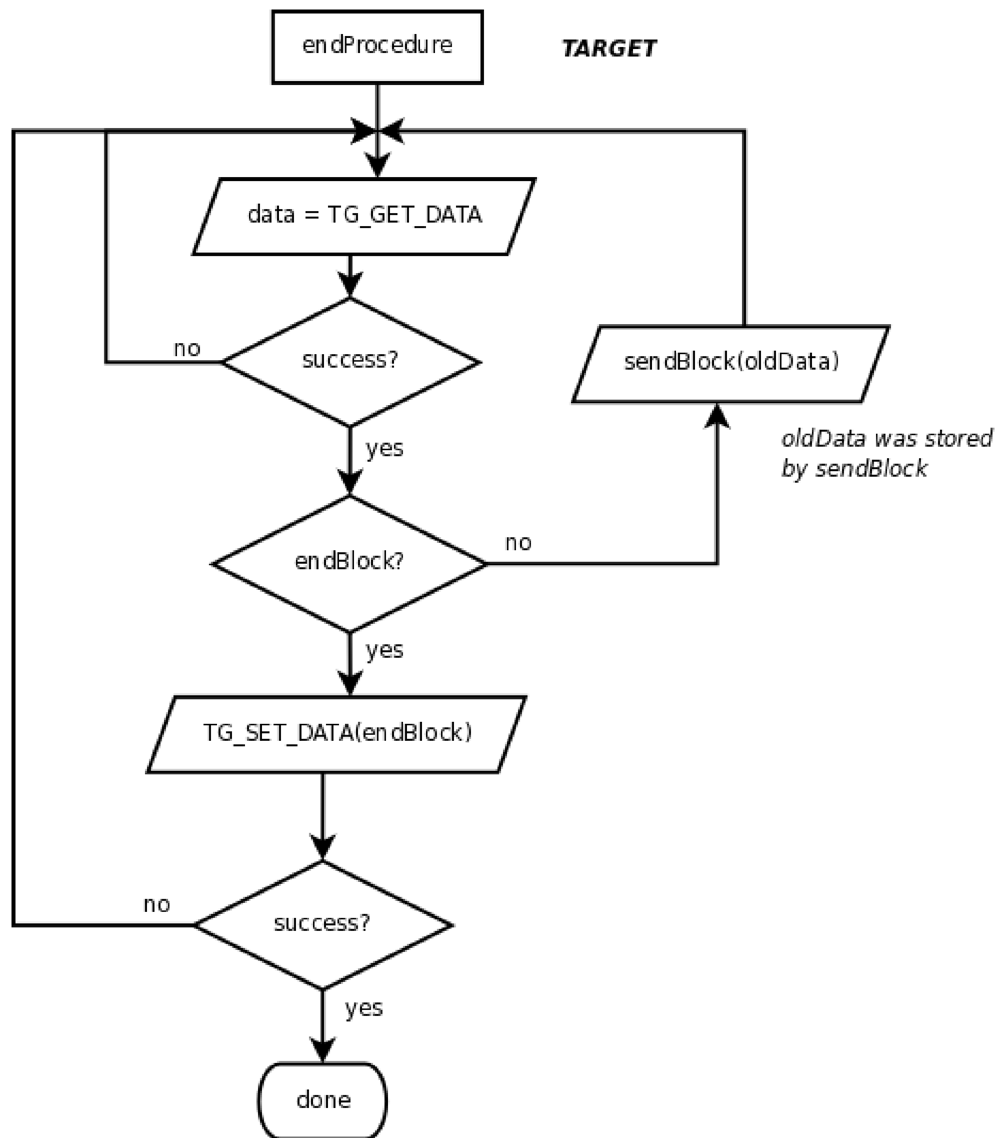

Figure 1.13: Target endProcedure

- In the flow charts not everything is checked, only the most crucial things. For example one should check in the case of the initiator `sendBlock` whether or not the data returned in the case of the send action where it is not the last block that the return value is actually `emptyBlock`
- When a low level send or receive is performed the flowchart basically indicates that one should retry to send forever. It may be reasonable to implement some limit for this
- The target `receiveBlock` function checks whether or not the block is expected instead of the high level receive as in the case of the initiator
- The initiator and target receive functions can be merged into one, there would be one conditional for the `endProcedure` that in case of the target needs to be executed after the send, and in case of the target after the receive

1.6.13 Testing

In order to test the communication between initiator and target we created an "echo test" application where the initiator sends some data to the target and expects the same back. The initiator and target both check the data they receive (they know what block number to expect) to see whether this matches.

We test sending a fixed amount of data, in this case 1000 bytes from initiator to target in multiple rounds. We chose multiple rounds to be able to get a meaningful average, see table 1.5. We chose 1000 bytes because it requires splitting data in multiple blocks of maximum size 240, which is a realistic work load for the security protocols that will use this connection.

We tested both the situation where the phones are target, and where the phones are initiator. Figure 1.14 shows the ACS ACR122 as initiator and the phones as target. Figure 1.15 shows the ACS ACR122 as target and the phones as initiator.

As can be seen from table and the figures the phone is very unreliable in the situation where it is target. Especially the Nokia 6212 Classic is bad. However, if the phones are the initiator the connection is very reliable. There are some connection resets with the Nokia 6131 NFC, but those can most likely be explained by the weak antenna, testing the Nokia 6131 NFC in initiator mode required keeping the phone very close to the NFC reader.

The connection resets can be explained by a problem in the phone's NFC operation. The `receive` method call throws a `java.io.IOException: No data in response` on the Nokia 6212 Classic and succeeds but returns `null` instead of the expected data on the Nokia 6131 NFC. The designed communication layer deals with these problems and makes it possible for the data communication to succeed anyway. It must however be noted that these connection reset problems are not acceptable for our intended purpose to run a security protocol over this connection. We came up with a solution for this in section 1.7.

Initiator / Target	ACS ACR122	Nokia 6131 NFC	Nokia 6212 Classic
ACS ACR122	$\bar{x}$: 1312 ms σ: 50 ms R: 0	$\bar{x}$: 3509 ms σ: 1935 ms R: 41	$\bar{x}$: 6810 ms σ: 1722 ms R: 200
Nokia 6131 NFC	$\bar{x}$: 1558 ms σ: 275 ms R: 11	X	X
Nokia 6212 Classic	$\bar{x}$: 1511 ms σ: 12 ms R: 0	X	X

Table 1.5: NFCIP performance and reliability, showing: average transmission time for a run ($\bar{x}$), sample standard deviation (σ) and number of connection resets during the entire test (R).

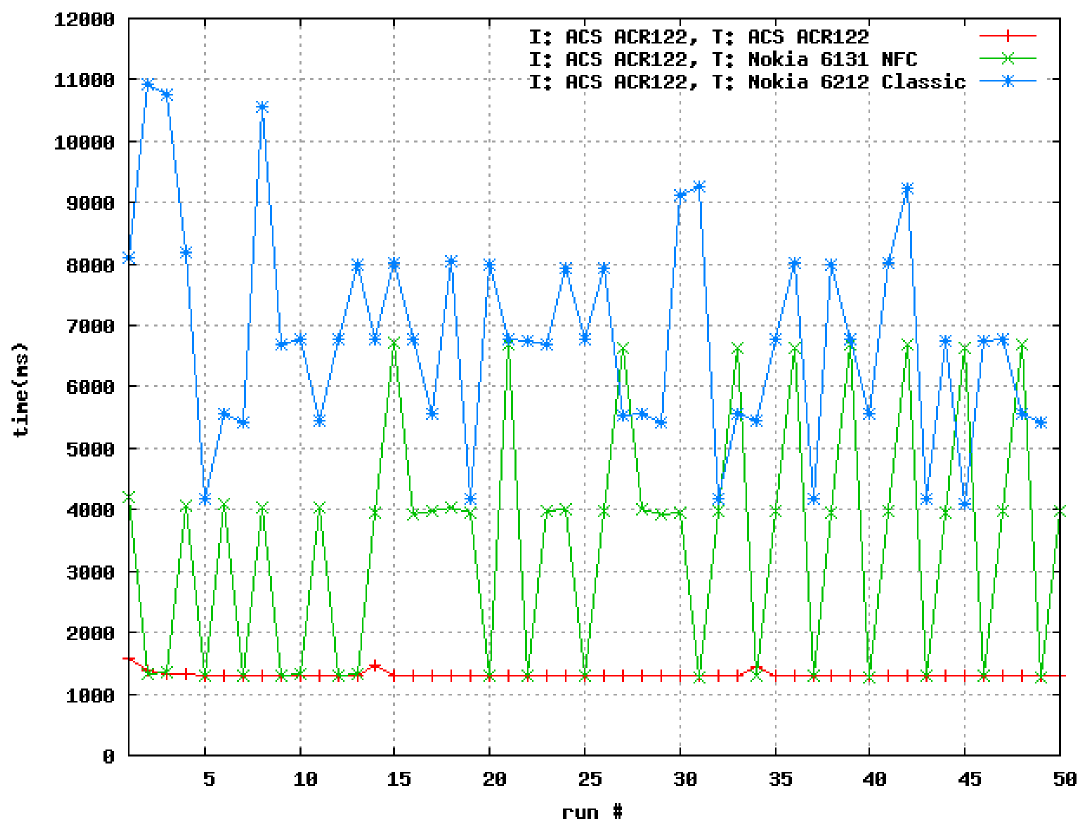

Figure 1.14: ACS ACR122 as initiator, phones as target

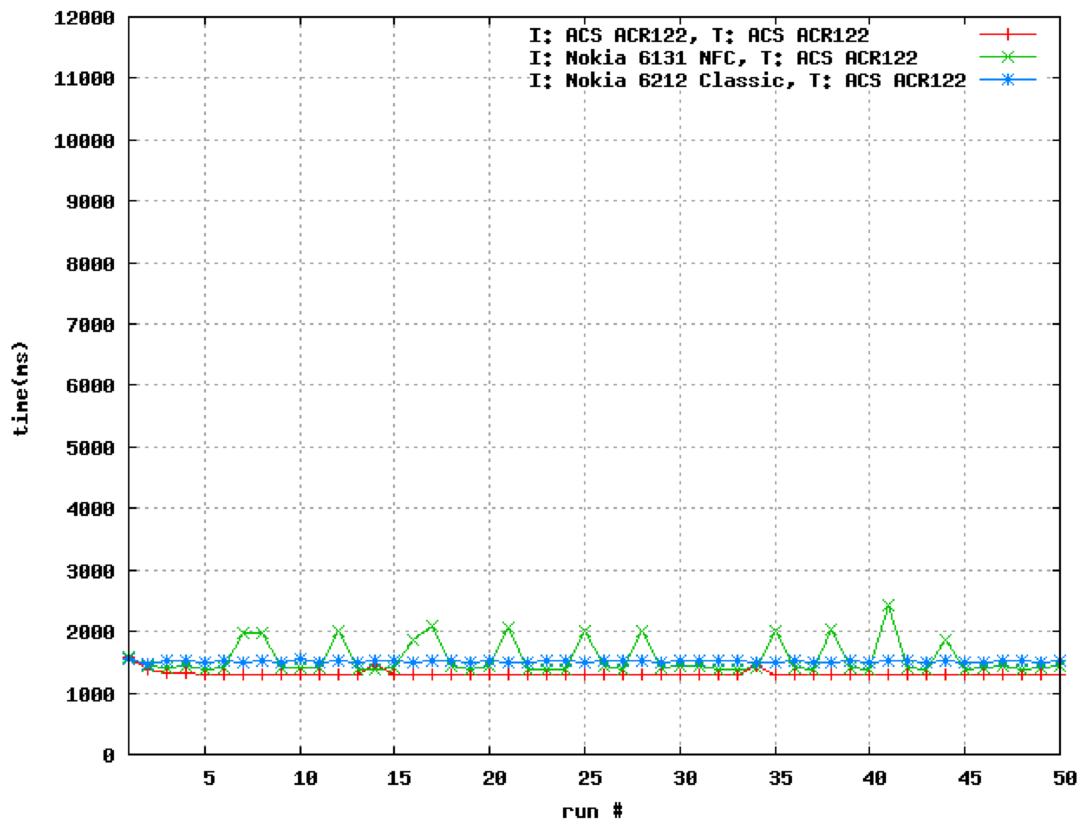

Figure 1.15: Phones as initiator, ACS ACR122 as target

1.7 Third approach

As shown above the packet loss is significant when the phone is configured as a target and the NFC reader is initiator, while the packet loss is negligible when the phone is initiator.

A possible solution for this would be to always configure the phone as initiator and the NFC reader as target. However, we still would like the NFC reader to start sending data. A solution for this is to create a `dummyBlock` that is being sent from the initiator to the target as shown in table 1.6. This will in effect 'reverse' the positions of the initiator and target in that it is now the turn of the target to send a message and for the initiator to wait for a response.

Fake Initiator		**Data**		**Fake Target**	*Notes*
TG_GET_DATA	←	0x10		IN_DATA_EXCHANGE	dummyBlock *the Fake Initiator starts with sending a message ...*
TG_SET_DATA		0x01 0x80 0x81	→		
TG_GET_DATA	←	0x08		IN_DATA_EXCHANGE	emptyBlock
TG_SET_DATA		0x03 0x82 0x83	→		
TG_GET_DATA	←	0x08		IN_DATA_EXCHANGE	emptyBlock
TG_SET_DATA		0x01 0x84 0x85	→		
TG_GET_DATA	←	0x08		IN_DATA_EXCHANGE	emptyBlock
TG_SET_DATA		0x02 0x86	→		
TG_GET_DATA	←	0x01 0x80 0x81		IN_DATA_EXCHANGE	
TG_SET_DATA		0x08	→		emptyBlock
TG_GET_DATA	←	0x03 0x82 0x83		IN_DATA_EXCHANGE	
TG_SET_DATA		0x08	→		emptyBlock
TG_GET_DATA	←	0x01 0x84 0x85		IN_DATA_EXCHANGE	
TG_SET_DATA		0x08	→		emptyBlock
TG_GET_DATA	←	0x02 0x86		IN_DATA_EXCHANGE	
TG_SET_DATA		0x04	→		endBlock

Table 1.6: Example of communication between the initiator and target using a dummy message, compare this to the situation in table 1.3.

The initiator uses `IN_DATA_EXCHANGE` to send a dummy message (this is all part of the communication setup) that the target receives with `TG_GET_DATA`. At this point the API send/receive calls can be used. This means that the fake initiator has to send data at this point and the fake target has to receive data, which is exactly the expected behavior of a 'real' initiator and target.

Due to the asymmetry of the protocols this will sometimes break restarting the connection cleanly after a connection tear. It would be really helpful in the future to create a truly

symmetric send/receive wrapper that can be switched without problems, this will require some more work.

However, because of the better reliability of the connection, connection tears are rare so this will increase the speed considerably and reduce the connection resets required. The packet loss will now drop to around 2% with the added benefit of resuming much faster after the connection loss than when the phone is target.

1.8 Conclusion

Eventually we managed to get reliable communication, although with a high error rate using the custom chaining solution (the second approach). A much (lower) error rate is achieved by using the fake initiator and target in the third approach.

At first it seemed like an almost trivial problem that could be solved easily. Unfortunately due to bugs in the NFC reader and NFC phone it was not as easy. To deal with these bugs and at the same time make the communication reliable was a non trivial problem.

Chapter 2

Security Analysis

Introduction

In this chapter we will analyze (security) problems of using a mobile phone as a means of (public) transport payment. We will not describe an actual payment protocol here, only define some general properties of this protocol.

Basic requirements for a public transport payment system:

1. The transport company (TC) wants financial security, i.e.: only offer services to customers[1], not lose money and not allow clones of tickets floating around to detect possible fraud.
2. The customer wants privacy, i.e.: the TC is not supposed to be able to link individual travelers to the trips made.

Assumptions:

1. The protocol uses some form of public key cryptography for verification of signatures and signing tickets.
2. The phone stores a private key which is unique for every subscription.
3. The TC provides an application (MIDlet) providing an interface to the customer for making public transport payments.
4. The protocol uses a direct connection between the phone and the terminal. There is no network operator involved.

[1] TCs usually have subscription services for (loyal) customers in offering them discounts on trips, or subscriptions to travel for free on certain connections. These subscriptions should be linked to individual customers and not (temporary) transferable to someone else.

To fulfill the requirements for the public transport company's financial security:

1. It must be impossible for anyone, including the customer, to extract the private key from the phone (it should be more expensive to extract the key than the benefit it will give).

2. It must be possible to verify that a customer is entitled to use the specific phone to travel.

And to fulfill the customer's privacy requirement:

1. It must be impossible for anyone involved in the PT protocol, including the TC, to uniquely identify the phone.

The phone has different communication channels with the outside world, we describe ways to uniquely identify a phone using those channels. We will also look at the capabilities a running MIDlet has to identify a phone.

Additionally, the phone offers different locations for (permanently) storing data. We analyze those locations and see if they can provide the security required for storing the private key (i.e.: it is impossible to extract it).

After this we will describe what is possible, with existing technical measures (capabilities), by some entity with those capabilities (actors).

Furthermore, we will look into the threats this poses to both the customer the TC and look into solutions for these threats.

2.1 Capabilities

We list the capabilities any possible actor has in relation to the requirements for the system state above. We look at capabilities relating to phone identification, the options for use (and misuse) of the (private key) data storage, linking phones to individuals and the possibilities for eavesdropping and man in the middle attacks.

2.1.1 Phone Identification

The Nokia 6131 NFC has a number of communication channels with the outside world:

1. GSM/GPRS network

2. Bluetooth

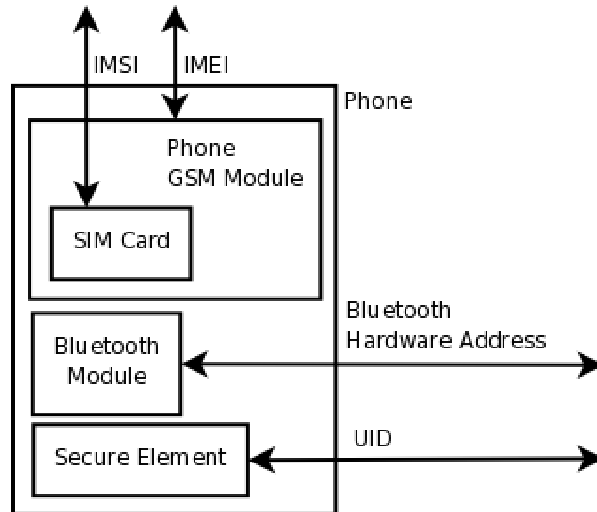

Figure 2.1: Information leaking channels on the Nokia 6131 NFC

3. NFC
4. Infrared

Some of the hardware involved with these communication channels has unique identifiers as shown in figure 2.1:

- IMEI number (number specific to the phone)
- IMSI number (number specific to the subscriber, SIM card)
- Bluetooth hardware address (unique for this specific phone's Bluetooth module)
- Secure Element (SE) identifier in the form of the UID that can be extracted through the emulated MIFARE card (see section 3.4.2)

These can be extracted by different kinds of actors:

1. An actor in the vicinity of the phone, but not necessarily directly involved in the protocol.
2. An actor providing a MIDlet running on the phone, which then leaks the identification over some communication channel.
3. An actor that has physical access to the phone.

An actor in the vicinity of the phone can extract the unique identifiers via:

1. IMSI and IMEI sniffing performed by the mobile network operator or equipment like an IMSI catcher[2] as sometimes used by law enforcement.
2. Bluetooth Device Discovery, assuming Bluetooth is enabled and the phone is set to be 'visible', to extract the Bluetooth hardware address.
3. A smart card reader to talk to the MIFARE area of the SE to extract the MIFARE UID from block number 0 (assuming the SE is activated) or during anti collision.

An actor able to run a MIDlet[3] on the phone can extract the unique identifiers via:

1. Reading device system properties exported to the Java VM to extract the IMSI and IMEI identifiers. This can only be done with a MIDlet suite in the Operator Domain or the Manufacturer Domain[4] (see chapter 7).
2. The LocalDevice class available through the JSR-82 API to extract the Bluetooth hardware address. Bluetooth needs to be enabled, but it is not required that the Bluetooth device is visible for device discovery.
3. The Nokia JSR-257 extension API to extract the MIFARE UID. The SE needs to be activated and the MIDlet should be placed in at least the Trusted Third Party Domain in order to be able to access the SE (see section 3.4).

An actor with physical access can extract the unique identifiers via:

1. The IMEI number can be extracted by "dialing" `*#06#`. It does not seem possible to obtain one's own IMSI number using just the phone.
2. The Bluetooth address can be extracted by "dialing" `*#2820#`.
3. The unique MIFARE UID can be extracted in the same way as with the actor in the vicinity.

2.1.2 Data Storage

Data on the phone can be stored in the following locations:

1. The phone internal file system

[2]An IMSI catcher is a device that performs a MITM attack on GSM communication by imitating a GSM base station with higher signal strength than legitimate base stations and thus make all mobile phones in the vicinity use the IMSI catcher as a base station.

[3]We have to take into consideration the MIDlet permissions as set by the user. We assume here that the user did not modify them. See chapter 7 for more information on MIDlet permissions.

[4]See http://wiki.forum.nokia.com/index.php/How_to_get_IMEI_in_Java_ME

2. A removable memory card in the phone
3. The Record Management System (RMS)
4. The secure element (SE), see section 3.4 on how to use it

The SIM card in the phone is also a smart card and could be used for data storage, but there is no possibility to use the data storage directly from within a MIDlet as it requires the JSR-177 API which is not available on the Nokia 6131 NFC. Furthermore this would require cooperation with (all) mobile network providers. However, the JSR-75 PIM API has the ability to access the address book on the SIM and to write to this address book (given sufficient write/read permissions). See section 3.3.3 for more information on the PIM API.

Some of the above mentioned locations are not secure against data extraction. An actor in the vicinity of the phone can try to extract data via:

1. The Bluetooth link from a computer to access the file system when the phone and the computer are paired, using the OBEX protocol[5]. The same holds for accessing the memory card as it (virtually) becomes part of the phone file system. The applications themselves and the RMS can also be accessed and extracted over the Bluetooth link, for example the tools Gnokii[6], Gammu[7] and MobiMB[8]. See section 3.3.5 and section **??** for a more extensive explanation.
2. A smart card reader[9]. The raw data on the SE can never be accessed, only indirectly through applets installed on the SE. It is designed to be tamper resistant and only allows the installation and removal of applets when the management keys are known. So it is impossible to access the (raw) data this way.

An actor able to run a MIDlet on the phone can try to extract data via:

1. The file system on the phone, which can be accessed from a MIDlet using the JSR-75 FileConnection API. The JSR-75 FileConnection API forbids file system access to the RMS[10]. It is possible to access the memory card as it (virtually) becomes part of the phone file system.

---

[5]Object Exchange (OBEX) Specification as defined by the Infrared Data Association (`http://www.irda.org/`)

[6]See `http://www.gnokii.org`.

[7]See `http://www.gammu.org`.

[8]See `http://www.logomanager.co.uk/` can access files on the phone file system, extract the applications and access the RMS. This was verified to work on the Nokia 6131 NFC and Nokia 6212 Classic.

[9]We assume here that the SE is accessed as a Java Card (see section 3.4 and appendix F)

[10]The JSR-75 FileConnection specification states: “Implementations MUST NOT (absolute requirement) allow a File Connection to access MIDP RMS databases.”

2. The RMS provides boundaries between MIDlet suites. MIDlets belonging to one MIDlet suite cannot access the RMS of another MIDlet Suite without explicit permission.

3. The JSR-257 API. The raw data on the SE can never be accessed, only indirectly through applets installed on the SE. It is designed to be tamper resistant and only allows the installation and removal of applets when the management keys are known. So it is impossible to access the (raw) data this way.

An actor with physical access can extract data via:

1. The phone internal file system can be accessed through Bluetooth or by using a USB cable connected to the phone and a PC. One can use OBEX (get) to access files on the phone. By using a tool like `gnokii` it becomes possible to access the (complete) phone file system.

2. A removable memory card can be extracted and placed into a memory card reader attached to a computer. This way all data can be extracted.

3. The RMS is stored on the phone file system and can be accessed by a tool like `gnokii`. The RMS files are located amongst the installed Java ME applications, usually in the `predefjava` folder or in a sub directory of `predefjava` on the phone.

4. Again the data on the secure element is safe against extraction due to the tamper resistant hardware design, so extracting data there will not be possible.

2.1.3 Linking a Phone To an Individual

It is impossible to directly link a phone to a customer as no biometric means of identification are available (there is no picture printed on the phone exterior). It is impossible for someone to prove the phone is really theirs as phones nowadays are commodities that are available, everywhere, to everyone for relatively low prices. A phone company might be able to link a phone to a customer who bought it with a subscription and was required to show an ID, but this is not true for a great number of phones in case a prepaid payment scheme is used.

2.1.4 Generic Capabilities

As with all secure protocols one should expect eavesdroppers and man in the middle actors trying to learn about the protocol and possible attack it. This can in principle be performed for all communication channels, as shown in figure 2.2.

Mallory has a fake relaying device, for example a phone with modified software that relays all the communication from the real terminal to the fake terminal, using Bluetooth

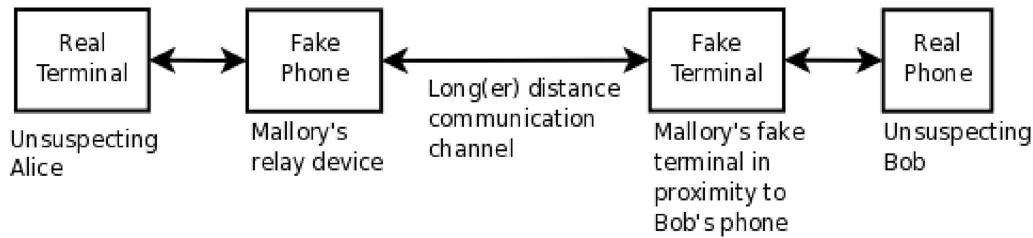

Figure 2.2: Overview of a relay attack

which has an extended range compared to NFC for instance, to a fake terminal which can also be a phone programmed as a terminal. If now the phone of Bob can be tricked into communicating with this fake terminal, Mallory can convince the real terminal she is Bob.

We do not consider replay attacks and other protocol level attacks here. We only focus on the lower level to what is in principle applicable to every protocol.

2.2 Actors

We present here all the relevant actors. We include the Phone Network Operator (PNO) here although it is not directly involved with the protocol. It has some capabilities that can affect customer privacy and is thus worth mentioning. Especially in case we look at collaborating actors.

1. Customer, phone owner, person traveling the public transport (Customer)
2. Friend or thief, either legitimately borrowing the phone, or using a stolen phone (Friend)
3. Transport Company (TC)
4. Phone Network Operator (PNO)
5. Eavesdropper (Eve)
6. Man in the Middle (Mallory)

We do not consider law enforcement an actor. Its capabilities include the capabilities of the PNO (as the worst possible adversary of the customer's privacy). It can summon the PNO access to real time location information of the phone. Information considering the payment of travels seems redundant compared to the information already available.

We also do not consider the manufacturer of the phone and SE an actor here as its is not involved anymore in any way. We assume of course there are no back doors built into the phone and/or SE.

2.3 Actors and Capabilities

In table 2.1 the capabilities, and actors that are capable of them, are shown (marked with an 'X').

Capability / Actor	Customer	Friend	PTC	PNO	Eve	Mallory
Extract IMSI/IMEI identification numbers	X	X		X		
Extract Bluetooth unique identification	X	X	X		X	X
Extract SE unique identification	X	X	X		X	X
Access internal file system	X	X	X			
Access data in RMS	X	X	X			
Access data in SE			X			
Proof ownership phone						
Perform Relay Attack						X
Perform NFC sniffing	X	X	X		X	X

Table 2.1: Actors and their capabilities

2.3.1 Customer

The customer is capable of extracting some identifying information about its own phone, this is not a threat to anyone. However, access to the file system and the RMS is a threat if those locations are considered for storing the private key.

2.3.2 Friend

This actor is identical to the customer. Assuming no PIN is required for actually traveling, or 'buying' a ticket the thief has the same capabilities.

2.3.3 Transport Company

The TC is able to run a MIDlet on the phone in at least the Third Party security domain. The unique Bluetooth hardware address as well as the unique number in the emulated

MIFARE area of the SE is a threat for the customer's privacy. A running MIDlet also has access to the file system on the phone and may use this to create a profile based on the contents of the file system.

In order for a MIDlet to be able to use certain functions on the phone, like accessing the file system (explicit) permission must be given for this. Every MIDlet suite has certain permissions that can be modified by the user. It is possible for instance to restrict a MIDlet from accessing the file system of the phone by changing the application permissions.

2.3.4 PNO

This is a powerful actor as it can locate and trace phones. Sometimes it has the capability to upload new versions of the phone operating system to the phone, providing the phone with more functionality like for example remote control for the PNO. With full control over the phone operating system most capabilities except extracting data from the SE become possible.

2.3.5 Eve and Mallory

Eve is capable of Bluetooth device discovery, and SE identification when Bluetooth is enabled and discoverable and the SE is activated. Furthermore, sniffing of NFC communication is possible if the sniffing equipment can be placed between the phone and the terminal. Mallory can also perform a relay attack (i.e.: any attack where modification or redirection of the data stream is required.

2.4 Threats

2.4.1 Key and Application Extraction

The phone poses a threat to the secrecy of the private key when the phone file system, memory card or RMS are used. When the key becomes known clones of the phone application including the private key can be made, which is undesirable, even though the clones could be detected on the protocol level.

Next to this, it is also possible to extract the application and analyze it. Secrets should not be stored in the application because they can (and will) be revealed given enough time. This can be a threat to the financial security of the public transport company.

2.4.2 Unique Identification of the Phone

Whenever Bluetooth is enabled, or the SE is activated, (a) unique identifier(s) of the phone can be obtained by an actor in the vicinity of the phone, and thus link travels to phones.

This is a threat to the privacy of the customer.

2.4.3 Inability To Link a Phone to a Customer

When it is impossible to link a phone to a customer it becomes possible for travelers to use a random phone, with possibly a subscription for discounts or free trips, to travel. This can be done by borrowing a phone or stealing one. This will make the subscriptions not exclusive anymore. This is a threat to the financial security of the public transport company.

2.4.4 Relay Attack

Relay attacks are a threat because they will cost an unsuspecting customer money (when someone buys a ticket using the customer's phone) or be inconvenient (when someone checks into the public transport system using the customer's phone leaving him unable to enter the public transport system).

2.4.5 Collaborating Actors

We also have to look at actors working together. What one actor cannot accomplish, two may. In case the TC and the PNO work together all privacy for the customer is lost. By combining the payment data with the location information of the customer all movements of the customer are known. We assume here that the customer uses his phone as a regular phone connected to the GSM network so the PNO actually has something to trace.

2.4.6 Insufficient Customer Trust

Trust of an incomprehensible and even insecure system seems easily earned nowadays with customers of public transport companies as demonstrated by the adoption of the current OV-Chip project. Convenience and efficiency (and not losing money) is more important than security or privacy for the average customer. Nevertheless, at least some trust in the system is required for the customer to accept it and use it.

2.5 Solutions

2.5.1 Prevent Key Extraction

In order to prevent key material from being extracted from the phone one has to use the SE for storing it. It should not be possible to access this applet using a card reader, or at

least not without explicit customer consent. One can envision a PIN, requested by the SE before signing, that needs to be entered every time.

2.5.2 Prevent Unique Identification of the Phone

The public transport company has a few ways to uniquely identify a phone. One can be prevented by the customer (disabling Bluetooth) the other is hard to avoid.

Activating the SE activates it for 60 seconds after which it can be accessed by both MIDlets and external card readers. There is no distinction possible on the Nokia phones.

When a MIDlet requires access to the SE to sign a message it needs to be activated which makes it possible to extract the MIFARE UID as well, either through the MIDlet, or remotely using the card reader.

It seems impossible to prevent this.

2.5.3 Linking a Phone to a Customer

As part of the protocol one can envision a system where authorized parties will be able to extract a (digital) picture from the phone that was stored in a secure place along with the private key. However, this will make it possible to construct a (centralized) database with pictures of travelers and again link customers to individual trips. This is still a better solution than immediately storing all pictures of the customers in a database and using them for verification.

Another solution would be to have an extra physical proof of subscription with the public transport company. One has for example a credit card sized card with a picture of the customer printed on it together with the details of the subscription. It can state for example "40% discount on all trips after 9am", or "free travel between Amsterdam and Utrecht". The 'best' solution is most likely the non technical solution.

Although this does not necessarily link a phone to a customer, it will be enough to guarantee the exclusiveness of the subscription and not cause any financial damages.

2.5.4 Relay Attack

There are some ways to deal with relay attacks:

1. Take note of the time it takes for an answer to come back to a request. If it is above a certain minimum the transaction should be terminated.
2. Design the protocol in such a way that nothing can be done without explicit customer consent every time a transaction request is triggered (like entering a PIN).

Having a phone makes it possible to protect against relay attacks as there is the possibility for user confirmation. When using a (simple) smart card, like the system currently in use, there is no such protection.

2.5.5 Trustworthy System

To increase the trust a customer has in the system, the source code can be published together with the design of the system.

Even when the protocol is open and the software open source, this does not mean anything when it cannot be verified what software is actually running on the device. This is a hard problem, but one that needs addressing as releasing the source code of the applications does not necessarily make it more trustworthy.

When a customer can actually verify that the software running on the phone and/or smart card is the same as the source version there is more reason to trust the system.

This problem can be solved with these steps:

1. Open the MIDlet build system for everyone to analyze. This way it will become possible to create the MIDlet from source and verify the hash of the generated MIDlet with the hash of the installed application. One can then be sure the MIDlet corresponds to the source code.
2. Have a small 'trusted' applet on the SE that is just used for signing purposes.

The customer can know exactly what functionality of the applet in the SE is used and whether or not that is justified. This system even allows for custom implementations of the MIDlet to offer more functionality or be more flexible. The private key will remain safely in the SE[11].

2.6 Protocol Design recommendations

Here we propose some design recommendations implementations should take into consideration:

1. Always ask the customer for permission before doing anything that can (indirectly) cost the customer money.
2. Do not rely on the PNO in any way to avoid sharing of data which is a threat to customer privacy.

[11]This will require that the developer is able to sign the MIDlet to place it in the Trusted Third Party Domain.

3. Provide the MIDlet source code including the exact build system so customers can verify the running application with the available source code.

4. Use a small 'trusted' applet on the SE with a clear interface and limited responsibilities. See appendix F.

2.7 Conclusion

It is possible to create a privacy friendly, trustworthy, financially reliable system for public transport payment. A problem to still overcome is to avoid customer tracking by the TC through the secure element's unique identifier. No solution for this was found. Data on the phone cannot be considered safe from extraction by the customer, or other parties having physical control over the phone, unless it is stored in the secure element. The secure element should be used for storing the private key and run a small applet responsible for ticket signing. Customers should disable Bluetooth on their mobile phones to avoid Bluetooth device discovery from picking up their phone in the vicinity of a public transport terminal which can register the movements of the individual throughout the public transport system.

Chapter 3

Mobile Programming

Introduction

This chapter will describe developing Java ME applications for devices with limited processing resources like mobile phone. In particular we will focus on the Nokia 6131 NFC and the Nokia 6212 Classic. First we will look at the different Java editions available, compare them briefly and then look into Java ME APIs. After that we will look at how to create an application that can run on the phone and end with a description of the secure element as found in the Nokia 6131 NFC and Nokia 6212 Classic.

3.1 Java Editions

There are four Java editions currently in (widespread) use:

- Java Platform, Standard Edition[1] (Java SE)
- Java Platform, Enterprise Edition[2] (Java EE)
- Java Platform, Micro Edition[3] (Java ME)
- Java Card Platform[4]

The standard edition is what runs on an average desktop system or notebook. The enterprise edition is used on servers, for example, to run web (server) applications, and has

[1]See http://java.sun.com/javase/
[2]See http://java.sun.com/javaee/
[3]See http://java.sun.com/javame/index.jsp
[4]See http://java.sun.com/javacard/

special APIs for this. The Micro Edition is used on embedded systems, or hand held devices like mobile phones. The Java Card Platform can be found on some (contactless) smart cards. We will focus on Java ME here and look at Java Card when we go into more detail regarding the secure element.

3.2 Java ME

The difference between Java SE and Java ME is the way of deployment and the available APIs. The language used for Java ME is the same as in Java SE version 1.4[5].

Some APIs, like the ones related to the graphical user interface (GUI), are replaced in Java ME with different APIs optimized for mobile devices that take into consideration the different input mechanisms and screen sizes. Additional APIs can be added (at the phone manufacturers discretion), like a FileConnection API (described below). We will limit ourselves to the APIs that will play a role in creating and using a Java ME public transport payment system.

Applications written for Java ME devices (MIDlets) are bundled in MIDlet suites, where a MIDlet suite can contain (one or more) MIDlets. A MIDlet can be compared to a Java class with a “main” method where execution starts.

3.3 API

First we will describe the APIs which are available (and required) for every Java ME implementation. This is described in the Connected Limited Device Configuration (CLDC) and the The Mobile Information Device Profile (MIDP) specifications. In addition we describe some of the extension APIs often available:

Personal Information Management (PIM) API: provides access to contacts, evens and notes

FileConnection API: access to the phone’s (internal) file system

Contactless Communication API: access to the “secure element” in the phone

Contactless Communication Extension API: access to NFC functionality and the secure element

Record Management Store: private and persistent MIDlet suite (limited) storage space on the phone

[5] Section 5.3.1 of the CLDC 1.1 specification states: “A CLDC implementation must be able to read Java class files in all the formats supported by Java Standard Edition, JDK versions 1.1, 1.2, 1.3 and 1.4.”

3.3.1 CLDC

The CLDC version 1.1 is defined in JSR-139[6]:

> The Connected Limited Device Configuration (CLDC) defines the base set of application programming interfaces and a virtual machine for resource-constrained devices like mobile phones, pagers, and mainstream personal digital assistants

The focus of the CLDC is to provide a basic (minimal) Java system[7]:

- Java language and virtual machine features
- Core Java libraries (java.lang.*, java.util.*)
- Input/output (java.io.*)
- Security
- Networking
- Internationalization

As stated before: the language is the same as the language used in Java SE 1.4.

3.3.2 MIDP

In addition to the CLDC there is an extension profile for devices like mobile phones, the Mobile Information Device Profile. Currently two version are in wide spread use. Version 2.0 (which is the version supported by the Nokia 6131 NFC) and version 2.1 (which is the version supported by the Nokia 6212 Classic[8].

The MIDP is defined in JSR-118. It is briefly described as:

> The Mobile Information Device Profile (MIDP) lets you write downloadable applications and services for network-connectable mobile devices.

It is built on top of CLDC and extends the functionality of CLDC by including functionality relevant for mobile devices. The specification includes areas such as[9]:

---

[6]See `http://jcp.org/en/jsr/detail?id=139`

[7]Taken from section 2.3 ('Scope') of the CLDC 1.1 specification.

[8]The difference between MIDP 2.0 and 2.1 can be found in the Changelog found in Chapter 1 of the MIDP 2.1 specification.

[9]Taken from section 1.2 ('Scope') of the MIDP 2.1 specification.

- Application delivery and billing
- Application lifecycle (i.e.: defining the semantics of a MIDP application and how it is controlled)
- Application signing model and privileged domains security model
- End-to-end transactional security (https)
- MIDlet push registration (server push model)
- Networking (higher level network protocols)
- Persistent storage (Record Management System)
- Sound
- Timers
- User interface (UI) (including display and input, as well as the unique requirements for games).

So to actually be able to create and deploy applications on the phone at least the CLDC and MIDP libraries are needed.

3.3.3 Personal Information Management (PIM) API

With this API[10], it is possible to access the contact, events and todo items on both the phone and SIM card.The example below shows how to extract a list of all contacts names that are located on the SIM card of the phone. The PIM API is more extensive than this. It is for instance also possible to add contacts (to the SIM) or to export them to vCard.

```
/**
 * Get a list of the names of all the contacts stored on the SIM of the
 * phone.
 *
 * @return the list of contact names
 * @throws PIMException
 */
private Vector getContactList() throws PIMException {
    PIM pim = PIM.getInstance();
    ContactList cL = (ContactList) pim.openPIMList(PIM.CONTACT_LIST,
            PIM.READ_WRITE, "SIM");
    Vector v = new Vector();
    Enumeration e = cL.items();
    while (e.hasMoreElements()) {
```

[10] The PIM API is specified in JSR-75, see http://jcp.org/en/jsr/detail?id=75

```
        Contact contact = (Contact) e.nextElement();
        int cV = contact.countValues(Contact.FORMATTED_NAME);
        for (int i = 0; i < cV; i++) {
            String contactName = contact.getString(Contact.FORMATTED_NAME,
                    i);
            v.addElement(contactName);
        }
    }
    return v;
}
```

3.3.4 FileConnection API

With this API[11], it is possible to access the phone's internal file system. This means access to data on the device, like for example documents, music or videos. In case a removable (memory) card is placed in the phone it is possible to access those files as well. It is not possible to access "special" files, like the MIDlet suite files themselves[12], or the RMS as described in section 3.3.5.

In order to list all the "roots" (i.e.: the available root directories on the phone) one can use this code fragment[13]:

```
Enumeration e = FileSystemRegistry.listRoots();
while (e.hasMoreElements()) {
  String root = (String) e.nextElement();
  /* The String "root" contains the name of the file system root */
}
```

The next example opens a file on the removable memory card, its root was first determined to be `E:/`, inserted in the phone and writes "Hello World" to it:

```
FileConnection fc =
  (FileConnection) Connector.open("file:///E:/file.txt");
if (!fc.exists())
  fc.create();
PrintStream ps = new PrintStream(fc.openOutputStream());
ps.println("Hello World");
ps.flush();
ps.close();
```

[11] The FileConnection API is specified in JSR-75, see http://jcp.org/en/jsr/detail?id=75

[12] This is described in Nokia's `Java Developer's Library 3.3 -> Implementation notes -> (JSR-75 Optional package) FC API -> Runtime environment notes -> Security settings` which can be found at http://library.forum.nokia.com/.

[13] Taken from JSR-75 FileConnection API documentation and slightly modified to make it compile.

The URL is typically specified by its root and the location on the file system. `C:` refers to the phone internal file system and `E:` refers to an (optionally installed) memory card.

In order to access the file system (from a signed MIDlet suite), some permissions need to be requested. For file system access there are two relevant permissions:

- Read access to files (and directories): `javax.microedition.io.Connector.file.read`
- Write access to files (and directories): `javax.microedition.io.Connector.file.write`

It should be noted that requesting information about the available roots requires the MIDlet suite to have the permission `javax.microedition.io.Connector.file.write` as well. See section 4.5.1 on how to request permissions.

A MIDlet is prohibited from accessing special directories like `C:/predefhiddenfolder/`. Even in the Manufacturer or the Operator security domain it is not possible to access these directories. We wrote a method for MIDlets using the FileConnection API to traverse the directory structure as far as permitted (from within the phone). The code is shown in Appendix C. The list returned is exactly the same whether the MIDlet suite is placed in the "Identified Third Party" security domain or the "Manufacturer" security domain.

3.3.5 Record Management Store

Although the RMS is part of the MIDP API[14] we discuss it in this section because we want to describe it in more detail. The MIDlets can use the RMS to persistently store data. It is intended for applications that want to store some settings or for example keep high scores.

A MIDlet can access all the record stores in its MIDlet suite. It is also possible to give other MIDlet suites access to the RMS, but we will not discuss that here. See for more information on the record store chapter 14 of the MIDP 2.1 specification.

The example here shows how to instantiate the RMS, add some data to it and retrieve it (at a later time). We start with opening the record store, or creating it when it does not exist yet:

```
RecordStore rs = RecordStore.openRecordStore("settings", true);
```

We want to store an integer value in the record store. Because the record store only supports storing byte arrays we convert the integer to string first before converting it to a byte array. This is not the most efficient way of storing integers, as every 32 bit integer can be represented by at most 4 bytes, but is easy:

```
byte[] s = String.valueOf(123456).getBytes();
```

Then we add the record to the store:

[14] See chapter 14 of the MIDP 2.0 specification.

```
rs.addRecord(s, 0, s.length);
```

The record store stores data and associates a unique identifier with the data. The first record will have the identifier 1, the second one 2, and so on. Every time data is added the identifier is increased by one. Even after deleting records the identifiers belonging to the deleted record will never be used again.
Here we retrieve the first record:

```
byte[] s = rs.getRecord(1);
```

Convert it back to string (assuming it was the integer stored in the previous example):

```
String t = new String(s);
```

And then convert it to integer:

```
int i = Integer.parseInt(t);
```

We left out all the exception handling here to focus on the API itself.

3.4 Secure Element

The Nokia 6131 NFC and Nokia 6212 Classic both have a "secure element". This secure element contains a tamper resistant chip that has a number of features:

1. It can act as a (NFC) smart card reader
2. It can act as a (NFC) smart card (e.g.: Java Card, MIFARE)
3. It can act as a (NFC) peer to peer device (NFCIP)

See figure 3.1 for the different means of communication.

The API for the first two use cases is standardized in JSR-257[15] except for emulating a MIFARE card, which is part of the Nokia JSR-257 extension API.

The smart card can be accessed from the phone itself (using JSR-257) or by an external card reader. The secure element needs to be activated (deactivated by default) in order for a MIDlet or an external card reader to be able to access it. This section will describe the APIs for using the secure element through the JSR-257 and JSR-257 Extension API.

The smart card (which is part of the secure element) can be used to store data that need to stay secret, for example encryption keys for the payment protocol described in chapter 6. The secure element is evaluated for private key storage in chapter 2. This section will describe how to use it.

[15] See http://jcp.org/en/jsr/detail?id=257.

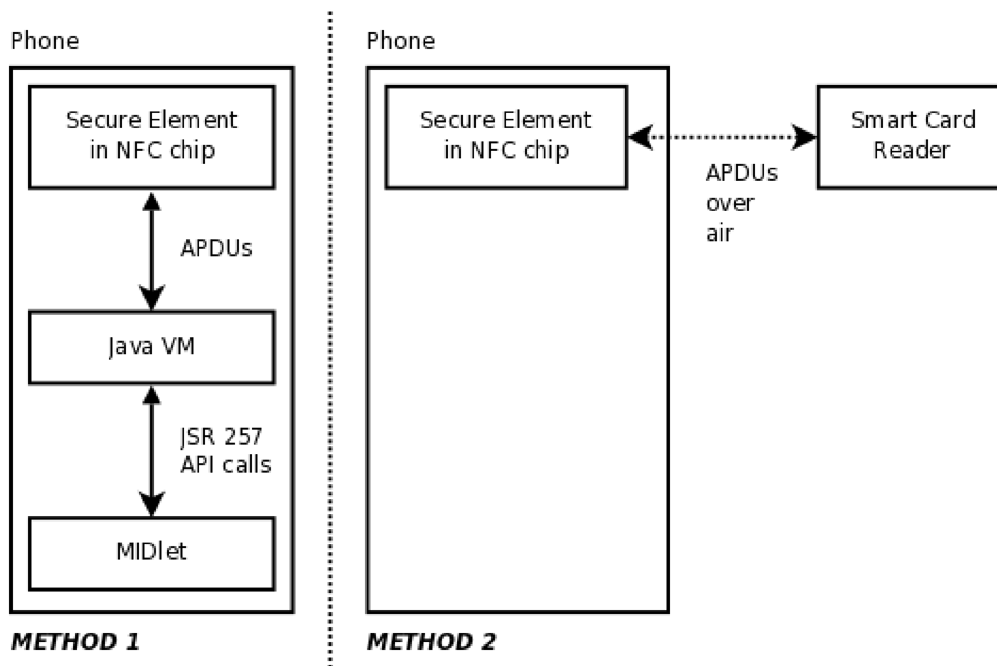

Figure 3.1: Secure Element Communication (using MIDlet or card reader)

3.4.1 Contactless Communication API

Using the Contactless Communication API[16], it is possible to access the secure element (SE) built into the phone, or any supported external smart card in proximity of the phone. There is support for accessing Java Card smart cards and other cards. This example shows how to connect to the phone's internal SE and select an applet by its AID:

```
private byte[] SELECT = {(byte) 0x00, (byte) 0xA4, (byte) 0x04, (byte) 0x00,
                        (byte) 0x09, (byte) 0x74, (byte) 0x69, (byte) 0x63,
                        (byte) 0x6B, (byte) 0x65, (byte) 0x74, (byte) 0x69,
                        (byte) 0x6E, (byte) 0x67, (byte) 0x00};

String url = System.getProperty("internal.se.url");
ISO14443Connection conn = (ISO14443Connection) Connector.open(url);
byte[] result = conn.exchangeData(SELECT);
```

Typically one starts by selecting an applet to talk to before communicating with it. This is done with the command where the APDU bytes are `CLA = 00`, `INS = A4`, `P1 = 04`, `P2 = 00`, `LC = 09`. The Applet ID is terminated with a `0x00`. The response to this command should be a `0x90 0x00` which indicates success. Any other code indicates problems selecting

[16]The Contactless Communication API is defined in JSR-257, see http://jcp.org/en/jsr/detail?id=257.

the applet. We try here to select an applet with ID `0x74 0x69 0x63 0x6B 0x65 0x74 0x69 0x6E 0x67`, which is a byte representation of the string "ticketing".

For communication with the internal SE, the MIDlet suite needs to be installed at least in the "Identified Third Party" security domain. See section ?? for an overview of the security domains.

3.4.2 Contactless Communication Extension API

Nokia added a (proprietary) extension API to the Nokia 6131 NFC and Nokia 6212 Classic phones to use the more advanced capabilities of the included NFC chip. Two are of importance here:

- NFCIP API, peer to peer NFC communication to exchange byte arrays between two NFC capable devices, see chapter 1 for details.
- MIFARE API, access external MIFARE cards, or the emulated MIFARE card in the SE

This example shows how one would establish a NFCIP Connection as a target to an initiator and send and receive some data:

```
NFCIPConnection nic =  (NFCIPConnection)
  Connector.open("nfc:rf;type=nfcip;mode=target");

byte[] recv = nic.receive();
nic.send("Hello World".getBytes());
```

The connection URL for target is `nfc:rf;type=nfcip;mode=target`. For initiator mode it is `nfc:rf;type=nfcip;mode=initiator`. A target starts with receiving while an initiator starts with sending. To modify the above code to an initiator one would in addition to changing the URL, switch the position of `nic.receive()` and `nic.send()` method calls.

In addition to NFCIP communication it is also possible to talk to the emulated MIFARE section on the secure element. The example below shows how to extract the UID of the MIFARE card:

```
MFStandardConnection msc = (MFStandardConnection)
  Connector.open(System.getProperty("internal.mf.url"));
byte[] uid = msc.getManufacturerBlock().getUID();
```

This will put the UID of the MIFARE tag in the byte array `uid`. For communication with the internal SE, the MIDlet needs to be installed in the Trusted Third Party Security Domain. See section ?? for an overview of the security domains.

3.4.3 Accessing the Secure Element With Card Reader

It is also possible to use the secure element with an external card reader. One can for example use GPshell[17], GlobalPlatformManager[18] or libnfc[19].

One can use the following GPshell configuration file to list the Java Card applets installed on the secure element:

```
mode_211
enable_trace
establish_context
card_connect -readerNumber 1
select -AID a000000003000000
open_sc -security 3 -keyver 42 -mac_key 404142434445464748494A4B4C4D4E4F
                               -enc_key 404142434445464748494A4B4C4D4E4F
                               -kek_key 404142434445464748494A4B4C4D4E4F
get_status -element 20
card_disconnect
release_context
```

For this to work the secure element needs to be "unlocked". This procedure is described in section 3.4.4.

One can install an applet by replacing the line `get_status -element 20` with `install -file applet.cap -priv 2` where `applet.cap` is the name of the applet to install. To delete an applet (and package) one uses:

```
delete -AID 010203dead060708090000
delete -AID 010203dead0607080900
```

where the first item is the applet AID and the second one is the package AID.

3.4.4 Unlocking the Secure Element

The secure element is in "locked" state by default. This means that the management keys are not known to the phone owner. Nokia provides a MIDlet suite that can reset the keys to default values (as used in section 3.4.3). The MIDlet suite can be downloaded from the Nokia website[20].

The unlocking is strictly speaking only required for the Java Card part of the secure element. The MIFARE key can be found on the phone file system itself. As part of our

[17]See `http://sourceforge.net/projects/globalplatform/`.

[18]GlobalPlatormManager was released as part of the OV-Chip 2.0 project (See `http://www.sos.cs.ru.nl/ovchip/`).

[19]See `http://www.libnfc.org`.

[20]See `http://wiki.forum.nokia.com/index.php/NFC` for a link to the Unlock MIDlet suite.

analysis of the security of the Nokia phones (see chapters 2 and 7) we found out that the MIFARE key can be found in `/predefhiddenfolder/predefscdata/mf key nokia.dat`.

For the unlock tool to work one needs to have a working Internet connection on the phone. The tool connects to a Nokia website in order to retrieve the (supposedly) device specific keys with which the default keys can be set instead of the device specific ones. After performing this, one can never restore the phone to its original state (i.e.: reset the keys to their factory values), so the secure element will not be trusted anymore by third parties for installing payment applets for example.

Chapter 4

MIDlet Suite Construction

Introduction

A MIDlet suite is a collection of (one or more) MIDlets together in a Java Archive (JAR file) accompanied by an optional Java Application Descriptor (JAD file).

A MIDlet is a Java class that extends the abstract class `javax.microedition.midlet.MIDlet` which gives the phone operating system the ability to control the MIDlet (start, pause, destroy)[1]. It can be seen as the entry point of execution (like the `main` method in Java SE applications).

The following steps are part of constructing a MIDlet suite:

1. Compile the source code to Java classes
2. Preverify the classes
3. Create the manifest
4. Package the preverified classes and manifest in a JAR archive
5. Create the application descriptor (JAD file) (optional)
    - Add the permissions required by the MIDlets (only when signing)
    - Sign the suite by adding the JAR file signature and certificate chain (optional)
6. Deploy the MIDlet suite

[1] See chapter 12 of the MIDP 2.1 specification, of which some parts have been summarized in this chapter.

4.1 Compilation

Compiling source code for MIDlet suites is the same as for Java SE. Some differences:

- The CLDC and MIDP APIs (and possible extension APIs) are part of the classpath instead of the regular Java SE classes
- The source and target compatibility is version ≤ 1.4 (for CLDC 1.1 devices)

These class files can now be preverified.

4.2 Preverification

Due to the limited processing power of most (legacy) mobile devices, the verification of Java classes a virtual machine performs is split in two parts. There is an "off-device" preverification phase and a (limited) verification phase on the mobile device.

The preverification modifies the byte code in the class files in the follow way[2]:

- Inline all subroutine calls (remove all the calls and returns)
- Add StackMap attributes to the class files to make verification 'on-device' easier

This preverification is done by software tools not available in the normal Java Development Kit (JDK). There are two preverification tools we know of:

- Sun's Wireless Toolkit[3] (WTK) `preverify` tool written in C and available for a limited number of platforms
- The ProGuard Preverifier/Obfuscator[4] written in Java

4.3 The JAR Manifest

The JAR file contains a manifest[5] which must contain at least the following attributes:

`MIDlet-Name:` Contains the name of the MIDlet suite

[2]Taken from section 5.2.1 of the CLDC 1.1 specification.

[3]See http://java.sun.com/products/sjwtoolkit/overview.html

[4]See http://proguard.sourceforge.net

[5]The manifest file format is described in http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html.

`MIDlet-Version`: Contains the version number of the MIDlet suite

`MIDlet-Vendor`: Contains the vendor name of the MIDlet suite

If there is no application descriptor (JAD file), the manifest must contain the following attributes as well[6]:

`MIDlet-<n>`: Contains comma separated the name of the MIDlet, the file name of the icon (relative to the root of the JAR archive) and the class name of the MIDlet (n is incremented for every MIDlet belonging to that suite).

`MicroEdition-Profile`: Contains the MIDP version number. This is usually either `MIDP-2.0` or `MIDP-2.1`.

`MicroEdition-Configuration`: Configuration attribute. Contains the CLDC version number. This is usually `CLDC-1.1`.

It makes sense to always include all these (six) items in the manifest file as that way the JAR file can be deployed without an accompanying application descriptor (assuming it does not require functionality that requires MIDlet suite signing). An example manifest looks like this:

```
MIDlet-Name: Test MIDlet Suite
MIDlet-Version: 1.0.0
MIDlet-Vendor: Test MIDlet Vendor
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.0
MIDlet-1: Test MIDlet,test_icon.png,com.example.Test
```

4.4 Packaging

One creates a JAR file with for example the `jar` tool. This tool is part of the Java JDK. The JAR file should containing the manifest from the previous section, the preverified classes and optionally the MIDlet suite icon.

4.5 Application Descriptor

The (optional) application descriptor (JAD file) must contain the following entries[7] if they are not in the manifest (see section 4.3):

[6] See section 12.2.3.3 of the MIDP 2.1 specification

[7] See section 12.2.4 of the MIDP 2.1 specification.

`MIDlet-<n>`**:** Contains comma separated the name of the MIDlet, the file name of the icon (relative to the root of the JAR archive) and the class name of the MIDlet (n is incremented for every MIDlet belonging to that suite).

`MicroEdition-Profile`**:** Contains the MIDP version number. This is usually either `MIDP-2.0` or `MIDP-2.1`.

`MicroEdition-Configuration`**:** Configuration attribute. Contains the CLDC version number. This is usually `CLDC-1.1`.

The application descriptor (if it exists) must contain the following entries:

`MIDlet-Name`**:** Contains the name of the MIDlet suite

`MIDlet-Version`**:** Contains the version number of the MIDlet suite

`MIDlet-Vendor`**:** Contains the vendor name of the MIDlet suite

`MIDlet-Jar-URL`**:** Contains the relative or absolute link to the JAR archive file

`MIDlet-Jar-Size`**:** Contains the size (in bytes) of the JAR archive file

In addition the application descriptor can contain additional entries. We will limit ourselves to describing the following entries, more can be found in the MIDP specification:

`MIDlet-Permissions`**:** Contains permission requests that are essential for MIDlets in this suite. If (one of the) permissions is not given the MIDlet suite will not be successfully installed (see section 4.5.1)

`MIDlet-Permissions-Opt`**:** Contains permission requests that are requested by MIDlets in the suite, but are not essential for successful operation (see section 4.5.1)

`MIDlet-Push-<n>`**:** Contains push registry entries (see section 4.5.2)

`MIDlet-Certificate-<n>-<m>`**:** Contains certificates (see section 4.5.3)

`MIDlet-Jar-RSA-SHA1`**:** Contains signature (see section 4.5.3)

4.5.1 Permissions

When signing MIDlet suites one needs to request permissions in the application descriptor if certain sensitive operations are performed. For example accessing the file system (using JSR-75) will require permissions. Failing to request those permissions will result in a `SecurityException`. A list of some permissions and their description can be found in table 4.1. The permissions can be requested by using either the `MIDlet-Permissions` or `MIDlet-Permissions-Opt` attribute, where the former specify essential permissions and the latter optional permissions not essential for operation.

Permission	Description
javax.microedition.io.Connection.file.read	Request general read permission (e.g.:file system and address book)
javax.microedition.io.Connector.file.write	Request general write permission (e.g.: file system and address book)
javax.microedition.io.PushRegistry	Request permission to use the push registry

Table 4.1: Some MIDlet suite permissions

4.5.2 Push Registry

With the push registry[8] it becomes possible to start applications when a certain external event occurs. For example a certain smart card is touched, or an SMS is received from a preconfigured number.

There are two types of registrations:

Static: a static registration is requested through the application descriptor is in effect during the lifetime of the MIDlet suite;

Dynamic: a dynamic registration is requested at run-time by a MIDlet. A MIDlet can (dynamically) register and unregister push events.

The attribute for push registrations was already shown in section 4.5. The `MIDlet-Push-<n>` entries have a comma separated value. The first part indicates the push URL (the kind of connection). The second part indicates the MIDlet that should be started when the URL gets triggered. The third part specifies the filter (who can initiate the connection). First we will give some examples on how to create static push registrations:

```
MIDlet-Push-1: nfc:undefined_format,com.example.Test,nfc:rf;type=mf4k;uid=*
```

The above entry launches the `com.example.Test` MIDlet whenever the phone touches a NFC tag with the url “undefined format” which happens to be any MIFARE 4K card (the filter).

```
MIDlet-Push-2: secure-element:?aid=001122334455667788,com.example.Test,*
```

The above entry launches `com.example.Test` whenever an external card reader accesses the secure element and requests a Java Card applet with AID `001122334455667788`.

[8]The push registry is described in chapter 7 of the MIDP 2.1 specification.

```
MIDlet-Push-3: sms://:10000,com.example.Test,*
```

The above entry waits for an SMS from anyone (filter is *) on port `10000`[9].

```
MIDlet-Push-4: socket://:5000,com.example.Test,192.168.1.*
```

This entry waits for a socket connection on port `5000` from an IP address in the range `192.168.1.*`.

For dynamic registrations one can use:

```
PushRegistry.registerConnection(url, midlet, filter);
```

Here the fields `url`, `midlet` and `filter` are of type String and is used like the static registrations. In order to find out the name of the currently running MIDlet one can use `this.getClass().getName()`.

When requesting permissions from a MIDlet in a signed MIDlet suite one needs to request the push registry permission, see section 4.5.1.

4.5.3 Signing

There are a few reasons why signing a MIDlet suite is beneficial (or necessary):

- It allows for verification whether or not a MIDlet suite should be "trusted" (i.e.: it was signed by someone who obtained the trust of one of the CAs whose root certificate is included on the phone. This does not necessarily warrant an increase in trust places on a signed MIDlet suite as the requirements for obtaining a signing certificate are not that strict).
- Allow access to restricted functionality like accessing the secure element on the Nokia 6131 NFC and Nokia 6212 Classic.
- Allow the user to give "Always allowed" permissions to certain sensitive operations like accessing the file system, sending messages or using the Internet connection.

Signing a MIDlet suite[10] involves adding the certificate chain (of trust) and signature to the application descriptor. The chain is required because usually only the root certificate is installed on the phone.

An example of a part of an application descriptor with certificates and signature added:

[9] The port can be specified by using for example the JSR-205 API "Wireless Messaging API (WMA)".

[10] Described in detail in chapter 4 of the MIDP 2.1 specification.

```
MIDlet-Jar-RSA-SHA1: YyDeVQHVMDmgeGAELWRiUUVViklohMMAsIstO+zEGRCXU2uD4+...
MIDlet-Certificate-1-3: MIIDJzCCApCgAwIBAgIBATANBgkqhkiG9w0BAQQFADCBzjE...
MIDlet-Certificate-1-2: MIIDTjCCAregAwIBAgIBCjANBgkqhkiG9w0BAQUFADCBzjE...
MIDlet-Certificate-1-1: MIIDUDCCArmgAwIBAgIQaE9OYK/OS6yhbP663nVJQDANBgk...
```

The certificates are Base64 encoded DER certificates. The signature is also Base64 encoded. Both the certificates and the signatures are added to the application descriptor without line breaks.

There are various ways to add the certificate chain and the signature to the application descriptor:

1. Use `JadTool` included in the Sun WTK
2. Use Antenna or JAD Ant Tasks for Ant
3. Use an IDE with Mobile Development plugin (e.g.: NetBeans or Eclipse)

4.5.4 Application Descriptor File

The application descriptor has almost the same structure as the manifest file. The important difference is that manifest files have a maximum line length of 72 characters while there is no such restriction for application descriptors. Application descriptors even require the use of longer lines for storing the certificates and the signature when signing MIDlet suites[11].

An example of a complete application descriptor (JAD file):

```
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.0
MIDlet-Vendor: Radboud University Nijmegen
MIDlet-Version: 1.0.0
MIDlet-Jar-URL: TestMIDlet-1.0.0.jar
MIDlet-Name: Test MIDlet Suite
MIDlet-Jar-Size: 12345
MIDlet-1: First Test MIDlet,suite_icon.png,com.example.TestOne
MIDlet-2: Second Test MIDlet,suite_icon.png,com.example.TestTwo
MIDlet-Certificate-1-2: MIIHPTCCBSWgAwIBAgIBADANBgkqhkiG9w0BAQQFADB5MRAw...
MIDlet-Certificate-1-1: MIIE6jCCAtKgAwIBAgIDBxSNMA0GCSqGSIb3DQEBBQUAMHkx...
MIDlet-Jar-RSA-SHA1: aEWOAoXgpXx4nJB+U+P3YGukDydSvAfaHE7KhXZmbvDjXXZ5scx...
MIDlet-Permissions: javax.microedition.io.PushRegistry
MIDlet-Push-1: nfc:undefined_format,com.example.Test,nfc:rf;type=mf4k;uid=*
```

[11]See section 4.1.5 and 4.1.6 of the MIDP 2.1 specification.

4.6 Installation on mobile phone

There are a few ways to transfer MIDlet suites to a mobile phone:

- Using Bluetooth (or USB)
- Put it on a memory card using a computer and install the memory card in the phone
- Download it through an Internet connection on the phone

Since the first option is the most convenient we describe this here.

Whenever a MIDlet suite is signed both the Application Descriptor (JAD file) and JAR file need to be transferred to the phone. One can use Bluetooth file transfer of the operating system (using Obex Push) or use Nokia PC Suite[12] when using Windows. In case the MIDlet suite is not signed just sending the JAR file will suffice.

4.7 Development Tools

In order to automate the construction of MIDlet suites one can make use of tools to take care of the steps that differ from general Java development, like preverification, creating the application descriptor and MIDlet suite signing. In addition to having a IDE (Integrated Development Environment) we want to have a (minimal) build system that converts sources into a working MIDlet suite with a single command. Preferably this is portable and does not depend on anything else than a working Java environment.

4.7.1 Sun Java ME SDK

Sun offers a solution in the form of the Java ME SDK 3.0 (formerly known as the Sun Wireless Toolkit, WTK)[13] which is currently at version 3.0 and only available for the Windows operating system. It includes a stripped NetBeans (see below) environment specialized for Java ME development which makes it possible to keep all development in one application. The old version (2.5.2) is available for other platforms as well (Linux and Solaris) although only for 32 bit systems. The old version includes the tools for preverification and signing, but not a complete IDE.

[12] See http://europe.nokia.com/get-support-and-software/download-software/nokia-pc-suites

[13] See http://java.sun.com/javame/index.jsp.

4.7.2 Nokia SDK

Nokia offers its S40 SDK[14] which is basically a separate device configuration for the Sun WTK with a different emulator skin and some additional APIs like the Contactless Communication API and Contactless Communication Extension API and in the case of the NFC SDKs for the Nokia 6131 NFC and Nokia 6212 Classic[15] an emulator.

4.7.3 NetBeans

Sun's Java ME SDK uses a minimal NetBeans[16] environment as its IDE, but also the full NetBeans installation can be used with the NetBeans for Mobile Development[17] plugin. That way NetBeans can use any of the supported SDKs like the one from Sun, Nokia or an alternative like MicroEmulator[18] (with ProGuard as preverifier).

4.7.4 Eclipse

One can use for instance Eclipse[19] with the Mobile Tools for Java plugin[20]. and use either the Sun WTK, Nokia SDK or MicroEmulator. ProGuard is unfortunately not yet supported as a preverifier which makes Eclipse still require Sun's WTK `preverify` tool.

4.7.5 Building MIDlet Suites

Ant

Ant[21] is a build system for (primarily) Java which can be used to deploy Java source code from the command line (or automated).

Antenna

Antenna[22], a project developed to integrate Sun's WTK with Ant. It provides all the Ant tasks necessary to deploy MIDlet suites.

[14] See http://www.forum.nokia.com/info/sw.nokia.com/id/cc48f9a1-f5cf-447b-bdba-c4d41b3d05ce/Series_40_Platform_SDKs.html.
[15] See http://www.forum.nokia.com/info/sw.nokia.com/id/5bcaee40-d2b2-4595-b5b5-4833d6a4cda1/S40_Nokia_6212_NFC_SDK.html.
[16] See http://www.netbeans.org.
[17] See http://www.netbeans.org/features/javame/.
[18] See http://www.microemu.org.
[19] See http://www.eclipse.org.
[20] See http://www.eclipse.org/dsdp/mtj/.
[21] See http://ant.apache.org.
[22] See http://antenna.sourceforge.net.

JAD Ant Tasks

We created our own Ant tasks (JAD Ant Tasks[23]) for creating the JAD file and (optionally) signing the MIDlet suite. ProGuard can be used for the preverification. ProGuard has an Ant task included so it is easy to interface with it from Ant. We use the APIs included with the MicroEmulator project so we have a Java only solution. We can use the JSR-257 and JSR-257 extension APIs from the Nokia SDK or create empty stub classes (from the API documentation) as they are only required for compilation.

4.7.6 Emulators

The Nokia SDK includes an emulator that has the ability to emulate virtual smart cards for testing the application being developed. The emulator seems to provide support for NFCIP communication, but this unfortunately does not work after various attempts. The Nokia SDK is limited to the Windows operating system.

MicroEmulator includes a (platform independent) emulator that can run general purpose MIDlet suites and in combination with the BlueCove[24] project has Bluetooth emulation[25] support.

[23]See the README file in JAD Ant Tasks archive at http://nfcip-java.googlecode.com/ for more information on how to use it.

[24]A JSR-82 implementation for Java SE, see http://www.bluecove.org.

[25]See http://www.bluecove.org/bluecove-emu/.

Chapter 5

The OV-Chip 2.0 Software

Introduction

As part of the OV-Chip 2.0 project software was developed to implement the protocol described in chapter 6. It targets Java SE (for the host) and Java Card for the smart card. The host communicates with a card reader through the `javax.smartcardio.*` API to emulate either a gate, conductor or recharge station.

Our goal here is to modify the software to use NFC(IP) on the Java ME platform. We have discussed the NFC communication in chapter 1 and the Java ME platform in chapter 3.

The reason for doing this is that we want to compare the performance of the Java Cards with the performance of mobile phones like the Nokia 6131 NFC and Nokia 6212 Classic. The performance issues of the Java Card are described in [5]. We compare the effect of porting to Java ME with Java Card in section 5.2.

This chapter will describe the procedure we followed in porting the software to Java ME and make it use the library we designed in chapter 1. We first describe the current structure of the software. For more information one can take a look at the released source code[1] and documentation of the OV-Chip 2.0 project.

Due to the extensive differences between the Java ME and Java Card platform we started looking at creating a much simpler design for the communication protocols in section 5.3. Due to the limited resources on the Java Card the code is very "low level" and very conservative in its use of resources. This is a good thing for the Java Card platform, but for the Java ME platform more resources are available and readability of the code and adhering to object oriented programming principles can also be a big win.

[1]The source code can be found on the OV-Chip 2.0 project website at `http://www.sos.cs.ru.nl/ovchip/`.

5.1 Porting Existing Software

The porting involves multiple steps. What currently exists is a "host driver" (running on a standard PC with Java SE) and an "applet" which is a Java Card applet designed to run on a smart card. To talk from a Java SE to a smart card reader the `javax.smartcardio.*` API is used. APDUs are sent back and forth. Due to Java Card 2.2.1 restrictions an APDU is limited in size to 255 bytes. So to deal with this, a custom layer was written to create an "unlimited" byte array transfer. Two main steps are required for successful porting:

1. Replace the existing communication layer with the NFCIP communication layer between the Java SE host and Java ME phone.
2. Port the code of the applet from Java Card to Java ME.

First we will describe how to get the current system to work for analysis. Then we start with analyzing the currently existing communication channel, in particular the data that is transferred between the host and applet and describe a method to port this to the NFCIP communication layer. Something that would make the porting easier would be the existence of an "applet" written for Java SE without any communication channels involved and just calling methods directly, so we look into creating this as well. We focus on porting the test project, not anything related to the real OV-Chip protocol yet as that is not essential to compare the performance of the Java Cards and the mobile phone.

5.1.1 Setup

We need the OV-Chip 2.0 code from CVS, Java Card SDK 2.2.1, the JCOP emulator `jcop`, the JCOP `offcard.jar` library, Sun Java 6 (or OpenJDK) and GlobalPlatformManager (included in the source distribution). The JCOP emulator and `offcard.jar` library are not publicly available anymore unfortunately.

The OV-Chip 2.0 source can be found on the OV-Chip 2.0 website[2]. To extract and start the configuration:

```
$ tar -xzf ov-chip-2-2009-06-26.tar.gz
$ cd ov-chip-2-2009-06-26/
$ cp ConfigMakefile.in ConfigMakefile
```

Set the configuration flags `JAVA_HOME`, `JCKIT221`, `GPM` and `OFFCARD` in `ConfigMakefile`.

Now, to compile everything:

```
$ cd test
$ make all jcopio-jar
```

[2]See http://www.sos.cs.ru.nl/ovchip/.

We can now start the JCOP emulator:

```
$ /path/to/jcop -port=8015
```

Now, we can run the host driver and test the applet:

```
$ ./test_host -jcop -ping
Applet selection failed, reinstall.
Load applet _java_build_dir/card/test/ov_test/Java Card/ov_test.cap.
DEBUG: packagePath: ov_test/Java Card/
DEBUG: package: ov_test
DEBUG: package AID: 6F 76 5F 74 65 73 74
DEBUG: applet AIDs: [6F 76 5F 74 65 73 74 2E 61 70 70 ]
Applet loading finished.
Install applet ov_test.app of package ov_test.
Installation arguments : C910.00F4.0102.0082.0005.0000.0122.EAAE.1CB8
Install finished.
################# ping card start
################# ping card finished
$
```

Verbosity of the messages can be increased with "-d", "-dd" or "-ddd". Other tests can be performed by changing "-ping" to "-data-check" or "-mult-check" (see "-help").

5.1.2 Analysis of Data Communication

We start with compiling the software "as is" and run the host application in combination with the JCOP emulator. This will let us analyze the low level data communication. Together with the existing documentation[3] we hope to get a better overview of the underlying communication.

Data Check

In order for "-data-check" to be a bit easy to understand what is going on, we removed all the tests except the first[4]. This can be done in `test/Check_data_transmission.java`:

```
$ ./test_host -jcop -data-check -dd
Connecting to the jcop emulator ... connected
Select applet CLA:00 INS:A4 P1:04 P2:00 NC:0B no NE
   data: 6F76.5F74.6573.742E.6170.70
HP Start applet status with 0 arguments 6 results
HP long apdu send 0 bytes receive 17 bytes
HP send bytes 0 - 0 receive bytes 0 - 17
```

[3]We look at the `hacker-guide.html` file located in the `doc` directory.

[4]The `test/Test_host.java` class contains a bug where the `-data-check` actually is `-data-perf`.

```
HP send apdu CLA:00 INS:01 P1:00 P2:00 no data NE:11
Status OK (0x9000), received 17 bytes data: 00F40102008200050000122EAC1913001
Connected to applet created at 5 August 2009 15:31:10 CEST
with maximal sizes 244, 258, 130 and 5.
Applet was compiled WITH assertion checks.
Applet status matches, continue.
################# check data start
######## check size 1 start
### set_size start
HP Start data set size with 2 arguments 1 results
HP long apdu send 21 bytes receive 20 bytes
HP send bytes 0 - 21 receive bytes 0 - 20
HP send apdu CLA:00 INS:05 P1:00 P2:00 NC:15 NE:14
   data: 0001.0001.0001.0001.0001.0001.0001.0001.0001.0001.00
Status OK (0x9000), received 20 bytes data: 0001000100010001000100010001000100010001
### set_size finished
### check data start
HP Start data check with 5 arguments 5 results
HP long apdu send 5 bytes receive 5 bytes
HP send bytes 0 - 5 receive bytes 0 - 5
HP send apdu CLA:00 INS:04 P1:00 P2:00 NC:05 NE:05
   data: 0001.0203.04
Status OK (0x9000), received 5 bytes data: 0001020304
### check data finished (88.664 ms)

######################################################
#################
################# check data successfully finished
#################
######################################################
$
```

This shows that first protocol 5 is executed, which is used to set the sizes of the buffers. There are 5 arguments, and 5 results. The value indicates the size of the argument. So first the size of the arguments is set, after which the actual arguments are constructed and sent. Here are 5 arguments of size 1 which makes 5 bytes. Which is exactly what is sent using protocol 4.

In addition to the data, what is important for us are the fields `INS` which is the protocol number, and `P1` which indicates the protocol step. We may also consider using the `NC` and `NE` bytes which indicate respectively the number of data bytes sent and expected, however these cannot be used in the same way because they only have a 1 byte size which limits it to denoting 255 bytes. The `P2` field is irrelevant as it contains the "batch" number which indicates the (255 byte limited) packet belonging to a protocol step.

5.1.3 Communication

We can take two possible approaches to modifying the communication layer:

1. We leave the APDU size limited messages intact (including all the administration of block numbers which would then also include the field `P2`)
2. We remove the APDU size limited messages completely and we take into consideration the use of an "unlimited" communication channel where only the bytes `INS` and `P1` are retained (indicating the protocol number and protocol step).

We chose the latter approach for efficiency reasons as our NFCIP communication layer, from chapter 1 already takes care of this splitting and merging by itself. Additional benefit is that we can also create a way of communicating that does not use the NFCIP communication layer, but instead calls directly in the MIDlet code as to avoid dealing with the real communication layer for testing purposes.

Host

The host code uses `javax.smartcardio.CardChannel` throughout the classes. We can replace the communication `CardChannel` with a class that implements the same interface as the `CardChannel` class and uses the new communication layer from that point on.

We modify the following:

- Remove `Card_terminal.java` (we do not need it anymore)
- Modify `Host_protocol.java` (remove `send_apdu_message` and modify `send_long_apdu` to create a byte array that contains the `args` and is prepended with `protocol_id` and `step_id` before initiating the `transmit`)
- Modify `Test_host.java` (this will initiate the communication channel and send data to the `Host_protocol.java` layer.
- Add `DataChannel.java` (this will contain the new low level communication channel, see below)

The `DataChannel` interface will have a `transmit` method with signature `byte[] transmit(byte[])` instead of `ResponseAPDU transmit(CommandAPDU)`, see appendix E.

MIDlet

The applet code uses the APDU class in combination with `CardChannel` throughout the applet. This needs to be modified to use the "new" communication channel.

We modify the following:

- Modify `Card_protocol.java` (replace the method `process` and remove the methods `process_send` and `process_receive`, see appendix E)

- Modify `Protocol_applet.java` to call into the modified `Card_protocol.java`. We no longer have APDUs but a byte array instead where the first byte indicated the protocol number (`INS`). This class would extend `MIDlet` and provide for all the MIDlet control methods (see appendix B for a minimum MIDlet example)
- Modify `Test_applet.java` to extend `Protocol_applet.java` and handle installation arguments. We use the installation arguments, but hard code them for now. See appendix E

5.1.4 Code

There are some things we have to consider when porting software from Java Card to Java ME:

1. We need to remove `RSA_exponent` and all its references as the phone has no cryptographic processor (modify `Bignat_protocols.id`)
2. Deal with assertions (use `Misc.myassert`) as Java ME does not support assertions
3. Replace all the signatures/hashes method calls with BouncyCastle[5] calls
4. Replace Java Card `Random` class with Java ME `Random` class
5. Take care of the installation arguments as Java ME does not have those (in the same way as Java Card)

This still leaves some loose ends untied, like imports that are only available on the Java Card platform, `JCSystem` calls in the code for copying arrays and (temporary) allocate memory. Also the `ISOException` class is not available, neither is the `ISO7816` class. These can easily be created for Java ME by looking at the Java Card API documentation.

The source code of some of the more intrusive modifications can be found in Appendix E. These should help in porting the source code of the official release as our porting effort was done using an old version from CVS.

5.2 Performance Comparison

Figure 5.1 shows a comparison between the Java Card (wired and wireless) and on the Nokia 6131 NFC and Nokia 6212 Classic.

For real world usage we would need a key of about 1200 bits (and an exponent of around 160 bits). As a comparison we will create a MIDlet that performs the same operation using the OpenJDK `java.math.BigInteger` class. We will use these exact same values (dump from performance testing Bignat):

[5]See http://www.bouncycastle.org.

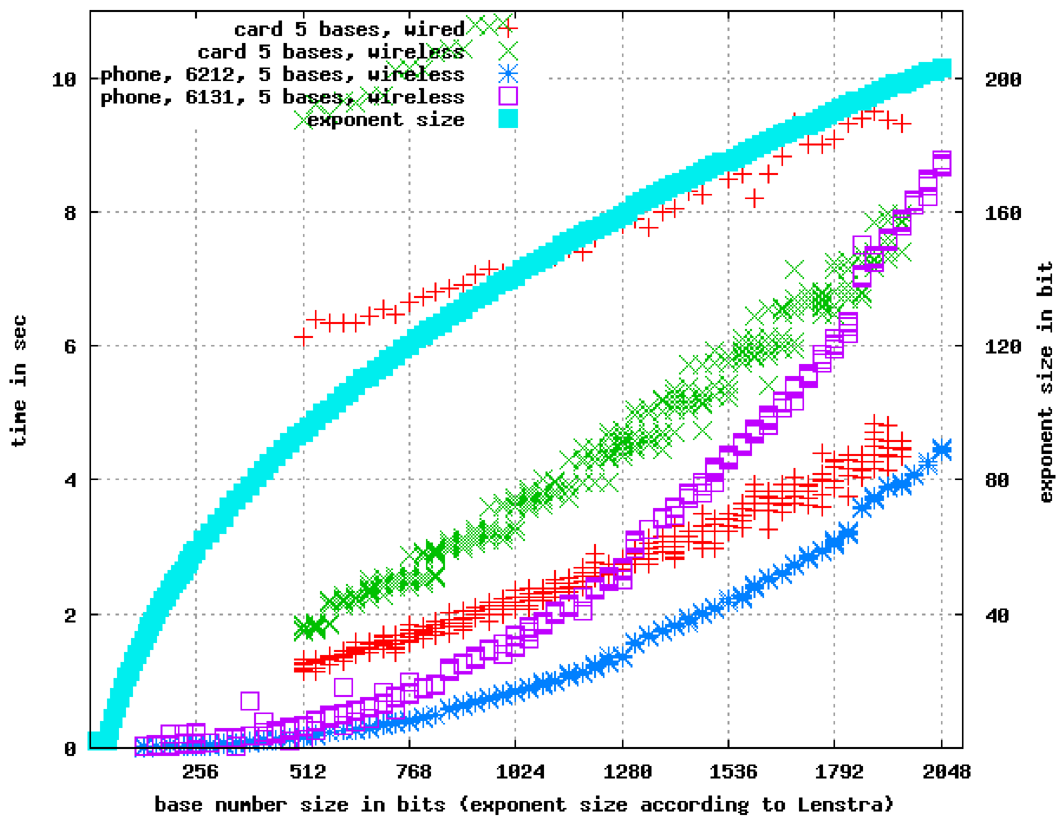

Figure 5.1: Performance Vector Exponentiation with 5 bases on Java Card and Nokia phones

```
## Effective size base 160 bytes (1280 bits) exponent 159 bits (in 20 bytes)
size 1280 ^ 159 1.351 s (0.081 s 1.432 s)
size 1280 ^ 159 1.337 s (0.087 s 1.424 s)
size 1280 ^ 159 1.352 s (0.080 s 1.433 s)
size 1280 ^ 159 1.359 s (0.081 s 1.440 s)
size 1280 ^ 159 1.355 s (0.085 s 1.440 s)
```

And for 2048 bits:

```
## Effective size base 256 bytes (2048 bits) exponent 203 bits (in 28 bytes)
size 2048 ^ 203 4.431 s (0.084 s 4.515 s)
size 2048 ^ 203 4.461 s (0.084 s 4.545 s)
size 2048 ^ 203 4.467 s (0.082 s 4.549 s)
size 2048 ^ 203 4.439 s (0.083 s 4.522 s)
size 2048 ^ 203 4.432 s (0.084 s 4.517 s)
```

We will generate 5 random numbers of 1280 bits and 5 random numbers of 159 bits and a modulus of 1280 bits.

As the `java.math.BigInteger` class does not support "vector exponentiation" we will calculate the exponentiations separately (using the `modPow` method) and then multiply the results again. This is not as efficient as the BigNat approach, but is the best available without hacking the `java.math.BigInteger` class. We will perform the calculations directly on the phone and not measure the generation of the random numbers.

We will compare Bignat (from the OV-Chip 2.0 project), BigInteger from the OpenJDK sources and BigInteger[6] from BouncyCastle[7] on the Nokia 6212 Classic. The results are shown in table 5.1.

In order to calculate the results with the OpenJDK and BouncyCastle BigInteger class we used the following code (we ran it 5 times in sequence to get an average):

```
for (int i = 0; i < VEC_LENGTH; i++) {
    result = result.multiply(b[i].modPow(e[i], m)).mod(m);
}
```

Where `VEC_LENGTH` is the number of exponents, which is 5 in this case, `b` and `e` are the bases respectively the exponents, `m` is the modulus and `result` is initialized with `BigInteger.ONE`. The bases, exponents and modulus were generated with the BigInteger constructor `BigInteger(int, Random)` where the integer represents the requested size (in bits) and Random is a random instance.

As we can see, the BouncyCastle implementation performs really bad compared to the Bignat and OpenJDK implementation. The efficient "vector exponentiation" in the Bignat library seems to provide some benefits to the "manual" exponentiation used with the OpenJDK library.

[6] We used `java.math.BigInteger` from OpenJDK sources (from Fedora OpenJDK source package: `java-1.6.0-openjdk-src-1.6.0.0-27.b16.fc11.x86_64`).

[7] We used BouncyCastle for J2ME version 143 (the latest available at time of testing).

Library	Bit Sizes (b/e/m) (1280/159/1280)	(2048/203/2048)
Bignat	≈ 1.4 seconds	≈ 4.5 seconds
OpenJDK	≈ 2.5 seconds	≈ 7.0 seconds
BouncyCastle	≈ 50 seconds	≈ 173 seconds

Table 5.1: Performance comparison between BigInteger libraries

5.3 Alternative Implementation

Because the Java ME and Java Card platform differ a great deal an attempt was made to create security protocols based from scratch aimed at the Java ME platform. It is possible to make the design a simpler if it is possible to create more abstractions because of the increased resources available. The problem with this approach is the lack of an efficient modulo power function. The one provided by the OV-Chip 2.0 implementation is more efficient as was shown in section 5.2.

5.3.1 Remote Method Invocation

To make a security protocol work one has to send and receive data. There are different ways to do this. When using a system like Java one can use “Remote Method Invocation“ (RMI) to access remote objects from a local system with a “client” and a “server” model. The client would for example be the terminal and the server the travel document smart card. Security protocols work usually in a sense that there is some data transfer back and forth between client and server in which they for example try to convince each other of their identity.

So what is needed for communication between terminal and travel document, whether this travel document is a smart card or a phone, is a transport layer for byte arrays. For smart cards this is already used in the form of APDUs. These are byte arrays containing serialized objects and some instructions for RMI to determine what method to execute at the client side.

For building security protocol, something extra is needed and not just plain RMI. A travel document can play different roles. The role being played is different when entering the public transport system by checking in at a gate, or when recharging the document at a reload station. So methods available through RMI belong to a role, and also to a certain ordering of calling those methods. It is also good security practice to make it impossible for the steps of the protocol to be called out of order.

Using RMI makes sense because it becomes easy to see how the protocol really works (a mapping between the (informal) description and the actual code). By abstracting all the

underlying data transfer, imposing security limitations (like making it impossible to call methods out of order) all one has left to do is program against the interface.

There are some problems with using Java ME for RMI in that it does not have Java features that would be really helpful in this case. There is no “serialization” support which means that objects have to be converted to and from byte arrays manually. Furthermore there is no RMI support on the phone so there also needs to be a custom RMI layer that takes care of calling the right methods and protocol on the phone using a low level data structure specifying which method and protocol to call at a specific time.

The concepts behind the implementation of such an RMI layer are relatively simple. One has the concepts of a client and server. There is an interface, or actually multiple interfaces for multiple protocols and there are the RMI stub classes that take care of handling the conversions to and from method calls and parameters and return values. The RMI system will be explained with a simple example in which all the steps will be shown.

5.3.2 Design

There is an interface specifying what the server needs to implement. The client can call these methods transparently (through the RMI system) on the server. For example we have the following interface:

```
package ds.project;

public interface EncryptionProtocol {
    public void setKey(BigInteger key) throws RMIException;
    public void setData(byte[] data) throws RMIException;
    public void encrypt() throws RMIException;
    public byte[] getResult() throws RMIException;
}
```

This is a simple example of what one could implement on the server. Of course this cannot be seen as a secure protocol, but it merely demonstrates how the RMI system works. In terminology used in the RMI system this interface is a *protocol* with public methods that are called *steps*. The steps belong to the protocol and need to be accessed in the order specified in the interface. So one has to call `setKey` before `setData`. In case there are multiple protocols (interfaces implemented by the server) it is impossible to switch to another protocol once a certain protocol has started, that is, the first method of that protocol was called.

Then there is server code that implements this interfaces and implements all the methods from this interface. The sequence of events:

1. The client instantiates a RMIClient object. The RMIClient takes care of converting method arguments to byte arrays for all methods specified in the interface.

2. The RMIClient opens a connection to the server (over whatever medium)
3. The client calls a method of RMIClient which implements the interface
4. The method in RMIClient serializes the arguments and adds a protocol and method number to be called on the RMIServer (running on the server)
5. The RMIServer receives the data from RMIClient and decodes the protocol number and jumps to the relevant method handling code
6. The data is checked (i.e.: correct number of parameters, correct method call, correct protocol)
7. The data is deserialized into its objects again and the corresponding method is called in the Server

We created a protocol generator that takes the interfaces one by one and extracts the information required to create RMIServer and RMIClient classes (using the Java Reflection API). All protocols (interfaces) and protocol steps (methods) get a unique number. For instance, the interface above would have the following step numbers:

```
private final byte SET_KEY = (byte) 10;
private final byte SET_DATA = (byte) 11;
private final byte ENCRYPT = (byte) 12;
private final byte GET_RESULT = (byte) 13;
```

This can all be done automatically by the reflection API. This is also of use when the conversion from and to byte arrays needs to be performed. With the reflection API one can determine the type of the arguments and thus specify a method for converting it to and from byte array. So we have for example methods to convert a `BigInteger` to and from byte array, but also a primitive type like `int`, or even arrays of objects or primitive types. We just have to make sure the conversion helper methods exists and the rest will be done transparently.

5.3.3 Communication

The data being sent and received is nothing more than a byte array with the (serialized) data and a header. The header contains the protocol number, the step number and the number of parameters. Format (`B` = byte, `S` = short):

```
<B protocol number> <B protocol step> <B number of parameters>
    <S length of param_1> ... <S length of param_n>
        <param_1> ... <param_n>
```

Examples

```
0x06 0x0F 0x00
     (protocol 6, step 15, no parameters)

0x05 0x0A 0x01 0x00 0x08 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07
    (protocol 5, step 10, 1 parameter (which happens to be a byte array of
     size 00 08 (8 bytes))
```

5.3.4 Protocol Implementation

Implementing the protocol itself becomes easy now. One has to make sure all the argument types are supported (manually) and then create an interface (or multiple interfaces) specifying the protocols with their steps. Once this is done the protocol generator can be run to create the RMIClient and RMIServer classes. The protocol developer only needs to worry about the interfaces and the code in the Server respectively Client classes.

This approach is less efficient for the Java Card platform, but perfectly suitable for the Java ME platform. We started out with this approach and got it to work well over both Bluetooth and NFCIP. However, due to the lacking performance of the OpenJDK BigInteger library we abandoned this project and focused on porting the existing OV-Chip 2.0 code to the Java ME platform.

Using the BouncyCastle BigInteger library we performed some tests (b = 1280, e = 159, m = 1280 bits, number of exponents = 3). The issuance phase takes 254 seconds. The verification phase takes 117 seconds (as described in section 6.1):

```
Issuance & Initialization: 254.22016825 seconds
Verifying attributes: 117.302834391 seconds
This is a valid Second Class ticket
```

Assuming the OpenJDK BigInteger library is a factor 20 faster (we do not have time to verify the results unfortunately), we would expect the time it takes to validate the attributes to be around 5 seconds, assuming the problem lies with the inefficiency of calculating the vector exponent with BouncyCastle.

If anyone ever continues with an implementation for the Java ME platform of security protocols we would seriously recommend using the approach described above[8].

[8] The current source code for the project is hosted in the repository of nfcip-java in the rmi-sp SVN repository directory at http://nfcip-java.googlecode.com.

Chapter 6

The Protocol

This chapter will detail the protocol as suggested by[4] to the extend necessary to understand the performance issues. The protocol is not finalized yet and some things are still missing or are not secure yet. However, in the current status of the protocol repeated exponentiations modulo a large number are required. We will take a look at the protocol and use an example (with small numbers) to show how it works.

Introduction

The goal of the protocol is to provide a privacy friendly and secure public transport payment system. The is accomplished by using cryptographic techniques like RSA, cryptographic hashes, zero knowledge proofs and selective disclosure protocols.

RSA: a public key cryptography system designed by Rivest, Shamir and Adleman. It works with two keys where one is publicly known and one is kept private.

Hashes: a mathematical one way function that generates an unique number based on the input where it is difficult to find different inputs that generate the same result

Zero knowledge proof: a (mathematical) proof where you prove knowledge of a fact without actually disclosing this fact

Selective Disclosure: from a set of knowledge you only want to provide one fact without revealing the rest or make it possible to link facts retroactively to the same owner

Signatures: an encryption (with the private RSA key) of the hash of a document one wants to sign. The public key can be used to restore the hash and one can calculate the hash of the document and compare it to the decoded hash

The protocol encompasses a wide field of cryptography. The basis for the protocol described here was devised by Stefan Brands in "Rethinking Public Key Infrastructures and Digital Certificates; Building in Privacy" [1] with protocols designed in "Protocols for Public Transport Payment Systems"[4].

In this chapter we will give an extensive example of the attribute proving and relevant information for an implementation of it that is programming language agnostic, but easy to implement in any language with the information presented here.

Some issues still don't have a clear solution in the protocol, like how to express that a value is for example larger than 10 but without disclosing this number.

We want to have a situation where a traveler does not have to disclose more than absolutely necessary, but cannot lie about this knowledge. For example one only wants to prove that one has a valid travel document, but not home address, date of birth or amount of money on the travel document or even the unique serial number. This is where the selective disclosure protocols come in. The protocol is described to some extend in Brands [1] and modified in "Protocols for Public Transport Payment Systems". So there are a few attributes linked to a travel document, like the type of card, the amount of money on the card and the expiry date.

6.1 Attribute Proving

6.1.1 Setting up RSA

We have two parties in this protocol. A verifier V and a prover P. The verifier sets up the RSA system by generating a key pair and a modulus. The public key and the modulus are public and will be used by the prover.

V chooses two prime numbers, p and q and performs pq to get the modulus n. V now picks a random v, the public key, with properties $v \in \mathbb{Z}^*_{\varphi(n)}$ and v is prime. Where $\varphi(n)$ is the Euler function, and can be calculated by $(p-1)(q-1)$ when p and q are both prime.

The private key d of the verifier can be found using Euclides' extended algorithm to calculate the inverse of the public key, $v^{-1} \pmod{n}$.

6.1.2 Setting up the Attribute Expression

We now choose k bases, for storing k attributes. The bases are generated once and made public. They are shared by all participants of the protocol. It is now possible to create the *attribute expression*:

$$A \equiv g_1^{a_1} \cdot \cdots \cdot g_3^{a_3} \pmod{n}$$

Here g_i are random numbers, co-prime to n and $g_i \in \mathbb{Z}_n^*$. The attributes can be unique for each prover in the protocol and contains for example balance, type of card and card number. For the attributes holds $a_i \in \mathbb{Z}_v$.

6.1.3 Proving knowledge of the RSA representation

P sends a *blinded attribute expression* to V, which is defined as $A'_c = b^v A$ where $b \in \mathbb{Z}_n^*$ and random. Also P sends a witness ω to V. V responds with a challenge $\gamma \in \mathbb{Z}_v$ and verifies the response of P to this challenge to determine whether P has indeed knowledge of the attributes. All the calculations are done modulo n except the calculation of r_i which is modulo v.

$$
\begin{aligned}
P \to V \quad &: \quad A'_c, \\
& \quad\;\; \omega \equiv \beta^v g_1^{\alpha_1} \cdot \ \cdots \ \cdot g_k^{\alpha_k} \\
V \to P \quad &: \quad \gamma \\
P \to V \quad &: \quad s \equiv \beta b^\gamma \prod_i^k g_i^{(\gamma a_i + \alpha_i) \text{ div } v}, \\
& \quad\;\; r_1 \equiv \gamma a_1 + \alpha_1, \\
& \quad\;\; \vdots \\
& \quad\;\; r_k \equiv \gamma a_k + \alpha_k
\end{aligned}
$$

Here $\beta \in \mathbb{Z}_n^*$ and random, $\alpha_i \in \mathbb{Z}_v$ and random. Also $\gamma \in \mathbb{Z}_v$ and random.

Now V checks whether the following holds:

$$
s^v \prod_{i=1}^k g_i^{r_i} \equiv (A'_c)^\gamma \omega
$$

If it holds then then P has proved knowledge of the attributes without disclosing them.

6.1.4 Disclosing certain attributes and proving them

Suppose now that one wants to disclose one attribute without disclosing the other attributes. It is important that this disclosed attribute can be verified to be correct without disclosing the other attributes. The protocol below shows disclosure for one attribute, a_t, but is easily extended to disclosing multiple attributes.

$$
P \to V \quad : \quad A'_c,
$$

$$
\begin{aligned}
& & a_t, \\
& & \omega \equiv \beta^v g_1^{\alpha_1} \cdot \ \cdots \ \cdot g_{t-1}^{\alpha_{t-1}} \cdot \ \cdots \ \cdot g_{t+1}^{\alpha_{t+1}} \cdot \ \cdots \ \cdot g_k^{\alpha_k} \\
V \to P & : & \gamma \\
P \to V & : & s \equiv \beta b^\gamma \prod_{i,\ i \neq t}^{k} g_i^{(\gamma a_i + \alpha_i) \text{ div } v}, \\
& & r_1 \equiv \gamma a_1 + \alpha_1, \\
& & \vdots \\
& & r_{t-1} \equiv \gamma a_{t-1} + \alpha_{t-1}, \\
& & r_{t+1} \equiv \gamma a_{t+1} + \alpha_{t+1}, \\
& & \vdots \\
& & r_k \equiv \gamma a_k + \alpha_k
\end{aligned}
$$

Now V checks whether the following holds:

$$
s^v \prod_{i=1,\ i \neq t}^{k} g_i^{r_i} \equiv \left(\frac{A'_c}{g_t^{a_t}} \right)^\gamma \omega
$$

6.1.5 Selective Disclosure example

We are demonstrating the protocol above for the case where $k = 3$, so there are three attributes and we want to disclose the second attribute, $a_2 \equiv 10$, which indicates the type of the card. With this we want to prove that we have a valid first class travel ticket when checked by a public transport conductor.

6.1.6 Setting up RSA and the System Parameters

The verifier V generates two random numbers that should be prime, p and q. For the example we will choose $p = 13$ and $q = 17$. This results in a public modulus $n = pq = 221$. Furthermore V generates a public key v and calculates the private key d from this public key. We choose as a public key 101 here. The complementing private key would be $101^{-1} \equiv 173 \pmod{n}$.

Now we also choose the public bases, $g_1 \cdots g_k$. P is customized by private attributes $a_1 \cdots a_k$ which remain secret during the proving process.

$$
\begin{aligned}
n &= 221 && \text{the public modulus} \\
v &\equiv 101 && \text{the public key of } V \\
d &\equiv 173 && \text{the private key of } V
\end{aligned}
$$

$$
\begin{aligned}
g_1 &\equiv 108 \\
g_2 &\equiv 55 \\
g_3 &\equiv 95 \\
a_1 &\equiv 25 \text{ (ticket number)} \\
a_2 &\equiv 10 \text{ (ticket type)} \\
a_3 &\equiv 100 \text{ (ticket balance)}
\end{aligned}
$$

6.1.7 Disclosing An Attribute

Now we can calculate the attribute expression A:

$$
\begin{aligned}
A &\equiv g_1^{a_1} g_2^{a_2} g_3^{a_3} \pmod{n} \\
&\equiv 108^{25} \cdot 55^{10} \cdot 95^{100} \\
&\equiv 160 \pmod{221}
\end{aligned}
$$

Now we calculate the blinded attribute expression A'_c with blinding vector b we randomly choose here as being 38:

$$
\begin{aligned}
b &\equiv 38 \\
A'_c &\equiv b^v A \\
&\equiv 38^{101} A \\
&\equiv 113 \pmod{221}
\end{aligned}
$$

We construct the witness expression by randomly choosing α_i for all the bases which attribute we don't publish. In this case we disclose a_2 so we leave out g_2 and thus there is also no α_2. We also randomly choose a β:

$$
\begin{aligned}
\beta &\equiv 16 \\
\alpha_1 &\equiv 54 \\
\alpha_3 &\equiv 44
\end{aligned}
$$

We can now calculate the witness expression ω:

$$
\begin{aligned}
\omega &\equiv \beta^v g_1^{\alpha_1} \cdot g_3^{\alpha_3} \\
&\equiv 16^{101} \cdot 108^{54} \cdot 95^{44} \\
&\equiv 66 \pmod{221}
\end{aligned}
$$

For the random challenge γ, we take $\gamma \equiv 87$. We now calculate the responses r_i and s:

$$
\begin{aligned}
r_1 &\equiv 87 \cdot 25 + 54 = 7 \pmod v \\
r_3 &\equiv 87 \cdot 100 + 44 = 58 \pmod v
\end{aligned}
$$

We calculate the response s:

$$
\begin{aligned}
s &\equiv 16 \cdot 38^{87}[108^{(87 \cdot 25+54)\ div\ v} \cdot 95^{(87 \cdot 100+44)\ div\ v}] \\
&\equiv 101 \pmod n
\end{aligned}
$$

Now V checks whether the following holds:

$$
\begin{aligned}
s^v \prod_{i=1,\ i \neq t}^{k} g_i^{r_i} &\equiv \left(\frac{A'_c}{g_t^{a_t}}\right)^{\gamma} \omega \\
101^{101} \cdot [108^7 \cdot 95^{58}] &\equiv \left(\frac{A'_c}{g_2^{a_2}}\right)^{\gamma} \omega \\
40 &\equiv \left(\frac{113}{55^{10}}\right)^{87} \cdot 66 \\
&\equiv 40 \pmod{221}
\end{aligned}
$$

It does, so this concludes the proof of knowledge and correctness of a_2.

Chapter 7

Nokia S40 Security

This chapter was removed from the public version of this document due to security issues we found on the Nokia S40 platform.

Chapter 8

Future Research

There are still many aspects open for research, we name a few:

- It would be interesting to investigate the JSR-177 API to see if the performance of the "vector exponentiation" can be improved by using the cryptographic functions (as was done on the Java Card platform)
- Combine Bluetooth with NFC to create fast and reliable connection and just use NFC to exchange keys for easy connection setup (this is also required for a possible attack as described in chapter 7)
- Investigate the possibility of the use of LLCP instead of NFCIP for the peer to peer communication between computer and phone[1]
- Try to improve the performance of "vector exponentiation" by improving the OpenJDK `BigInteger` source
- Take a look at the Elliptic Curve variant of Brands' algorithm to improve the performance of the protocol as smaller numbers are required there.
- **This item was removed from the public version of this document due to security issues we found on the Nokia S40 platform.**
- It would be nice if we could find a way to create a symmetric `nfcip-java` library where both initiator and target have the same code that does not have too many conditionals
- We would like to find a way to register push registrations immediately on MIDlet suite installation, now they require a reboot (or a manual launch of the MIDlet suite before it is registered)

[1] See http://wiki.forum.nokia.com/index.php/Simple_NFC_LLCP_peer-to-peer_Chat.

Chapter 9

Conclusion

The way to achieving the test results that we presented in chapter 5 was a long one. We had significant difficulty in creating stable communication between the NFC reader and the phone which took huge amounts of time analyzing, programming and testing. A serious problem with this was that there was no emulator available so we had to test on real hardware. This work resulted in the NFCIP-JAVA project that makes it possible to reliably communicate between a PC with card reader and the mobile phone.

While analyzing the phone we found out that it is insecure to use the Record Management Store (RMS) for storing keys as they can easily be extracted by widely available software, so we had to focus on the secure element available in the Nokia 6131 NFC and Nokia 6212 Classic. That way the keys are safe against extraction. See for more details chapter 2.

Creating software for the mobile phone also took us along a number of different software projects, libraries and IDEs where in the end we decided to create our own solution to create MIDlet suites for the mobile phone to be able to automatically build the suites from the command line in one step.

To come back to our initial question **"Can a mobile phone be used to efficiently implement the** OV-CHIP 2.0 **protocol?"** we have to answer that the phone (if we are talking about just the Nokia 6212 Classic) is indeed faster than the smart cards that were used for the performance test as was shown in figure 5.1. However, the margin of improvement is close to negligible. The Java Card solution has the benefit of using the cryptographic co-processor to calculate the vector exponents where the phone needs to rely on just the main CPU. Therefore, using a phone as a replacement for a Java Card smart card at this time does not lead to an increased performance.

This paragraph was removed from the public version of this document due to security issues we found on the Nokia S40 platform.

Bibliography

[1] Stefan A. Brands. *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy.* MIT Press, Cambridge, MA, USA, 2000.

[2] ECMA. *ECMA-340: Near Field Communication — Interface and Protocol (NFCIP-1).* ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, December 2004.

[3] Flavio D. Garcia, Gerhard de Koning Gans, Ruben Muijrers, Peter van Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs. Dismantling mifare classic. In Sushil Jajodia and Javier Lpez, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 97–114. Springer, 2008.

[4] Wouter Teepe, Bart Jacobs, and Jaap-Henk Hoepman. Protocols for public transport payment systems. Unpublished, 2009.

[5] Hendrik Tews and Bart Jacobs. Performance issues of selective disclosure and blinded issuing protocols on java card. Paper accepted at WISTP 2009, 2009.

Appendix A

Padding Scheme

When sending the last block we know it has length less than the maximum block size otherwise it would have been sent in the preceding loop. So we can always pad this block with an indicator as to whether it was padded or not, of size 1 byte. The first byte is either 0x00 or 0x01 indicating padding. When the padding is enabled, after the indicator byte we have a number of 0x00s followed by a 0x01 indicating the end of the padding. Suppose we want to send 10 bytes in the last block. We can not do this as the NFC reader breaks in that case. We add padding to increase the length of the data block to 23. We need to add 13 bytes, of which one is the padding indicator, 11 are padding zeros and 1 is the end of padding indicator as shown in table A.1.

Data (10 bytes)	0xa0 0xa1 0xa2 0xa3 0xa4 0xa5 0xa6 0xa7 0xa8 0xa9
Padded Data (23 bytes)	**0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01** 0xa0 0xa1 0xa2 0xa3 0xa4 0xa5 0xa6 0xa7 0xa8 0xa9

Table A.1: Padding scheme

Table A.2 shows some paddings for other amounts of data one can send in the last block.

Padding (for 20 bytes)	0x01 0x00 0x01 *20 bytes of data*
Padding (for 21 bytes)	0x01 0x01 *21 bytes of data*
Padding (for 22 bytes)	0x02 *22 bytes of data*
Padding (for 0 bytes)	0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 *no data*

Table A.2: Some corner case paddings

In case we want to send larger amounts of data the padding indicator is set to 0x00 and there is no padding, the other side just has to remove this padding indicator.

For this to work both sides have to send the amount of data that is being transferred by the NFC chip. So when DID is used this means blocks of 236 bytes, and without DID 240 bytes per block. This way one can predict the splitting behavior of the NFC chip, which is required for this padding to work.

Appendix B

Hello World MIDlet

This is (minimal) complete example code for a functioning MIDlet that displays “Hello World!” on the phone screen.

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Form;

public class MyMIDlet extends MIDlet {
        private Form form;
        private Display display;

        public MyMIDlet() {
                form = new Form("MyMIDlet");
                display = Display.getDisplay(this);
                display.setCurrent(form);
        }

        protected void destroyApp(boolean arg0) throws
                MIDletStateChangeException {
        }

        protected void pauseApp() {
        }

        protected void startApp() throws MIDletStateChangeException {
                form.append("Hello World!\n");
        }
}
```

Appendix C

Obtaining File List From Phone

```
/**
 * List all file and directory entries under the current directory.
 *
 * @param ps
 *           the PrintStream to write the list to
 * @param startPath
 *           the start directory to list from (e.g.: "C:/")
 * @throws IOException
 *            if opening directory or obtaining file list fails
 */
public void recurseDirectory(PrintStream ps, String startPath)
        throws IOException {
    recurseDirectory(ps, startPath, 0);
}

private void recurseDirectory(PrintStream ps, String dirPath, int depth)
        throws IOException {
    String url = "file:///" + dirPath;
    /* indenting to structure output */
    for (int i = 0; i < depth; i++)
        ps.print("  ");
    ps.println(dirPath);
    FileConnection fc = null;
    try {
        fc = (FileConnection) Connector.open(url);
    } catch (SecurityException e) {
        /* we don't have access ignore this entry, keep going */
    } catch (IllegalArgumentException e) {
        /* file name/path seems invalid, ignore it */
    }
    if (fc != null && fc.isDirectory()) {
        Enumeration f = fc.list("*", true);
        while (f.hasMoreElements()) {
            String file = (String) f.nextElement();
            recurseDirectory(ps, dirPath + file, depth + 1);
```

```
            }
        }
    }
```

Appendix D

Creating a Java Key Store

In order to sign MIDlet Suites one has to create or obtain a code signing certificate. The procedure will be described here in detail, however the signing (by a Certificate Authority) is outside the scope.

An easy way to create a certificate for code signing is to use `keytool` included with the Java Development Kit.

First we create a private/public key pair using the RSA algorithm. We use the default location of the key store which is `.keystore` in the user home directory. One can specify a different location for the key store with `-keystore /path/to/key store` added to every command:

```
$ keytool -genkeypair -keyalg RSA -alias MyCert
```

This will ask for Distinguished Name values like “Common Name”, “Organizational Unit”, “Organization”, “Country”, “State” and “Locality”:

```
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  F. Kooman
What is the name of your organizational unit?
  [Unknown]:  DS
What is the name of your organization?
  [Unknown]:  Radboud University Nijmegen
What is the name of your City or Locality?
  [Unknown]:  Nijmegen
What is the name of your State or Province?
  [Unknown]:  Gelderland
What is the two-letter country code for this unit?
  [Unknown]:  NL
Is CN=F. Kooman, OU=DS, O=Radboud University Nijmegen, L=Nijmegen,
   ST=Gelderland, C=NL correct?
```

```
  [no]:  y

Enter key password for <MyCert>
(RETURN if same as keystore password):
```

Now we can generate a Certificate Signing Request (CSR) in PEM format to be signed by the Certificate Authority (CA). This should be done by a CA whose root certificate is installed on the mobile phone so the phone actually trusts the application and places is in the "Identified Third Party Security Domain".

```
$ keytool -certreq -alias MyCert -file MyCert.csr
```

The CSR file will be signed by the CA which will return a signed certificate, we assume here that the file we receive is called `MyCert.crt`. We need to convert this certificate to DER format for `keytool` to be able to handle it. We use OpenSSL[1] for this:

```
$ openssl x509 -inform PEM -in MyCert.crt -outform DER -out MyCert.der
```

Now this signed certificate can be imported in the keystore:

```
$ keytool -import -alias MyCert -file MyCert.der
```

In case this fails because the (root) certificate that was used for signing this certificate is not available in Java[2]. One can add root certificates to the key store (it needs to be in DER format) as well:

```
$ keytool -import -trustcacerts -alias MyRootCA -file MyRootCA.der
```

After importing the root certificate, importing the (signed) certificate should work.

Now the key store is ready to be used by applications like Eclipse with the MTJ plugin, NetBeans with the Mobility modules, Antenna and JadTool. These are all described elsewhere.

[1] See `http://www.openssl.org`.

[2] A number of CA root certificates are included in the default installation of the Java implementation. Certificates for Thawte and Verisign are included for instance, but not necessarily all root certificates that are available on the phone. The list of certificates can be obtained by using `keytool -list -keystore /path/to/jre/lib/security/cacerts`.

Appendix E

OV-Chip 2.0 Porting Code

E.1 DataChannel.java

We create an interface `DataChannel` that has two implementations. One for using NFCIP and one for calling into the MIDlet code class. The interface specifies the `byte[] transmit(byte[])` method. We use the `nfcip-java` library[1]. The implementation for `NFCIPDataChannel` looks like this:

```
import ds.nfcip.se.NFCIPConnection;
import ds.nfcip.NFCIPException;

public class NFCIPDataChannel extends DataChannel {
    private NFCIPConnection m;

    public NFCIPDataChannel() {
        m = new NFCIPConnection();
        try {
            m.setTerminal(0);
            m.setMode(NFCIPConnection.FAKE_INITIATOR);
        } catch (NFCIPException e) {
            e.printStackTrace();
        }
    }

    public byte[] transmit(byte[] data) {
        try {
            m.send(data);
            return m.receive();
        } catch (NFCIPException e) {
            e.printStackTrace();
            return null;
        }
```

[1] See http://nfcip-java.googlecode.com.

```
    }
}
```

The implementation for calling directly into the MIDlet code looks like this:

```
public class FakeDataChannel extends DataChannel {
    private Test_MIDlet z;

    public FakeDataChannel() {
        z = new Test_MIDlet();
    }

    public byte[] transmit(byte[] data) {
        return z.process(data);
    }
}
```

With `FakeDataChannel` we can now test our porting efforts without running the code on the mobile phone. We could also easily write an alternative implementation using Bluetooth instead of NFCIP.

E.2 Host_protocol.java

We replace the method `byte[] send_long_apdu(CardChannel, byte[], int)` with `byte[] send_long_apdu(DataChannel, byte[] args)`:

```
private byte[] send_long_apdu(DataChannel ds, byte[] args) throws Exception {
    byte[] data = new byte[2 + args.length];
    data[0] = protocol_id;
    data[1] = step_id;
    System.arraycopy(args, 0, data, 2, args.length);
    return ds.transmit(t);
}
```

We can remove the method `int send_apdu_message(CardChannel, CommandAPDU, byte[], int, int)` as it is no longer needed with the replacement `send_long_apdu` method.

E.3 Card_protocol.java

For the MIDlet we replace the `void process(APDU, byte[])` method with `byte[] process(byte[])`. The contents of the method are as below:

```
public byte[] process(byte[] buf) {
    ASSERT_TAG(protocol != null, 0x09);

    Protocol_step step = protocol.steps[protocol_step];

    // Check protocol and step identifier and batch.
    if ((buf[0] & 0xff) != protocol.protocol_id) {
        clear_protocol();
        ISOException.throwIt(Response_status.OV_UNEXPECTED_PROTOCOL_ID);
    }

    if ((buf[1] & 0xff) != step.step_identifier) {
        clear_protocol();
        ISOException.throwIt(Response_status.OV_UNEXPECTED_PROTOCOL_STEP);
    }

    // Deserialize
    Convert_serializable.array_from_bytes(buf, (short) 2, step.arguments);

    // Execute step
    process_method(buf, step);

    // Serialize and return results
  return Convert_serializable.array_to_bytes(step.results);
}
```

E.4 Installation Arguments

The Installation arguments consist of some (maximum) sizes of the used data structures. The installation arguments used for example in chapter 5 are:

```
00F401020082000500000122EAC1913001
```

The first 8 bytes indicate the sizes of `short_bignat_size`, `long_bignat_size`, `double_bignat_size` and `max_vector_length`. The rest of the bytes indicate the version of the applet.

In `Test_applet.java` they are passed as arguments to the constructor. In the MIDlet we hard code `bytes`, `start` and `len` like this:

```
byte[] bytes = new byte[] { (byte) 0x00, (byte) 0xF4, (byte) 0x01,
        (byte) 0x02, (byte) 0x00, (byte) 0x82, (byte) 0x00, (byte) 0x05,
        (byte) 0x00, (byte) 0x00, (byte) 0x01, (byte) 0x22, (byte) 0xEA,
        (byte) 0xC1, (byte) 0x91, (byte) 0x30, (byte) 0x01 };
short start = 0;
byte len = (byte) Misc.length_of_serializable_array(install_arguments);
```

We also have to call the `process` method in `Protocol_applet` with a `byte[]` instead of an `APDU` and expect return data to be sent back over the NFCIP communication channel.

Appendix F

Java Card Example

In this appendix we show an example of how to program with Java Card. We create an applet that can run on the Secure Element of the Nokia 6131 NFC and Nokia 6212 Classic. We show the complete Java Card code and the accompanying code for use on a Java SE platform. This could easily be ported to Java ME using the BouncyCastle library or JSR-177.

It is not a complete solution as issuer verification is missing. This would require the storage of a signature over the public key and modulus (or possibly something like a X.509 certificate). However the example is complete enough to show how one would construct such a Java Card applet.

We created the applet using EclipseJCDE[1] in combination with Java Card 2.2.1 Development Kit because the Secure Element in the Nokia 6131 NFC works only with Java Card 2.2.1. We had to patch[2] EclipseJCDE to work with the Java Card 2.2.1 Development Kit because by default it only works with version 2.2.2.

Some notes for successful programming with EclipseJCDE:

- Set "Compiler compliance level" to version 1.4 or lower
- Set the Applet AID by right clicking on the source file (Java Card Tools, Set Applet AID)
- Set the Package AID by right clicking on the package (Java Card Tools, Set Package AID)
- Convert the project to `cap` file by right clicking on the package (Java Card Tools, Convert)

Once this is complete the `cap` file can be uploaded to the SE using `GPshell` as described in section 3.4.3.

[1] See `http://eclipse-jcde.sourceforge.net/`.

[2] Patch is available at `http://www.tuxed.net/eclipsejcde`.

F.1 Java Card

Here is the code for the Java Card. We left out some exception handling that may be required if the card does not supported the used cryptography. But this works, as is on the Nokia 6131 NFC.

```
package ds.jc;

import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
import javacard.security.KeyBuilder;
import javacard.security.KeyPair;
import javacard.security.RSAPublicKey;
import javacard.security.Signature;

/**
 * Stores a public/private key pair that is used to sign messages with the
 * private key.
 *
 * @author F. Kooman <F.Kooman@student.science.ru.nl>
 *
 */
public class JCSign extends Applet {
    final static byte INIT = (byte) 0x10;
    final static byte GET_PUB_MOD = (byte) 0x20;
    final static byte GET_PUB_EXP = (byte) 0x30;
    final static byte SIGN_DATA = (byte) 0x40;

    /**
     * Our custom exception codes
     */
    final static short ALREADY_INITIALIZED = 5;
    final static short NOT_INITIALIZED = 6;

    /**
     * The RSA key length to use.
     */
    final static short KEY_LENGTH = KeyBuilder.LENGTH_RSA_1536;

    /**
     * The algorithm used for the signature.
     */
    final static byte SIGN_ALG = Signature.ALG_RSA_SHA_PKCS1;

    private KeyPair keyPair;
    private boolean initialized = false;

    private JCSign() {
```

```
}

public static void install(byte bArray[], short bOffset, byte bLength)
        throws ISOException {
    new JCSign().register();
}

/**
 * The entry point for running the applet.
 */
public void process(APDU apdu) throws ISOException {
    if (selectingApplet())
        return;

    byte[] buffer = apdu.getBuffer();

    switch (buffer[ISO7816.OFFSET_INS]) {
    case INIT:
        /* generate a RSA key pair */
        generateKeyPair();
        break;
    case GET_PUB_MOD:
        /* return the public modulus */
        returnPubMod(apdu);
        break;
    case GET_PUB_EXP:
        /* return the public exponent */
        returnPubExp(apdu);
        break;
    case SIGN_DATA:
        /* sign the data */
        signData(apdu);
        break;
    default:
        ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
    }
}

/**
 * Generate a RSA key pair.
 */
private void generateKeyPair() {
    if (initialized)
        ISOException.throwIt(ALREADY_INITIALIZED);
    keyPair = new KeyPair(KeyPair.ALG_RSA, KEY_LENGTH);
    keyPair.genKeyPair();
    initialized = true;
}

/**
 * Get the public modulus.
```

```
     *
     * @param apdu
     *           the APDU
     */
    private void returnPubMod(APDU apdu) {
        if (!initialized)
            ISOException.throwIt(NOT_INITIALIZED);
        byte[] buffer = apdu.getBuffer();
        /* put the modulus in the buffer */
        short pubModSize = ((RSAPublicKey) keyPair.getPublic()).getModulus(
                buffer, (short) 0);
        /* return the modulus */
        apdu.setOutgoingAndSend((short) 0, pubModSize);
    }

    /**
     * Get the public exponent. This will most likely be 2^16+1.
     *
     * @param apdu
     *           the APDU
     */
    private void returnPubExp(APDU apdu) {
        if (!initialized)
            ISOException.throwIt(NOT_INITIALIZED);
        byte[] buffer = apdu.getBuffer();
        /* put the public exponent in the buffer */
        short pubExpSize = ((RSAPublicKey) keyPair.getPublic()).getExponent(
                buffer, (short) 0);
        /* return the exponent */
        apdu.setOutgoingAndSend((short) 0, pubExpSize);
    }

    /**
     * Sign data.
     *
     * @param apdu
     *           the APDU
     */
    private void signData(APDU apdu) {
        if (!initialized)
            ISOException.throwIt(NOT_INITIALIZED);
        byte[] buffer = apdu.getBuffer();
        Signature s = Signature.getInstance(SIGN_ALG, false);
        s.init(keyPair.getPrivate(), Signature.MODE_SIGN);
        short sigSize = s.sign(buffer, ISO7816.OFFSET_CDATA,
                buffer[ISO7816.OFFSET_LC], buffer, (short) 0);
        apdu.setOutgoingAndSend((short) 0, sigSize);
    }
}
```

F.2 Java SE

This application uses the `javax.smartcardio.*` API introduced in Java 6. It starts by querying existing smart card readers and selecting the first available one. It then sends a command to select the Applet. After this it initializes the Applet. If this fails (because it was already initialized) it continues by requesting the public key components (modulus and exponent) and using that to reconstruct the public key of the card. Using this it becomes possible to validate the signature generated by the card.

```
import java.math.BigInteger;
import java.security.InvalidKeyException;
import java.security.KeyFactory;
import java.security.NoSuchAlgorithmException;
import java.security.PublicKey;
import java.security.Signature;
import java.security.SignatureException;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.RSAPublicKeySpec;
import java.util.List;

import javax.smartcardio.Card;
import javax.smartcardio.CardChannel;
import javax.smartcardio.CardException;
import javax.smartcardio.CardTerminal;
import javax.smartcardio.CommandAPDU;
import javax.smartcardio.ResponseAPDU;
import javax.smartcardio.TerminalFactory;

/**
 * Interacts with the Java Card sign applet to sign some data.
 *
 * @author F. Kooman <F.Kooman@student.science.ru.nl>
 *
 */
public class SignatureValidation {
    final static byte INIT = (byte) 0x10;
    final static byte GET_PUB_MOD = (byte) 0x20;
    final static byte GET_PUB_EXP = (byte) 0x30;
    final static byte SIGN_DATA = (byte) 0x40;

    /**
     * Our custom exception codes
     */
    final static short ALREADY_INITIALIZED = 5;
    final static short NOT_INITIALIZED = 6;

    private static CardChannel channel;
    private boolean result;

    public SignatureValidation() throws CardException,
```

```
        NoSuchAlgorithmException, InvalidKeySpecException,
        SignatureException, InvalidKeyException {

    /* get the (first) card terminal */
    TerminalFactory factory = TerminalFactory.getDefault();
    List<CardTerminal> terminals = factory.terminals().list();
    CardTerminal terminal = terminals.get(0);
    Card card = terminal.connect("T=0");
    channel = card.getBasicChannel();

    /* the data of which we want to verify the signature */
    byte[] data = new byte[] { (byte) 0x00, (byte) 0x01, (byte) 0x02,
            (byte) 0x04, (byte) 0x05 };

    /* the Java Card Applet (AID) we want to use */
    byte[] appletAID = new byte[] { (byte) 0x6a, (byte) 0x63, (byte) 0x73,
            (byte) 0x69, (byte) 0x67, (byte) 0x6e };
    sendAPDU((byte) 0x00, (byte) 0xa4, (byte) 0x04, (byte) 0x00, appletAID);

    /* initialize the card (this will internally generate an RSA key pair */
    sendAPDU((byte) 0x00, INIT, (byte) 0x00, (byte) 0x00, null);

    /* get the public modulus stored on the card */
    byte[] pubModData = sendAPDU((byte) 0x00, GET_PUB_MOD, (byte) 0x00,
            (byte) 0x00, null);
    BigInteger pubMod = new BigInteger(1, pubModData);

    /* get the public exponent stored on the card */
    byte[] pubExpData = sendAPDU((byte) 0x00, GET_PUB_EXP, (byte) 0x00,
            (byte) 0x00, null);
    BigInteger pubExp = new BigInteger(1, pubExpData);

    /* get the signature over the data */
    byte[] cardSignature = sendAPDU((byte) 0x00, SIGN_DATA, (byte) 0x00,
            (byte) 0x00, data);

    /* validate the signature generated by the card */
    RSAPublicKeySpec pks = new RSAPublicKeySpec(pubMod, pubExp);
    KeyFactory kf = KeyFactory.getInstance("RSA");
    PublicKey pubKey = kf.generatePublic(pks);
    Signature sig = Signature.getInstance("SHA1withRSA");
    sig.initVerify(pubKey);
    sig.update(data);
    result = sig.verify(cardSignature);
    card.disconnect(false);
}

/**
 * Get the result from the signature check.
 *
 * @return true if signature is valid, false is signature is invalid
```

```
     */
    public boolean getResult() {
        return result;
    }

    /**
     * Send the APDU to the card and verify the results.
     *
     * @param cla
     *            the CLA byte
     * @param ins
     *            the INS byte
     * @param p1
     *            the P1 byte
     * @param p2
     *            the P2 byte
     * @param data
     *            the data to send
     * @return response data
     */
    private static byte[] sendAPDU(int cla, int ins, int p1, int p2, byte[] data)
            throws CardException {
        ResponseAPDU r = channel.transmit(new CommandAPDU(cla, ins, p1, p2,
                data));
        /* if card is already initialized we don't complain */
        if (r.getSW() != 0x9000 && r.getSW() != ALREADY_INITIALIZED)
            throw new CardException("instruction failed (error code: "
                    + r.getSW() + ")");
        return r.getData();
    }

    public static void main(String[] args) {
        System.out.println("Validating Signature...");
        try {
            SignatureValidation s = new SignatureValidation();
            if (s.getResult())
                System.out.println("   OK");
            else
                System.out.println("   FAILED");
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

Appendix G

Gjokii Design

This appendix was removed from the public version of this document due to security issues we found on the Nokia S40 platform.