MiPass: MicroPayment for Android Smartphones

by

Hendri Appelmelk

THESIS

for the degree of

MASTER OF SCIENCE

vrije Universiteit amsterdam

Faculteit der Exacte Wetenschappen
Vrije Universiteit

August 9, 2011

MiPass: MicroPayment for Android Smartphones

by

Hendri Appelmelk

THESIS

for the degree of

Master of Science

Faculteit der Exacte Wetenschappen
Vrije Universiteit
Amsterdam, The Netherlands

August 9, 2011

Supervisors

Dr. Mauro Conti

Prof. Dr. Bruno Crispo

Acknowledgement

This Master's thesis uses NFC readers and RFID tags which are offered by Arygon Technologies[1]. I want to thank them for their support.

[1] www.arygon.de

Abstract

This Master thesis provides a novel micropayment scheme called MiPass, which prevents the user from overpaying when parking a car on the side of the road.Once the car is parked, the user checks-in and pays a deposit, and checks out on departure. In doing so, any excess payment done in advance by deposit is booked back to the user. MiPass is designed for NFC (Near Field Communication). NFC offers an improvement for all short range communication between mobile phones, which is interesting for developing faster and more efficient mobile payment solutions. With MiPass, the user does not store money on his mobile phone. Rather, the mobile phone stores a proof of a certain amount of money. Therefore, MiPass also offers the advantage that a malicious user cannot invent his own coins. Beyond addressing the problem of overpayment and security, MiPass is also suitable for extension into the field of regular payments. This thesis also contains a comprehensive literature review of existing payment schemes, and offers a comparison between them. Furthermore, it provides a step-by-step explanation of how libnfc is ported to Android. Finally, as a proof of concept, this thesis describes the implementation of the discussed payment scheme as an Android application.

Contents

List of Figures

List of Tables

1 Introduction

The fast development of today's mobile phones continues to open the door to new applications for these phones. One crucial element of this development is NFC (Near Field Communication). Since the introduction of NFC, a new type of mobile phone application is possible, because it enables faster communication between mobile phones and other devices it is also designed with intuitivity, user friendliness and security in mind. Even though there is no NFC-equipped mobile phone on the market at present, it is expected that NFC will be a standard within the near future. Because it is significant, NFC is standardized by the NFC-forum, a consortium of companies including, for example, Sony or Philips [7]. Members of the NFC-forum aim to develop a new wireless protocol that is characterized by security, intuitivity, and simple communication. NFC operates on the unlicensed ISM band (Industrial, Science, Medical), on a frequency of 13.56 MHz, and offers a transmission speed of 424 $Kbit/s$, although faster connections are expected in the future.

In essence, NFC is an extension of RFID, which renders it compatible with the ISO14443 A/B standard and Sony's Felica tag (which is a de-facto standard). NFC compatibility with the ISO15693 standard is, furthermore, expected in the future. All standards define the communication protocol between reader and tag. Communication between two NFC devices can occur in active and passive mode. In active mode, the NFC device is allowed to communicate with passive devices such as an RFID tag. In passive mode, the NFC device acts as a passive RFID tag.

With the pending introduction of NFC-equipped mobile phones, new and innovative applications such as mobile payment solutions, are necessary. Based on this assumption, this thesis introduces a new payment solution for mobile phones, mainly to illustrate the usability of NFC in mobile applications. Because there is, to date, no suitable NFC equipped mobile phone available on the market, this project uses the Nexus One, which has been extended with nfclib[2].Specifically, this thesis develops a micropayment scheme for the payment of on-the-road parking tickets. The term micropayment typically refers to small value transactions, which includes such parking tickets. The application aims at enabling precise payment of such tickets, by reducing the risk of over-payment, if a user cannot estimate the precise parking time for her or his vehicle.

The application is based on the structure of a typical micropayment, which has three entities: the broker that acts as a trusted third party, the vendor as the entinty who sells items, and the user as the entity who buys the items. Most existing micropayment system work with an offline broker, meaning that transactions are processed between the vendor and the costumer. This has the disadvantage that the broker can only verify the legitimacy of all payments at the end of the day. This means that a malicious person can only be traced long after the actual misbehaviour has happened. However, such micropayment

[2]Available at: www.libnfc.org

schemes are in use, mainly because the lack of a broker during the transaction,decreases the total cost of the transaction. Yet, this also means that the degree of safety of the respective payment system is lower. Therefore, the micropayment developed in this thesis offers a flexible and lightweight solution for payment. It is protected against coin invention, by storing a hash value on the device which represents the current balance, instead of a balance itself, but it also has the possibility to trace fraudulent users. Since it does not store a real balance on the device, a tamper proof device is not required.

1.1 Contribution

This thesis proposes a new micropayment mechanism based on NFC. In addition, it provides an extensive overview of existing RFID and NFC standards, and includes an examination of available NFC devices and an extensive explanation of the implementation on Android Froyo 2.2 with NFC, which is not equipped with any kind of NFC support by default. The new micropayment scheme is primarily designed to enable fair and comfortable payment for parking spots on the side of the road. The payment mechanism works as follows: When someone wants to park a car, he or she pays a deposit to park for a certain time. If the user returns before this time has expired, the remaining part of the deposit is booked back to the user. That way, overpayment–which is a common problem of roadside parking–is prevented.

To our knowledge, this is the first micropayment scheme that is entirely focused on addressing and solving this problem. It is also the first to work with a deposit, and to be implemented on Android, using NFC. This micropayment scheme is also resilient to untrusted devices.

The implementation of our payment solution on an Android device required specific modifications to the Android operating systems. Most importantly, Android 2.2 (Froyo) had to be modified to work with libnfc, a library that was not originally developed to run within the Android operating system.

1.2 Organization

The sequel of this Thesis is as following: in Section 2, various RFID standards are described. Those standards are an important part of NFC. In Section 3, the underlying principles of NFC are explained. Section 4 provides an overview of the existing devices which are equipped with NFC. This section also explains which device was chosen and why. In Section 5, the internal structure is of the Android operating system is discussed. In Section 6, our new solution is proposed, by first examining the existing literature and build a new payment protocol using ideas which explained in this literature. In Section 7, our idea is implemented on an Android mobile device. This section also covers the porting

of libnfc to Android. Section 8 provides an evaluation from the new proposal. Finally, Section 9, conclusions are drawn and ideas for further research are proposed.

2 RFID

NFC (Near Field Communication) can be seen as an extension of RFID (Radio Frequency Identification). This means that an NFC-equipped device should be backwards compatible with RFID. It should thus, for example, be possible to use a NFC device and emulate a RFID tag or RFID reader. To ensure that NFC really does fullfil this need for backward compatibility, it needs to be compatible with the ISO 14443A/B standard and Sony's FeliCa. This chapter elucidates this ISO standard and the FeliCa card.

In Section 2.1, a short history of the development of RFID is provided. Section 2.2 describes how a basic RFID system works. Specifically, it contains an explanation of the role each component plays within the system. In Section 2.3, the RFID standards NFC needs to comply with are described, as is the ISO 15693 standard. This standard is not officially supported at the time of writing this thesis. However, new developments by NFC-chip manufacturer NXP do already support this standard, as does [8]. ISO 14443A/B is outlined in Section 2.3.1, and the FeliCa card is discussed in Section 2.3.2. In Section 2.3.3, the ISO 15693 is described.

2.1 A brief history of RFID

Radio Frequency IDentification (RFID) is a combination of radio broadcast and radar technology, both of which were developed during the early 20th century. However, microprocessors and communication network RFID only came into frequent use after the development of semiconductors. While academic interest in RFID was sparked slightly earlier [37], the commercial potential of RFID became apparent only during the 1960s. Companies such as Sensormatic and Checkpoint where founded, and invented the so-called electronic article surveillance (EAS), which was used to fight product theft.

From the 1970s on, the usage of RFID became more common and scientists, inventors and companies began to extensively research it. American researchers predominantely conducted studies dealing with the part RFID plays in access control for buildings and transportation, whereas European studies focused more on industrial and business applications, and animal tracking.

From 1990 on, RFID became mainstream, probably due to the wide scale of deployment in, for instance, highway toll ports. With these toll ports, it was possible to pay toll without stopping a car. Other uses of RFID included ski passes in the European Alps or payment in train travel in countries such as Australia, China, Japan, Brazil or Argentina.

Today, RFID is regularly discussed by the media, and is therefore becoming a part of everyday life (for a detailed history of RFID, see [37]).

2.2 RFID system overview

NFC is an extension of RFID and is therefore compatible with RFID standards. Before an in depth overview is given about the various standards NFC needs to comply with, attention is paid to a RFID system overview in order to get a general sense of a RFID system. A typical RFID system consists of three elements: tag, reader, and verifier. Figure 1 shows the relation between these components.

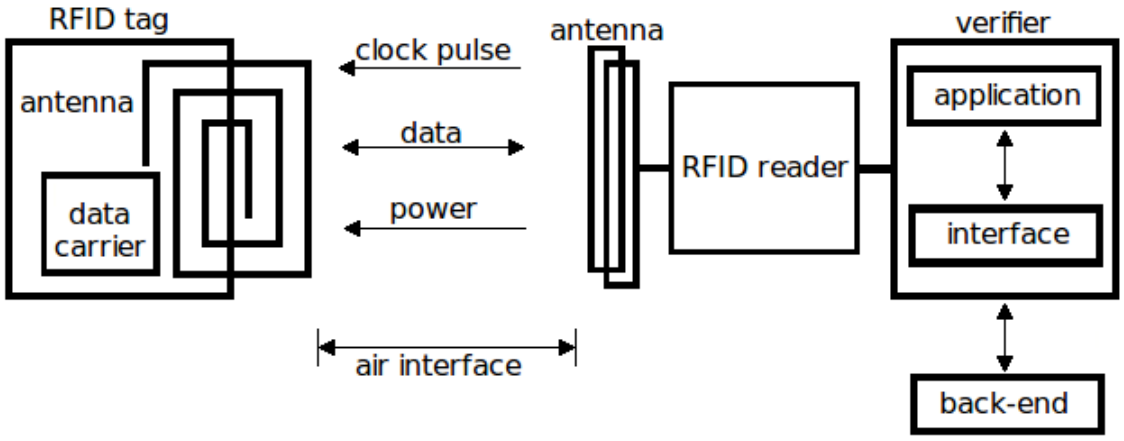


Figure 1: Overview of a simple RFID system

Tag. There are active and passive tags available. An active tag is powered by an on-board power source. A passive tag is powered by the RFID reader, as shown in Figure 1. The reader emits radio waves, which are picked-up by the antenna of the tag. The radio waves induce an electrical current in the antenna wire located on the tag. The power is necessary to feed the data carrier. The reader provides the tag also with a clock pulse to process data. Both the power supply and the clock pulse are needed to communicate back to the reader.

An active tag uses its antenna only for communication with the reader. The tag is typically powered by a small on-board battery. Therefore, it can communicate to the reader on its own, without the radio waves emitted from the reader. Active tags are larger in physical size compared to passive tags, simply because the active tag has a power source.

Reader. The RFID reader is also equipped with an antenna. The primary task of the reader is to process the radio signals from the tag and to pass these processed electronic signals to the verifier (and vice versa); this is shown in Figure 1. The reader can be seen

as an interface between the verifier and the tag. Apart from the interface function, the reader also has the task of powering the tag if it is passive.

Verifier. A verifier interacts with the data that the tag has received from the reader. This information can be passed to an application. This application can make decisions based on the data received from the reader. The verifier can also contain a back-end system, an example for which is a database.

For clarification, assume a situation where a user presents a tag with a unique number to a reader in order to gain access to a barrier (e.g. a door of a specific building). The reader extracts this unique number from the tag, and passes this number on to the verifier. The verifier, in turn, queries the back-end to obtain the access rights belonging to this user. The application will open the barrier, if the user has proper access rights.

2.3 RFID standards

As mentioned before, NFC is an extension of RFID. This means that NFC should be able to behave just like a tag and a reader. To make NFC work as part of an RFID system, and to guarantee backward compatibility for RFID systems, NFC must comply with existing RFID standards. The authors of [25] also argue that NFC implementations must be compatible with the ISO 14443A/B standards, and Sony's FeliCa tag, in order to comply with the NFC protocol definition. In this section, an overview of those RFID standards is provided.

2.3.1 ISO 14443A/B

This standard is widely used in RFID tags available today. Examples are MIFARE, PayPass, and the Biometric passport. The standard is subdivided into two sub-standards, the ISO 14443A and the ISO 14443B. The ISO 14443A is the most widely used tag. However, the ISO 14443B standard has an advantage over the ISO14443A variant [11]. The type B variant has no patents on the communication encoding.

The standard itself contains four major parts. Of those four parts, the last three parts are subdivided into two sections, "type A" and "type B" (this refers to the A or B extension behind the standard name). The distinction between the standards is mainly focused on specific signal properties, such as modulation, bit representation, carrier modulation, error correction, and others.

The first part [26] describes the physical properties of a tag. These properties are, for example, the physical size and the operating temperature. The second part [27] is devoted to the signal interface and shows how data is transmitted between reader and tag. The third part [28] defines an anti-collision and session initiation. Anti-collision is needed, when more than one tag is presented to the reader. The fourth part [29] shows a transmission protocol used.

Part One. The physical properties of the tag are explained in this section of the standard [26]. A tag is typically embedded in paper or plastic. This ISO standard assumes an ID-1 type card. The ID-1 type card, which is a card without a tag attached, is defined in ISO 7810. This ISO 7810 standard requires the physical dimensions of an ISO 14443A/B tag to be be $85,60mm * 53,98mm$. This size is comparable to the size of most credit cards.

Examples of physical properties described in this standard refer to the tag. One example is the property that the communication between reader and tag is contactless. Moreover, it must be possible to bend the RFID card to a certain extent, and this bending should also not compromise the function of the RFID tag. The tag must also be able to resist a low amount of X-ray radiation without this affecting the functioning of the tag itself. Lastly, it should be capable of operating between $0°C$ and $50°C$. A complete version of this standard can be found in [26].

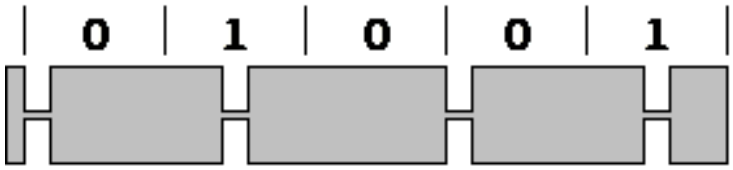
Part Two. The second part of the ISO14443A/B discusses two types of signal interfaces between a tag and reader: a type A and a type B interface [27]. This part covers both the power supply and the signalling of the tag. An RFID signal that operates according to the ISO 14443A/B standard operates on $13.56Mhz$ with a variation of $7Khz$. The maximum throughput is $\sim 106Kb/s$.

The construction of the signal differs between type A and type B tags. Both types use ASK (Amplitude Shift Key) modulation to transmit data signals from the reader to the tag. The data signals sent from the tag to the reader use "load modulation". Load modulation is used to sent data signals from the tag to the reader, by using the reader as a power source. An extensive description of load modulation and ASK can be found in [9].

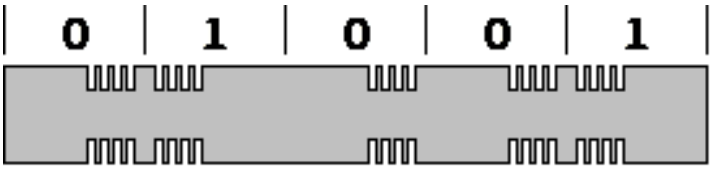
To make the difference between tag type A and type B clearer, both types are subsequently explained in separate paragraphs. First, the signal characteristics of tag type A are explained, followed by a paragraph on the type B tag.

- **Type A.** A signal sent from the tag to the reader utilizes a 100% amplitude modulation on the RF (Radio Frequent) signal. This signal is encoded with Modified Miller. The data is modulated on top of ASK (Amplitude Shift Key), and is shown in Figure 2a. The idea is that, when a logical "0" is followed by another logical "0", the power level stays high. If a "0" is followed by a "1", the power level is low in the first part of the bit period. If a "1" is followed by a "0", the power level is low in the middle of the bit period.
  When the signal is sent from the tag to the reader, it modulates a sub-carrier using a $847.5Khz$ OOK (On-Off Keying) modulation, and the data is encoded with Manchester Encoding. As shown in Figure 2b, in this encoding scheme, a logical "1" has four subsequent low power level periods at the beginning of the bit period. A

logical “0” has four subsequent low power level periods at the end of the bit period. In type A signalling, the RF field is turned off for a short time every time data is sent (i.e. the power level is low for short intervals). The internal circuit needs to save enough power to bridge this time.



(a) 100% ASK, used in ISO 14443 A tags.

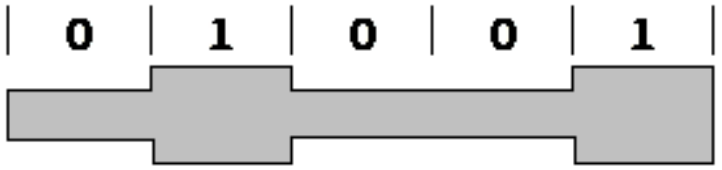


(b) BPSK subcarrier load modulation with NRZ-L encoding.

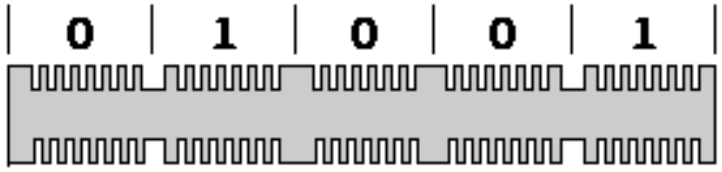
Figure 2: Carrier and sub-carrier modulation for type A tags

- **Type B.** The signal utilizes a 10% amplitude modulation on the RF field in order to communicate between the reader and the tag. Figure 3a shows that this communication uses NRZ-L (Non Return to Zero Level) encoded data. This type of data encoding is straight forward: A logical “0” has a lower value than a logical “1”.
  The communication from the tag to the reader is done using BPSK (Binairy Phase Shift Keying) modulation on a sub-carrier with NRZ-L encoded data, as depicted in Figure 3b. At the start of each bit period it is determined, if a logical “0” or logical “1” is encoded. If the bit period starts with a high value, a “0” is encoded. If the bit period starts with a lower value, a “1” is encoded. The signal is continuous, meaning that there are no small periods of time without a signal (as is the case in type A).

Part Three. The third part defines the anti-collision used in ISO-14443A/B and session initialisation [28]. The anti-collision scheme is in fact a simple state machine. The anti-collision and initialization are constructed to permit multi-protocol readers to operate with type A and B tags. The state machines for type A and type B tags differ slightly. The tag A type tags connection setup can be distinguished into five subsequent steps, whereas the type B tag divides into seven steps. First, the states for type A tags are explained, followed by the type B states.



(a) 10% ASK, used in ISO 14443 B tags



(b) BPSK subcarrier load modulation with NRZ-L encoding

Figure 3: Carrier and sub-carrier modulation for type B tags

- **POWER-OFF.** In this state, no power is fed to the tag. Due to a lack of power, no sub-carrier is emitted by the tag.

- **IDLE.** When the tag is in the POWER-OFF state, but enters the readers RF-field, it becomes IDLE. This state is also entered, if the tag is inactive for a maximum delay. IDLE means that there is still power is, and the tag is capable of demodulating and recognizing the REQA (Request A) or WUA (Wake Up type A) command. The REQA command is sent to determine whether the tag is a type A tag. The WAKE-UP command sets the tag back into ready mode.

- **READY.** The READY state is entered, if the REQA or WAKE-UP command have been received from the reader. The tag returns ATQA (Answer To Request of type A). When the reader receives one ATQA command, it knows that one tag is present. If a collision occures, the collission avoidance sequence is started.
  In this case a search tree algorithm is used to select a tag. The reader starts with sending a SELECT command, NVB (Number of Valid Bits) and a bit mask. The length of the bit mask depends on the NVB. Every tag compares the bit mask to its own unique id. If the mask matches the tags id, the tag sends back the rest of its id. The algorithm is repeated until one tag is selected. If one tag is selected, the reader sends a SELECT command with the id of that tag. The tag in turn responds with a SAK (Select Acknowledge) command and enters the ACTIVE state.

- **ACTIVE.** In this state, the tag and the reader negotiate about the protocol parameters used for data transmission. This tranmission protocol is defined in ISO 14443 A/B part four. However, other (proprietary) protocols are also possible. The reader can send a HLTA (Halt type A) command to the tag. This moves the tag

into the HALT state.

- **HALT.** The tag can be put into READY mode when a WAKE-UP command has been sent to the tag while the tag is in HALT state, this command moves the tag back to READY state.

The type B tag differs from the type A tag to some extent. In particular with the type A card, the type B tag has 7 states. The POWER-OFF and HALT state here are identical to the type A tag states. The IDLE state is nearly the same, with the difference being that the tag now waits for a REQB command instead of a REQA command. The READY state is divided into the READY-REQUEST and READY-DECLARED sub-states. In the READY-REQUEST state, the tag listens to REQB (Request type B) commands and slot-MARKER messages, this message indicates the end of a time slot. The tag enters the READY-DECLARED state as soon the ATQB (Answer To Request type B) command is sent.

Part Four. The fourth part defines negotiation regarding the data transmission protocol used [29]. This phase assumes that the tag is in the READY state, as defined in the state machine in part three. Once in the READY state, the reader sends a RATS (Request Answer To Select). The tag sends an ATS (Answer To Select) back to the reader. If the tag has PPS (Protocol Parameter Selection) support (i.e. the tag is ISO 14443-4 compliant), the reader can query the tag for its settings by sending a PPS request. The communication protocol, which is placed in the application layer is defined in ISO 7816-3.

2.3.2 FeliCa

The FeliCa tag was initially meant to be named "ISO 14443C". However, Sony failed to receive an ISO standard for their product. Therefore, the IC-card is known as FeliCa [11]. It is important to note that the FeliCa is a de facto standard, because it is widely accepted in the industry rather than an international standard, such as the ISO 14443A/B. However, the Felica card does comply with the ISO/IEC 15408 EAL4 security standard [16]. This ISO standard deals with security of computer systems.

The FeliCa RFID system has much in common with the ISO 14443A/B standard, particularily when it comes to properties related to signal transmission. FeliCa also communicates on $13.56MHz$, although, the FeliCa is a little slower than the ISO standard. It does not require a sub-carrier as is the case by both ISO 14443 standards. It relies on symmetric communication instead.

The signal is encoded with Manchester encoding and uses an modulation depth of 10%. The Manchester encoding is chosen to avoid the impact of noise. This noise can occur, if the tag is not at a stable distance from the reader.

Atomicity can be guaranteed by FeliCa. If, during a transaction, the FeliCa tag moves

outside the coverage area of the reader, it is able to recover itself to a previous correct state. This makes the transaction atomic on the side of the tag. This is not the case for the ISO14443A/B standard. The communication during the transaction is encrypted, and keys are generated when the tag authenticates itself successfully to the reader.

One of the key features of the FeliCa tag is the employment of authentication between the reader and the tag. FeliCa is also capable of having different data sets on one card, as depicted in Figure 4. This means that the FeliCa card is able to suit multiple purposes. The internal structure of the FeliCa tag consist of "services" and "areas", which are organized in a tree structure. If this internal structure is comparable to an ordinary file system in an operating system, areas can be compared to directories and services to files.

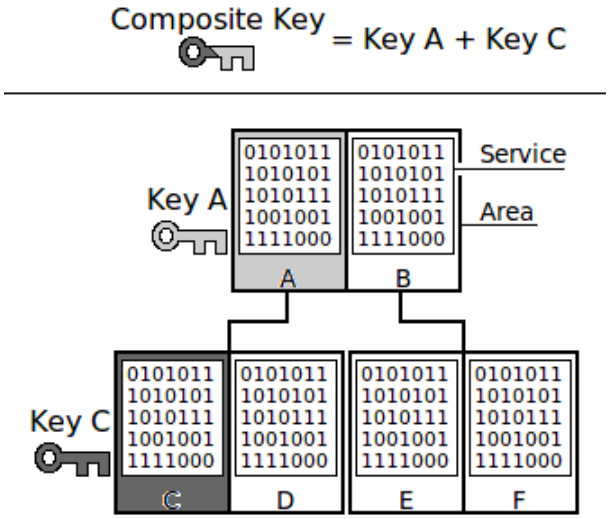


Figure 4: Felica security model

Normally, if different sets of data need to be entered, different authentications should be processed. The problem with this approach on RFID tags is that processing time might increase if more than one authentication has to be done. To tackle this problem, the FeliCa tag uses a composed key. With this key, different sets of data can be accessed without any individual authentication for each data set. In Figure 2.3.2, the composite key (which is a combimation of key A and key C) gives access to both area A and area C. Both keys are combined to increase authentication efficiency.

2.3.3 ISO 15693

The ISO 15693 standard has similarities to the ISO 14443A/B standard. It has, for example, the same operating frequency, which is $13.56Mhz$. However, the ISO-15693 is

applicable for vicinity tags. This type can operate from 1.5 meters, which is 15 times the distance reached by $10cm$ proximity tags. The communication speed is therefore also slower and is either $26.48Kbits/s$ or $1.65Kbit/s$, depending on the modulation. This lower data throughput is caused by the cpu, which receives less power from the reader compared to a proximity tag. This type of tag is perfect for applications with requirements of a certain distance between reader and tag.

Just as the ISO 14443A/B standard, this standard is subdivided into three separate sections. The first part is devoted to physical requirements [33]. The second part discusses the RF power and signal interface [31]. The last part describes the anti-collision and transmission protocol [32].

Part One. The vicinity tag is commonly attached to a plastic or paper card. This part mostly describes the contactless card with the integrated circuit attached. As with ISO 14443A/B, the tag and reader communicate using inductive coupling. The standard is concerned with physical properties. These physical properties are, for example, the minimum and maximum temperature, the amount of X-ray radiation, and the electric magnetic field that a tag must handle. This part is not only protecting the tag from external influences. It also describes the quality of the surface of the card.

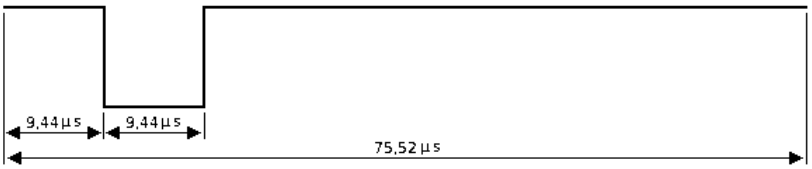
Part Two. Details of the transmitted signal are the data carrier frequency, the modulation type, power levels, data rates, and encoding algorithms. These values are necessary in order to allow a standardized communication between the reader and the tag. The result of this standard is a protocol that defines method to transmit data between the reader and tag.

This standard uses ASK (Amplitude Shiftkey) as a modulation technique. With a modulation depth of 10 % or 100 %. The encoding is either one out of four- or one out of 256 pulse code modulation. One out of four encoding is depicted in Figure 5.

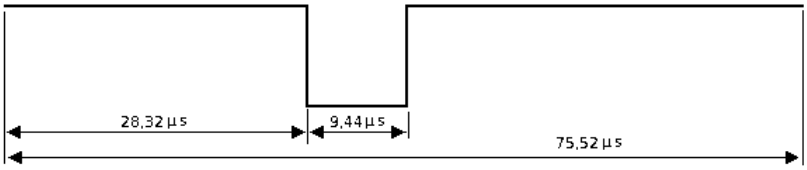
In one out of four encoding, a whole period is 75,52 μS (seconds) long and is divided into four quarters that are 18,88 μS long. These quarters represent the value of the whole period. When the signal is low in the first quarter, as can be seen in Figure 5a, a “00” is encoded. In the second quarter, as can be seen in Figure 5b, a “01” is encoded, “10” and “11” are encoded on a low signal in respectively the third and fourth quarter. In fact, one 75,52 μS period encodes two bits. This generates a throughput of:

$$1S/75,52\mu S * 2 = 13241 * 2 = 26,48\ Kbit/S$$

The ISO 15693 standard can also operate with one out of 256 encodings. It works essentially the same as a one out of four encoding, the only difference being that one full period distinguishes 256 elements instead of four. Therefore, the period takes 4,833 mS, but it can encode 256 distinguishable values in one period or one byte. In Figure 6, the value 200 is encoded.



(a) Encoding of “00”



(b) Encoding of “01”

Figure 5: Two bits encoded in one out of four pulse modulation.

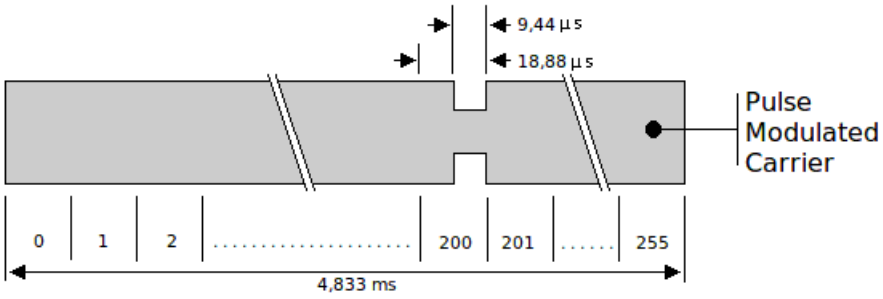


Figure 6: One out of 256 modulation, encoding of “200”

The maximum throughput in one out of 256 encodings is:

$$1S/0,004833S * Log_2(256) = 1,65Kbit/S$$

Part Three. This standard also uses an anti-collision scheme to prevent tags from sending simultaneously. To achieve this, each tag has a UID (Unique Identifier). Each tag also has a Application Family Identifier. This filter enables the reader to determine, if a tag belongs to a certain application. For example, a book equipped with a RFID tag from the city library may not interfere with a theft protection system from the library of the university. This tag from the city library may not be “seen” by the reader from the university library. Using these values, the reader can efficiently and quickly select the appropriate tag to communicate with.

The algoritm works as follows, and is extensively discussed in [44]: Assume a scenario with four vicinity tags. These tags have the following hexadecimal UID: 0x12A , 0x32A,

0x45A, and 0x345. In Table 1, sixteen slots are presented and labelled with a hexadecimal number. These numbers represent slot-numbers. These slots are used as a time frame, in which a tag is allowed to send. The second row shows the tags which corresponds with that specific slot number. This slot number is determined by the tag itself, by comparing their UID to the slot number plus mask. This mask is set to zero in the first round.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
					0x345					0x12A 0x32A 0x45A					

Table 1: Tags divided amongst the provided slots in the first round

In Table 1, the UID with the value 0x345, corresponds to slot 5, the others to slot A. Slot 5 does not have a collision, since it is the only UID placed in this slot. However, the other three values do suffer from a collision. The reader will notice this and mark slot where collision occurred.

A collision occurred in slot number A. In the second round, the tags from slot A are placed in another slot by comparing their second least significant hexadecimal value. This is achieved by obtaining a new mask value, which is: $new\_mask\_value = old\_mask\_value + slotnumber$. The new mask length is incremented by four bits. This results in taking the second least significant hexadecimal value of the tags UID, which is shown in Table 2.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
		0x12A 0x32A			0x45A										

Table 2: Tags divided amongst the provided slots in the second round

This continues until all slots are divided amongst the provided slots. Other important things of this part of the standard are the communication flow of requests and responses. It provides a complete command set that allows an application to read, write, and lock single or multiple blocks of data. It defines the complete communication interchange for requests and responses. To prevent the data-transmission from errors, CRC (Cyclic Redundancy Check) is used.

Section three also defines the complete communication interchange for requests and responses. A complete command set defines reads, writes and the locking of data for

single and multiple blocks. A CRC (cyclic redundancy check) block is included to assure the integrity of the data received.

2.3.4 Standards compared

All different standards have different properties. The Felica tag has better properties when it comes to security. It offers an encrypted channel between reader and tag. However, the ISO 14443A/B also offers security, but proprietary mechanisms are required, examples of these proprietary mechanisms are: DESfire, MiFare and SmartMX developed by NXP[41].

Technically seen are there no big differences. All types work on the same frequency. However, the data throughput differs greatly, the ISO 14443A/B can transmist 848 $Kbit/s$ where the Felica tag do half of that rate, the ISO 15693 is 30 to 500 times slower than the ISO14443 A/B standard. In Table 3, the main differences are pointed out.

	ISO14443A	ISO14443B	Felica	ISO15693
Operation Frequency	13.56Mhz	13.56Mhz	3.56Mhz	13.56Mhz
Maximum throughput	848$Kbit/s$	848$Kbit/s$	424$Kbit/s$	26.48 or 1.65$Kbit/S$
Needs sub-carrier	yes	yes	no	yes
Sub-Carrier Modulation	ZOO	BPSK	None	ASK
Modulation Depth	100%	10%	10%	10 or 100%
Encoding	Manchester	NRZ-L	Manchester	1 out of 4, 1 out of 256
Atomicity	No	No	Yes	No
Mutual Authentication	No	No	No	No
Encrypt communication	No	No	Yes	No
Maximum distance	$\sim 10cm$	$\sim 10cm$	$\sim 10cm$	$\sim 1,5m$

Table 3: RFID standards, which are compatible with NFC, compared

3 Near Field Communication

The adoption of NFC (Near Field Communication) strongly depends on the interoperability between NFC devices. Therefore, standardisation is required. In 2004, the NFC–forum has been founded. The NFC–forum is a consortium of companies, such as Sony, NXP, and Microsoft. The members of this forum are responsible standardizing NFC.

The standard is described in ISO/IEC–18092 NFCIP-1 (Near Field Communication Interface-1) [34]. However, there are several other standards NFC needs to be compatible with: ISO/IEC-14443A/B, FeliCa (not an ISO standard), and ISO/IEC-15693. ISO/IEC-14443A/B and FeliCa are related to RFID (as explained in Section 2.3.1). ISO/IEC-15693 is a standard for vicinity cards (as explained in Section 2.3.3).

To integrate these standards into NFC, NFCIP-2 (Near Field Communication Interface Protocol-2) is developed by the NFC-forum. This standard is also known as ISO/IEC-21481 [30]. This standards specifies a mechanism to support and select either ISO/IEC-14443A/B, FeliCa, or ISO/IEC-15693. All these standards allow NFC to operate in three different modes. These modes are:

- Card Emulation: The NFC device emulates a tag. The reader can only read this tag.
- Reader/Writer: The data is read from or written to the tag.
- Peer-to-Peer: A bidirectional data connection between two NFC-devices can be setup using a peer-to-peer connection.

In Section 3.1, the NFCIP-1 protocol is explained, this part is important to understand how a peer-to-peer connection is setup using NFC. Section 3.2, describes how a particular communication mode is selected, these modes are explained in the previous section. Security measures which prevents a NFC connection from for example eavesdropping, is discussed in Section 3.3.

3.1 Near Field Communication Interface and Protocol-1

The ISO/IEC-18092 / ECMA-340 defines the standard for NFCIP-1. In this standard, requirements are set for modulation, bit encoding schemes and frame architecture [46]. This standard also specifies requirements, regarding the types of data rates that are supported, 106 Kb/s, 212 Kb/s, and 424 Kb/s. The standard also describes the data frame architecture, collision avoidance, error correction, single device detection and transport protocol [42]. This protocols describes the process to let two devices communicate in peer-to-peer mode.

To avoid collisions during session initialisation, the NFCIP-1 standard provides collision avoidance. This requires the device to not generate a RF signal as long as another

RF signal is detected. The initiator is obligated to probe for other RF signals during initialisation of the connection. Thus, if a NFC device does not detect another RF signal during timeframe τ, the RF signal can be switched on.

When a device can safely send, without causing a collision, the RF signal is switched on. The next step is to select the appropriate device needs. This depends on whether a NFC device enters passive or active mode. A passive mode device can not select other devices, it is selected by an active device. If more than one device is provided to the initiator, and thus one device needs to be selected out of several devices, the NFCIP-1 describes a state machine to successfully select one device.

This state machine, starts in a state where the targets are powered off. This is known as the POWER-OFF state in the state machine. When the de target is successfully powered, it enters the SENSE state. At this state, the targets only listens to the SENSE_REQ (Sense Request) or ALL_REQ (All Request). Upon reception of this command, the target sends the command SENS_RES (Sense Resolution) back to the initiator. At this moment, the target enters the RESOLUTION state. In this phase, the Single Device Detection algorithm (SDD) is used to select one target at the same time. In this algorithm, every target generates a random id and sends this id to the initiator. When a Tag receives a SEL_REQ (selection request) command together with its id, the target sends SEL_RES (Selection Resolution) back to the initiator. The target is now in the SELECTED state. In this mode, the target is allowed to communicate with the initiator. The initiator can put his target into SLEEP mode by issuing a SLP_REQ (Sleep Request) command.

3.2 Near Field Communication Interface and Protocol-2

The NFCIP-2 protocol determines which standard can be used for data transmission. Recall that there are three different ISO standards for modes of operation: the ISO/IEC-14443A/B for RFID cards, the ISO/IEC-15693 for vicinity cards and ISO/IEC-18092 for peer-to-peer communication.

The NFCIP–2 standard describes a mechanism to detect one communication mode out of the three possibilities described above. The standards also defines the steps which should be taken after the selection of a communication mode. Initially, the RF field from a NFC device is switched off. The device probes if there is an RF field active. If this is the case, the communication mode is set according the definitions described in NFCIP-1. If the NFC device does not detect a RF field, it can either switch to NFCIP-1, VCD or PCD. Once NFCIP-1 mode has been selected, the definitions in NFCIP-1 should be used. If VCD or PCD mode is selected, there should again be a RF detection. If an RF has been found, the whole procedure starts again. If no RF signal is found, the device will generate an initial RF signal and enter the the appropriate communication protocol.

3.3 Secure Near Field Communication

NFC comes with security concerns. The authors from [18, 23] have distinguished four threats NFC is vulnerable to:

Eavesdropping. NFC communicates over the air using RF signals. This implies that the RF signals can be captured by an attacker, if the attacker uses an antenna. It is important to note that it is easier to eavesdrop an active sender than a passive sender. By an active sender, the signal basically goes one way, from the sender to the target. However, a passive tag needs the presence of the active reader. This reader powers the tag. Thus, an attackers needs to separate the power RF field from the data channel when eavesdropping a passive tag. NFC is vulnerable to eavesdropping.

Data Corruption. An attacker can, instead of listening to the data, simply damage the data to corrupt it. The goal of an attacker can thus be to disturb the communication between two parties. The attacker does not necessarily need to understand the transmitted signal. Data can be corrupted by sending data on the same frequency of the frequency spectrum.

Data Insertion. When a communication channel between two devices is set-up, an attacker is able to insert messages into this communication channel. However, the authors [18] argue that there are some restrictions to this attack. It is only possible to insert data when the answering device needs a long time to reply on a request. In this case, the attacker can send this answer earlier than the answer from the valid receiver. Further, it is important that the valid target not sends during the transmission of the fraudulent device. Otherwise, all data will be corrupted.

Man-in-the-Middle-Attack. In this type of attack, two parties, say Bob and Alice, communicate and do not know that a third party, Eve, is involved. This third party tends to be malicious and the two legitimate parties do not know that this third party is involved. The communication from Alice to Bob passes Eve. Eve, can pickup this signal, modify it and passes it to Bob. This works the same in the other way around. Where Bob sends data to Alica and passes Eve.

So, NFC is not protected from these threats by default. It is not hard to hide an antenna and intercept communication. Moreover, those threats are taken so seriously that an official standard was designed. In 2010, the ECMA developed this standard, the EMCA–386 (standard described in [22]) also called NFC-SEC-01. This standard defines cryptographic mechanism for key exchange and data transmission. NFC achieves a secure key exchange by using ECDH (Elliptic Curves Diffie–Hellman) algorithm. The AES (Advance Encryption Standard) encryption algorithm is often used to protect data during communication.

3.3.1 Key Agreement

In figure 7, the key exchange protocol, used in NFC-SEC is shown. This protocol consists of two phases. In the first phase, both parties agree upon a key. In the second phase, the key is confirmed. Before the protocol can start, both the sender S and de target T should be in possession of each other's ID. This procedure is described in Section 3.1. Both parties have generated their private and public key using an Elliptic Curve. The public key from the sender is called K_S^{pub} and the public key from the target is called K_T^{pub}.

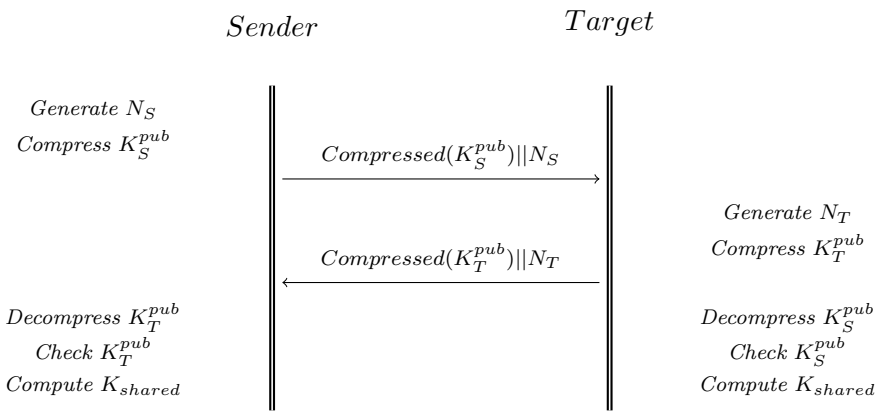


Figure 7: Key agreement protocol, used in NFC-SEC

The sender starts with generating a nonce N_S. After that, the sender compresses its elliptic curve public key K_S^{pub}. Both values are sent to target T. Target T, generates, even as the sender, a nonce N_T. The target also compresses its public elliptic key K_T^{pub} and sends both values to the sender. The sender and the target decompress the received key value and verify them. If these values are verified, the shared secret K_{shared} is calculated. This shared secret is later used as an input for the key confirmation process.

3.3.2 Key Confirmation

The calculated secret K_{shared}, is used as an input for the KDF (Key Derivation Function) to calculate the master key K_m, as shown in Figure 8.

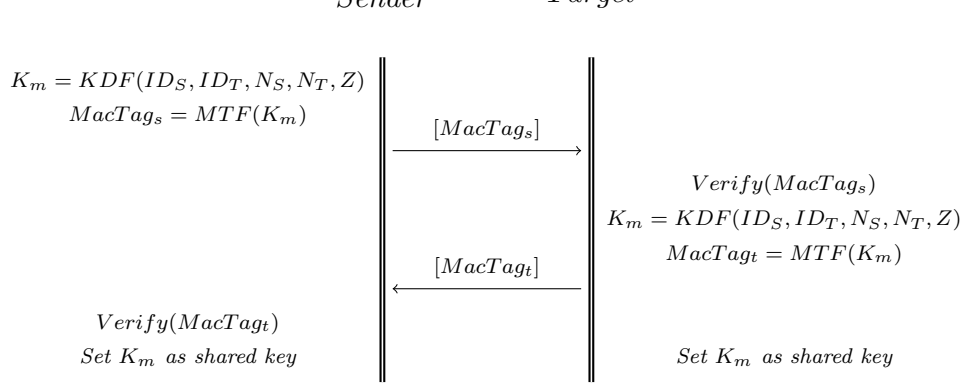


Figure 8: Key confirmation protocol, used in NFC-SEC

This master key is later used to encrypt the communication between both parties. To make sure that both parties agree on the key, a key confirmation phase has to be made. This is done using the MFT (Mac Tag Function), where the master key is an input. The result $MacTag_s$ is sent to the target. The target verifies the result and calculates $MacTag_t$ and sent this to the Sender. If this value is also successfully verified, the master key K_m is set as a shared session key.

3.3.3 Encryption

Data communication is more straightforward, compared to the key-exchange and key confirmation part. A schematic overview is given in Figure 9.

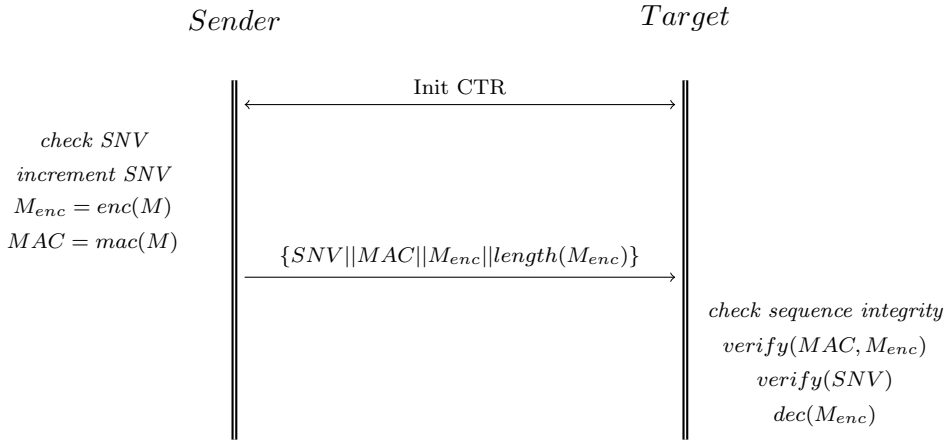


Figure 9: Protocol overview

As shown in Figure 9, the first thing to do on both sides is to set the CTR (counter). This counter is necessary to make AES run in AES-CTR mode. Thereafter the SNV (Sequential Number Variable) is initialised.

A sender S, sends a message M by first checking its current SNV value. If this value has reached 2^{24-1}, the protocol terminates. Otherwise it proceeds by incrementing the SNV. It then encrypts the message M, using the AES-CTR encryption algorithm, this results in M_{enc}. The next step is to calculate the MAC (Message Authentication Code), by calculating $MAC = f(M_{enc}, length(M_{enc}), SNV)$. The sender S sends M_{enc}, $length(M_{enc})$, SNV and MAC to target T.

The target T receives its data and extracts M_{enc}, $length(M_{enc})$, SNV and MAC. The target T checks the integrity of those values. If the SNV value is valued 2^{24-1}, terminate the algorithm. If all values are valid, the data can be decrypted.

4 Available NFC devices

This section explores the possibilities for using NFC on a smartphone running on Android. As it turns out, at presence there is no out of the box Android smartphone equipped with NFC available. However, there are other possibilities, such as porting Android to NFC equipped mobile phones, or using external devices what extend a mobile phone with NFC. In this section, potential solutions are explored and discussed. In section 4.1 existing mobile phones are explored and evaluated. Section 4.2 discusses external hardware which can be adapted to a mobile phone. In section 4.3, the possibilities are evaluated and a decision made which solution is adopted for this project.

4.1 Mobile phones

Today, only a handful of mobile phones equipped with NFC are available [20], these mobile phones are listed below:

- Samsung's SHW-A170K;
- Sagem Wireless Cosyphone;
- Samsung S5230 NFC;
- Hedy;
- Shanghai Simcom;
- Nokia C7;
- Nexus S;
- Casio IT-800RGC-35;
- Axia A306.

All mobile phones listed above, except for the Nexus S, have their own proprietary operating system. Those operating systems are developed by the manufacturer of the phone itself. The main drawback of those operating system is that they lack flexibility, because there closed nature. Another problem is that none of these mobile phones are ported to Android. The Casio IT-800RGC-35, Axia A306 and Nokia C7 do have their own development environment. With this development environment, developers can build their own applications. However, a requirement for this project is that the mobile phone should run on Android. The Nexus S runs on Android Gingerbread. At the moment of writing, this mobile phone is only available in United States.

4.2 External devices

There are external devices which can equip mobile phones with NFC. It is possible to extend an "ordinary" smartphone with an external device in order to use NFC. The idea is to connect an external NFC device to an Android smartphone. Various external NFC devices exist:

The iCarte. This device [6] is a hardware extension exclusively made for Apple's iPhone. This external reader can than be attached to the bottom of the iPhone and enable NFC. This hardware extension is only available for the iPhone 3GS and iPhone 3G.

microSD. The companies: Tyfone [13], SDiD [10] and Cell-Idea [2] have developed a microSD. This solution is to equip an ordinary microSD card with an NFC chip. This microSD card can be inserted in the sd card slot of a mobile device. The SDiD device only runs with Windows Pocket 2003/2004, Windows Mobile 5.0/6.0/2003 and Palm OS 4.1. The Cell–Idea solution is only compatible with BENQ T80 and Fonelabs X3 devices. The solution offered by Tyfone has a drawback of using a closed device driver. This is a problem when this device drivers needs to be ported to Android.

With–me. Cell-Idea has also developed the "With–me" device. This is an separate device and communicates with the mobile phone using Bluetooth [3]. The drawback of this device is that it is no device drivers released for Android. There is also no open source driver available.

MyMax. Twinlinx has developed a sticker with a built-in battery, Bluetooth- and NFC-antenna. This sticker is called "My–Max" [12]. This sticker can be attached on an ordinary mobile phone equipped with Bluetooth. It then transforms NFC signals into Bluetooth signals. A platform independent development kit is expected but not yet available.

External NFC reader. This reader is made available by Arygon [1] for desktop PC's. It is an USB device which has a NFC chip build in. They support different operating systems, for example Windows XP/Vista/7 and Linux kernels 2.6 and up have a build in device driver. The NFC reader can be places in a USB socket from the desktop PC and used for NFC transaction. The drawback is that this type of external NFC reader is not suitable to run on a mobile phone. However, with additional hardware it is possible to connect an external USB device to an Android smartphone.

A smartphone does not provide enough power to power the external reader. To tackle this problem, a dual-USB cable can be used. This cable has an extra connector for power

supply. This extra connector can be attached to an ordinary computer, pure for power supply.

Modifications to the Android operating system are also needed. The mobile phone should be able to detect the NFC reader. This means that the device should be turned into USB-host mode. To use the reader, the open source device driver from the reader needs to be installed, the usb- and nfc library (both written in C) need to be ported to Android. Lastly, write a JNI-wrapper (Java Native Interface) to cover the functionalities provided by the NFC library.

4.3 Evaluation

After summarizing all the possibilities, one can say that there is working a solution available. The provided mobile phones that are equipped with NFC are not ported to Android. It is not an option to port Android to those platforms. Porting an operating system to another hardware platform is rather complicated and very time consuming. This means, that those mobile phones are not suitable for this project. Most fitting would be the Nexus S. This mobile phone is suitable for this project: it has build in NFC and a programmable API. Unfortunately, this mobile phone is not available in Europe at the time of writing this thesis and it is not clear when this mobile phone will be released.

The external devices are also not promising. The iCarte only runs on Apple's iPhone. The microSD cards are not compatible with Android and are not open source. Both arguments also apply to the MyMax sticker.

An external NFC reader seems to be the most promising solution. Since this reader is compatible with Linux kernels 2.6 and higher. It is clear that it takes effort to modify Android in such a way that this solution will work.

However, Android does not have this driver by default. This means that the Android kernel needs to be recompiled.

5 Android

Android is an operating system for mobile phones, which was initially developed by Android Inc., situated in Palo Alto, California. The aim of the Android developers is to build a system which is flexible and upgradable at the same time. In 2007, Google was interested in a company related to software development for mobile phones, because it wanted to enter the mobile market with their own mobile platform. In 2007, Google bought the Android Inc. company and released their own mobile platform. Since then, Android is a rapidly growing mobile platform, and has a market share of 25.5 % in the third quarter of 2010, according to Gartner market research [5].

Today, Google's Android is the center point of the, so called "Open Headset Alliance" (abbreviated as: OHA). The goal of this alliance is to create an open platform, which accelerates innovation. This alliance was founded on the 5^{th} of November 2007 and consist of 79 companies which can be subdivided into five subgroups:

- **Mobile Operators**, e.g.: Vodafone and T–Mobile;
- **Semiconductor Companies**, e.g.: Intel and ARM;
- **Software Companies**, e.g.: eBay and Google;
- **Handset Manufactures**, e.g.: Asus and Acer;
- **Commercialisation companies**, e.g.: Noser and Wind River.

The aim of the OHA is to commercially deploy the handset and services for the Android platform.

5.1 Android architecture

The Android architecture is subdivided into five main parts, (1) the Linux kernel, (2) the libraries, (3) Android runtime, (4) application framework and (5) applications. In Figure 10 a logical placement of all these components is illustrated, an in depth explanation about the Android architecture can be found in [14, 39].

5.2 Linux kernel layer

The heart of the Android platform is the Linux kernel,seen as the bottom layer in Figure 10. This layer provides the core services, including Memory management, security models, networking, process management, power management and hardware drivers (e.g. USB, display, WiFi, etc.). These services are handled by a Linux 2.6 kernel. The Linux kernel is used as a HAL (Hardware Abstraction Layer). This HAL improves the ability to move Android to an other hardware platform by changing hardware specific software which is placed in the Linux kernel.

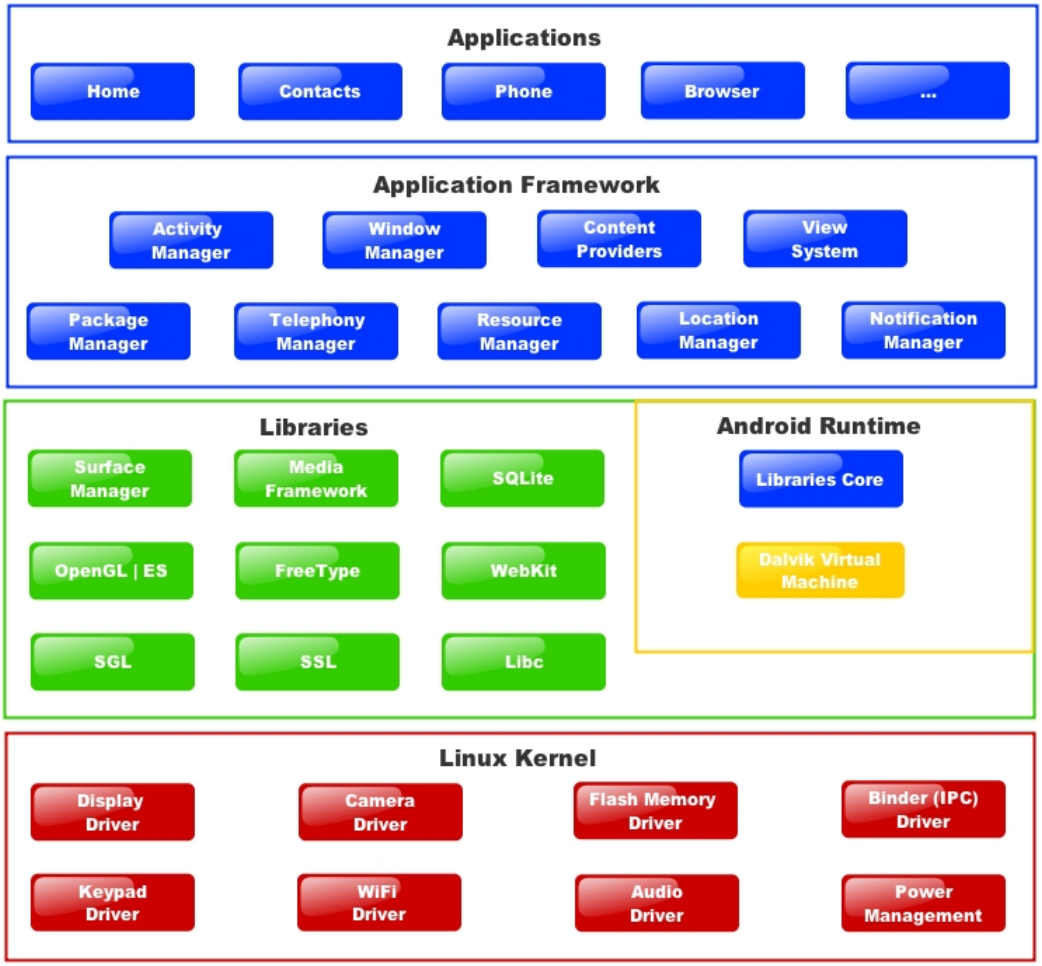


Figure 10: The Android architecture (adapted from [14])

5.3 Native libraries layer

Running on top of the kernel, the libraries give an application the functionality of the Java programming language. All libraries are written in C and C++. Examples of these libraries are:

- System C libraries. This is an implementation of the standard C system library (libc);
- Media libraries. These libraries support the playback of common audio and video formats;
- SQLite. This is a database system which is available for all applications.

5.4 Android runtime layer

This layer is specific designed for embedded devices. Such devices have specific requirements for battery life, memory and CPU. The main component of this layer is the DVM (Dalvik Virtual Machine), every process has its own instance of the DVM. The DVM has been designed to efficiently handle all these instances.

The DVM runs .dex files. These are byte codes that are the result of converting .class and .jar files at built time. When these files are transformed to `.dex` files, they become much more efficient byte code and run more efficient on small processor. For example they use memory more efficiently and data structures are designed to be shared between processes.

The core libraries are written in C and rely on the functionalities of the underlying Linux kernel. The Java libraries provide the majority of the functionality provided by C libraries and Android specific libraries, such as: utilities, IO, data structures, etcetera.

5.5 Application framework layer

The entire Application framework is written in the Java programming language. This framework can be seen as the toolkit every application uses. It provides classes which are used when an Android program is built. The framework provides an abstraction for hardware access, it manages the application resource and user interface. This layer also provides a set of services and systems, for example:

- Views, which are used as a building block for Android Applications;
- ContentProvides, which let applications share data between each other;
- Activity Managers, which controls the live cycle of Android activities. An activity is a class which interact with the user.

5.6 Application layer

All applications, both native and third party, are built using the same available API, and are logically placed in the application layer of the system. Every application runs in a so called sandbox. With this sandbox approach, Linux separates each process from each other and the operating system. Every application has a unique ID, and the operating system does have a set of rules based on these ID's.

6 MiPass Payment

This Master Thesis proposes a new payment system called MiPass. MiPass is a payment system designed for payment of parking spaces. The main goal of this payment system is to create a replacement for existing parking tickets-system. The main advantage of this system is that users always pay only for the time they really parked their car, which means that each user is charged per minute. Normally, if a user such as, for example, Bob wants to park his car, he buys a ticket at a parking meter. However, in this system, Bob has to pre-estimate the exact time he will park his car, which could be "two hours". Based on this guess, Bob then inserts the required money into a parking meter. The parking meter in turn prints a ticket, which is valid for two hours exact. However, if Bob returns after only one hour, he cannot get his money back, so Bob paid too much. On the other hand, if Bob returns after three hours, he faces the risk of a fine. Ideally, Bob returns after precisely two hours.

The ideal scenario is unrealistic while other scenarios are unfair, simply because it is quite hard to know exactly how long a parking space is needed. So, MiPass is based on the idea that Bob can use his NFC-equipped mobile phone to pay for his parking spot. This means that he has to "check-in" when parking his car. Checking-in in this context means that Bob swipes his phone at the parking meter upon arrival. In doing so, information about the payment is exchanged. When Bob leaves the parking spot, he again swipes his phone in front of the parking meter. This action is referred to as "checking-out". This action finalises the payment, the exact price of the parking is determined by calculating the difference in time between the check-in and check-out.

This section is structured as follows: in Section 6.1, existing academic knowledge about micro payments is summarized and discussed. In Section 6.2, the systems' architecture is presented. In Section 6.3, the interaction between different parts of this architecture is elucidated.

6.1 Related Work

At closer look, it appears that there is a rather large body of research on micro payments. This Section covers an analysis of this existing work. First, in Section 6.1.1, quality attributes are discussed. These attributes explain which properties are important, if a new system is designed that has the objective to replace real money. In Section 6.1.2, an overview is given of the basic protocol structure of virtually all payment protocols. In Section 6.1.3 the difference between online and offline protocols is explained. Finally, Section 6.1.4 summarizes existing payment protocols.

6.1.1 Quality attributes

To design a new and flexible mobile payment system, and a replacement for "real" money in form of notes and coins, it is necessary to take properties of real money into consideration. In [43, 17, 21], a list of attributes is provided. An overview of these properties can be seen in Table 4:

Attribute	Description
Acceptability	All parties involved should accept the payment system;
Cost	The transaction costs should be as low as possible;
Anonymity	No party involved should be able to pinpoint who has spend which money where, fraudulent users should be caught;
Intraceability	No party should be able to trace money in the system;
Speed	The transaction should be processed in an acceptable amount of time;
Invention	It should be impossible to create coins by any other party than a trusted third party, e.g. a bank;
Offline	Transactions should preferably be processed without invoking a trusted third party;
Overspending	It should be impossible to spend more money than a user has.

Table 4: Quality attributes of "real" money which should be preserved in e-cash systems

6.1.2 Protocol structure

A typical electronic cash (abbreviated as e-cash) system has three entities; (1) the customer, or the person who spends money, (2) the vendor, who sells items or services to the costumer; and (3) the broker (in some existing literature referred to as "bank"), which is responsible for maintaining the bookkeeping, fraud detection, and blocking users from the system.

A typical payment system has three or four phases as depicted in Figure 11, namely the initialization phase, the transaction phase, the redemption phase, and sometimes a revocation phase. In the initialization phase, all users and vendors set up an account with the broker. The user is required to deposit some money into an account. The broker in turn sends a certificate signed by the broker to the user and vendor. The transaction phase is the phase where the transaction takes place. The implementation of this phase can differ from system to system. The redemption phase describes the process of transferring the money to the vendor from the bank. This is processed in the following way: the vendor sends a request for open payments to the broker. The broker then administrates these payments. Some protocols do have a revocation phase. This phase is added to avoid that

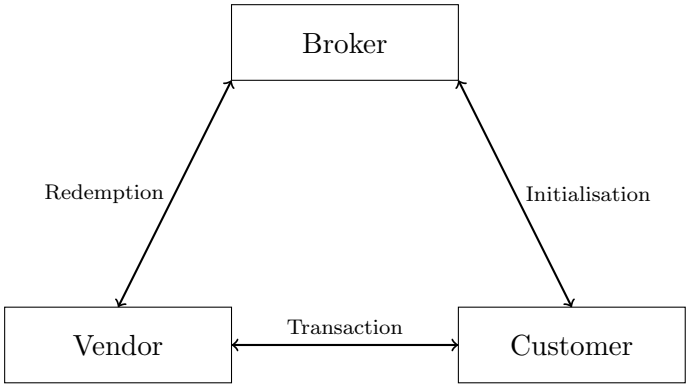


Figure 11: Micro-payment basic architecture

digital money can live forever in the system. If, for example, a broker expects a digital coin to live with a maximum lifespan of one month and that digital coin is still in the system after that period of time, the coin is removed from the system and the value of that coin is returned back to the users account.

6.1.3 Offline and Online protocols

In the existing literature, two groups of payment protocols are pointed out: offline and online. In online payment, every transaction between a vendor and a user is sent to the broker, this means that the broker needs to be online. The broker is then able to ensure that the coins are not spent more than once, or even invented by a malicious person in the system. However, following this approach, the central server can easily become a sort of bottleneck for this system. If the broker is not functioning for one or the other reason, the system cannot perform any transactions at all. Examples of online payments can be found in [19, 15].

Offline payment means that a vendor does not need to consult the broker on every payment. Instead, the transaction is performed between the client and the vendor. Offline system contact the broker on certain intervals e.g. once a day (other intervals are also possible), the vendor sends all transactions from that day to the broker. The broker uses this data as an input to maintain all administration. Not contacting the broker implies problems in, for example, preventing the user from overspending or double spending his money. Overspending can usually only be detected after it has happened. Also, only when the vendor synchronises with the broker at the end of the day, the broker is able to see if a certain coin has been spent twice.

6.1.4 Existing protocols

One of the first offline payment protocols came from Ronald Rivest and Adi Shamir [45]. They proposed "PayWord", which is based on a chain of hash values. Every link in this hash chain is called a payword. Every payword is a representation of money, for example, one cent. Every payword has the same value, and all items that are bought must be worth one, or a multiple of one payword.

Initially, both the user and the broker have to set up an account with the broker. The broker also issues certificates to the users. These certificates contain, for example, users name, or user public key, and are used to generate a user- and vendor-specific payword chain $\omega_1, \omega_2, \ldots, \omega_n$. The user creates a payword chain in reverse order, by picking a random ω_n. Then for $i = n-1, n-2, \ldots, 0$, using a hash function $h()$:

$$\omega_i = h(\omega_{i+1})$$

During the first transaction between the user and the vendor after the vendor contacted the broker last, the user first sends a "commitment". This commitment contains the root ω_0 of the payword chain, the users certificate obtained by the broker and the expiration date of the certificate. Upon the $i-th$ payment, the user sends (ω_i, i) to the vendor. Together with the commitment from the user that has been received beforehand, the broker is able to verify the payment. It is important to note that the system proposed by Rivest and Shamir is post paid, meaning that the user pays real money after the purchases, e.g. on the end of the month.

At the end of every time-frame, the most recent (ω_i, i) pairs and commitments from all users are sent to the broker. The idea proposed in [45] has several shortcomings, payments are not anonymous, every payword has exactly the same value and, more importantly, a single hash chain can be spent by only one vendor. However, the idea of paywords is an important aspect for further research in this field, see for example: [36, 49, 50, 40].

The payword protocol proposed by the authors of [36] makes it possible to spend the same hash chain with multiple merchants, using a single hash chain. It also provides non-repudiation and atomicity. This protocol is designed to buy content over the Internet. When a user buys, for example, a news article the system functions by sending the product id to the vendor. The vendor responses with the encrypted news article, which was just bought. The key to decrypt this article is sent after the payment is finalised by the customer. When the customer buys an article at the next vendor, the customer has to provide the certificate of the previously seen vendor together with the payword spent there to the current vendor. Once the current vendor wants to get the required payment, it sends its own certificate, and the certificate from the previously seen vendor by the user, plus its received payword and the payword from the previous vendor to the broker. The broker is then able to construct and verify the whole hash chain.

Another limitation of PayWord [45] is that every hash element is worth the same

amount of money. The authors of [47], proposed to introduce multiple hash-chains, and this multi hash chain property introduces less hash calculations. The key idea is that every hash chain supports a different value of money. For example, one hash-chain is built for quarters, one for dimes and one for "whole" dollars. Those hash chains are also suitable for spending by different vendors.

The authors of [24], proposed an idea to use a single hash chain, which can be used by multiple vendors. In contrast to the idea of, e.g. [36, 45], the proposal of [24] does offer anonymity. When the user buys credit, the broker stores the length of the hash chain, together with the root ω_0, the length of the hash chain, the timestamp of the chain, and also an additional random number. When a user wants to pay, it sends this random number to the vendor, together with the hash value ω_i and i. The vendor invokes the broker and sends the information received from the user to the broker. The broker can identify the record from the user by its random number. The broker verifies this value and sends the result back to the vendor. The main shortcoming of this protocol is that the system is online instead of offline, since every payment is passed to the broker. However, it does prevent double and overspending.

In contrast to payword-based solutions, the authors of [17] suggest a differing protocol, called PSP (Private Secure Payment), which is developed as a payment system for RFID systems. In PSP, the main idea is that the RFID tags do not contain any money, instead they have information to create money. A user initially "charges" an RFID tag at a broker. In exchange for real money, he or she receives information to generate coins. The readers are prepared with a so-called "bloom-filter". This filter has the property of always accepting items which belong to the collection that a particular filter maintains, but it also has a small chance of accepting items which do not belong to the collection the filter maintains. However, to keep the size of the bloom filter manageable, the system is divided into so-called "epochs". One epoch can, for example, be one month. Every e-coin has a lifetime of τ e-pochs. When an e-coin expires, it is removed from the system and given back to the user.

None of the payment protocols discussed above prevents from real time double spending, with the exception of [24]. In other words, fraud is detected only eventually. All schemes perform fraud checks after the synchronisation with the broker. As it turns out, real time double spending checks in offline payment systems are extremely hard to perform, especially with multiple vendors.

There are a few solutions for double spending. However, most of these solutions demand the vendor to make a connection to the broker in some cases when a certain criteria is met. Most authors base their approach on risk management. This essentially means that the cost of breaking the system exceeds the benefit expected of theft [48]. The author of [48] proposed a solution to decrease the benefit of double spending and overspending, by creating an upper-bound on the value a user can earn with fraud. The approach the author chose is by partial real time fraud detection. The partial is the number of transactions

which are verified, the smaller the partial, the higher the upper-bound, but the cheaper the fraud prevention. The bigger the partial, the more money is paid to detect fraud, but the upper-bound is decreased. The authors of [35] proposed the idea let the vendor verify a transaction by the bank with a certain probability, the probability increases when the value of the transaction rises.

6.2 Architectural overview

The protocol proposed is a new solution for a parking space payment system. The system makes use of NFC for such payments. The scheme is based on a chain of paywords, as in [45]. In MiPass, every payword has the same value of one cent.

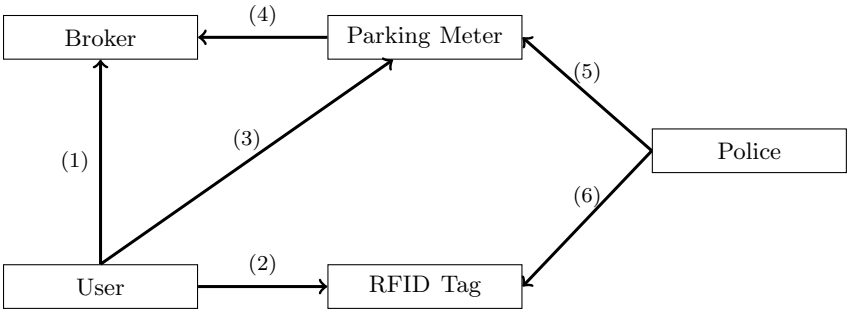


Figure 12: Architectural overview

Figure 12 provides an overview of the different actors in the system. The actors are: the broker, the RFID tag, the parking meter, the user, and the police. The arrows from the user are communication session initiated by the user, whereas the arrows which start at the parking meter, are communication sessions initiated by the parking meter. Arrows originating from the police are initiated by them. Those connections can work independent from each other. More details on these actors are provided in the following.

1. **User.** Every user has a mobile phone with NFC. The user interacts with the broker to add money on his phone in step (1) in Figure 12, which can be spent at the parking meter (3). The user also interacts with the RFID tag, which is attached to the car window (2);

2. **Broker.** The main role of the broker is transforming real money (e.g. Euro) to virtual e-coins. The broker also blocks fraudulent users, performs fraud checks, etcetera;

3. **RFID tag.** This tag is attached to the car window. It contains information about the car, such as color, brand, and license plate, but also personal details of the car owner;

4. **Parking meter.** The parking meter stores information about the cars parked in its zone. For every car parked, it keeps the car properties, deposit, and time of arrival. The parking meter synchronises every time τ with the broker, τ can, for example, be one day. This is step (4) in Figure 12;

5. **The police.** The police can check everybody's payment. It can do so by loading information from the local parking meter to a police phone, in step (6). This information contains properties about the cars parked (a copy of the data on the cars RFID tag) and payment info (deposit, time of arrival, etc.). When the police has this information, the officer can check every car by scanning its RFID tag, step (7). The officer can also double check the content stored in the RFID tag or parking meter by physically observing the car and comparing this to the information he knows.

6.3 Interaction between the actors

In Figure 12, various actors are displayed. The main focus of this section lies in the interaction between all actors and the values which are exchange between them. This section only focuses on the payment itself, which has the task to perform random checks to see if someone has paid or not. The symbols used in this protocol are explained in Table 5.

6.3.1 Basic protocol idea

As in [45], this protocol is also based on hash chains, with the difference that our hash chains are generated by B, instead of U. Also, our protocol let U pay before instead of after the purchase. Our protocol is also able to spend the same hash chain at multiple vendors (in the context of this thesis, vendors can be replaced by "parking meter"). A drawback of this protocol is that it is vulnerable to double spending.

The idea is that B produces a pool of seeds upon forehand. In fact, all seeds are generated by B and are identified by j, without loss of generality, let us say that B generates J seeds, for: $j = 0, 1, \ldots, J$. When a hash chain is generated later on, this hash chain is also identified by the same j as the seed. For instance ω_i^j means the i^{th} element of the j^{th} hash chains.

The seeds are propagated to all P in de system, before they can be used to generate a hash chain. If U wants to "charge" a phone with some credit, U sends a certain amount of money to the broker. Lets assume that one hash value is worth one cent, and U pays $\underline{A}$ cent to the broker. The broker then, picks a the j^{th} seed from its pool and calculates ω_0^j: $\omega_0^j = h(h^{A-1}(\omega_L^j), \omega_L^j)$. Thus, value of ω_0^j is a hash of the preceding value ω_1 and the

Symbol	Description
B	Broker, also trusted third party
U	User
P	Parking meter
$h()$	A cryptographic hash function, $h^2(x)$ is equivalent to $h(h(x))$, $h^3(x)$ is equivalent to $h(h(h(x)))$, etc.
l	The current length of the hash chain
L	The original length of the hash chain
ω_i^j	The $i-th$ element of hash chain j, calculated as $h(\omega_{i+1}^j)$
$\overline{\omega_i^j}$	The current root of the hash chain, calculated as $h(\omega_{i+1}^j, \omega_L^j)$
ω_L^j	The seed of the hash chain
K_X^{priv}	Private key from party X
K_X^{pub}	Public key from party X
CP	Properties of a car, those properties make a certain car uniquely identifiable (e.g. color, license plate, brand etc.)
γ	Cents per minute
D	Deposit, what the user pays in advance
ΔT	Check-out time minus check-in time, expressed in minutes
A	Value of the transaction, in expressed in Euro-cent (e.g. €0.01)
$\{Q\}_{sig_X}$	Message Q signed with the private key of X
TS_X	Time stamp created by X
$Cert_X$	Certificate belonging to X, issued by the broker

Table 5: Notation table

seed of the hash chain, ω_L^j. By doing so, the parking meters have enough information to verify whether the provides $\overline{\omega_0^j}$ is valid or not. More importantly: the user is unable to generate his own money or learn the original hash value, since the seed of the hash chain is kept secret by B and P.

6.3.2 Initialisation phase

In the initialisation phase, the user U opens an account with the broker B. In this scheme, as illustrated in Figure 13, U provides B with personal details. This means, full name, address, etc. (e.g. a copy of a passport, belonging to U). This step is indicated with "Hi, I am U". In turn, B provides U a certificate $Cert_U$. This certificate contains the public key of U, expiration date E and unique account identification AI and is signed by B, such that: $Cert_U = \{K_C^{pub}, E, AI\}_{sign_B}$. This certificate is required to be able to identify users efficiently. If a malicious person, Eve, performs, for example, a fraudulent action, B can revoke or blacklist her certificate. The user U can also contact B if his mobile phone has

been stolen, in which case B blocks or revokes the certificate of U.

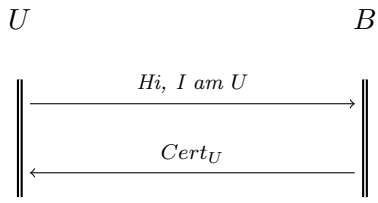


Figure 13: Initialisation phase between B and C

Once the account has been established by B, U is able to deposit money on the account, as shown in Figure 14. A typical scenario of depositing money can be where U interacts with a vending machine with a NFC reader. Examples of such machines are used in the United Kingdom and the Netherlands to respectively charge the Oyster and OV-Chipcard card. A user does this by transferring an amount of money, A, to B together with $Cert_U$. This $Cert_U$ is verified by B. The next step B performs is picking a complete pre-generated seed j from its pool and generates a hash chain from this seed, which represents a value of A cents. The root element, ω_0^j, of this hash chain is not only a hash value of its preceding ω_1^j but also from ω_L^j, this value is indicated as: $\overline{\omega_0^j}$. The $\overline{\omega_0^j}$ value is encrypted with the public key of U and signed by B. In this case, only U can decrypt this value, but U can also verify whether this value really comes from B. The encrypted and signed value of $\overline{\omega_0^j}$ is sent to U together with the length of the hash chain, l and the index j. A copy of the complete hash chain is stored at B and linked to $Cert_U$.

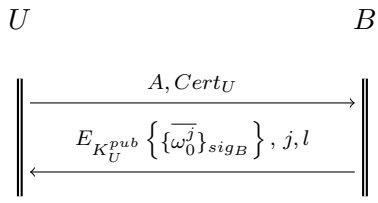


Figure 14: Obtaining credit

However, the interaction depicted in Figure 14 is still vulnerable to the man-in-the-middle attack. An adversary ADV can replace the $Cert_U$ with its own $Cert_{ADV}$ and change the the reply of B to its own advantage, i.e. replacing it with another value which is signed by the bank, but is worth less that the value A sent to the broker by U. Only when U contacts a parking meter, U will notice that something went wrong during the transaction with B. An improvement could be that broker asks the device of U to display a "QR" (Quick Response) code on its screen, which represents a fingerprint of $Cert_U$. Now B is able to compare the digitally received certificate with the optical scanned certificate

fingerprint of U. When those values matches, B can be sure that the claimed identity is true.

6.3.3 Payment phase between parking meter and user

When U pays in the system, U needs to pay a deposit D to the parking meter P. This deposit is a guarantee that U pays for at most τ minutes, where $\tau = D/\gamma$. To perform a complete transaction, U sends the next message to the parking meter, as depicted in Figure 15.
The value of j is used to lookup the proper ω_L from the local data-store of P. Then, P calculates $h^{l-1}(\omega_L^j, \omega_L)$ and compares this to the received $\overline{\omega_i^j}$. This check proves validity of the received $\overline{\omega_i^j}$ by P. The next step is to determine, whether U has enough credit to pay D, by verifying whether $(l - D) \geq 0$. To subtract D from the balance of U, P calculates: $i = i + D$ and $l = L - i$. The value of $\overline{\omega_i^j}$ is calculated by P: $\overline{\omega_i^j} = h(h^{l-1}(\omega_L^j), \omega_L)$. Then, the new value of l and $\overline{\omega_i^j}$, even as the TS_P, is sent back to U.

P stores D, l, CP, ω_L^j and $Cert_U$. The CP value is stored to make life for the police easier. This value contains properties of the car owned by U, e.g. , color, license plate, brand. The TS_P that is sent from P to U, is for the administration of U (e.g. an alarm may sound on the device of U if the parking time is almost elapsed).

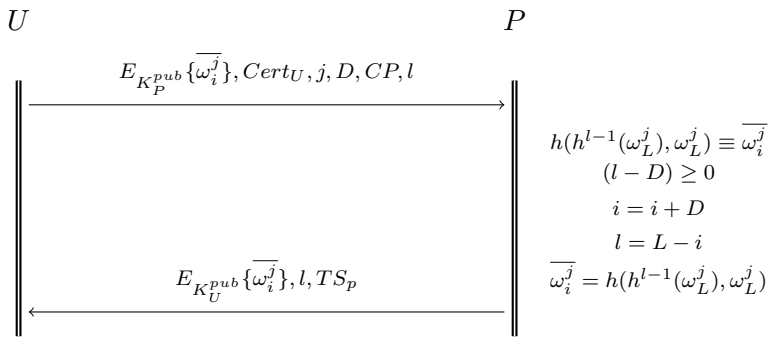


Figure 15: Check-in on the parking meter

When U returns back to P, and wants to have the remaining part of D, U sends $\overline{\omega_i^j}, Cert_U, j$ and l back to P, as shown in Figure 16. A lookup is performed by P, using j. The parking meter P verifies whether the received ω_i^j is valid by comparing is to $h(h^{l-1}(\omega_L^j), \omega_L^j)$. Then, P calculates the remaining deposit R, $R = D - \Delta T * \gamma$. To give U the remaining D, R back.

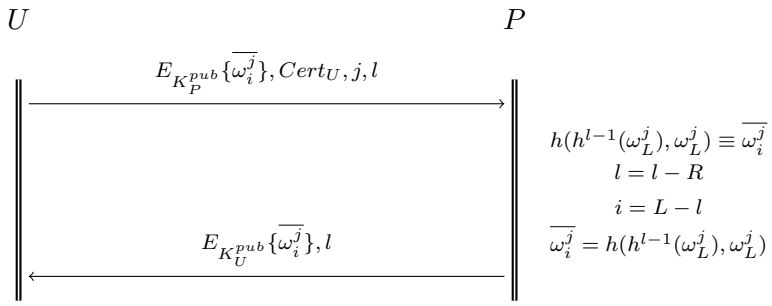


Figure 16: Check-out on the parking meter

6.3.4 Role of the police

The police has to verify whether U left a sufficient D, taht is, if he paid enough money or to verify whether U paid at all. The police does so by first loading all records of all parked cars that are checked in from that particular parking meter. Then, the police verifies all cars by scanning their RFID tag and compare this information, with the information loaded from the parking meter. If the time to park is elapsed, the police may give U a fine.

6.3.5 Possible extension

This protocol is also suitable for payment of ordinary consumer goods, such as: a bottle of soda or a candy bar. However, then the interaction is between the user and the vendor. The transaction is also slightly different, as depicted in Figure 17. In this overview, the symbol V is introduced. This symbol represents the vendor. The *Amount* indicates the amount of money a user has to pay, expressed in cents. Initially, the user sends $E_{K_P^{pub}}\{\overline{\omega_i^j}\}, Cert_U, j, l$ to the vendor. The vendor verifies the values of ω_i^j, by calculating $h(h^{l-1}(\omega_L^j), \omega_L^j)$. Then, the vendor checks whether the hash chain of U has enough value left to perform the payment. The length of the hash chain is then decreased, and the new $\overline{\omega_i^j}$ is calculated and sent back to U.

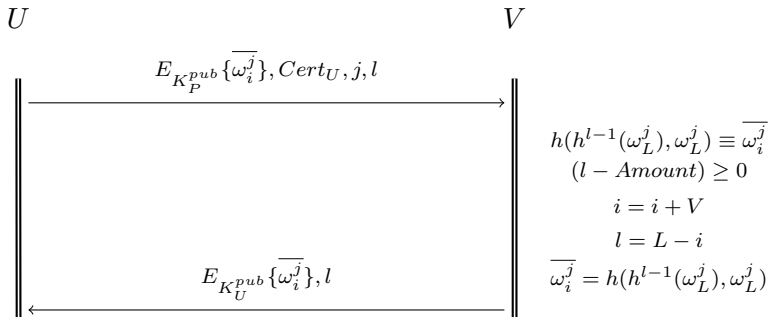


Figure 17: A normal payment

6.3.6 Auditing

At the end of every fixed time-frame (a possible time-frame could be a day), every parking meter sends the most recent ω_i from every hash chain j which P has seen that day, together with l and j belonging to that particular hash chain. The broker verifies all received hash values, using the chain it already knows. It verifies, if no double spending is performed, by looking for duplicate values, B also lowers the current value of the chain A, in its own administration. If, for example, one day later a payment is made where the received l value from a parking meter is larger than the value of the chain A, B has detected double spending. The broker sends back a blacklist with certificate which should be rejected by the parking meters.

6.3.7 Evaluation

In Table 6, the payment systems as discussed in Section 6.1.4 as well our new proposal are compared on different properties. In this table, the bolded words are considered as good properties. In the existing literature, no payment scheme is able to handle payments which need to have a deposit. A scenario where this might be possible (besides parking a car) is, for example, a ride with the bus, where the user pays a deposit when he enters the bus and receives his deposit back at his destination, with the price of the bus fare subtracted from the deposit.

As with many offline payment protocols, it is extremely hard to trace double spending in real time. The MiPass protocol also lacks a possibility to avoid this. A possible scenario would be, that a malicious person Eve charges her phone with a certain amount of money, and then stores the message received from the broker in her malicious application. She then can replay the received message from the broker to different parking meters that same day. However, it is possible, to check on double spending if a user spends the same hash values on the same parking meter. The hash chain is also linked to the users certificate. The only thing a user can do is to send his hash chain, together with his

certificate to all her friends. Then, the users friends can spend this money at different parking meters – only in such way can a user benefit from fraud. However, all of the users friends would need to use this hash chain on the same day. If a user performs fraudulent actions, his identity is revealed and rejected by the system. Thus, while the scheme does not protect against double spending, it is able to detect double spending, only after it has taken place. However, the solution proposed in [24] does prevent from double spending, but this is achieved by invoking the broker on every transaction. This drops the "offline" criteria and is therefore slower in processing the transaction compared to the other payment systems. In exchange, double spending can be totally prevented. Most payment schemes are used for online payment, for example, to buy content on a website. Only MiPass and PSP can be used in daily life. PSP is still vulnerable to coin invention, although with a small probability. However, this is not the case with MiPass.

	Payword and Micromint [45]	A micro-payment system for multiple-shopping [36]	AMVPayword [24]	PSP Private Secure Payment [17]	Micro-payment Protocol Based on Multiple Hash Chains [47]	MiPass
Application	Online purchases	Online purchases	Online purchases	Public transport	Online purchases	Parking & (small)products
Is it implemented?	No	No	No	No	No	**Yes**
Able to handle deposit	No	No	No	No	No	**Yes**
Technology used	Internet	Internet	Internet	RFID	Internet	NFC
Cost	**Low**	**Low**	High	**Low**	**Low**	**Low**
Offers anonymity	No	No	**Yes**	**Yes**	No	No
Offers intraceability	No	No	Yes	**Yes**	No	**Yes**
Speed	**Fast**	**Fast**	Slow	**Fast**	**Fast**	**Fast**
Avoid invention	**Yes**	No	**Yes**	No	No	**Yes**
Offline	**Yes**	**Yes**	No	**Yes**	**Yes**	**Yes**
Avoid double spending	**Yes**	No	**Yes**	No	No	No
Avoid overspending	**Yes**	No	**Yes**	**Yes**	No	**Yes**
Pre/Post paid	Post	**Pre**	**Pre**	**Pre**	**Pre**	**Pre**
Used data structure	HC	HC	HC	BF	MHC	HC

HC Hash chain

MHC Multiple Hash Chains

BF Bloom Filter

Table 6: Micropayment systems compared

7 Implementation

In this section, the implementation of the project is discussed. Therefore, `libnfc` (a open source nfc library) needs to be ported to Android, this is discussed in Section 7.1. In order to use this library, changes to the Android internals have to be made, this is explained in Section 7.2 and explained by an example to obtain a good understanding of this procedure. In Section 7.3, the implementation of the broker and the parking meter is discussed. They are placed in one section, since they both use the same NFC java project.

7.1 Getting NFC to work on Android

In order to let Android work with NFC, a number of steps have to be made. The idea for this project is to use an external NFC reader from Arygon[3] and connect this reader to the Nexus One. The NFC reader has to be supplied with additional power, since the Nexus One alone is not capable of powering this reader. This solution does not work out of the box. This section explains the steps which have to be taken in order to let the Nexus One operate with an external reader. The components used in this project are:

- Nexus One mobile phone with Android Froyo 2.2.1 (build FRG83D);
- Ubuntu 10.04 as an Operating System on the host computer;
- External Arygon ADRB NFC-reader;
- USB dual cable type A;
- USB Female-to-Female connector type A;
- USB type A to micro-USB type B cable.

The items are connected as depicted in Figure 18. In this picture, the NFC-reader is connected to the female connector of the USB-dual cable and the Nexus One to the male connector. However, the other connector from the USB-dual cable is also a male connector. To solve this problem, a female-to-female connector type A is used. The host computer is used as a power supply for the reader, this could be replaced with a power supply.

The host computer also has the task of compiling the Android framework and additional Android applications. Therefore, the host computer needs to be prepared to perform this task This is discussed in Section 7.1.1. The Nexus One itself also needs modifications, since it is unable to support and detect the external NFC reader. This is explained in Section 7.1.2. The additional libraries, which are required for the external NFC-reader are ported to Android and this is discussed in Section 7.1.3. The Android

[3] www.arygon.de

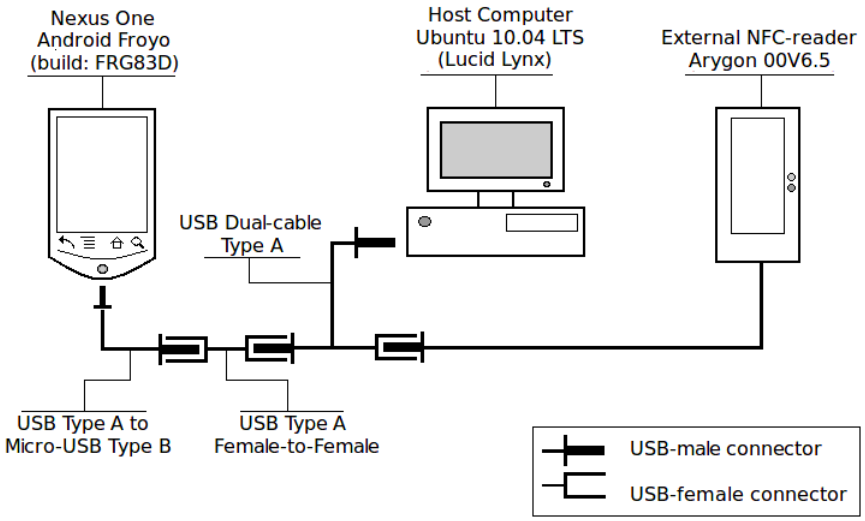


Figure 18: Project setup

kernel is also changed, since the Nexus One needs to run as a USB-host. Section 7.1.4 explains this. Lastly, in Section 7.1.5, the implementation is tested.

7.1.1 Preparing the host

Before the Nexus One can be modified, additional packages needs to be installed on the host computer. To start with, packages related to compiling and building programs are installed directly from Ubuntu's repository. However, Ubuntu 10.04 comes with `OpenJDK` (Open Java Development Kit) by default, this is an alternative for texttt JavaSDK (Java Software Development Kit). However, for Android, JavaSDK is needed. To do this, additional repositories need to be added to the `source.list` file to install it.

When all necessary packages are installed, the Android SDK has to be downloaded. This SDK provides tools which can be used to communicate to Android. This SDK is available as a starter package. To install the actual components, the `android` program needs to be started in the shell. It is important to install at least SDK platform-tools. This package contains programs related to developing and debugging programs but also the `adb` (Android Debug Bridge) utility. This utility is a powerful tool to make a connection to an Android device and execute commands on the device or install programs.

Before the new framework van be build, the `Android NDK` (Native Development Kit) needs to be installed. This software can cross-compile the Android source code. Cross-compile is a process where program code is compiled to function on a platform, other than the one on which the compiler runs. In this case, the compiler runs on an `i386` processor architecture, the Nexus device on an `arm` architecture. For convenience reasons,

it is recommended to add both, the NDK and SDK paths, to the Linux `$PATH` variable.

The next step is to download the source code from the Android code repository. This action can be taken with the `git` program. Downloading and compiling the source code can take a long time, up to three hours is not unusual. An important step is to comment the permission check in `su.c`. This piece of code checks if the user has enough permissions to gain root privileges.

7.1.2 Preparing the phone

The Nexus One needs to be rooted first. Rooted in this context means: gaining root privilege on the device. This privilege is necessary to modify the kernel and the framework. First, it is important to enable `USB-Debugging` on the Nexus One. USB-Debugging enables the ability to communicate between the Nexus One and the host computer. By doing so, it is possible to enter the Android shell and execute commands in Android or use the Android device as a replacement for the emulator during development.

To actually unlock the Nexus One, a program called `fastboot` is used. This program lets the Nexus One device boot into the bootloader. Once the Nexus One has arrived in the bootloader, the unlock command can be entered. By doing so, the device is "rooted". It is also required to install `ConnectBot` on the Nexus One. ConnectBot is a shell on the device itself, this shell enables the user to type commands on the command line. This is needed to enter commandos later, when the "usb-host" kernel is loaded. Building of this kernel is explained in Section 7.1.4. This kernel disables the possibility to connect to the Nexus One with a USB-cable, hence the shell can not be entered via the Android Debug Bridge (ADB). This is the reason that an additional application, which provides a shell and is placed on the Nexus One, is needed

7.1.3 Add `libusb`, `libnfc` and `libfreefare`

The most difficult part is porting the usb, nfc and freefare libraries. The three different libraries depend on each other. Libusb is a library which allows userspace programs to communicate with USB-devices. The NFC-library communicates with NFC-readers by invoking the USB-library. This library uses the API provided by the USB-library and provides an API to communicate with tags connected to the reader and the reader itself. The freefare library is an abstraction layer on top of the NFC-library.

After downloading these libraries, all unnecessary files can be removed, all files which a nog a C source or header file. Those files are no longer needed within the Android framework. By porting libraries to Android, it is enough, to write an `Android.mk` file for each library. Android.mk can be compared with the `make` program. It is a tool to build efficient large software projects.

Besides the `Android.mk` files, modifications also needs to be made to the existing source files. Some macro's which are assumed to be in the included header files which do

not exist in the Android environment. For example: in Ubuntu 10.04 the `bf MIN()` and `MAX()` macro's are defined in `sys/param.h`. This is not the case in Android. Thus, those libraries rely on macro's which are not there. To tackle this problem, the macro's can be added in an existing `.h` file which is included in every `.c` file which needs it.

The last step is adding this the compiled library to the existing framework. It is not required to clean and build the whole framework as a whole. It is enough to use the `mmm` command. This command can be used to compile one project (e.g. the libraries), instead of the whole framework, which is the case by the `make` command.

7.1.4 Build the kernel

The last step in this process is to compile the kernel. For this project, it is not possible to use the "standard" kernel which comes with the Android framework. To make the Nexus One able to detect and operate on external USB devices, the kernel should operate as a USB-host. A developer named: Sven Killig [4] adjusted the Android kernel in order to do so and released the source code on his website. The source code can be checked-out from the git repository, in order to compile it.

It is necessary to copy the kernel modules which we need from the website. The "old" ones, the ones which are compiled together with the framework do not work any more, since they are kernel specific. Those modules can be pushed to the storage card from the device.

7.1.5 Testing the libraries

When the kernel is compiled, the new kernel image should replace the kernel image. This image is compiled in the Android framework as part of the boot image. This boot image can be unpacked. The kernel image, which is part of this boot image, can then be removed. The new compiled kernel image can be copied to the boot image. As a final step, the boot image needs to be repacked.

All images are putted into place. To load these images on the device, the Nexus One needs to be booted into the boot loader. Second, the system image from the framework is flashed onto this device. The system image contains the libraries that are ported to android. It is important to note that no power loss of the Nexus One may occur during this process, otherwise it will make the mobile phone unusable for further use. The third step is to boot the new kernel.

Once the system is booted, all modules can be loaded. First, the `usbcore` module is loaded. This module supports the USB host controller and devices. Second, the `ehci-hcd.ko` module is loaded to enable high speed communication to USB 2.0 devices. Then, the USB filesystem has to be mounted. Thereafter, the `usbserial.ko` and `cp210x.ko` modules are loaded. The `usbserial.ko` module is able to communicate

to a USB devices as it is a serial device. The `cp210x.ko` module enable a virtual `com` port.

7.2 Using the NFC library

Simply porting the NFC library to Android is not enough in order to use the functionality of this library in applications. There are two main problems that need to be tackled: First, libnfc is not able to operate in user mode, because it invokes restricted API's that are only accessible by the kernel. Therefore, a so-called "native service" has to be implemented. Second, porting only the libnfc to Android is not sufficient in order to use the functionality in application. Instead, a JNI (Java Native Interface) wrapper has to be written. This wrapper "translates" Java method calls into C or C++ method calls. This section provides an overview of all necessary actions that need to be taken in order to let Android work with NFC from the application layer. In other words, it shows how to extend the Android framework with libnfc in such a way that an application can use libnfc. In Section 7.2.1, an overview is given of the particular functions that actually have to be wrapped, because not all functions that are provided by libnfc require a change. In Section 7.2.2, an overview is given of the way a specific call from the application level can be executed in libnfc, using one library function as an example.

7.2.1 LibNFC API overview

The NFC library the offers functionality of letting two devices communicate with each other. In order to do so, one devices has the role of the "initiator", and the other functions as a "target". The initiator sends a request to the target, the target then answers to that request by sending the result back to the initiator. This approach resembles client-server architecture. Where the server processes requests from the client.

In the present project, the Android device takes the role of the initiator. Therefore, the Android platform only needs to contain initiator-related functions, simply because it does not use any target-related functions. However, the target functions will be used by the application that runs on the computer. Thus, these functions need to be wrapped as well for use on the computer only. The implementation on the computer lacks the initiator functions. For the sake of completeness, this section also discusses the functions that are implemented on the computer as a target. Generally, the initiator and target differ by the order of functions which are invoked during a process. Figure 19 provides an overview of this order of functions. Figure 19a shows an overview of the initiator, and 19b gives an overview of the target.

The `nfcInit` function initializes variables and should therefore always be invoked first. A typical task of this function is to allocate memory to store structures, which store properties of the connected NFC readers. Typical examples of this properties are the name of the device or the baud rate. However, regarding this point, only the pointer

(a) Initiator (b) Target

Figure 19: NFC primitives for peer to peer communication

to this allocated memory is returned. The allocated memory is filled-in later. The programmer can give a parameter to this function. This parameter defines the maximum number of NFC-devices that can be detected. The function returns a pointer to this memory location. The next function is the `nfcListDevices`, this function actively probes for connected NFC-devices, which are supported by libnfc. The properties coming from the detected NFC-devices are stored in the memory allocated by `nfcInit`. The `nfcListDevices` function returns the number of connected NFC-devices found. Then, the by using the `nfcConnect` function, the library can connect to a device. Therefore, the programmer needs to provide an id to select a device. This id is a number between 0 and the returned value of `nfcListDevices` minus one, including both values. From there on, the initiator and target invoke different functions. The initiator calls `nfcInitiatorInit`, which initializes the NFC-device as an initiator. The target is initialized as an emulated tag by invoking `nfcTargetInit`. The initiator needs to select a target and determine whether this target is active or passive by invoking `nfcInitiatorSelectDepTarget`. Then, the initiator is ready to send and receive

data by invoking `nfcInitiatorTransceiveBytes`. This function takes a byte array as data, which has to be sent to the target. It also receives a byte array from the target. The target has two separate functions in order to receive and send data. It receives data with `nfcTargetReceiveBytes`. Following, the target can process this data and send the result back using `nfcTargetSendBytes`. Then, both parties can close NFC, if they wish to do so, by invoking `nfcDisconnect`. This call removes the allocated memory, deselects -, and disconnects from the device.

7.2.2 Implementing the native service

In the Android framework, a new native service needs to be added. This native service executes the libnfc code in root mode, on behalf of the user. A typical native service consist of three main components, namely a server, a client to the server, and the services which the native service offers. In our case, function calls to libnfc. The service has to run in root mode. To do so, the permission of this server has to change accordingly. This action can be taken in the `android_filesystem_config.h` file. A record { `06750, AID_ROOT, AID_SHELL, ''system/lib/libnfc_server''`} has to be added.

The libnfc_server is connected to the Binder. This Binder is an IPC driver (Inter Process Communication) in Android. The Binder marshals data, using the parcel objects. These marshalled objects are sent by means of transactions. A Binder communicates in a synchronous way. The Binder transactions are between the server and client.

Java Application The `nfcVersion()` function is provided in the `NfcWrapper` class. This wrapper class provides an interface for the functions implemented in the JNI wrapper. The `NfcWrapper` class is implemented as a `Singelton` object, to be sure that no two threads can instantiate two different objects. To obtain the object, the code has to call the `NfcWrapper.getSingletonObject` function, as depicted in Listing 1. This function takes the type of the tag, baud rate, and operation mode as parameters.

```
1  private NfcWrapper nfcWrapper
2  ...
3
4  public class Payment extends Activity implements Runnable {
5
6      @Override
7      public void onCreate(Bundle savedInstanceState) {
8              ...
9              nfcWrapper = NfcWrapper.getSingletonObject (
10                     TagType.ISO14443A,
11                     BaudRate.NBR_424,
12                     OperationMode.PASSIVE);
13
14             Log.i("NFC-library version number = " + nfcWrapper.getVersion());
15     }
16 }
```

Listing 1: Instantiate the nfcWrapper.class

When the `NfcWrapper` object is successfully required, `getNfcVersion()` can be invoked. This is shown in line 16. However, before the call can be used, the `NfcWrapper` class needs to load the library with all the JNI functions first; this is shown in Listing 2 in line 3-14. In line 16, the `nfcVersion()` is declared a native function.

```
1  public class NfcWrapper {
2
3      static {
4          try {
5              String path = "/system/lib/libnfc_jni.so";
6              System.load(path);
7              Log.i("\"" + path + "\" Successfully loaded");
8          } catch (UnsatisfiedLinkError e) {
9              Log.e("Native code library failed to load.\n" + e);
10             System.exit(1);
11         } catch(Exception e) {
12             e.printStackTrace();
13         }
14     }
15
16     public native String nfcVersion();
17     ...
18 }
```

Listing 2: Loading the library in nfcWrapper.class

JNI Wrapper JNI is a powerful tool to integrate a Java application with existing code that has been, for example, written in C/C++. The author of [38] provides an extensive description of this procedure and a practical guide to implementing JNI. For the `nfcVersion()` example, the JNI wrapper header file has to be generated first. This can be done using the command: `javah -jni NfcWrapper.class`. This generated header file has to be copied to the jni project directory. This file has prototypes of all native function as declared in the `NfcWrapper.class`.

```
1 JNIEXPORT jstring
2 JNICALL Java_android_mipass_jni_nfclib_NfcWrapper_nfcVersion (
3         JNIEnv *env,
4         jobject obj) {
5
6         android::Nfc nfc_service;
7
8         return env->NewStringUTF (nfc_service.nfcVersion());
9 }
```

Listing 3: The JNI implementation of nfcVersion()

In Listing 3, first, in line 6, an example is provided from the NFC class. In line 8, the C string is transformed to a jstring, and returned to the Java environment.

onTransaction In Listing 4, the actual transaction to the server is created. Since no data is sent to `nfcVersion` as a parameter, no values are loaded into `data`. The `data` and `reply` are of the type `Parcel`. The `Parcel` class marshals the values before they can be sent to the nfc service. The transaction is done in line 10. The `NFC_VERSION` is send as a flag. This is necessary in order to perform the appropriate action in the nfc service. In line 13, the resulting string is unmarshalled. This string is placed there in the nfc service.

```
1  #define NFC_VERSION     0
2  ...
3
4  namespace android {
5
6      const char * Nfc::nfcVersion() {
7
8          getNfcService ();
9          const char *nfc_version;
10         Parcel reply, data;
11
12         binder->transact (NFC_VERSION, data, &reply);
13         nfc_version  = reply.readCString();
14
15         return nfc_version;
16     }
17 }
```

Listing 4: Invoking the onTransaction method call

Libnfc service The actual work is done in the nfc service. In Listing 5, the first thing that has to be done is to set the user permission to "root" mode. This allows this service to execute the libnfc code as a root. Therefore, the `setuid()` function is invoked. Then, the case is selected for `NFC_VERSION`, which invokes the `libVersion()` function. In line 29, the actual libnfc call is made. The resulting string is placed into the `reply` parcel object, which can then be read by the caller.

```
1  #define ROOT    0
2  ...
3
4  namespace android {
5
6      status_t NfcService::onTransact (
7          uint32_t nfc_transaction_code,
8          const Parcel & data,
9          Parcel * reply,
10         uint32_t flags) {
11
12         uid_t uid_on_entrance = getuid();
13
14         setuid(ROOT);
15
16         switch (nfc_transaction_code) {
17         ...
18             case NFC_VERSION:
```

```
19                  libVersion(data, reply);
20                  break;
21          ...
22          }
23
24          setuid(uid_on_entrance);
25
26      return NO_ERROR;
27      }
28
29      void NfcService::libVersion(const Parcel & data, Parcel * reply) {
30          const char * nfc_lib_version = nfc_version();
31
32          reply->writeCString(nfc_lib_version);
33      }
34  }
```

Listing 5: Using the libnfc service

Run the NFC infrastructure In Figure 20, an overview is given about the whole process to use NFC on the device. Most steps are already discussed in Section 18. However, it is only possible to use libnfc in Android by also applying the steps taken in this chapter. Therefore, in addition to the steps which are discussed in 18, the `libnfc_server` also needs to start after boottime. Figure 20 also assumes that the system image, as well as the boot image, have to be written to the device.

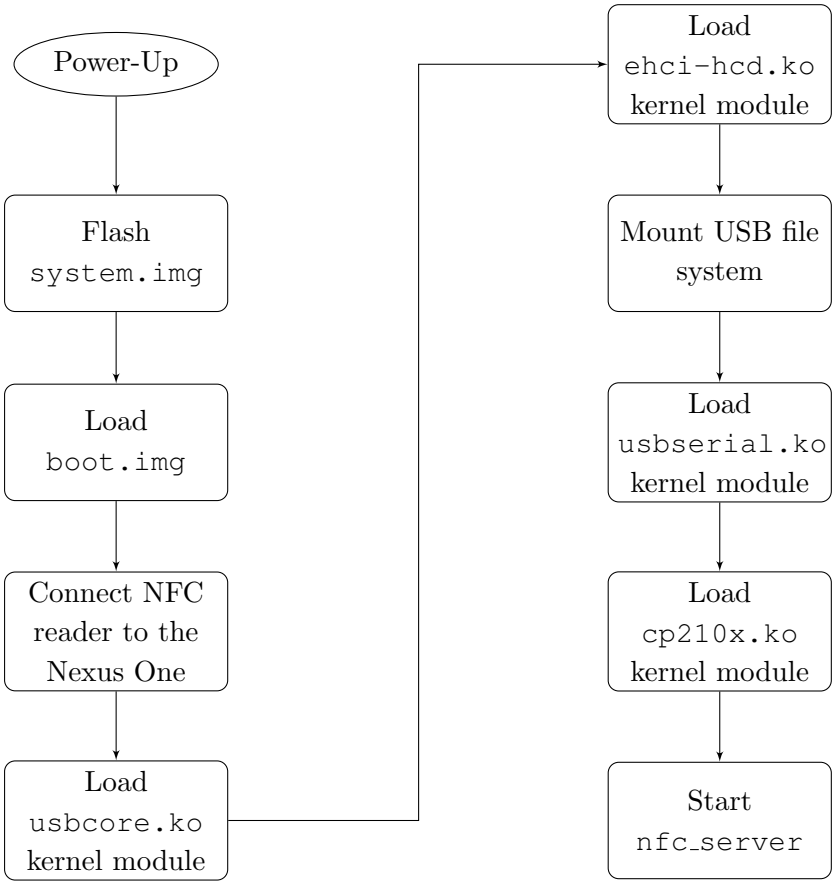


Figure 20: Steps to use NFC on the Nexus One

7.3 Broker and Parking meter

Both the broker and parking meter implementation use the Java project for NFC support. They both have to operate as a target and both entities are executed on a GNU/Linux computer. The implementation for NFC in these two projects is far less complicated than the NFC support on Android. What had to be done to make the two projects work was to write a JNI wrapper that is linked with libnfc, and to write a Java class with native function calls. In this section, `nfcGetPointerDeviceList()` and `nfcConnect()` are used as an example. The full source code is available in Appendix B.

7.3.1 Java native class

In this section, a part of the Java class is discussed. In Listing 6, the `SingeltonNfcWrapper` class is displayed.

```
1  public class SingletonNfcWrapper {
2
3      /* Integers to store pointer values from native code */
4      private int pointerDeviceList = 0;
5      private int pointerCurrentDevice = 0;
6      ...
7
8      /* Declaration of native functins */
9      private native int nfcGetPointerDeviceList (int maxDeviceCount);
10     private native int nfcConnect (int deviceIndex, int pointerDeviceList);
11     ...
12
13     /* Public available functions */
14     public boolean nfcGetPointerDeviceList (int maxDeviceCount) {
15         pointerDeviceList = nfcGetPointerDeviceList(maxDeviceCount);
16         return (pointerDeviceList > 0);
17     }
18
19     public boolean nfcConnect(int deviceIndex) {
20         pointerCurrentDevice = nfcConnect (deviceIndex, pointerDeviceList);
21         return (pointerCurrentDevice > 0);
22     }
23     ...
24 }
```

Listing 6: Java class with native functions

The private integers `pointerDeviceList` and `pointerCurrentDevice` are used to store the pointers that are returned from `nfcGetPointerDeviceList()` and `nfcConnect()` respectively. These pointers can later be returned to the native code. The returned pointer

from `pointerDeviceList` is used as an input for `nfcConnect()`. In other words, the administration of the pointers, obtained in native code, is maintained in Java code. The private functions are prototypes of the native functions. These functions are private because the pointer administration needs to be kept away from the developer.

7.3.2 Java Native Interface

In the JNI wrapper code snippet, as shown in Listing 7, the JNI implementation of both `nfcGetPointerDeviceList()` and `nfcConnect()` is shown.

The `nfcGetPointerDeviceList()` function allocates a chunk of memory to store the device list. With the value `max_devices_count`, this value indicates the maximum number of devices that can be stored in this table. Also, this value determines the total size of the allocated memory. The pointer to this allocated memory is returned back to Java.

```
1  #include "libnfc_jni_wrapper.h"
2
3  JNIEXPORT jint
4  JNICALL Java_mipass_nfc_SingletonNfcWrapper_nfcGetPointerDeviceList(
5          JNIEnv * env,
6          jobject obj,
7          jint max_devices_count) {
8
9      nfc_device_desc_t *device_descriptor = 0;
10
11     /* Malloc the device list */
12     if (!(device_descriptor = malloc (
13                     max_devices_count * sizeof (*device_descriptor)))) {
14
15         _DEBUG("Malloc failed %d\n", __LINE__);
16         //TODO: Exception Handling
17         return -1;
18     }
19
20     /* Return the pointer to the device list, nfc_connect needs this */
21     return (jint) device_descriptor;
22 }
23
24 JNIEXPORT jint
25 JNICALL Java_mipass_nfc_SingletonNfcWrapper_nfcConnect(
26     JNIEnv *env,
27     jobject obj,
28     jint device_index,
29     jint ptr_device_list) {
30
31     static nfc_device_t *current_device;
32
```

```
33      /* Point to the device list */
34      nfc_device_desc_t *device_descriptor =
35                          (nfc_device_desc_t *) ptr_device_list;
36
37      /* Execute the nfc_connect function from libnfc */
38      current_device = nfc_connect (
39                          &(device_descriptor[device_index]));
40
41      /* Successfull? */
42      if (current_device == NULL) {
43          //Error, return -1, handle it in java.
44          _DEBUG ("Unable to connect to NFC device.");
45          return -1;
46      }
47
48      /* Print the name of the device, where libnfc has connected to */
49          _DEBUG("Device Name = %s\n", current_device->acName);
50
51      /* Return pointer to the device, nfclib has connected to */
52      return (jint) current_device;
53  }
```

Listing 7: JNI implementation of nfcGetPointerDeviceList() and nfcConnect()

The `nfcConnect()` uses an `device\_index` and the pointer, which is returned by `nfcGetPointerDeviceList()`, `ptr_device_list` as an input. This pointer indicates the list, and `device_index` points to the slot that has to be used. Then, the `nfc_connect()` is invoked, and returns a pointer to the slot of that device in the device list. This pointer is then returned to Java.

8 Evaluation

This section evaluates the performance of our proposed scheme. The central question it answers is, how much can a malicious user benefit from this system? In other words, if a malicious user modifies a mobile device to make double spending possible, how much damage can this user then do? This section first states all assumptions and used symbols for the evaluation, in Section 8.1. Second, it elucidates the scenario of a single malicious user, in Section 8.2.Lastly, it shows an analysis of a second scenario, where a certain fraction of all users is malicious, in Section 8.3. Finally, several solutions are introduced that have the potential to limit fraud.

8.1 Assumptions and Symbols

The scenarios pointed out in this Section use specific assumptions, these assumptions are listed in the following:

- This scheme is deployed in a city with 1 million users;
- The parking fee is €2.50 per hour;
- One percent of all users in the system is malicious (10.000 users);
- On an average day, 40 percent of the users use the system;
- Average travel time between two parking meters is ten minutes;
- Average parking time is half an hour;
- All parking meters synchronize every day with the broker;
- $mins$ is time expressed in minutes, hr is time expressed in hours;
- Parking is free between 00:00hr and 06:00hr, in this time the parking meters a synchronised with the broker.

8.2 Analyse of one malicious user

When a user upgrades his or her balance with the broker, and consequently stores the received $E_{K_U^{pub}}\left\{\{\overline{\omega_0^j}\}_{sig_B}\right\}, j, l$, the user is able to replay this message at different parking meters. How much damage would this cause? First, a user cannot replay this message at the same parking meter, so the user needs to move to another parking meter to replay the same message. The first message sent to the parking meter will be accounted. After that, a replayed message is marked as "fraudulent", and can therefore not be accounted. This is not valid, if the user paid more deposit on the replayed message. In such cases, the previous message is marked as fraudulent. Thus, the transaction that provides the highest deposit

to the meter is accounted, the replayed messages with a lower deposits are consequently fraudulent. Because this means that the optimal fraud is committed when repeating equal parking lengths, all scenarios described in this chapter work with such a scenario. Thus, in the following we will investigate two scenarios. First, we investigate what happens if a malicious user performs many short parking movements during one single day. Second, we explain what happens if a malicious user performs two long parking movements during the day. Following the analysis, the findings are compared.

Many short parking movements

If a malicious user performs many short parking actions, and replays the same message over and over again, the losses for the system are relatively high. If a user drives through the city from 06:00 a.m. to 00:00 a.m., and parks on average 15 minutes with 10 minutes necessary to travel from meter to meter, then this user can maximally park $(24hr - 6hr) * (15mins + 10mins) = 43$ times that day for 15 minutes. However, because the user has to pay for the first visit to a parking meter, 43 can be subtracted by one, which makes 42. This means that there are 42 fraudulent transactions that day performed by this user. The effective time a user parks that day is $42 * 1/4$ =10:45hr and a maximum of (10:30hr*€2.50) =€24.38 is taken away from the system.

Two long payments

Another scenario describes a user that parks a car two times on the same day, and replays the first message the second time he or she parks. If the first deposit cover the first half of the day minus $5mins$, the user can replay this message for parking the second half of the day also, at a different parking meter. This means, that the user has paid enough deposit to cover the rest of the day, minus $10mins$ travel time. In this case, the fraud costs the system (24:00hr−06:00hr − $10mins$)/2*€2.50 =€22.25.

Conclusion

If we compare the two scenarios, the first scenario displays the largest losses for the system. This means that it is more profitable for a malicious user to perform short parking actions, drive through the city the whole day, and therefore move from meter to meter. However, a real-life scenario for a person who drives through the city the whole day, and parks his car every 15 minutes is hard to imagine. Therefore, it is more realistic that a person will use the second scenario. A possible scenario would be that a user starts the day at his or her office, pays for the first half of the day, and then moves the car to another parking meter during lunch-break. There, the user replays the same message to the new parking meter, and essentially parks free of charge the rest of the day. Therefore, for the remaining part of this Section, the second scenario is assumed.

8.3 Analyse of a fraction of malicious users

A system itself does not fail if only one user performs a fraud, the loss for the system in such a case is rather small. Therefore, in this Section, the assumption is used that 1 percent of all users is fraudulent. This means that, in a population of 1 million users, 10.000 users are malicious. However, only 40 percent of all people make use of the system during an average day. These 40 percent is randomly divided into benevolent and malicious users. This means that there are only 4000 malicious user on a single day in our scenario. This implies that the total loss of the system, when fraudulent transaction takes place is: $4000 * €22.25 = €90.000$.

8.4 Solutions

A possible solution to limit the loss for the system is to divide the group of users into different categories:

- Bronze;
- Silver;
- Gold.

The purpose of these categories is to restrict the maximally allowed balance a user can have. If a user is new to the system, the user obtains the "Bronze" status, meaning that he can only charge his or her balance to, say €10.00. This means that this user can damage the system up to a maximum of €10.00. If a user has proven to be honest, the user will enter the "Silver" status, meaning that the user can have a maximum of, say €15.00. The "Gold" status could mean that a user can have a maximum balance of say, €100.00.

This solution results in extra administration for the system and also functions as a barrier or limitation for honest users that are simply new to the system. Therefore, we assume that a better solution involves connecting all parking meters that are in close proximity. For example, this could be useful for parking spots aside a long road that has multiple meters on it. If all meters on the same road are connected and synchronise every now and then, say once every hour with the values received in that hour, it might be harder to perform fraud in the system.

An even simpler solution is to require the payment of a deposit for a new user that wants to enter the system. This means that if a user can have a maximum benefit of €22.50 by doing fraudulent actions, the user needs to pay this amount to the broker even before the user can actually use the system. In this case, if a user is caught in fraudulent action, the broker can reject the user from the system and keep the deposit. In any case, no harm is done.

9 Conclusion

This thesis proposes a new payment scheme called MiPass, designed for use in purchasing parking spots aside of the road. The thesis begins by presenting an extensive summary of NFC-related RFID standards, an overview of available NFC mobile devices, and an implementation of `nfclib` into Android. Following, we have a comprehensive overview of existing payment schemes, which leads us to propose the design of the MiPass protocol. MiPass is the first payment scheme using NFC, designed specifically for Android mobile phones. It protects the user from overspending on parking by working with a refundable deposit. Also, it shields the system from malicious coin invention by storing a proof of the user balance as a hash value. MiPass as a payment scheme also has the potential to be extended into other areas of payment (e.g., for small articles such as convenience food or snacks). One existing flaw of the system is that it does not prevent malicious users from double spending. However, this shortcoming could be easily prevented by introducing a pre-paid deposit to the broker, or by dividing users into different stages of clearance, where only trusted users are allowed to obtain a high monetary balance on their device. Further research could also extend this protocol to make peer-to-peer payments possible. This would mean that a user can transfer money to a friend by using his or her device. Another future improvement would also be to to make the protocol atomic, meaning that if a user removes his phone from the parking meter's NFC range, before the transaction is completed, the whole performed part of that transaction should be rolled back.

References

[1] Arygon. http://www.Arygon.com. *Visited on:* 24 November 2010.

[2] Cell-Idea – microSD . http://www.cell-idea.com/NFC\%20Micro\%20SD.htm. *Visited on:* 24 November 2010.

[3] Cell-Idea – With Me. http://www.nearfieldcommunicationsworld.com/2009/03/19/3865/cell-idea-adds-nfc-to-any-bluetooth-phone/. *Visited on:* 24 November 2010.

[4] Connect USB devices to your Nexus One. http://sven.killig.de/android/N1/2.2/usb_host/. *Visited on:* 12 Februari 2011.

[5] Gartner market research. http://techcrunch.com/2010/11/10/gartner-android-share-jumps-to-25-5-percent-now-second-most/-popular-os-worldwide/. *Visited on:* 7 January 2010.

[6] iCarte. http://www.icarte.ca/. *Visited on:* 24 November 2010.

[7] NFC Forum. http://www.nfc-forum.org. *Visited on:* 06 June 2011.

[8] NFC Phones Can Read ISO 15693 Tags. http://www.rfidjournal.com/blog/entry/8182/. *Visited on:* 12 Februari 2011.

[9] Operating principles of RFID systems. http://www.rfid-handbook.de/rfid/types_of_rfid.html. *Visited on:* 05 December 2010.

[10] SDiD – microSD. http://www.wdi.ca/docs/SW06-0007-DS%20-%20SDiD%201010.pdf. *Visited on:* 24 November 2010.

[11] Silone Learning Center. http://www.silone.com/learning/index.htm#ISO15693. *Visited on:* 06 December 2010.

[12] Twinlinx – NFC-sticker. http://www.twinlinx.com/. *Visited on:* 24 November 2010.

[13] Tyfone – microSD. http://www.tyfone.com/index.html. *Visited on:* 24 November 2010.

[14] What is Android? http://developer.android.com/guide/basics/what-is-android.html. *Visited on:* 05 December 2010.

[15] A practical anonymous payment scheme for electronic commerce. *Computers & Mathematics with Applications*, 46(12):1787 – 1798, 2003.

[16] Rohit Patali Abhijat Agarwal, Meshal Almashan. Mobile Wallet Using FeliCa. 03 2010.

[17] Erik-Oliver Blass, Anil Kurmus, Refik Molva, and Thorsten Strufe. PSP: Private and Secure Payment with RFID. In *Proceedings of the 8th ACM workshop on Privacy in the electronic society*, pages 51–60, Chicago, Illinois, USA, 2009. ACM.

[18] U. Biader Ceipidor, R. Gomes, G. Malfará, C. M. Medaglia, A. Moroni, S. Montrucchio, F. Prato, and A. Vilmos. Security in nfc system: Strengths and weaknesses. 02 2007.

[19] David. Chaum. Blind signatures for untraceable payments. page 199203, Plenum, NY, USA, 1983. Springer-Verlag.

[20] Mike Clark. A definitive list of NFC phones, 12 2010. `http://www.nearfieldcommunicationsworld.com/nfc-phones-list/`.

[21] Ziba Eslami and Mehdi Talebi. A new untraceable off-line electronic cash system. *Electronic Commerce Research and Applications*, 10(1):59 – 66, 2011.

[22] European Computer Manufacturers Association, Geneva, Switzerland. *NFC-SEC Cryptography Standard using ECDH and AES*, 04 2004.

[23] Ernst Haselsteiner and Klemens Breitfub. Security in Near Field Communication: Strengths and Weaknesses. 04 2006.

[24] Mona Hosseinkhani, Ebrahim Tarameshloo, and Mehdi Shajari. Amvpayword: Secure and efficient anonymous payword-based micropayment scheme. *Computational Intelligence and Security, International Conference on*, 0:551–555, 2010.

[25] Innovision PLC Research & Technology PLC / NFC-forum, Gloucestershire, United Kingdom. *Near Field Communication in the real world*, 09 2007.

[26] International Standard Organisation / International Electrotechnical Commission, Geneva, Switzerland. *ISO/IEC 14443, part 1*, 06 1999.

[27] International Standard Organisation / International Electrotechnical Commission, Geneva, Switzerland. *ISO/IEC 14443, part 2*, 06 1999.

[28] International Standard Organisation / International Electrotechnical Commission, Geneva, Switzerland. *ISO/IEC 14443, part 3*, 06 1999.

[29] International Standard Organisation / International Electrotechnical Commission, Geneva, Switzerland. *ISO/IEC 14443, part 4*, 06 1999.

[30] International Standard Organisation / International Electrotechnical Commission, Geneva, Switzerland. *Near Field Communication Interface and Protocol - 2 (NFCIP-2)*, 01 2005.

[31] International Standard Organisation / International Electrotechnical Commission, Geneva, Switzerland. *ISO/IEC 15693, part 2*, 03 2006.

[32] International Standard Organisation / International Electrotechnical Commission, Geneva, Switzerland. *ISO/IEC 15693, part 3*, 09 2007.

[33] International Standard Organisation / International Electrotechnical Commission, Geneva, Switzerland. *ISO/IEC 15693, part 1*, 06 2009.

[34] International Standard Organisation / International Electrotechnical Commission, Geneva, Switzerland. *Near Field Communication Interface and Protocol - 1 (NFCIP-1)*, 06 2010.

[35] Stanisaw Jarecki and Andrew Odlyzko. An efficient micropayment system based on probabilistic polling. In Rafael Hirschfeld, editor, *Financial Cryptography*, volume 1318 of *Lecture Notes in Computer Science*, pages 173–191. Springer Berlin / Heidelberg, 1997.

[36] S. Kim and W. Lee. A micro-payment system for multiple-shopping. SCIC '02, pages 229–234, Shirahama, Japan, 2002.

[37] Jeremy Landt. The history of RFID. *IEEE Potentials*, 24(4):8–11, 2005.

[38] Sheng Liang. *The Java Native Interface*. Addison Wesley, Reading, MA, USA, 1st edition, 1999.

[39] Reto Meier. *Android Application Development*. Wiley Publishing, Indianapolis, IN, USA, 1st edition, 2009.

[40] Khanh Quoc Nguyen, Yi Mu, and Vijay Varadharajan. Digital coins based on hash chain, 2008.

[41] NXP. *MIFARE DESFire contactless multi-application IC*, 12 2007.

[42] Annika Paus. Near Field Communication in Cell Phones. Master's thesis, Ruhr-Universit, Bochum, Germany, 7 2007.

[43] Codruţa Poenar. A Study Looking the Electronic Payment Market. *Informatica Economică*, 1(45):120–123, 2008.

[44] Shre Harsha Rao. *Implementation of the ISO15693 Protocol in the TI TRF796x*. Texas Instruments, Dallas, Texas, 4 2009.

[45] Ronald Rivest and Adi Shamir. Payword and micromint: Two simple micropayment schemes. In Mark Lomas, editor, *Security Protocols*, volume 1189 of *Lecture Notes in Computer Science*, pages 69–87. Springer Berlin / Heidelberg, 1997.

[46] Enrico Rukzio, Ulrich Dietz, Paul Holleis, Albrecht Schmidt, Oliver Falke, Enrico Rukzio, Ulrich Dietz, Paul Holleis, and Albrecht Schmidt. Communications 1 mobile services for near field communication. 2007.

[47] Hong Wang, Jialin Ma, and Jing Sun. Micro-payment protocol based on multiple hash chains. *Electronic Commerce and Security, International Symposium*, 1:71–74, 2009.

[48] Yacov Yacobi. Risk management for e-cash systems with partial real-time audit. *NETNOMICS*, 3:119–127, 2001.

[49] Sung-Ming Yen and Yuliang Zheng. Weighted one-way hash chain and its applications. In *Proceedings of the Third International Workshop on Information Security*, ISW '00, pages 135–148, London, UK, 2000. Springer-Verlag.

[50] Yang Zongkai, Lang Weimin, and Tan Yunmeng. A new fair micropayment system based on hash chain. *e-Technology, e-Commerce, and e-Services, IEEE International Conference on*, 0:139–145, 2004.

A Getting the environment to work

This Appendix is a step-by-step guide to prepare a Nexus One phone to operate with an external NFC-reader. This guide is a step-by-step guide and an extension of Section 18. To begin with, one must be in of the following items:

- Nexus One mobile phone with Android Froyo 2.2.1 (build FRG83D);
- Ubuntu 10.04 as an Operating System on the host computer;
- External Arygon ADRB NFC-reader;
- USB dual cable type A;
- USB Female-to-Female connector type A;
- USB type A to micro-USB type B cable.

Preparing the host computer

This section sets up the necessary utilities on the computer in order to build the Android framework and kernel.

Preparing Ubuntu 10.04

Install the necessary packages on the host platform, using the following command:

```
sudo apt-get install git-core gnupg flex bison gperf zip curl
gcc-multilib g++-multilib build-essential g++ zlib1g-dev
libx11-dev x11proto-core-dev
```

Install Java SDK

Install Java Development Kit version 5.

1. Open the `sources.list` file:

   ```
   sudo gedit /etc/apt/sources.list
   ```

2. Add the `Jaunty` repositories to this file:

   ```
   deb http://us.archive.ubuntu.com/ubuntu/ jaunty multiverse
   deb http://us.archive.ubuntu.com/ubuntu/ jaunty-updates multiverse
   ```

3. Update the package list:

   ```
   sudo apt-get update
   ```

4. Install Sun Java 1.5:

   ```
   sudo apt-get install sun-java5-jdk
   ```

5. Check the default version of Java:

   ```
   java -version
   ```

6. If the output of `java -version` is not `Java version "1.5.0_*`, set Java 1.5.0_* to default using the following command:

   ```
   sudo update-java-alternatives -s java-1.5.0-sun
   ```

Installing Android SDK

In this chapter, the Android SDK will be installed:

1. Download the SDK from `http://developer.android.com/sdk/index.html`.
2. Pick the Linux version and unzip it in `~/android-sdk-linux_86`.
3. Start the `android` program: `~/android-sdk-linux_86/tools/android`.
4. Go to `Available Packages` in the left panel.
5. Unfold the `Android repository` menu. Select at least `Android SDK Platform-Tools, revision2` and `SDK Platform Android 2.2 API 8`.
6. Click on the `Install Selected` button.

Installing the Android NDK

The NDK cross compiler is installed in this chapter:

1. Download the NDK from `http://developer.android.com/sdk/ndk/index.html`.
2. Pick the Linux version and unzip it in `~/android-ndk-{release}-linux-x86`.

Exporting $PATH

For convenience, it is recommended to add NDK's cross compiler and the the SDK `platform-tools` folder to `$PATH`:

```
export PATH=$PATH:/home/user/android-sdk-{release}-linux_86/tools/:
/home/user/android-ndk-{release}-linux-x86/toolchains/
arm-linux-androideabi-4.4.3/prebuilt/linux-x86/bin/
```

Creating Android Rulefile

Create a rulefile in Ubuntu. Add a rule to this file, which is applicable for HTC mobile phones (The Nexus One is manufactured by HTC):

1. Create a file in `/etc/udev/rules.d` named `50-android.rules`:

   ```
   sudo gedit /etc/udev/rules.d/50-android.rules
   ```

2. Add the following text:

   ```
   SUBSYSTEM=="usb", SYSFS{idVendor}=="0bb4", MODE="0666"
   ```

3. Change file permission:

```
sudo chmod a+rx /etc/udev/rules.d/50-android.rules
```

Downloading the Android Sources

The host computer is now prepared for the project. The next step is to download the Android source code from the repository (which takes +/- 1.5 hours) and compile the source code (which takes another +/- 1.5 hours).

1. Download and install the `repo` client:

```
mkdir ~/bin
curl http://android.git.kernel.org/repo > ~/bin/repo
chmod a+x ~/bin/repo
export PATH=$PATH:~/bin
```

2. Synchronize with the Android source code repository:

```
mkdir myandroid
cd myandroid
repo init -u git://android.git.kernel.org/platform/manifest.git \
-b android-2.2.1_r2
repo sync
```

   NOTE: It may happen that, during this process, a `connection time-out` occurs. Reissue `repo sync`, the process will continue from where it stopped.

3. Everything should be downloaded at this point. However, one modification in the source code is required to gain `su` privileges later on. Therefore, open de `su.c` file and remove the user permission check:

```
gedit ~/myandroid/system/extras/su/su.c
```

4. Comment line 62 - 65 with the following content:

```
//    if (myuid != AID_ROOT && myuid != AID_SHELL) {
//        fprintf(stderr,"su: uid %d not allowed to su\n", myuid);
//        return 1;
//    }
```

5. Enter the flowing commands to compile the Android Framework:

```
cd ~/myandroid
. build/envsetup.sh
lunch full_passion-eng
make -j4 CROSS_COMPILE=arm-linux-androideabi- ARCH=arm
```

Prepare the Nexus One

To modify the operating systems internals, it is required to obtain root privilege. This section explains how to obtain this privilege. It is important to note that this operation erases all data on the Device. First, the `ConnectBot` program is installed to use a console on the Nexus One.

Install ConnectBot

Install `ConnectBot` on the Nexus One:

1. Enable `USB Debugging` on the device:

   ```
   Applications menu -> Settings -> Applications -> Development
   ```

2. Download the `ConnectBot` application, go to: `https://code.google.com/p/connectbot/`, download the `.apk` file, save this file in `~/Download`.

3. Install `ConnectBot.apk` on the Nexus One:

   ```
   adb install ~/Download/ConnectBot.apk
   ```

Unlock the Nexus One

1. Lookup the build number. This number can be found on the Android device:

   ```
   Applications menu -> Settings -> About Phone -> Build Number
   ```

2. Download the appropriate fastboot program. Select this the appropriate version by its build number, and place it in a directory (e.g. `~/fastboot/`).

   `http://android.modaco.com/content/google-nexus-one-nexusone-modaco-com/298782/23-nov-superboot-erd79-frg83d-rooting-the-nexus-one/`

3. Reboot the Nexus One into the bootloader menu:

   ```
   adb reboot-bootloader
   ```

4. When the Nexus One has been rebooted into the boot loader, the device can be unlocked using the `fastboot` program:

   ```
   ~/fastboot/fastboot-linux oem unlock
   ```

5. The instructions on the screen can be followed.

Add NFC and USB support to Android

This section explains how USB and NFC support is added to Android.

Add USB Library to Android

This section explains how to port the usblib to the Android platform. This part is adapted from `http://android.serverbox.ch/wp-content/uploads/2010/01/android_industrial_automation.pdf`.

1. Download `libusb-0.1.12` from `http://downloads.sourceforge.net/libusb/libusb-0.1.12.tar.gz` and extract it in `~/myandroid/external`.
2. Remove all unnecessary files, i.e. , all files which do not have a `.h`, `.cpp` or `.c` extension.
3. Create `Android.mk` in `./external/libusb-0.1.12`, and add the following code:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SRC_FILES:= \
    error.c \
    linux.c \
    descriptors.c \
    usb.c

LOCAL_C_INCLUDES := external/libusb-0.1.12
LOCAL_PRELINK_MODULE := false
LOCAL_MODULE := libusb
include $(BUILD_SHARED_LIBRARY)
```

4. Add the usb library to the framework:

```
mmm ~/myandroid/external/libusb-0.1.12
```

Add NFC library

This chapter explains how the NFC library is ported to the Android environment.

1. Obtain `libnfc-1.4.1` from `https://code.google.com/p/libnfc/downloads/detail?name=libnfc-1.4.1.tar.gz` and unpack this file in: `~/myandroid/external/libnfc-1.4.1`.

2. As in `libnfc-1.4.1` (recursively) remove all files which do not have a `.c` or `.h` extension.

3. Add Android.mk in `~/myandroid/external/libnfc-0.1.12`:

```
LOCAL_PATH := $(call my-dir)
subdirs := $(addprefix $(LOCAL_PATH)/,$(addsuffix /Android.mk, \
libnfc ))
include $(subdirs)
```

4. Add Android.mk in `~/myandroid/external/libnfc-0.1.12/libnfc`:

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES:= \
   buses/uart.c  \
   chips/pn53x.c \
   drivers/arygon.c \
   nfc.c \
   iso14443-subr.c \
   mirror-subr.c

LOCAL_CFLAGS := -O2 -g -std=c99 -DSERIAL_AUTOPROBE_ENABLED
LOCAL_CFLAGS += -DHAVE_CONFIG_H -DHAVE_LIBUSB -DDRIVER_ARYGON_ENABLED

LOCAL_C_INCLUDES += \
    external/libnfc/ \
    external/libnfc/include/ \
    external/libnfc/libnfc/buses/ \
    external/libnfc/libnfc/chips/

LOCAL_SHARED_LIBRARIES += libusb

LOCAL_MODULE:= libnfc
LOCAL_PRELINK_MODULE:= false

include $(BUILD_SHARED_LIBRARY)
```

5. Add a `MIN()` and `MAX()` macro in `~/myandroid/external/libnfc-0.1.12/include/nfc/nfc.h`

```
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

6. Add "`#include nfc/nfc.h`" to:
   `./libnfc-1.4.1/libnfc/drivers/arygon.c`
   `./libnfc-1.4.1/libnfc/buses/uart_posix.c`
   `./libnfc-1.4.1/libnfc/chips/pn53x.c`

7. Add the NFC library to the framework:

   ```
   mmm ~/myandroid/external/libnfc-1.4.1
   ```

Add nfc-list

To test whether the `libnfc` implementation works, `nfc-list.c` is compiled in the same way as `lsusb`.

1. Create a new directory `nfc-list` in `~/myandroid/external/nfc-list`. Copy the `nfc-list.c`, `nfc-utils.c` and `nfc-utils.h` to this directory:

   ```
   mkdir ~/myandroid/external/nfc-list
   cp ~/myandroid/external/libusb-1.4.1/example/list-nfc.c \
      ~/myandroid/external/nfc-list/
   cp ~/myandroid/external/libusb-1.4.1/example/nfc-utils.* \
      ~/myandroid/external/nfc-list/
   ```

2. Create `Android.mk` in `~/myandroid/external/nfc-list` to compile the source files:

   ```
   LOCAL_PATH := $(call my-dir)
   include $(CLEAR_VARS)
   LOCAL_SRC_FILES     :=  \
       nfc-list.c
       nfc-utils.c

   LOCAL_MODULE        := nfc-list
   LOCAL_C_INCLUDES    += \
       external/libnfc-1.4.1/ \
       external/libnfc-1.4.1/include/
   ```

```
LOCAL_SHARED_LIBRARIES := libc libusb libnfc
include $(BUILD_EXECUTABLE)
```

3. Add `nfc-list` to the framework:

```
mmm ~/myandroid/external/nfc-list
```

Add libfreefare

The libfreefare is needed as an abstraction layer above libnfc. This layer provides features to deal with DESfire and mifare-classic.

1. Download the libfreefare from `https://code.google.com/p/nfc-tools/downloads/detail?name=libfreefare-0.3.0.tar.gz` and unzip it in `~/myandroid/external`.
2. Remove all files which do not end with `.c` or `.h`.
3. Create `Android.mk` in `~/myandroid/external/libfreefare-0.3.0` to compile the source files:

```
LOCAL_PATH  := $(call my-dir)
subdirs  := $(addprefix $(LOCAL_PATH)/,$(addsuffix /Android.mk, \
libfreefare ))
include $(subdirs)
```

4. Create `Android.mk` in `~/myandroid/external/libfreefare-0.3.0/libfreefare` to compile the source files:

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES:=           \
    mifare_application.c    \
    mifare_classic.c        \
    mifare_ultralight.c     \
    mifare_desfire.c        \
    mifare_desfire_key.c    \
    mifare_desfire_crypto.c \
    mifare_desfire_aid.c    \
    mifare_desfire_error.c  \
    freefare.c \
```

```
    freefare.h \
    tlv.c      \
    mad.c

LOCAL_CFLAGS := -O2 -g -std=c99 -DHAVE_LIBNFC

LOCAL_C_INCLUDES += \
    external/libfreefare-0.3.0/ \
    external/libnfc/include     \
    external/openssl/include/

LOCAL_SHARED_LIBRARIES := libnfc libc libssl

LOCAL_MODULE := libfreefare
LOCAL_PRELINK_MODULE := false

include $(BUILD_SHARED_LIBRARY)
```

5. Comment `#define HAVE_ENDIAN_H 1` (line 15) in file `./libfreefare-0.3.0/config.h`.

6. Open `./libfreefare-0.3.0/libfreefare/freefare_internal.h` and comment the `MIN` and `MAX` macros:

```
// #define MIN(a, b) (((a) < (b)) ? (a) : (b))
// #define MAX(a, b) (((a) > (b)) ? (a) : (b))
```

7. Add `libfreefare` to the framework:

```
mmm ~/myandroid/external/libfreefare
```

Android Kernel and Kernel Modules

This section is a guide to obtain and compile the Android kernel source. The standard source code is not suitable in this case. The goal of the new kernel is to let the Android device act like a USB-host, this is not the case by the default source code. However, the German developer Sven Killig (`http://sven.killig.de/android/N1/2.2/usb_host/`) has developed and released a kernel source which lets the Android device work like a USB-host.

Compile Android Kernel

- Obtain the modified Android kernel sources:

```
cd ~/
git clone http://github.com/sonic74/kernel_msm.git
```

- Download the `config.gz` file from the device and unzip it:

```
cd kernel_msm
adb pull /proc/config.gz
gunzip config.gz
mv config .config
```

- Compile kernel sources:

```
make -j4 ARCH=arm CROSS_COMPILE=arm-linux-androideabi-
```

Obtaining Kernel Modules

Several kernel modules need to be downloaded first. These modules load the USB and NFC driver, and are therefore necessary to use the external NFC reader.

- Download these kernel modules from Sven Killig's website:

```
usbcore.ko
ehci-hcd.ko
usbserial.ko
cp210x.ko
```

- Copy these modules to the sdcard, placed inside the device.

```
adb push usbcore.ko /sdcard/usbcore.ko
adb push ehci-hcd.ko /sdcard/ehci-hcd.ko
adb push usbserial.ko /sdcard/usbserial.ko
adb push cp210x.ko /sdcardcp210x.ko
```

Putting it all together

After using this appendix, the following things should now be ready:

- Ubuntu is ready;
- SDK and SDK are installed;
- The Android Framework is compiled;
- The Android kernel is compiled;
- The kernel modules are loaded on the SDcard;
- The `ConnectBot` application is downloaded;
- The Nexus One is unlocked;
- The Fastboot program is available on the system.

The next step is, to load the new compiled kernel and the framework on the Nexus One. The fist step is to place the kernel into the boot image. Second, flash system image to the device, and boot the Nexus One with the new boot image.

1. Make a directory for the boot image. This image needs to be modified:

   ```
   mkdir ~/images
   ```

2. Copy the system and kernel image from `~/myandroid/out/target/out/target/product/passion/` to `~/images`:

   ```
   cp \
   ~/myandroid/out/target/out/target/product/passion/boot.img \
   ~/images/
   cp \
   ~/myandroid/out/target/out/target/product/passion/system.img \
   ~/images/
   ```

3. The `boot.img` needs to be unpacked, in order to replace the kernel with the kernel build in Section A. Therefore, download the unpack and repack script from respectively: `http://android-dls.com/files/apps/unpack-bootimg.zip` and `http://android-dls.com/files/apps/repack-bootimg.zip`. Unzip these scripts and place them in /script. Modify line number 19 from `repack-bootimg.pl`: add `--base0x20000000` right after `mkbootimg`.

```
cd ~/images
~/script/unpack-bootimg.pl boot.img
~/script/repack-bootimg.pl ~/kernel_msm/arch/arm/boot/zImage \
boot.img-ramdisk new-image.img
```

Flashing the Nexus One

4. Restart the nexus one into the bootloader:

   ```
   adb reboot-bootloader
   ```

5. Flash the system partition:

   ```
   ../fastboot/fasboot-linux flash system ~/images/system.img
   ```

6. Load the kernel, do not flash it.

   ```
   ../fastboot/fasboot-linux boot ~/images/new-image.img
   ```

7. This automatically boots the device with the new kernel and new system image.

Last step, make it work

When the Nexus is successfully started, make sure all cables are connected in the proper way.

1. Start `ConnectBot` to load the kernel modules:

   ```
   su
   cd /sdcard
   insmod usbcore.ko
   insmod ehci-hcd.ko
   mount -t usbfs none /proc/bus/usb
   insmod usbserial.ko
   insmod cp210x.ko
   ```

2. To test if everything works, execute `nfc-list`.

B Project source code

JNI wrapper, parking meter and broker

```
1  #include "libnfc_jni_wrapper.h"
2
3  //nfc_device_t *global_current_device;
4
5  JNIEXPORT jstring JNICALL
6  Java_mipass_parkingmeter_nfc_SingletonNfcWrapper_nfcVersion(
7          JNIEnv *env,
8          jobject obj) {
9
10     _DEBUG("nfc_version() = %s\n", nfc_version());
11     return (*env)->NewStringUTF(env, (char *) nfc_version());
12 }
13
14 JNIEXPORT jint JNICALL
15 Java_mipass_nfc_SingletonNfcWrapper_nfcDisconnect
16 (
17         JNIEnv * env,
18         jobject obj,
19         jint ptr_current_device,
20         jint ptr_device_list) {
21
22     nfc_device_t *current_device   = (nfc_device_t *) ptr_current_device;
23     nfc_device_desc_t *device_list = (nfc_device_desc_t *) ptr_device_list;
24
25     nfc_disconnect (current_device);
26
27     free(device_list);
28
29     _DEBUG("Device %s is disconnected\n", current_device->acName );
30
31     return 0;
32 }
33
34 JNIEXPORT jint JNICALL
35 Java_mipass_parkingmeter_nfc_SingletonNfcWrapper_nfcInitiatorListPassiveTargets
36 (
37         JNIEnv * env,
38         jobject obj,
39         jint ptr_current_device,
40         jint tag_type,
41         jint max_target_count,
42         jint number_of_devices_found) {
43
44     int result = 0;
45     nfc_device_t *current_device = (nfc_device_t *) ptr_current_device;
```

```
46      nfc_target_t ant[max_target_count];
47
48      nfc_modulation_t nm = {
49          .nmt = get_modulation(tag_type),
50          .nbr = get_baud_rate(tag_type),
51      };
52
53      result = nfc_initiator_list_passive_targets (
54                                  current_device,
55                                  nm, ant, max_target_count,
56                                  (size_t *) &number_of_devices_found);
57
58      _DEBUG("\"nfc_initiator_list_passive_targets\" returned: %d\n", result);
59
60      return result;
61  }
62
63  JNIEXPORT jint JNICALL
64  Java_mipass_parkingmeter_nfc_SingletonNfcWrapper_nfcInitiatorInit(
65      JNIEnv *env,
66      jobject obj,
67      jint ptr_current_device) {
68
69      int result;
70      nfc_device_t *current_device = (nfc_device_t *) ptr_current_device;
71
72      result = nfc_initiator_init (current_device);
73
74      _DEBUG ("Connected to NFC device: %s\n", current_device->acName);
75
76      return result;
77  }
78
79  JNIEXPORT jint JNICALL Java_mipass_nfc_SingletonNfcWrapper_nfcConnect(
80      JNIEnv *env,
81      jobject obj,
82      jint device_index,
83      jint ptr_device_list) {
84
85      static nfc_device_t *current_device;
86      nfc_device_desc_t *device_descriptor =
87                                          (nfc_device_desc_t *) ptr_device_list;
88
89      current_device = nfc_connect (&(device_descriptor[device_index]));
90
91      if (current_device == NULL) {
92          _DEBUG ("Unable to connect to NFC device.");
93          return -1;
94      }
```

```
95
96      _DEBUG("Device Name = %s\n", current_device->acName);
97
98      return (jint) current_device;
99  }
100
101 JNIEXPORT jint JNICALL
102 Java_mipass_nfc_SingletonNfcWrapper_nfcListDevices(
103         JNIEnv *env,
104         jobject obj,
105         jint max_devices_count,
106         jint ptr_device_descriptor) {
107
108     int device_found = 0;
109
110     nfc_device_desc_t *device_descriptor =
111                                     (nfc_device_desc_t *) ptr_device_descriptor;
112
113     nfc_list_devices (device_descriptor, max_devices_count,
114                                                     (size_t *) &device_found);
115
116     _DEBUG("Number of devices found is: %d\n", device_found);
117
118     return device_found;
119 }
120
121 JNIEXPORT jint JNICALL
122 Java_mipass_parkingmeter_nfc_SingletonNfcWrapper_nfcInitiatorDeselectTarget(
123         JNIEnv * env,
124         jobject obj,
125         jint ptr_current_device) {
126
127     nfc_device_t *current_device = (nfc_device_t *) ptr_current_device;
128
129     nfc_initiator_deselect_target (current_device);
130
131     return 0;
132 }
133
134 JNIEXPORT jint JNICALL
135 Java_mipass_nfc_SingletonNfcWrapper_nfcGetPointerDeviceList(
136         JNIEnv * env,
137         jobject obj,
138         jint max_devices_count) {
139
140     nfc_device_desc_t *device_descriptor = 0;
141
142     if (!(device_descriptor = malloc (max_devices_count *
143                                             sizeof (*device_descriptor)))) {
```

```
144         _DEBUG("Malloc failed %d\n", __LINE__);
145         return -1;
146     }
147 
148     return (jint) device_descriptor;
149 }
150 
151 JNIEXPORT jstring JNICALL
152 Java_mipass_parkingmeter_nfc_SingletonNfcWrapper_nfcDeviceName(
153         JNIEnv * env,
154         jobject obj,
155         jint ptr_current_device) {
156 
157     nfc_device_t *current_device = (nfc_device_t *) ptr_current_device;
158 
159     return (*env)->NewStringUTF(env, (char *) nfc_device_name(current_device));
160 }
161 
162 JNIEXPORT jint JNICALL
163 Java_mipass_parkingmeter_nfc_SingletonNfcWrapper_nfcInitiatorSelectDepTarget(
164         JNIEnv * env,
165         jobject obj,
166         jint ptr_current_device,
167         jint baud_rate,
168         jint operation_mode) {
169 
170     nfc_target_t nt;
171     nfc_device_t *current_device = (nfc_device_t *) ptr_current_device;
172 
173     if (!nfc_initiator_select_dep_target (current_device, operation_mode,
174                                                         baud_rate, NULL, &nt)) {
175 
176         nfc_perror(current_device, "nfc_initiator_select_dep_target");
177         return EXIT_FAILURE;
178     }
179 
180     return 0;
181 }
182 
183 
184 JNIEXPORT jint JNICALL
185 Java_mipass_nfc_SingletonNfcWrapper_nfcTargetSendBytes(
186     JNIEnv * env,
187     jobject obj,
188     jint ptr_current_device,
189     jstring send_message) {
190 
191     nfc_device_t *current_device = (nfc_device_t *) ptr_current_device;
192     size_t send_mesg_length = 0;
```

```
193     jboolean isCopy;
194
195     const char *buffer = (*env)->GetStringUTFChars(env, send_message, &isCopy);
196
197     send_mesg_length = strlen(buffer);
198
199     _DEBUG("Message to send: %s\n", buffer);
200
201     if (!nfc_target_send_bytes (current_device, buffer, send_mesg_length)) {
202
203         _DEBUG("ERROR int function \"nfcTargetSendBytes\", LINE: %d",__LINE__);
204         return -1;
205     }
206
207     (*env)->ReleaseStringUTFChars(env, send_message, buffer);
208
209     return 0;
210 }
211
212 JNIEXPORT jstring JNICALL
213 Java_mipass_nfc_SingletonNfcWrapper_nfcTargetReceiveBytes (
214     JNIEnv * env,
215     jobject obj,
216     jint ptr_current_device) {
217
218     byte_t  abtRx[MAX_FRAME_LEN];
219     size_t szRx;
220     nfc_device_t *current_device = (nfc_device_t *) ptr_current_device;
221
222
223     if (!nfc_target_receive_bytes (current_device, abtRx, &szRx)) {
224         nfc_perror(current_device, "nfc_target_receive_bytes");
225         return (jint) EXIT_FAILURE;
226     }
227
228     abtRx[szRx] = '\0';
229
230     return (*env)->NewStringUTF(env, (char *) abtRx);
231 }
232
233 JNIEXPORT jint JNICALL
234 Java_mipass_nfc_SingletonNfcWrapper_nfcTargetInit(
235     JNIEnv * env,
236     jobject obj,
237     jint ptr_current_device,
238     jstring send_message) {
239
240     byte_t  abtRx[MAX_FRAME_LEN];
241     size_t  szRx;
```

```
242
243     nfc_device_t *current_device = (nfc_device_t *) ptr_current_device;
244
245     nfc_target_t nt = {
246         .nm.nmt = NMT_DEP,
247         .nm.nbr = NBR_UNDEFINED, // Will be updated by nfc_target_init
248         .nti.ndi.abtNFCID3 = {  0x12, 0x34, 0x56, 0x78, 0x9a,
249                                 0xbc, 0xde, 0xff, 0x00, 0x00 },
250         .nti.ndi.szGB = 4,
251         .nti.ndi.abtGB = { 0x12, 0x34, 0x56, 0x78 },
252         .nti.ndi.btDID = 0x00,
253         .nti.ndi.btBS = 0x00,
254         .nti.ndi.btBR = 0x00,
255         .nti.ndi.btTO = 0x00,
256         .nti.ndi.btPP = 0x01,
257     };
258
259     if(!nfc_target_init (current_device, &nt, abtRx, &szRx)) {
260         nfc_perror(current_device, "nfc_target_init");
261         return EXIT_FAILURE;
262     }
263
264     return EXIT_SUCCESS;
265 }
```

Listing 8: JNI wrapper, parking meter and broker

Java Class with Native functions

```
1  package mipass.nfc;
2
3  import java.util.logging.Logger;
4
5  public class SingletonNfcWrapper {
6
7      private final static Logger LOGGER = Logger.getLogger(
8                                           SingletonNfcWrapper.class.getName());
9
10     private static SingletonNfcWrapper singletonReference;
11
12     private TagType tagType;
13     private BaudRate baudRate;
14     private OperationMode operationMode;
15
16         /* Connection Variables, they will be filled in by the native code */
17     private int numberOfDevicesFound = 0;
18     private int pointerDeviceList = 0;
19     private int pointerCurrentDevice = 0;
20     private int pointerNfcTarget = 0;
21
22     /* constructor, keep it private */
23     private SingletonNfcWrapper( TagType tagType,
24                                  BaudRate baudRate,
25                                  OperationMode operationMode) {
26
27         this.tagType = tagType;
28         this.baudRate = baudRate;
29         this.operationMode = operationMode;
30     }
31
32     /* singleton implementation, first invocation? create instance */
33     public static SingletonNfcWrapper getSingletonObject(
34                                                 TagType tagType,
35                                                 BaudRate baudRate,
36                                                 OperationMode operationMode) {
37         if (singletonReference == null)
38             singletonReference = new SingletonNfcWrapper(tagType,
39                                                          baudRate,
40                                                          operationMode);
41
42         return singletonReference;
43     }
44
45         /* first the load the library*/
46     static {
47         try{
```

```
48              String path = String.format(
49                              "/home/hendri/project/NfcWrapper/lib/libshared.so");
50              System.load(path);
51              LOGGER.info("lib has been loaded, path : " + path);
52          } catch (UnsatisfiedLinkError e) {
53              LOGGER.severe("Native code library failed to load.\n" + e);
54              System.exit(1);
55          } catch(Exception e){
56              e.printStackTrace();
57          }
58          }
59
60          /* Native functions of LibNFC */
61          private native int nfcDisconnect(
62                              int pointerCurrentDevice,
63                              int pointerDeviceList);
64
65          private native String nfcTargetReceiveBytes(
66                              int pointerCurrentDevice);
67
68          private native int nfcTargetSendBytes(
69                              int pointerCurrentDevice,
70                              String sendString);
71
72          private native int nfcListDevices(
73                              int maxDeviceCount,
74                              int pointerDeviceList);
75
76          private native int nfcGetPointerDeviceList(
77                              int maxDeviceCount);
78
79          private native int nfcInitiatorInit(
80                              int pointerCurrentDevice);
81
82          private native int nfcConnect(
83                              int deviceIndex,
84                              int pointerDeviceList);
85
86          private native int nfcInitiatorListPassiveTargets(
87                              int pointerCurrentDevice,
88                              int tagType,
89                              int MAX_TARGET_COUNT,
90                              int numberOfDevicesFound);
91
92          private native int nfcTargetInit(
93                              int pointerCurrentDevice,
94                              String sendMessage);
95
96
```

```
97      public boolean nfcTargetInit(String sendMessage) {
98          pointerNfcTarget = nfcTargetInit(pointerCurrentDevice, sendMessage);
99          return (pointerNfcTarget > 0);
100     }
101
102     public boolean nfcInit(int maxDeviceCount) {
103         pointerDeviceList = nfcGetPointerDeviceList(maxDeviceCount);
104         return (pointerDeviceList > 0);
105     }
106
107     public int nfcListDevices (int maxDeviceCount) {
108         return nfcListDevices(maxDeviceCount, pointerDeviceList);
109     }
110
111     public boolean nfcConnect(int deviceIndex) {
112         pointerCurrentDevice = nfcConnect(deviceIndex, pointerDeviceList);
113         return (pointerCurrentDevice > 0);
114     }
115
116     public int nfcDisconnect() {
117         return nfcDisconnect(pointerCurrentDevice, pointerDeviceList);
118     }
119
120     public String nfcTargetReceiveBytes() {
121         return nfcTargetReceiveBytes(pointerCurrentDevice);
122     }
123
124     public int nfcTargetSendBytes(String sendString) {
125         return nfcTargetSendBytes(pointerCurrentDevice, sendString);
126     }
127
128     public int getNumberOfDevicesFound() {
129         return numberOfDevicesFound;
130     }
131
132     public TagType getTagType() {
133         return tagType;
134     }
135
136     public void setTagType(TagType tagType) {
137         this.tagType = tagType;
138     }
139
140     public BaudRate getBaudRate() {
141         return baudRate;
142     }
143
144     public void setBaudRate(BaudRate baudRate) {
145         this.baudRate = baudRate;
```

```
146     }
147
148     public OperationMode getOperationMode() {
149         return operationMode;
150     }
151
152     public void setOperationMode(OperationMode operationMode) {
153         this.operationMode = operationMode;
154     }
155 }
```

Listing 9: JNI native class, parking meter and broker

C User Manual

This Appendix describes the Android application. The main purpose of this guide is to show how the application works and what its functionalities are. The application consists of eight screens, every screen will be discussed separately. This guide does not provide details regarding the implementation, or about code.

Main Screen

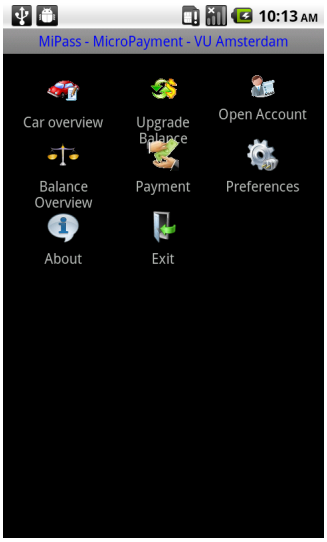


Figure 21: MiPass, Main Screen

Figure 21, shows the application once it starts. This screen provides a total of eight icons:

1. **Car Overview**, a user can choose a car from the list or read a new one from the RFID tag;
2. **Upgrade Balance**, add new credit to the current balance;
3. **Open Account**, open a new account from the Broker;
4. **Balance Overview**, see how much credit is left on the device;
5. **Payment**, check in or check out from/to the parking meter or pay for ordinary articles;
6. **Preferences**, adjust the application behaviour;
7. **About**, application credits;
8. **Exit**, quit the application.

Car Selection

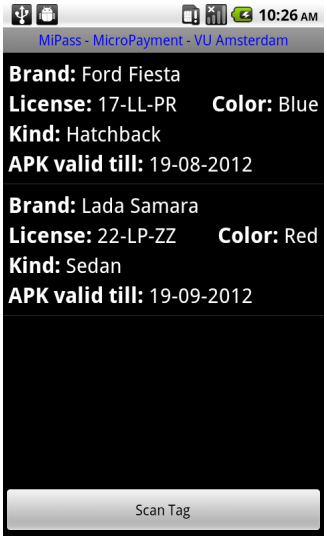
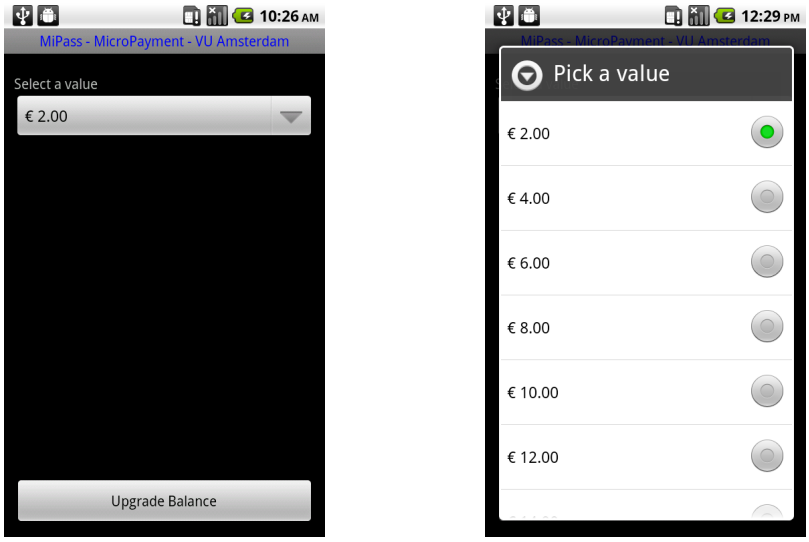


Figure 22: Pick a car

Figure 22 provides the screen of the **Car Overview**. This screen allows a user to scan an RFID tag from a car by pressing the **Scan Tag** button. This RFID tag contains information about the car – its car properties. These properties make the car unique and provide enough information to let the police efficiently determine the validity of the data,. This can be done by simply observing the car and comparing this visible information to that of the data placed on the RFID tag. This data is sent to the parking meter during check-in, but (as mentioned before) can also be used to help the police. When a users car is scanned, it is stored in the list. In this screen shot, A *Lada Samara* and *Ford Fiesta* are already scanned. Thus, a user can select a car by scanning its RFID or by picking a car from the list. A car can be selected by tapping on it; then, the application asks for a confirmation. This can be answered **Yes** or **No**.

Upgrade Balance

In Figure 23, and overview of the **Upgrade Balance** screen is provided. This screen allows a user to update the balance by pressing the **Upgrade Balance** button, as shown in Figure 23a. But first, a value is chosen by pressing the spinner, then, Figure 23b will appear. A user can pick a value to upgrade by tapping on this value. If the NFC reader is connected to the NFC reader of the broker, the value will be added to the current balance of the application.



(a) Upgrade Balance Screen (b) Pick an amount

Figure 23: Upgrade Balance

Preferences

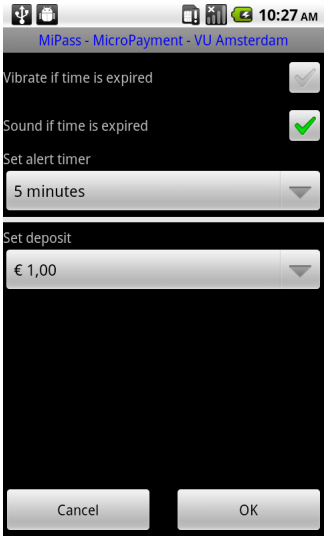


Figure 24: Preferences

The preferences are displayed in Figure 24. These preferences influence the behaviour of the application. The properties are explained from the top of the screen to the bottom. First, the two check buttons can be used to set the behaviour of the device when the parking time has elapsed. It can vibrate, sound an alarm or both. Second, the timer can be set. This timer alarms the user some time before the parking time elapses. This time span can be chosen by the user. Lastly, the user can set a deposit which is sent to the parking meter during check-in.

Payment

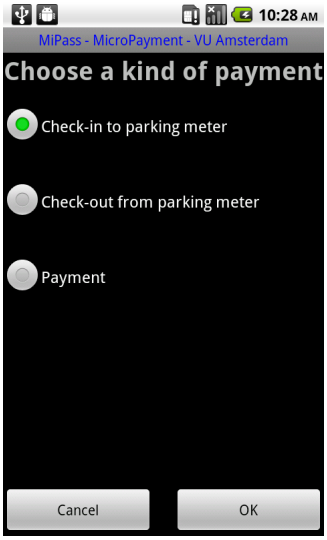


Figure 25: Payment

In Figure 25 the payment screen is displayed. This screen can perform three actions:

1. **Check-in**, to park a car, the check-in option should be chosen. A car should be selected from the **Car Selection Menu** before a car can be checked-in. The user should also have set a deposit in the **Preference Menu**. The NFC reader from the Android device should be placed on the NFC device from the Parking meter.

2. **Check-out**, when the user leaves the parking spot, it has to check out.

3. **Payment**, every ordinary e.g. buying a bottle of soda payment can be done using this option.

Open An Account

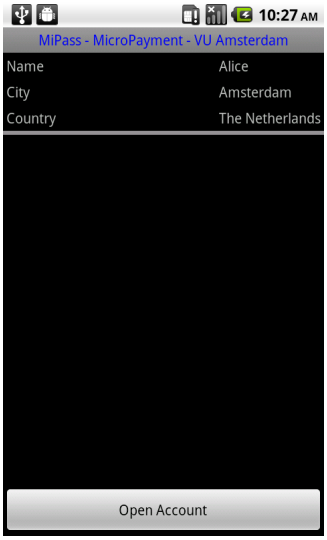


Figure 26: Open an Account

Before a user can use the application, he or she needs to create an account on the side of the broker first. Therefore, the device has to be placed on the NFC reader of the broker. In Figure 26, the screen to open an account is provided. The values that are used to open the account (e.g. Name, City, Country) are already placed in the application, since this application is a prototype. However, when the **Open Account** button is pressed and the device is placed on the brokers NFC reader, an account for Alice will be opened.