Masterarbeit

Authentication Performance Evaluation based on NFC Host Emulation

Matthias Schwarz, BSc.

Institut für Technische Informatik
Technische Universität Graz

Begutachter: Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger
Betreuer: Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger
Dipl.-Ing. Manuel Menghin

Graz, im September 2012

Kurzfassung

In heutigen Radio Frequency Identification (RFID) Systemen werden oft so genannte Contactless Smart Cards wie zum Beispiel der NXP MIFARE DESFire EV1 verwendet, die Sicherheitsstandards wie Common Criteria Certification CEAL4+ oder höher erfüllen. Um in diesen Systemen die Authentizität von Karte und Lesegerät sicher zu stellen kommen Authentifizierungsprotokolle basierend auf offenen Kryptographie Standards wie Data Encryption Standard (DES), triple DES oder Advanced Encryption Standard (AES) zum Einsatz. Die dafür benötigte Rechenleistung wird im Falle der Smart Cards von dezidierten Kypto Coprozessoren erbracht. Diese ermöglichen es, die erforderlichen kryptografischen Operationen in einem Zeitrahmen, der in der ISO14443 Norm beschrieben wird, abzuarbeiten.

Auf Basis dieser RFID Technologie entstand in den letzten Jahren die sogenannte Near Field Communication (NFC) Technologie. Heutige NFC-fähige Mobiltelefone sind rückwärtskompatibel zum ISO 14443 Standard und sind mittlerweile mit leistungsfähigen CPUs bzw. Basisband Prozessoren ausgestattet. Im sogenannten Karten Emulationsmodus sind diese in der Lage, sich wie konventionelle Contactless Smart Cards zu verhalten. Die dafür nötigen Rechenoperationen werden in einem sogenannten Trusted Execution Environment (TEE) durchgeführt wie z.B. einer SIM Karte oder einem Secure Access Module (SAM) e.g. NXP SmartMX.

Diese Arbeit zeigt die Fähigkeit eines modernen NFC-fähigen Mobiltelefons diese Karten Emulation auerhalb des Trusted Execution Environment eines Secure Element (SE) über den Basisband Prozessor durchzuführen. Am Beispiel eines AES Authentifizierungsprotokolls, wie es der MIFARE DESFire EV1 verwendet, wird die Performance der Authentifizierung auf einem Mobiltelefon gezeigt und durch freischalten des sogenannten Host Emulationsmodus eine Proof of Concepte Implementierung erstellt. Dies Implementierung zeigt, dass es möglich ist die AES Authentifizierung in vergleichbarere Zeit zur Emulation mittels SIM Karte durchzuführen.

Abstract

Contactless smart cards are commonly used in nowadays Radio Frequency Identification (RFID) systems. These credentials e.g. the NXP MIFARE DESFire EV1 meet security standards like Common Criteria Certification CEAL4+ and higher. To ensure the authenticity of the credential and the reader these systems rely on certain authentication protocols based on open cryptography standards like Data Encryption Standard (DES), triple DES and Advanced Encryption Standard (AES). Usually contactless smartcards use a dedicated crypto coprocessor to compute these cryptographic functions in a time frame defined by ISO14443 standard.

The upcoming Near Field Communication (NFC) technology is based on this RFID technology and therefore NFC enabled cell phones are backwards compatible to the existing ISO14443 standard and come along with fast CPUs or base band processors. In a card emulation mode they can act like a conventional Contactless Smartcard whereby the emulation is hosted in a Trusted Execution Environment (TEE) like a SIM card or a Secure Access Module (SAM) e.g. SmartMX.

This work shows the abilities of a modern NFC smart phone to host this card emulation not in the Trusted Execution Environment of Secure Element (SE) but on the base band processor. Using the example of the AES mutual authentication protocol of a MIFARE DESFire EV1 this work shows the timing behavior on a smart phone. By enabling the so call Host Emulation Mode a Proof of Concept implementations is provided. The measurements on this implementation show that the authentication protocol can be executed in comparable time to existing SIM card emulations.

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....
date   (signature)

Acknowledgment

Diese Diplomarbeit wurde im Studienjahr 2011/2012 in Zusammenarbeit mit NXP Semiconductors Austria und dem Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

Hiermit möchte ich allen, die mich bei dieser Arbeit unterstütz haben meinen Dank aus sprechen. Besonders möchte ich meinem Betreuer bei NXP Semiconductors Austria Dipl.-Ing. Wolfgang Steinbauer sowie Dipl.-Ing., Diplomé Ingénieur (ECP), M.A. Rainer Lutz für die Ermöglichung diese Projekts danken sowie allen andern Mitarbeitern bei NXP die zum Gelingen dieser Arbeit beigetrage haben. Des Weiteren möchte ich mich auch bei meinen Betreuern an der Universität Ass.-Prof.Dipl.-Ing.Dr.techn. Christian Steger, Dipl.-Ing. Manuel Menghin und Dipl.-Ing. Norbert Druml herzlich bedanken die mich mit Verständnis und konstruktiven Anregungen unterstützt haben.

Ganz besonders möchte ich mich bei meiner Freundin Yvonne für ihr Verständnis und ihre Unterstützung in dieser nicht immer leichten Zeit bedanken. Ohne sie wäre der Abschluss diese Arbeit nicht möglich gewesen.

Nicht zuletzt möcht ich noch meinen Eltern für die Unterstützung während des Studiums herzlich danken!

Graz, im Mai 2012 Matthias Schwarz

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Objectives 2
1.3 Structure of this work 3

2 State of the Art and Related Work **4**
2.1 RFID 4
2.1.1 Basics 4
2.1.2 Smart Card 5
2.1.3 Near Field Communication 9
2.1.4 Use Case: Access Management 14
2.2 Crypto Algorithm 17
2.2.1 Advanced Encryption Standard AES 17
2.2.2 Related Work on AES Implementations 20
2.3 Security Aspects of RFID Systems 21
2.3.1 Attacks on Cryptography 21
2.3.2 Attacks on the RFID System 21

3 Design **23**
3.1 Project Plan and Tasks 23
3.1.1 Starting Situation 23
3.1.2 Work Flow 24
3.2 Hardware and Platform Selection 25
3.2.1 Selection of Processor 25
3.2.2 NFC Phone 26
3.2.3 Existing devices for Comparison 26
3.3 Protocol Analysis 27
3.4 Common Elements for all Implementations 28
3.5 Design of C Implementation 29
3.6 Design of Application Level Implementations 29
3.6.1 Interface 29
3.6.2 Java Implementation 30
3.6.3 JNI implementation 30

3.7 Design of Proof of Concept . . . 31
3.7.1 Platform Analysis . . . 31
3.7.2 Stack Adoptions . . . 34
3.7.3 Implementation Levels . . . 37
3.7.4 Profiling and Timestamps . . . 38
3.7.5 PoC Use Case . . . 40

4 Implementation **42**
4.1 Used Hardware Platforms . . . 42
4.1.1 LPC1343 LPCXPresso board . . . 42
4.1.2 Nexus S . . . 44
4.2 Used Developing Tools . . . 45
4.2.1 LPCXpresso IDE . . . 45
4.2.2 Eclipse . . . 45
4.2.3 Visual Studio . . . 46
4.2.4 Testbench . . . 46
4.2.5 Android System Build Environment . . . 46
4.2.6 Used Versions of Developing Tools . . . 47
4.3 C Implementation . . . 47
4.3.1 Gladman Implementation . . . 48
4.3.2 Structure of Implementation . . . 48
4.4 Application Level Implementations . . . 48
4.4.1 User Interface . . . 48
4.4.2 Java Implementation . . . 49
4.4.3 JNI Implementation . . . 51
4.5 Proof of Concept Implementation . . . 52
4.5.1 Stack Adaptation . . . 52
4.5.2 PoC Stack Level . . . 54
4.5.3 Routing . . . 55
4.5.4 SDK Adoption . . . 57
4.5.5 PoC Application Level . . . 57
4.5.6 PoC Use Case Implementation . . . 59

5 Experimental Results **60**
5.1 Equipment and Tools . . . 60
5.1.1 ISO Setup . . . 60
5.1.2 Android Debug Bridge and LogCat . . . 61
5.1.3 Time Measurement . . . 62
5.2 Devices for Comparison . . . 64
5.3 C Implementation . . . 65
5.4 Application Level Implementations . . . 66
5.4.1 Multiple Repetitions . . . 67
5.5 Proof of Concept Measurement . . . 69

5.5.1 PoC Stack 69
5.5.2 PoC Java 70
5.5.3 PoC JNI 71
5.6 Discussion of Measurement Results 72
5.7 Impact on the Integrity of Existing RFID System 77
5.8 Usage Possibility for the Host Emulation 78

6 Conclusion and Future Work **79**
6.1 Conclusion 79
6.2 Future Work 80

Bibliography **81**

A Abbreviations **A 1**

B Measurement Tables **B 1**
B.1 PoC Stack . . . . B 1
B.2 PoC JAVA . . . . B 2
B.3 PoC JNI . . . . B 4

List of Figures

2.1 Block Diagram of MIFARE DESFire [NXP10a] 6
2.2 Level 3 and 4 Activation after ISO 14443 [ISO08] 8
2.3 Overview of NFC Forum Architecture [LR10] 9
2.4 Passive Mode - Card Emulation [Fin10] 10
2.5 Basic Components of the NFC-functionality of a Mobile Device [LR10] 11
2.6 Components of PN65 [NFC11a] 13
2.7 PN544 Block Diagram [NXP10b] 13
2.8 Generic Access System [RNP10] 14
2.9 Principle Access Control Procedure with Mutual Authentication . . . 16
2.10 Input to, and output from, the cipher state array [Gla07] 17
2.11 SubBytes Transformation [Gla07] 18
2.12 ShiftRows Transformation [Gla07] 19
2.13 MixColumns Transformation [Gla07] 19
2.14 XorRoundKey Transformation [Gla07] 20

3.1 Work Flow Task and their Dependences 25
3.2 Schematic AES Authentication Protocol 27
3.3 Android Stack 31
3.4 AndroidGlue [Mad11] 32
3.5 FRI and HAL Architecture [NXP11c] 33
3.6 HCI Activation Sequence 35
3.7 HCI Protocol for Host Emulation 36
3.8 Schematic Position of Timing Points for PoC Stack 38
3.9 Schematic Position of Timing Points for PoC Java 39
3.10 Schematic Position of Timing Points for PoC JNI 40

4.1 LPC13xx Block Diagram [NXP11b] 43
4.2 Screenshot Graphical User Interface 49
4.3 Application Class Diagram of Java Implementation 50
4.4 Sequence Diagram of Authentication with Java Implementation . . . 51
4.5 Application Class Diagram of JNI Implementation 52
4.6 Call Trace for Initializing Host Emulation 53
4.7 Structure AES Authentication Communication Blocks 55

4.8 Data Flow after HCI Receive Event for PoC Application Level Implementations 56
4.9 Class diagram of PoC Java 58
4.10 Sequence diagram for PoC JNI with callback 58
4.11 Structure of Selcect Application Command and Response 59
4.12 Structure of Read Command and Response 59

5.1 Complete Test Bench in Detail [GBBM08] 61
5.2 Average Measured Computation Times of JNI and Java Implementation 67
5.3 Repetition effect on JAVA Implemetation 68
5.4 Measured FDT of PoC Stack 69
5.5 Measured FDT of PoC Java with intend notification 70
5.6 Measured FDT of PoC JNI with intend notification 71
5.7 Callback and Intent comparison for PoC JNI 73
5.8 Measured FDT for PoC Use Case on PoC Stack 74
5.9 Comparison Devices and PoC Implemetations PICCBlock1 75
5.10 Comparison Devices and PoC Implemetations PICCBlock2 76

List of Tables

2.1 List of Available NFC Phones 12

3.1 Elements of the User Interface 30

4.1 Card Emulation Type A Register Values 54

5.1 Profiling Time Consumption 63
5.2 Correction Values for Profiling 64
5.3 Measured FDT of Comparison Devices 64
5.4 Computation Cycles for Different Compiler Settings 65
5.5 Computation Cycles for Different Number of Lookup Tables and Code Unroll 66
5.6 Computation Cycles for Different Number of Lookup Tables with JNI Implementation 67
5.7 Measured FDT of PoC Java with Intent Notification for PICCBlock1 71
5.8 C vs JNI 72
5.9 FDTs of PoC Use Case Commands for Different PoC Implementations 74

B.1 Total Measurement Results for PoC Stack . . . . B 1
B.2 Total Measurement Results for PoC Java with Intent Notification . . B 2
B.3 Total Measurement Results for PoC Java with Callback Notification . B 3
B.4 Total Measurement Results for PoC JNI with Intent Notification . . . B 4
B.5 Total Measurement Results for PoC JNI with Callback Notification . B 5

Chapter 1

Introduction

1.1 Motivation

It's impossible to exclude Radio Frequency Identification (RFID) Systems from our daily live. They play important roles in many areas e.g. logistics, animal identification or anti-theft protection. While early systems just had the possibilities to more or less store a certain amount of data on a transponder and recall this if needed or indicate their presence in a reader field, nowadays systems have abilities to encrypt data and carry out complex computation work. Therefore they can be used in security relevant areas like access management, ticketing or payment solutions. State of the art secure RFID systems rely on chip card transponders with micro controllers. They guarantee the authenticity of the two counterparts (reader and card) with encrypted data transfer. An example of this micro controller card is the MIFARE DESFire EV1[NXP10a], which can use an AES authentication protocol for this purpose. On this so called contactless smart card the crypto function are typically carried out on specific crypto hardware.

The introduction of the NFC Standards resolved the strict isolation of active and passive component in terms of dedicated reader or transponder. New NFC devices like smart phones are able to emulate card. Usually this is done in a secure environment like in additional hardware modules e.g. SmartMX and Universal Integrated Circuit Card (UICC) or Subscriber Identity Module (SIM) card to exclude from untrusted access. The recent speed up in mobile processors used in NFC enabled mobiles and smart phones and the performance evaluation on other AES implementations [Sha06] make it likely that this computations can be done on the base band processor of this devices as well.

1.2 Objectives

The new possibilities of NFC enabled mobile phones bring up the following questions:

- Is it possible to compute the AES base authentication protocol on a mobile phone's base band processor in comparable time to existing smart card and NFC solutions?
- How does the context of a mobile phone environment affect the computation time and total delays?
- Can the computation on the base band processor affect the security of systems relying on this AES based authentication?

Therefore it is the main objective to give an answer to these questions. This should be done by providing different implementations of the AES authentication protocol as it is used by a MIFAR DESFire EV1[1] and a evaluation of their performance. Thereby each one of these implementations aims to show a partial aspect.

The first implementation on a plain ARM processor should show the basic performance on typical mobile phone CPU without the capability of hardware crypto acceleration and without the overhead of an operating system (OS).

A second implementation should show the influence of an OS. This should be done on a free available NFC enabled smart phone and should be compared to a third implementation in a different programming language for the same hardware platform and OS.

To provide a proof of concept the hidden card emulation features of a NFC mobile phone via the base band processor should be enabled and the authentication protocol should be attached.

A comparison to state of the art smart card and card emulation solutions should show whether the host emulation could be used in a practical access management system or if it could be a gateway for possible attacks.

[1] Will be explained in Chapter 2

1.3 Structure of this work

Chapter 2 starts with the introduction into the state of the art including the relevant basics of RFID and NFC technology plus a description of the access management use case. Describing the AES crypto algorithm and its related work will round up this chapter.

In **Chapter 3** the necessary steps for achieving the objectives are defined and the chosen platforms are lined out as well as the design of the protocol implementations and the proof of concept.

Later on, in **Chapter 4**, it will be described how the actual implementations and the proof of concept are carried out.

The actual measurements and results are presented in **Chapter 5** followed by an analysis of the impact on existing RFID systems as well as statements to the usability of the host emulated AES authentication.

In the last **Chapter 6** the work shall be completed with an all-embracing conclusion and future perspective.

Chapter 2

State of the Art and Related Work

In the first part of this chapter the basics of RFID and NFC systems are described. Later on the focus moves to AES cypher and the chapter is closed by an assumption of possible attacks on RFID systems.

2.1 RFID

2.1.1 Basics

Generally spoken Radio Frequency Identification (RFID) is a technology that uses electromagnetic fields to transmit data for the prose of identification. Such an RFID system in principle consists of two components [Fin10].

- A transponder or tag that contains information and can be attached to other objects to identify them.
- And a reader to contactessly reads and/or writes the information on this tag.

RFID systems can use different methods and frequencies to transfer the data. Depending on that the reading range can be from a few centimeters up to more than 100 meters. Beside others they can roughly be categorized in system using a Backscatter principle to perform the data transfer from the transponder to the reader and systems using load modulation.

Backscatter

In a backscatter system the transponder is able to modulate the reflected electromagnetic wave sent from the reader. Usually this system is operating in the Microwave area as the energy of the reflected wave rises with the frequency [Fin10].

Load Modulation

Load modulation is normally used for short rage devices where the tag has to be in the near field of the reader and therefore is inductively coupled. By switching a load resistor the tag can influence the electromagnetic field provider by the reader. This can be sensed by the reader and be treated like an ordinary amplitude modulation.

The field of application for RFID systems is manifold. They reach from animal identification,logistic solutions over anti-theft to contactless measurement applications. There the physical appearance if the tags can be very different e.g. Glass tube transponder, Keys and Key Fobs, smart labels [Fin10]. For personal identification and accesses management often credentials (RFID transponders) in check card format are used.

2.1.2 Smart Card

Smart cards are plastic cards with normed dimensions that are featured with a microchip [LR10, translated].

They can be used for a range of applications like identification cards or bank cards. In principle they can be categorized in Storage cards or Processor cards. Processor cards contain a micro controller and the functionality depends on the software running on it [LR10, c.f.]. These cards can also contain additional hardware blocks like coprocessors for cryptography.

Contactless Smart Card

If such a microprocessor card features a RFID interface it is a contactless smart cards. Even if there are a number of different contactless smart card types we want to concentrate on cards compatible to ISO/IEC 14443. A representative of this card type is the MIFARE DESFire EV1.

MIFARE DESFire EV1

The MIFARE DESFire EV1 is compatible to ISO/IEC 14443A that fulfils Common Criteria Certification EAL4+. Some of its additional features are [NXP10a, c.f.]:

- ISO/IEC 7816 compatibility
- 7 bytes or RANDOM ID
- Mutual three pass authentication
- Hardware DES using 56/112/168 bit keys featuring key version, data authenticity by 8 byte CMAC

- Hardware AES using 128-bit keys featuring key version, data authenticity by 8 byte CMAC
- Up to 28 applications with 32 files each

The principle function blocks of the MIFARE DESFire EV1 can be seen in figure 2.1.

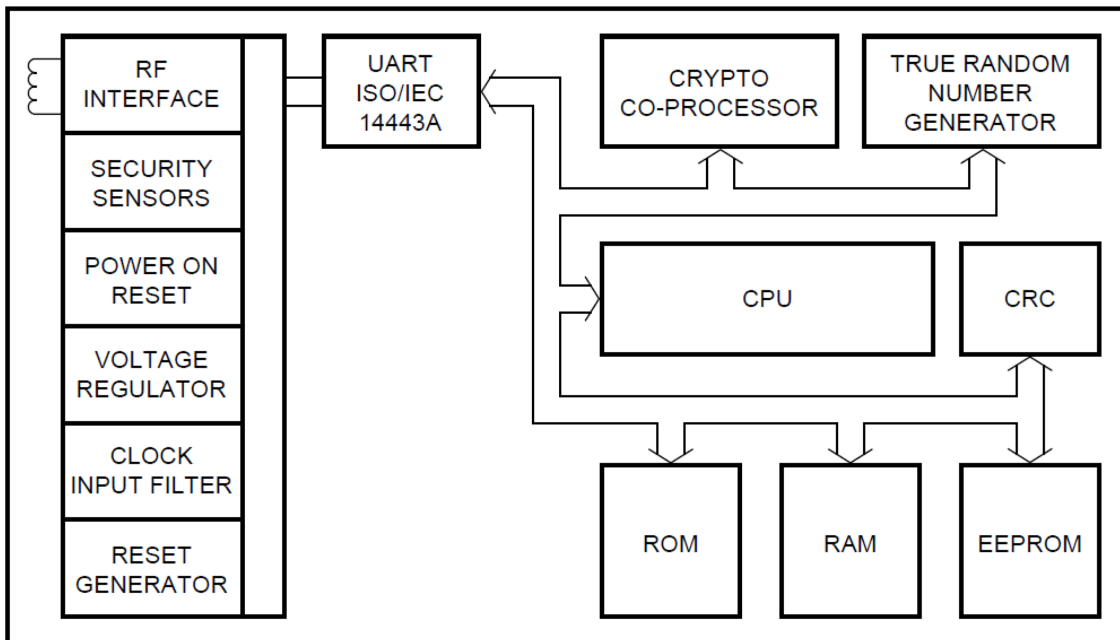


Figure 2.1: Block Diagram of MIFARE DESFire [NXP10a, p.5]

ISO 14443

This standard describes the transponder types relevant for this work, the so called Proximity Cards. They are powered by alternating magnetic field of a reader at frequency of 13.56MHz and operate in range of about 10 cm. Their physical appearance and behaviour are standardized in the ISO 14443 [ISO08] which consists of four parts. The first part describes the physical characteristics and defines different classes of tags according to their antenna size and shape. In the second part the field and signal shapes the reader or proximity coupling device (PCD) and the tag or proximity card (PICC) use to exchange data are described. It also distinguishes between two types of tags, Type A and Type B. The two types differ in signal shape and bit coding. To reference a certain PICC type in this woke it will be indicated by a letter e.g. 14443A for type A.
The framing and the anticollision mechanisms for operating with more than one

PICC in the reader field are described in Part 3. Commands of this part will be called Level 3 commands. The so called Level 4 commands are defined in the last part of the standard as well as the framing for upper layer commands and application data.

In order to exchange higher level data the PICC has to be in a certain active state. The steps needed to activate a certain type A tag to level 4 is shown in Figure 2.2.

First the PCD sends a "Request" (REQA). All PICCs in the field will answer to this with the "Answer to Request" (ATQA). In the anticollision loop a PICC certain UID is address and selected. The received "Answer to Select" (SAK) tells the PCD if the PICC is compliant to ISO/IEC 14443-4. If so it sends the "Request for Answer To Select" (RATS) command. In the "Answer To Select" (ATS) the PICC can define how much time it needs to for responding to a Level 4 command. This is the so called Frame Waiting Time (FWT). This can have a value between 302 s and 4949 ms. If the PICC supports the "Protocol and Parameter Selection" (PPS)' command (also coded in the ATS), this can be used to change the transmission rate in both directions. Once the card is activate to this level it can communicate transparently with previously defined (or default) parameters. Additionally, if the allocated FWT is not sufficient for responding to a specific command, the PICC can extend that time temporarily with a special respond (Waiting time extension WTX) up to 292 seconds.
As this standard does not include security mechanisms, this will be considered on a higher abstraction level.

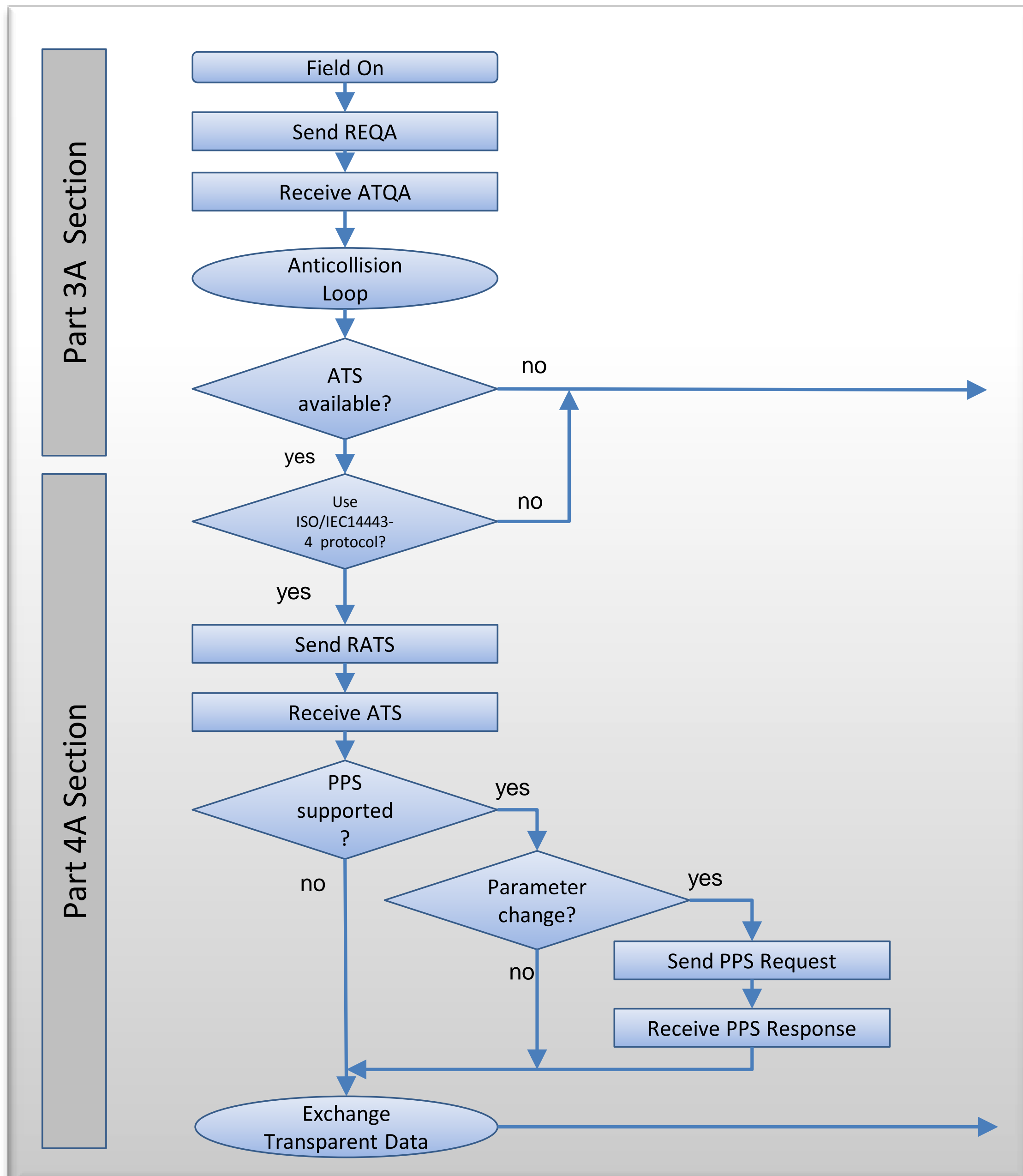


Figure 2.2: Level 3 and 4 Activation after ISO 14443 [ISO08]

2.1.3 Near Field Communication

The idea of NFC is to harmonize today's contactless technologies [NFC11b, c.f.].

This technology tears down the strict separation between active reader and passive tag [LR10]. Such devices are able to switch between active polling and passive listening. It combines the existing RFID Standards ISO14443 Type A and B and JIS X 6319-4 (FeliCa) with additional protocols for the communication between two NFC devices in ISO/IEC 18092 or ECMA-340.

A NFC device should support three modes of operation. Figure 2.3 shows them with the according standards.

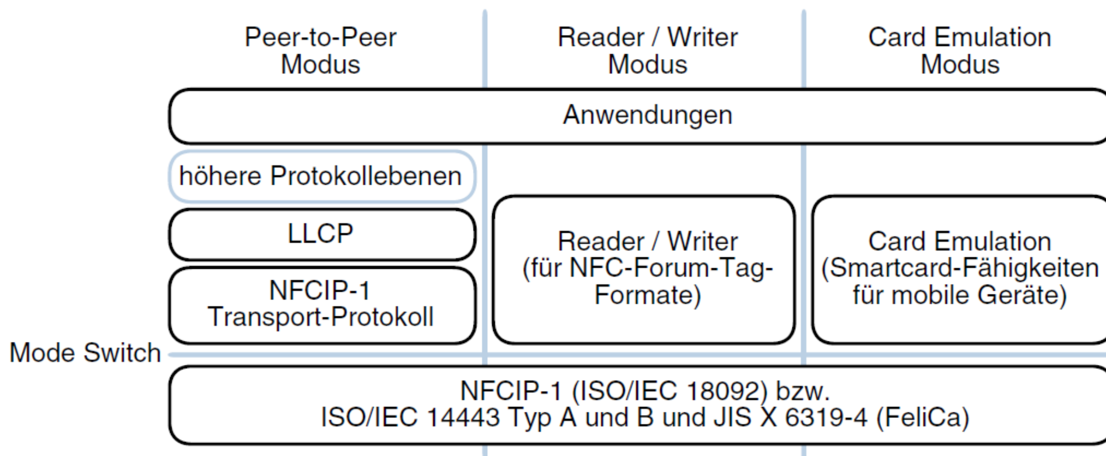


Figure 2.3: Overview of NFC Forum Architecture [LR10, p.90]

- The peer to peer (P2P) Mode is used for the communication between two NFC devices. This can either be done in a passive communication mode where one device (the initiator) provides the field and the other (the target) answers with load emulation or active communication mode where both devices provide their own field one after the other. The used Protocol is called NFCIP-1 defined in ISO/IEC 18092.

- In the Reader/Writer-Mode the device acts like a convention PCD device. It provides the field and senses the load modulation of a PICC.

- The most relevant mode for this work is the card emulation mode. In that mode the NFC device acts like a PICC and modulates the field of a PCD as shown in figure 2.4.

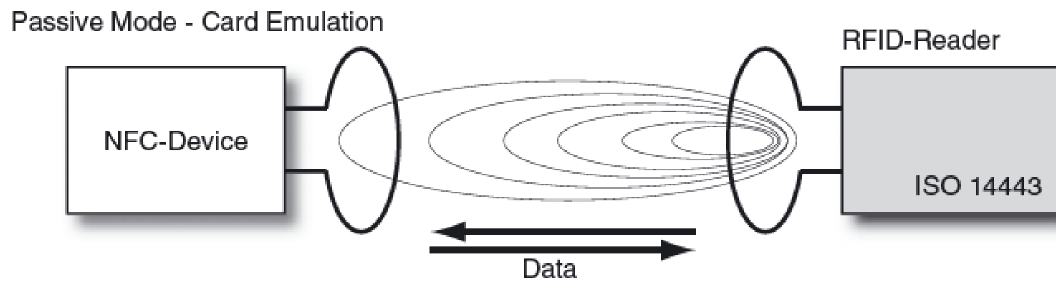


Figure 2.4: Passive Mode - Card Emulation [Fin10, p. 58]

These abilities of NFC devices offer a lot more possible application areas than RFID tags could on their own. Especially when combining those devices with internet connection like this is done on NFC enabled smart phones. This allows for examle to:

- download tickets and use the mobile to pass the access control.
- display informations from a smart poster
- Exchange business card
- Get and use coupons

The list of possible application is still growing.

Mobile phones

To provide the described functionalities a NFC enabled mobile phone should have at least these four components you can see in figure 2.5 [LR10]

- A base band processor to host applications and control the other components
- A NFC controller to control the RF Interface
- A Secure Element to provide a trusted Execution environment
- And a NFC Antenna

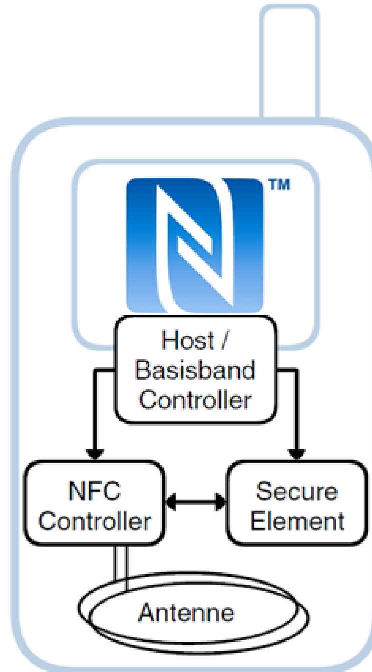


Figure 2.5: Basic Components of the NFC-functionality of a Mobile Device [LR10, p.146]

For the connection between the NFC controller and the Secure Element (SE) two protocols have been specified. If the SE is a UICC or SIM, the communication is done over the so called Single Wire Protocol (SWP) defined in ETSI TS 102 613. The second possibility is to connect a SE, e.g. at SmartMX over the NFC- WI as defined in ECMA-373. By principle more than one of this secure elements could be used at the same time and each could host several cards but in some cases this can lead to problems for example when the reader specification allows just one RFID card in the field. In [LR10] and [MDLS08] several approaches for solving this problem are discussed but no standardized solution has been found so far.

There are already a number of NFC enabled smart phones commercially available. A collection of them can be seen in table 2.1. All of these phones could fulfil the above mentioned requirements.
It is not always known which actual NFC chip is used within this phones. In case of the Google Nexus S documented tear down [ifi11] shows that this device uses an NXP PN65 chip. Therefore they will be described briefly.

Name	Hersteller	OS
Nexus S	Samsung	Android
Galaxy Nexus S	Samsung	Android
Galaxy S2 NFC	Samsung	Android
Galaxy Note	Samsung	Android
Sonic U8650 / T20	Huawei / Turckcell	Android
Droid Razr	Motorola	Android
Ruby	HTC	Android
Liquid Express	HTC	Android
C7	Nokia	Symbian3
N9	Nokia	MeeGo
700	Nokia	Symbian Belle
701	Nokia	Symbian Belle
Wave 578	Samsung	Bada
Wave Y	Samsung	Bada
Wave M	Samsung	Bada
Bold 9900	BlackBerry	BB 7 OS
Curve 9350	BlackBerry	BB 7 OS

Table 2.1: List of Available NFC Phones

NFC Controller Chip

In the schematic picture of the PN65 in figure 2.6 you can see, that it hosts a PN544 NFC chip which is connected to a SmartMX in the same package. For the purpose of this work this SmartMX is not relevant and the PN65 can be treated the same way as the PN544. Therefore only this chip is described in detail

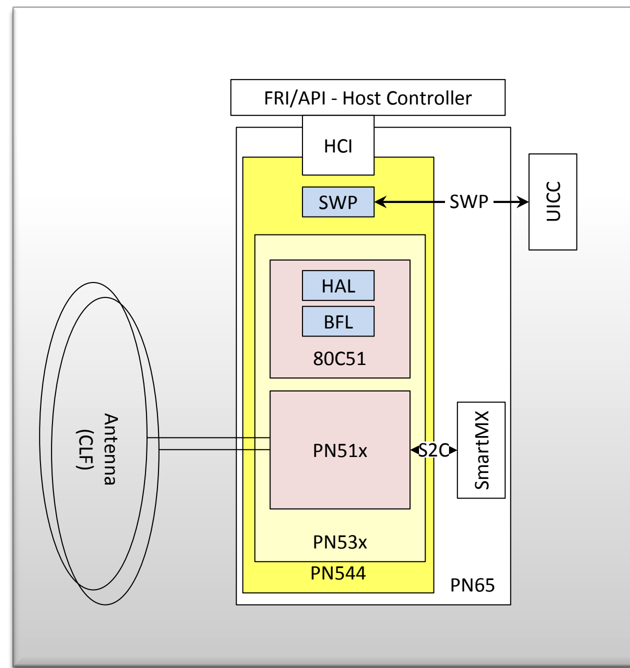


Figure 2.6: Components of PN65 [NFC11a]

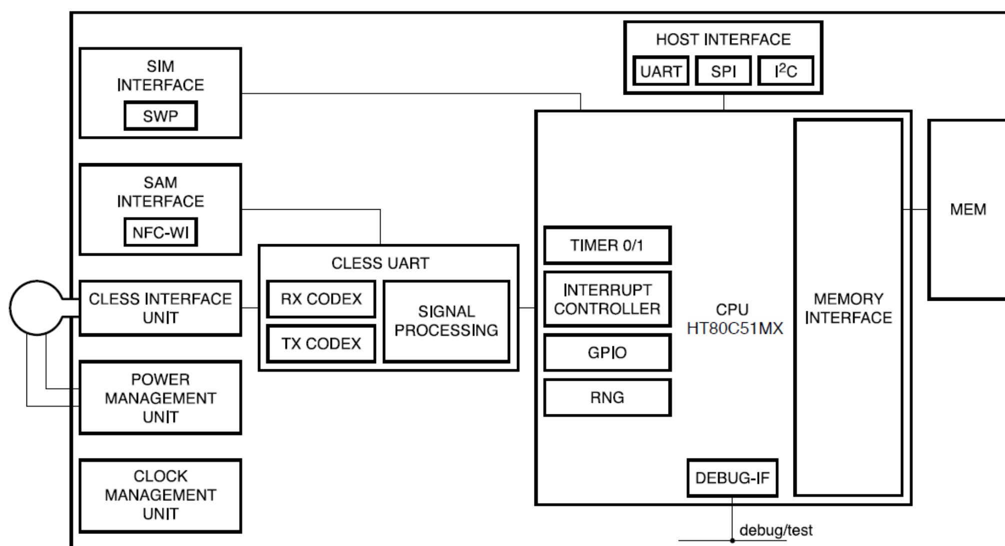


Figure 2.7: PN544 Block Diagram [NXP10b, p. 3]

If you look at the composition this PN544 in figure 2.7 it can be seen, that in principle the SE can be connected in different ways.One is to do it over the NFC-WI interface for the Secure Application Module (SAM) like the SmartMX. This interface is connected directly to the Contactless UART Block whereas the connection over the Single Wire Protocol (SWP) interface and the Host interface are established over the chips CPU. This means, at least theoretically, that the last two should have similar access possibilities. The relevant functional abilities of the PN544 will be described later in chapter 3 as well as the Host Controller Interface (HCI) defined in [ETS08].

Card Emulation Solutions

As described before, a NFC enabled cell phone should be able to emulate a RFID card. In fact there are not many solutions available on the commercial market. This is most likely because of the not fully developed ecosystem that has to deal with proprietary smart card solutions and restricted access to the secure elements.

Nevertheless the company Gemalto announced in 2010 to be the first to integrate a MIFARE DESFire card emulation in a SIM card [Gem10]. Also emulations on SmartMX exist but because of the mentioned restricted access rights they are not available on a phone.

2.1.4 Use Case: Access Management

One of the key applications in the truest sense of the word of RFID systems is the access control. Even if most of the following points hold for logical access management as well this work will focus on providing physical access to restricted areas. Therefore the schematic setup of a access management system can be seen in figure 2.8.

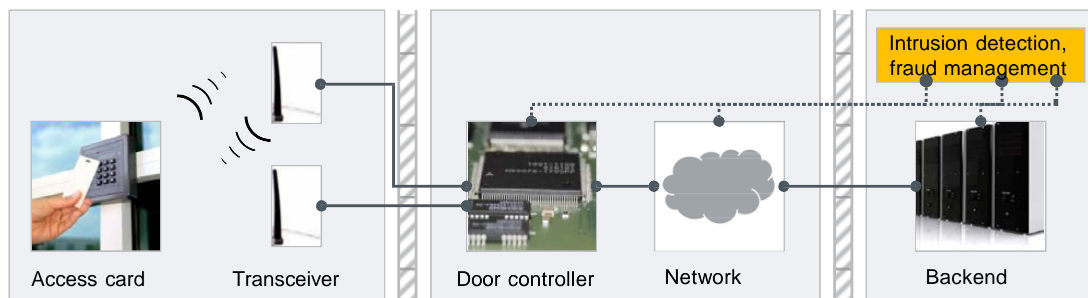


Figure 2.8: Generic Access System [RNP10, p. 15]

You can distinguish between an online and an offline system.

The offline system just contains the left parts of figure 2.8. The door controller is not connected to a network or backend system. Therefore it has to store all the relevant keys and white listed card IDs. This topology is useful for small systems because a high number of these standalone controllers comes along with a high managing effort.

In an online system the backend system is connected to all controllers. This kind systems is easier to maintain and manage and it gives much more possibilities to detect frauds and intrusion e.g. by comparing the locations where a tag is used and if the time between the usages is plausible.

Authentication Flow

Old RFID access management systems often just use the card UID for checking the cards authenticity. These systems can be spoofed easily as there already exist cheap solutions for cloning the UID. To achieve a state of the art security level a certain Authentication Protocol is recommended in [RNP10].

The practical implementation of this protocol for a Type A contactless smart card like the MIFARE DESFire EV1 is shown in figure 2.9. It highlights the relevant information the reader receives from the card during this procedure. For authentication it is recommended to use diversified key. That means each credential is using a different key that is derived from a master key and e.g. the cards UID. After this mutual authentication all messages should be protected by a message authentication code (MAC) or full encryption.

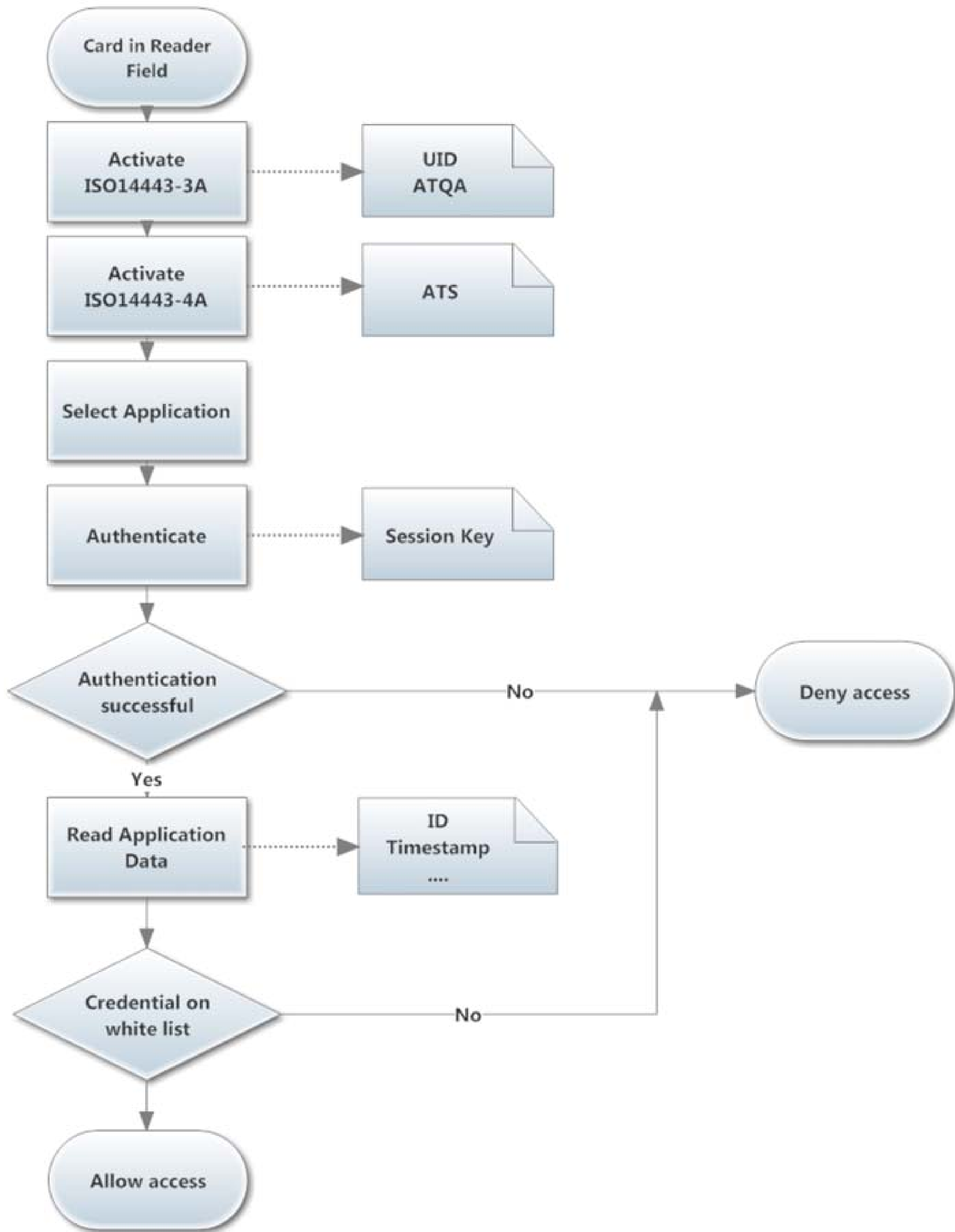


Figure 2.9: Principle Access Control Procedure with Mutual Authentication

2.2 Crypto Algorithm

The relevant encryption standard in this work is the Advanced Encryption Standard (AES). Therefore this section gives a summary of this algorithm as well as the known implementations and evaluations. This is important to see the possible performance switches to boost the performance of the algorithm.

2.2.1 Advanced Encryption Standard AES

The AES was originally called Rijndael algorithm and was developed by Joan Daemen and Vincent Rijmen. It was adopted by the U.S. National Institute of Standards and Technology NIST after a five-year standardization process to find a follower to the Data Encryption Standard DES. The Algorithm that is described in the AES is a specific variant of the Rijndael algorithm where the block size is fixed at 128. Therefore it is specified for key lengths of 128, 192 and 256 bits.

For doing an encryption the input data are stored in a four by four byte matrix which is called the *state* as you can see in figure 2.10.

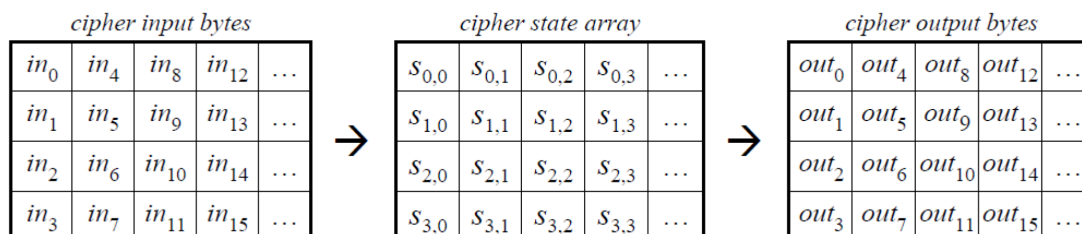


Figure 2.10: Input to, and output from, the cipher state array [Gla07, p. 2]

Next this state is treated according to the following structure of the AES algorithm [Gla07, Sha06, c.f.]:

1. Key schedule
2. Initial round
   (a) XorRoundKey Transformation
3. (Nr - 1) Rounds (where Nr is 10, 12 or 14, depending on the key length)
   (a) SubBytes Transformation
   (b) ShiftRows Transformation

   (c) MixColumns Transformation
   (d) XorRoundKey Transformation

4. Final round
   (a) SubBytes Transformation
   (b) ShiftRows Transformation
   (c) XorRoundKey Transformation

Key schedule

To do the AES encryption a key for each round has to be derived. Therefore the original key is expanded with the Rijndael key schedule as described in [Gla07]. The expansion can be done totally in advance or stepwise before each round.

SubBytes Transformation

At the *SubBytes Transformation* on all elements of the state array a non-linear byte substitution is performed. The used substitution table is the so called s-box and is constructed as described in [Gla07, p. 7].

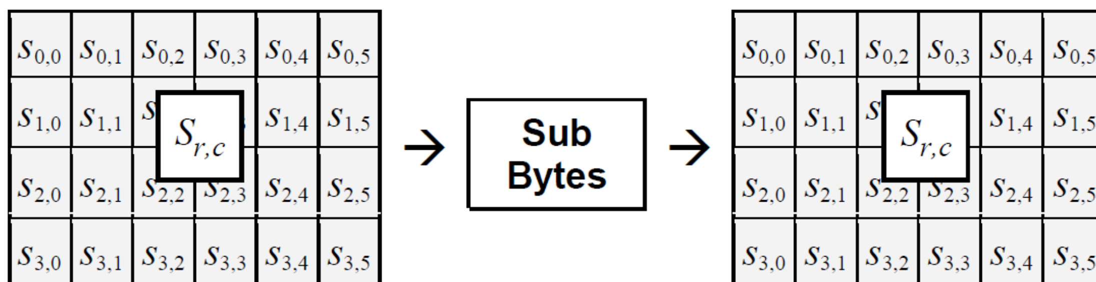


Figure 2.11: SubBytes Transformation [Gla07, p. 7]

ShiftRows Transformation

For the *ShiftRows Transformation* the rows of the state get cyclically shifted. The nth row is shifted n-1 bytes to the left as shown in figure 2.12.

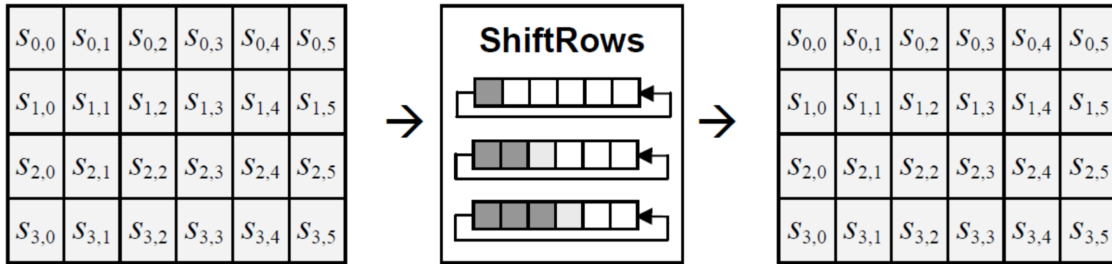


Figure 2.12: ShiftRows Transformation [Gla07, p. 9]

MixColumns Transformation

In the *MixColumns Transformation* each column of the matrix gets multiplied with a fixed polynomial.

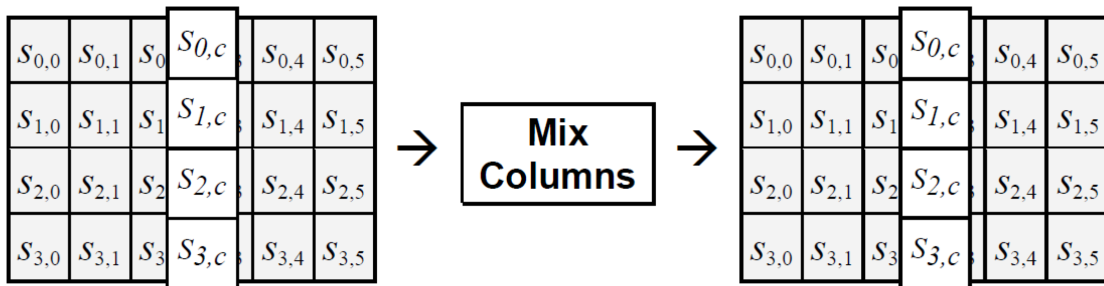


Figure 2.13: MixColumns Transformation [Gla07, p. 9]

XorRoundKey Transformation

In the last step the rows of the state matrix get XORed with the round keys derived during the key schedule as shown in figure 2.14.

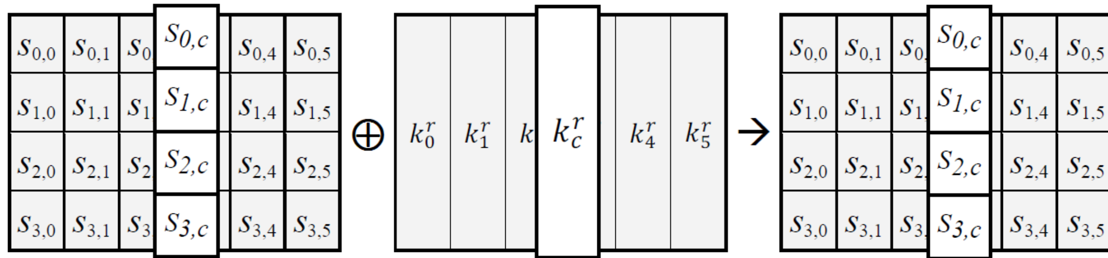


Figure 2.14: XorRoundKey Transformation [Gla07, p. 10]

Decryption

The presented transformations for encryption are invertible and therefore the inverse transformations can be performed in the reverse order to execute a decryption.

2.2.2 Related Work on AES Implementations

As the AES algorithm is in use for some years now e.g. for securing WLAN and the SSH protocol, numbers of implementations and works about the performance and optimization of the algorithm are available. Even if the performance of the cypher could be accelerated with a hardware implementation the attention in this work is focused on software implements.

The usual approach to accelerate those AES software implementations is to adapt the algorithm to the special abilities of the hardware e.g. in [BBF+03] it is done by introducing a new way of calculating MixColumns transformation.
In this context papers like [OBSC10] or [BKG11]and a master thesis [Sha06] show performance evaluations on different kinds of platforms including ARM processors.

As there is no implementation on a mobile device including an Operating System (OS) these works can just give a hint that this platform is fast enough for the purpose of this work. But the previously mentioned works list a number of existing implementations. One of them is from Gladman [Gla11]. It is the mostly referenced one and also one of the fastest as the measurements in [Sha06] show. This fact and the good availability of the Gladman implementation make it a proper candidate for this work. Details of this implementation will be presented in Chapter 4.

2.3 Security Aspects of RFID Systems

As RFID systems are often used in high security areas they face a latent risk to be attacked. On the one hand such an attack could try to break the cryptographic algorithm to gain information about the secret keys and on the other hand it could use weaknesses in the implementation of the RFID system. Therefore the existing attacks on both sides are described here briefly.

2.3.1 Attacks on Cryptography

As the AES algorithm is used in many security critical applications a number of theoretical works is written about breaking the cypher. The approaches to do these attacks are various e.g. a related-key attacks like in [BK09] or attacks based on bicliques like in [BKR11]. So far none of these attacks is for practical use but in conclusion you can say that at the current state the 128-bit key version is at least as secure as the 192-bit and the 256-bit versions.

2.3.2 Attacks on the RFID System

For RFID systems you can group the attacks into three categories. Attacks on the transponder, on the reader or on the interface [Fin10]. As the reader is not in focus of this work we will concentrate on the other two.

Spoofing and Cloning

The first principle attacks is to clone a tag with all its functionality and including the data stored on it. Therefore either blank cards which allows to set the UID or devices that act like a card are needed. Such a device is for example the Chameleon emulator of [KvMOP11] which is able to emulate a series of ISO14443 cards. Never the less, for executing this attack the information stored on a transponder has to be extracted. Of course in modern smart cards this is protected from unauthorized reading by secret keys which have to be extracted from the card with other attacks e.g. Side-channel analysis

Side-channel analysis

In this attack the actual hardware implementation of a cypher algorithm is examined. Thereby additional information to break the cypher can be gained from e.g the consumed power like in a differential power analysis or timing behaviour. In [OP11] for example such a side-channel attack is successfully performed on an older version of the MIFARE DESFire the MF3ICD40.

Replay authentications

At a replay attack the communication between the card and the reader is eavesdropped and is replayed later on. This attack can be countered by using strong random numbers for the authentication.

Relay Attack

The principle of this attack is to extend the range between the reader and the transponder. Such an attack can be seen in [Han06]. For this attack two devices are required. One device, the proxy, that is located near the reader to remotely transceive data to and from the second device, the leech, that is located close to the transponder and simulates a reader [Fin10, c.f.]. For ISO 14443 compliant systems the first barrier comes with the level 3 activation where the time-out constraints are very strict. Therefore the level 3 activation could be done autonomously by the two devices in order to match the constrains. This requires to counter this attack on ISO1443-4 level with a so called proximity check. The command uses similar crypto functions as used during the mutual authentication with the derived session key. Thereby a restrictive time out vale is set. The unknowing of the session key makes a real time imitation of the card repose impossible. This command also disqualifies cards that need too long for the crypto computation.

Chapter 3

Design

The first part of this chapter describes how the work should give answers to the objective questions of section 1.2 and defines the necessary steps to be made. Further on the selection of the required hardware is described and later on the design of the required implementations and the proof of concept will be explained in more detail.

3.1 Project Plan and Tasks

3.1.1 Starting Situation

If doing a rough conclusion of the aim of this work you can say it should give an insight into the capability of modern NFC smart phones base band processor to provide AES based secure authentication and to act like a conventional proximity smart card.

To be able to do this two components are required:

- Software implementations of this authentication protocol in appropriate programming languages.
- And a NFC enabled phone that is able to host a card emulation on its base band processor.

The problem is: none of them is available.
So in order to acquire knowledge about the performance of the authentication protocol these two components have to be created.

3.1.2 Work Flow

As a first step, it is important to set up a work flow for the required work. This can be seen in figure 3.1.

The attempt to bring light into the matter starts with a C implementation. To create a reference base that is not influenced by an operating system this will be done on a plain ARM processor that is comparable to a typical mobile phone base band processor.

Performance measurements on this implementation will show optimization potentials and whether it computes within the time limit stated in ISO 14443-4. As described in Section 2.1.2 this is a broad border and it would make no sense to continue the work if the computation time exceeds that limit.

If the C implementation is fast enough it is time to do implementations on a mobile phone. Therefore a proper hardware has to be selected.

On top of this hardware and its associated operating system two implementations on application level will be done. One will base on the C implementation to show the influence of the OS and another in a different programming language to show the language influence.

These should fulfil the ISO limits as well to justify the effort of enabling the hidden card emulation feature.

As these C and application level implementations just show the calculation time and give no information about communication time overhead the next step is to enable the host emulation and combine it with the previous implementation to a Proof of Concept (PoC).

Provided that the enabling of this feature is possible the proof of concept gets expanded to satisfy the requirement the of access management use case and to shows the practical usability.

Measuring the authentication performance of existing smart card and card emulation solution will give reverence for a concluding analyses of security impact.

All this steps are designed in more detail in the following chapter. With respect to a better readability and understandability this is not done in chronological order. This also forecloses that all implementations satisfy the ISO 14443-4 limits and the enabling of the host emulation feature is possible.

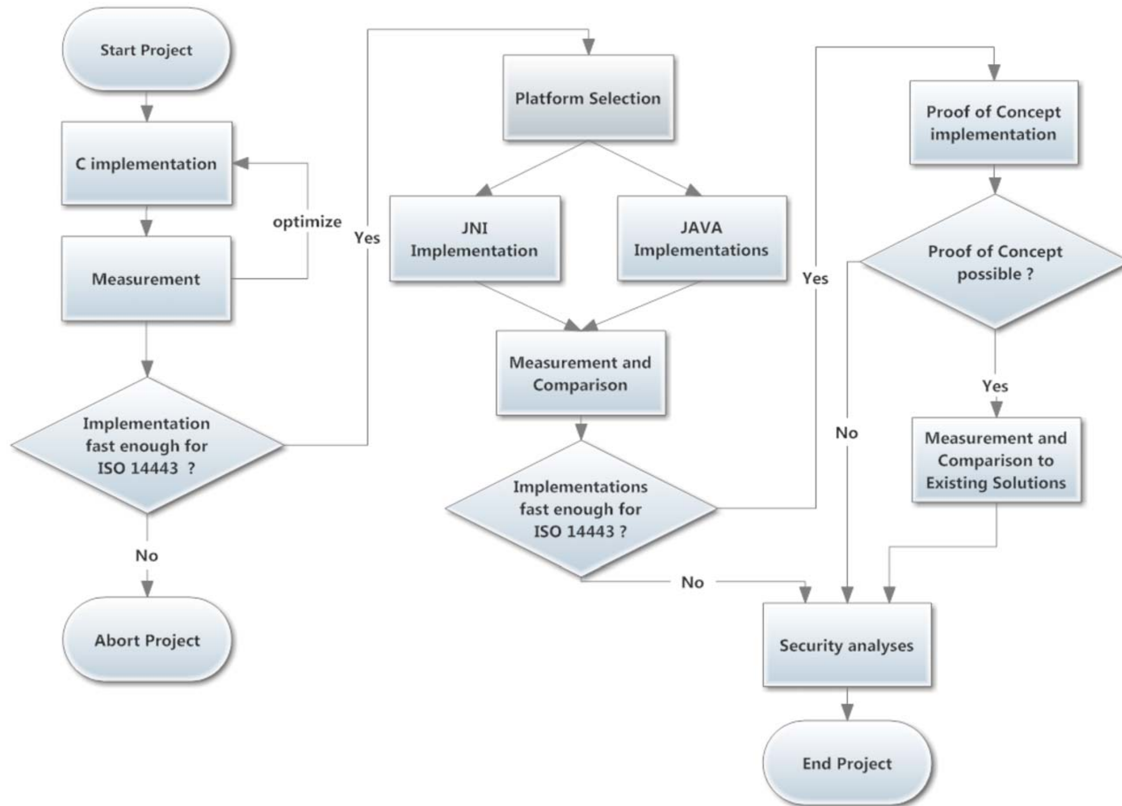


Figure 3.1: Work Flow Task and their Dependences

3.2 Hardware and Platform Selection

A basic idea of this work is to show the computation of the authentication protocol on real hardware. That makes it important to find devices that have a representative character. Therefore the reason for taking the particular devices used in this work will be outlined now.

3.2.1 Selection of Processor

For the first implementation a hardware platform is required that is close to a mobile phone's hardware but does not come with the overhead of an operating system.

Many actual smart phones use ARM Cortex A processors which are designed to *"provide an entire range of solutions for devices hosting a rich OS platform and user applications..."* [ARM11]. As an OS application is not wanted in this case the choice fell on another processor type of the ARM Cortex family namely a member of the ARM Cortex M series.

This can be found on the NXP LPC1343 LPCXPresso board with its 32-bit ARM Cortex-M3 microcontroller. Additionally to the proper processor this board comes together with an Eclipse base IDE called LPCXpresso which makes it easy to setup a build tool chain.

3.2.2 NFC Phone

The choice of the mobile platform fell on the Google Nexus S. This choice was mainly driven by two factors, the OS and the NFC Chip.

The Nexus S is able run an Android OS in the most current version (Icecream-sandwich) and as seen in table 2.1 this is the mostly used OS on NFC enabled smart phones. It hosts applications written in Java or in C with the Java Native Interface (JNI). Therefore those will be the languages used for the application level implementations. Additionally Android is an open source platform and a Source Developing Kit (SDK) is provided by Google. Together with the PN65 this makes the enabling of the host emulation feature possible. Another point that talks for this devise as an implementation platform is that on the one hand the Nexus S is a mass market consumer product and on the other hand it is a designated Android developer phone. This fact makes it easier to do changes in the OS.

3.2.3 Existing devices for Comparison

Measuring the performance of the designed implementations is not the only objective of this work. An important part is the comparison to existing PICCs and card emulation solutions. The principles of this solutions were already described in Section 2.1.3. Therefore we take a look at the particularly chosen devices.

The first reference object of course will be a conventional smart card. As the protocol is directly derived from the NXP MIFARE DESfire this will be the object of choice. In the actual case it will be a MIFARE DESfire EV1 which is the current representative version of this smart card family

As there exists no DESFire emulation for SmartMx on a NFC phone the next solution is a JCOP SmartMx card that hosts the DESFire functionality and is provided by NXP as well.

The last two test objects use the Gemalto SIM card to emulate the MIFARE DESFire. In one case the SIM card is hosted by the NXP PN544 Design Kit board in the other case by a Huawei Sonic U8650 / Turckcell T20.

As written in table 2.1 the Huawei Sonic U8650 / Turckcell T20 is an NFC enabled smart phone with Android 2.3.3. The NFC functionality is provided by the same NFC chip namely the NXP PN544.

In principle the Nexus S could be the hardware platform for the SIM solution as well. The reason for not using it in this case is the fact that it does not support this feature in its delivery state. Enabling it would be beyond the scope of this work especially when having a proper pendant with the T20.

3.3 Protocol Analysis

Before starting with the work flow of section 3.1.2 the actual authentication protocol has to be examined. As already described an AES authentication procedure comparable to the MIFARE DESFire is used to take a look at the authentication performance. It is a symmetric authentication. This means that the reader and the card have to assure that they hold the same secret keys. This type of authentication sometimes is called three pass authentication. A schematic flow of the protocol is shown in figure 3.2.

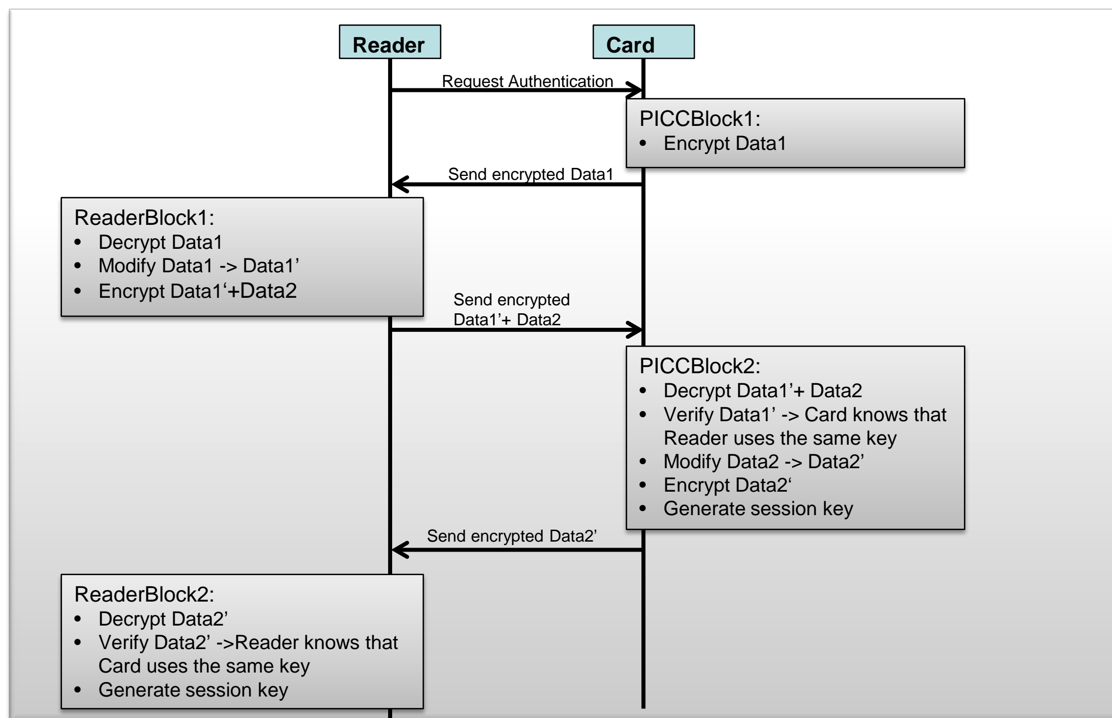


Figure 3.2: Schematic AES Authentication Protocol

The RFID reader sends a command to request the authentication of the card.

The smart card generates a random number (RNDB). This number is encrypted with the secret key and sent back to the reader. This block of computation is the first interesting performance part of this work and will be referenced with PICCBlock1 for the rest this work.

The read decrypts the number (RNDB). As it does not know the original number it does not know if it uses the same key as the card at this moment. It modifies the decrypted number in a predefined way, e.g. rotate some bits. Then it generates a random number (RNDA) as well and encrypt both, the modified number from the card and its own number and sends them back to the card.

The card decrypts these numbers. Undoing the modification of the received number tells the card if the reader has used the same key for the de- and encryption. If so the card modifies the number from the Reader (RNDA) the same way, encrypts it and sends it back to the reader. In addition it generates a session key from the RNDA and RNDB for securing the further communication. This will be called PICCBlock2 in the remaining work.

Similar to the card the reader decrypts the received data and undoes the modifications. If this equals the previously generated number the authentication is completed and the reader generates a session key as well.

In the consider case an AES128 algorithm is used for de- and encryption.

3.4 Common Elements for all Implementations

The work should show the performance of different implementations and therefore should give the ability to measure the computation time of PICCBlock1 and PICCBlock2. As this work starts from far a way of having an actual communication with a reader device this parts had to be written in software as well. For the test cases these software reflected parts are designed to communicate by method calls. Another element that all implementations have in common is the fact that they are not designed for secure computation in the first case. That means that they have to work accordingly to the protocol but no additional measures e.g. for secure storage of the key were considered. This decision was made in the context of possible intruders that will not care about the software security but only about the fastest and easiest possible solution.

3.5 Design of C Implementation

The first planned implementation builds the corner stone of the performance analysis. It is designed to run on the ARM Cortex M3 and therefore has to be written in plane C. From a design point of view there is not much to describe but it should satisfy at least some basic requirements. It should:

be simple: no other security features like secure random generation or secure key storage will be considered.

use an existing AES Implementation: As there are plenty of existing C implementations the tradeoff between performance an free availability has to be considered.

be reusable for the JNI implementation: A modularly design will improve the reusability.

3.6 Design of Application Level Implementations

Application level implementations will be the combining notation for the Java and the JNI implementations which are designed to run as an Android applications on the Nexus S.

3.6.1 Interface

To make the measurements and interaction easier the application level implementations require a user interface. As this is the only purpose it does not need to contain any design highlights but some basic elements. These are shown in table 3.1.

Name	Type	Comment
Key Input	Input field	To manually change the authentication key.
Set Key to Zero	Button	For a quick reset of the authentication key.
Generate Random Key	Button	Generates a random key and updates the **Key Input** field.
Start Authentication	Button	To start the actual authentication.
Rounds Input	Input field	Defines the number of authentications to be performed in sequence.
Start Authentication Rounds	Button	Starts the authentication with a newly generated random key as often as defined by the **Rounds Input** field.
Result	Text field	To display the used computation time of the last authentication or the mean value in case of more than one rounds and the generated session key.

Table 3.1: Elements of the User Interface

3.6.2 Java Implementation

In principle the Java implementation should not do much more than the C implementation but as Java is an object oriented programming language it is obvious to divide the application into the three following classes:

Application Class: controls the application, initialization and the user interface.

Card Class: hosts the actual computation part of PCCBlock1 and PICCBlock2 for the actual performance measurement.

Reader Class: contains the reader computation parts.

For reasons of simplicity the message passing between two classes should be done by public method calls.

3.6.3 JNI implementation

The JNI implementation should be quite similar to the Java implementation. The only differences should be the reuse of the C implementation and the replacement of the card emulation class with JNI methods calls.

3.7 Design of Proof of Concept

The planned implantations in section 3.5 and 3.6 will give a good picture of how fast the actual computation of the crypto function could be. This is not enough to make statements according to how long the actual Frame Delay Time (FDT) would be on a smart phone because the communication times between the different software and hardware components play a significant role in this context. Unfortunately the functionality of card emulation over the host controller interface of a NFC chip is not implemented in nowadays mobile operating systems. Therefore this has to be done as a part of this work to enable a proof of concept. This requires changes deep in the heart of the operating system and the following analysis of the chosen platform, the Samsung Nexus S with Android 4.0.3, will show where and how they have to be made.

3.7.1 Platform Analysis

To get an orientation let us take a look at the principle setup of Android. In figure 3.3 you can see that Android is composed of several software levels. It starts with the application layer and goes down to the Linux kernel on which Android is built on. In principle this is the same for all Android versions.

Figure 3.3: Android Stack [Goo11c]

The Nexus S gets shipped with Android 2.3 and is equipped with a NXP PN544 as described in Section 2.1.3. This Android version already holds the basic NFC functionalities like reading and writing of tags and these functions are accessible from the application level. But it does not cover the full functionalities of the PN544 NFC chip as the card emulation features are not supported. Never the less, some of the functionality is already prepared beneath the application framework level but not accessible from an application. To cover all the latest preparation work the decision was made to us Android 4.0.3 as the implementation basis which is the latest available OS version at this time.
For doing the further analysis in principle three source were available. The major part of information had to be extracted from the Android source code itself. The second is the PN544 user manual [NXP11d] (not public) provided by NXP and the third the ETSI standard [ETS08]. Parts of the analysing work were already done by Madlmayr [Mad11] who provides a schematic of how the NFC functionality is glued into the system as you can see in figure 3.4.

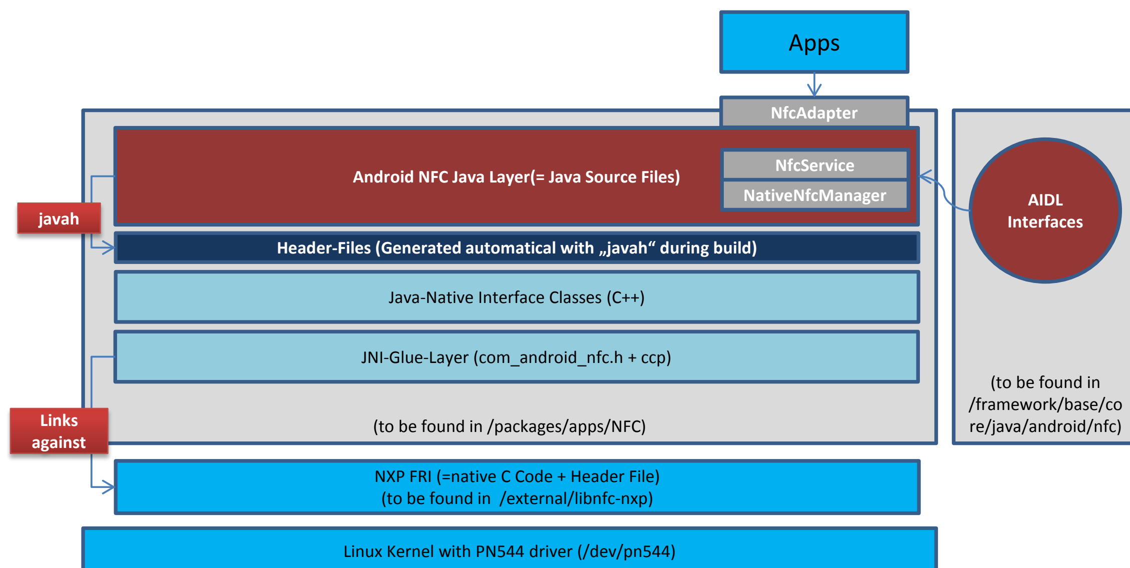


Figure 3.4: AndroidGlue [Mad11]

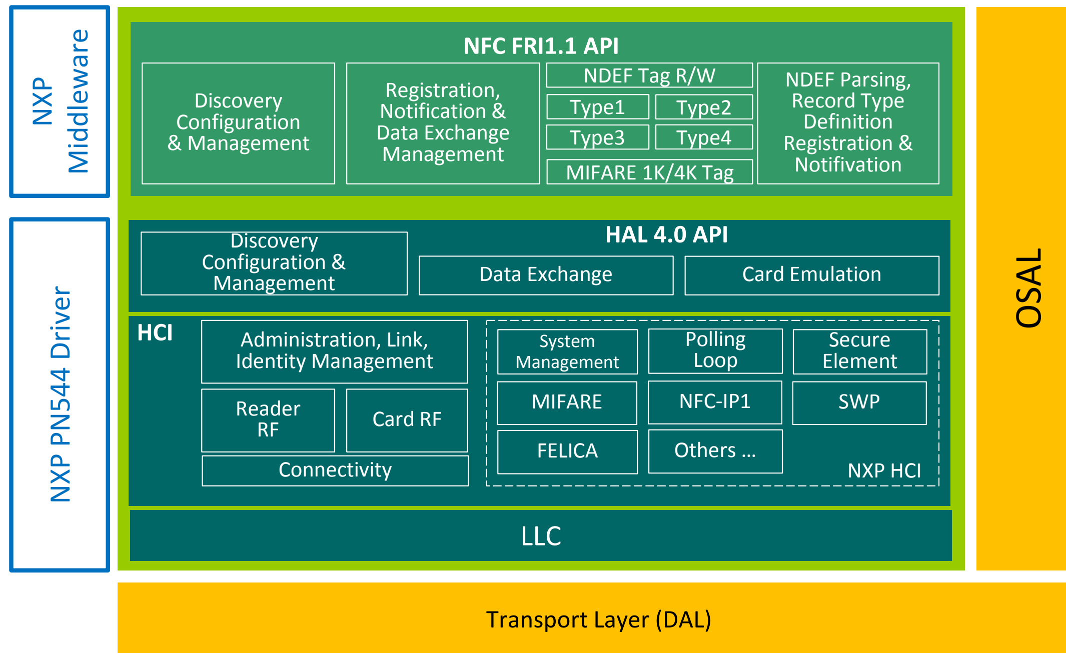


Figure 3.5: FRI and HAL Architecture [NXP11c, p.17]

It illustrates the route from the application level to the NXP Forum Reference Implementation (FRI). This is a software stack for controlling the NFC chip over the Host Controller Interface (HCI). As you can see in figure 3.5 it already contains the functionality of the card emulation (HAL level). This stack is part of the open Android source code as well.

A more detailed inspection of the source code shows that the functions for the activation of card emulation over SWP and NFC-WI are already routed up to the NativeNFCManager class which is part of the application framework. These interfaces could be enabled during the activation sequence of the NFC chip. This activation sequence is as followed:

1. Get hardware Module
2. Open PN544
3. Initialize Driver
4. Initialize Stack
5. Get Stack CAPABILITIES

6. (Update Firmware)
7. Update EEPROM settings
8. (Activate SECURE ELEMENTS)
9. Configure LLCP and P2P mode

The steps in brackets are not performed at every initialization or not activated at all.

As the activation of secure elements is already prepared and the activation of Host Controller Emulation (HCE) is similar to this the NativeNFCManager class seems to be a good starting point.

When looking closer into the source code of the NFC stack it comes up that there is already a class prepared for the HCE that can be extended. This class sets up on the HCI communication as described in the ETSI Standard [ETS08].

3.7.2 Stack Adoptions

After the analysis of the current NFC stack structure has shown a possible point to build on the required action for reaching the goal of performing the card emulation on the base band processor over the host controller interface have to be brought to the plan. They can be separated into three different steps:

1. Activation and configuration of the host interface during the enabling sequence of the NFC controller.
2. Adoption and implementation of the actual authentication protocol function.
3. Routing the data through the different software layers.

For achieving the first point a communication pipe between the host controller and the RF frontend of the PN544 has to be established. As described in the ETSI Standard [ETS08] the pipe has to be set up between the “Type A Card RF Gates“ of the host controller and the Gate of the PN544. This pipe then can be used to set the Gate register on the PN544 side. These register entries control the ISO 14443 Level 4 activation which is then performed by the NFC chip in an autonomic way. The sequence to enabling the gate and setting of the parameters can be seen in figure 3.6. This is the same procedure a UICC would use to establish a connection with the difference that it is not possible to set a specific UID if using the Host Controller Interface. Therefore in the following figure this step is set in brackets. In case of emulation over the HCI a random UID is used that changes every time the device enters the RF field of the reader.

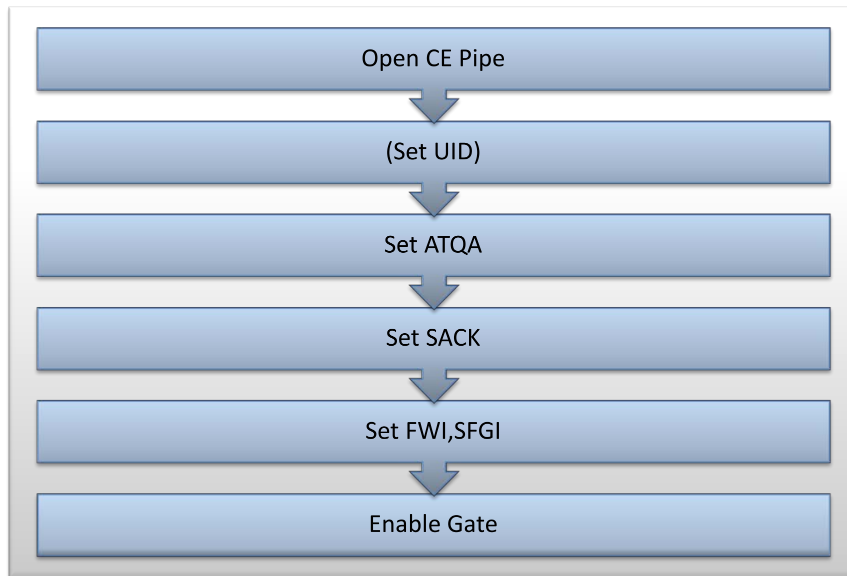


Figure 3.6: HCI Activation Sequence

If the Gate is enabled the communication has to follow the protocol as shown in figure 3.7. When PN544 detects the external reader field it notifies this to the host controller (Host) and performs the level 4 activation and again reports this. If the reader then sends an ISO 14443-4 command the PN544 Type A Card RF Gate rises an “send data“ event that contains the sent data.

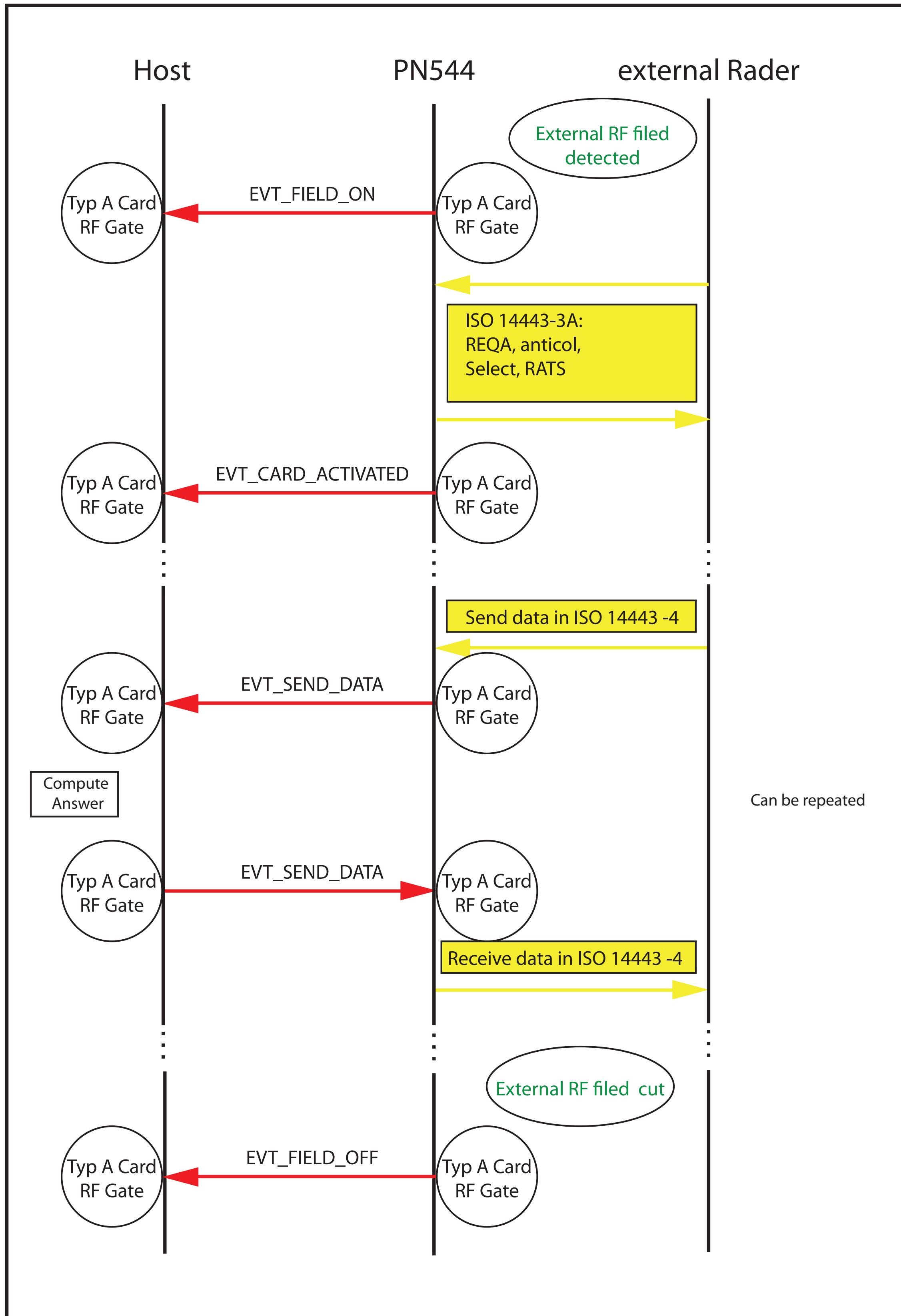


Figure 3.7: HCI Protocol for Host Emulation

When receiving this event the host can compute an answer and send it back with another “send data“ event on the pipe. The PN544 relays this data to the reader by load modulating the reader field. This can be repeated until the reader field gets cut. The computing of the answer is the actual point where the emulation functionalities for the AES authentication protocol come in.

3.7.3 Implementation Levels

The actual setup of the android operating system allows more than one option for placing the emulation functionality.

The first possible implementation is to add the emulation code to the NFC stack. This will probably be the fastest PoC solution, because it does not require to pass the data though the different software layers of the OS. As the stack is written in C it is possible to reuse the code of the C implementation. The disadvantage of this kind of implantation is less flexibility and no chances at system run time. This might be drawback for future application. To differ from the other implementation this will be called the PoC Stack implementation.

The second approached will be on application level and will use the previously described Java and JNI Applications. This requires to route the data through the whole OS stack. The base of these implementation is an adoption of the Android API to provide the functions to send and receive the data on this level. The two resulting implementations will be called PoC Java and PoC JNI implementation or PoC application level implementations if referring to both.

As the NFC chip sends the data asynchronously when receiving data from the reader a certain mechanism is needed to trigger the computation. The Android OS therefore offers different mythologies to do this. On the one hand the intent approach and on the other the a callback method. Both of them get implemented for comparison.

The first and simpler method is to use a messages object called intents. They are used to pass data between activities (the application) and services (in this case the NFC Service). The data is attached to this intent object and it is broadcast. With the use of filters this intents can be caught by the application and can trigger continuing actions. The problem of the approach is the large and varying time between the broadcast and reception of the intent as the measurements in section 5.5 will show.

The other possible way is to use a kind of callback mechanism. As Java does not support callbacks directly a workaround has to be used. Instead of registering a call back an addition interface is used that is implemented on application level.

In more detail this will be described in the implementation section 4.5.3.

3.7.4 Profiling and Timestamps

Now that the operative functionality of the proof of concept implementations are defined the focus comes to the definition of the profiling point positions. As the generation of timestamps in the program code produces distortion the number of them has to be as small as possible. Of course, all three PoC implementation concepts (PoC Stack, PoC Java and PoC JNI) require a different number of profiling points but with respect to the comparability of them, times on same software level will be named similarly. This means that the time between t0 and t11 will be the total frame delay time (FDT). t1 will be the time of receiving the data from the NFC chip on the I2C bus and t10 when sending the response on the bus. This also means that according to the different level of the implementations the name of the profiling points around the pure calculation of the authentication procedure will change.

As you can see in figure 3.8 these are the planned positions for the PoC Stack implementation.

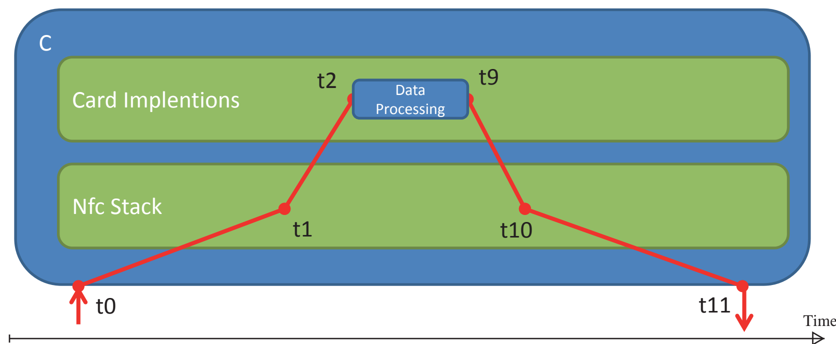


Figure 3.8: Schematic Position of Timing Points for PoC Stack

In the ideal case this number of profiling points would be sufficient for the other implementations as well but unfortunately the timestamps provide by the C function differ from those provided by Java functions. These circumstances require additional timestamps on the boarder between Java and C code as you can see in figure 3.9 for the PoC Java implementation.

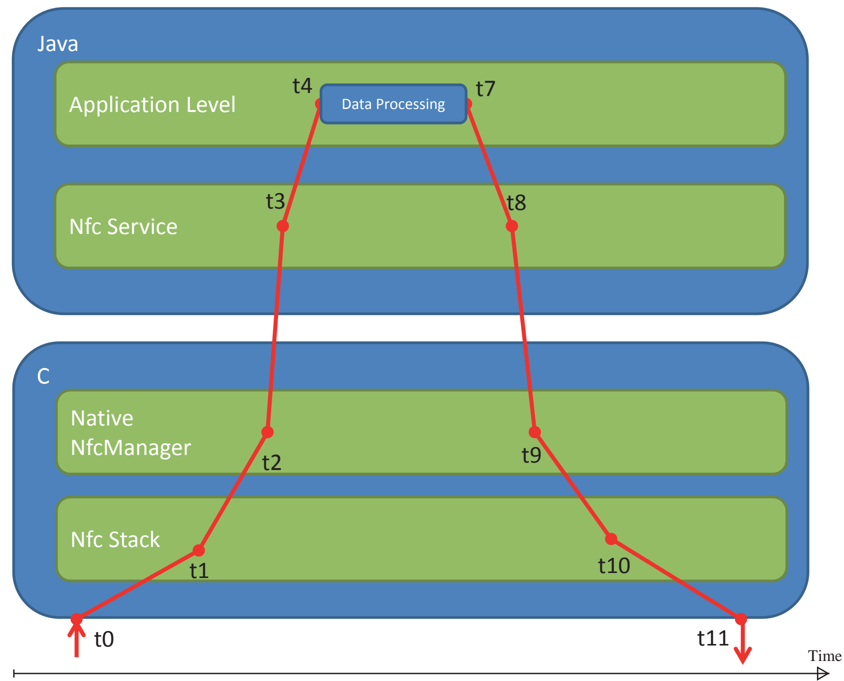


Figure 3.9: Schematic Position of Timing Points for PoC Java

As the PoC JNI implementation generates an additional Java / C boarder as you can see figure 3.10 the timestamps t5 and t6 are introduced.

Figure 3.10: Schematic Position of Timing Points for PoC JNI

More details concerning the used profiling functions and the differences will be described in section 5.1.3.

3.7.5 PoC Use Case

All of these previous parts of the proof of concept implantations will allow a good picture of the authentication performance but they are not sufficient for a practical use case. In order to be able to give estimations about the usability of this PoC implementations in a real case they are going to be extended to satisfy the requirements for the access management use case as described in Section 2.1.4.

Description

The goal of this implementation is to provide the full functionality that is used in access management systems to provide privileged access as seen in figure 2.9. This requires two additional commands:

Select Application Usually there can be more than one application on a smart card. With this command a specific one is selected e.g. the access management application.

Read Data The selected application can have a number of data files. The data in this files can be gathered with this command. For this use case the authenticity of the read data also has to be secured by a MAC that is generated with the session key derived during the authentication.

In access management the credential should identify the person it belongs to. Therefore the data responded on a read command has to contain a user ID. To demonstrate that this ID can be changed on the fly the User Interface has to be extended by an input field for this user ID.

Chapter 4

Implementation

The following chapter covers the practical implementation of the designed parts. In the first section the actually used hardware is described more closely followed by an overview of the used developing environments and tools. The remaining sections will illustrate how the actual implementations were done and how they look like in the end.

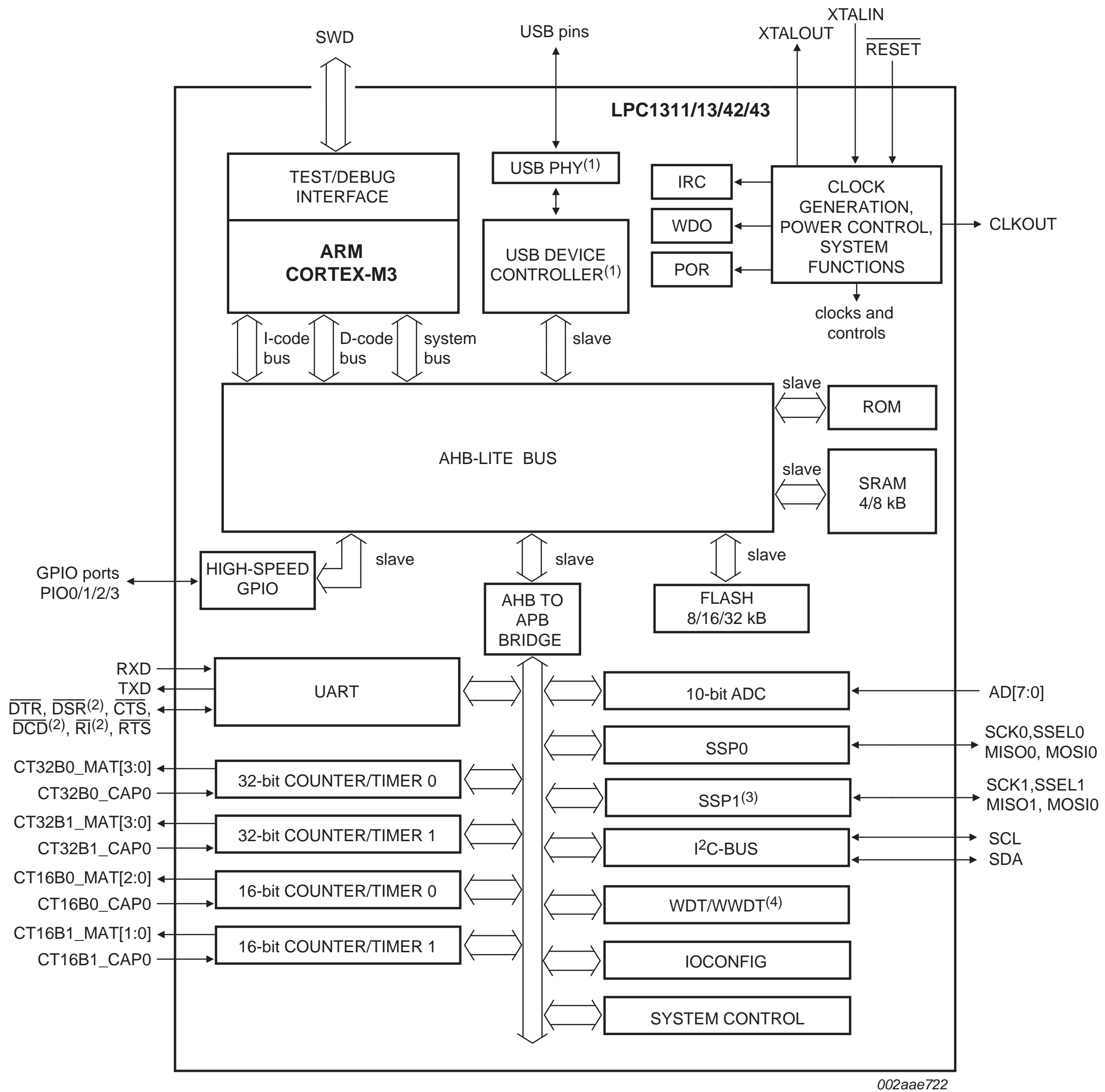
4.1 Used Hardware Platforms

For hosting the authentication protocol two devices were used. As mentioned in section 3.2.2 the designated hardware for the C implementation is the LPC1343 LPCXPresso board. And as a NFC phone platform the Google Nexus S was chosen. Therefore these devices are described now briefly.

4.1.1 LPC1343 LPCXPresso board

The LPC1343 LPCXPresso board is like the rest of the LPC13xx LPCXpresso family a combination of an ARM Cortex-M3 target and a JTAG Debugger interface called LPC-Link^{TM} on one board. This device which jointly developed by NXP, Code Red, and Embedded Artists gives combined with the LPCXpresso IDE a good development platform. The schematic structure of this target platform and its various in and output interfaces can be seen in figure 4.1.

As the name indicates the processor on this board is a NXP LPC1343. This is running at frequencies of up to 72 MHz and it has 8kB SRAM. The program code is stored in a 32kB flash memory and the test and debug interface allows direct control of the core register.



(1) LPC1342/43 only.

(2) LQFP48 package only.

(3) On LPC1313FBD48/01 only.

(4) Windowed WatchDog Timer (WWDT) on LPC1311/01 and LPC1313/01 only.

Figure 4.1: LPC13xx Block Diagram [NXP11b, p. 7]

4.1.2 Nexus S

The Google Nexus S was the first Android device that supported NFC and was released in November 2011. It was manufactured by Samsung Electronics in four different versions mainly differing in the used display and mobile radio (3G and 4G). The version used for this evaluation has the product identification GT-I9023 and is original shipped with Android 2.3. (Gingerbread) but was updated to version 4.0.3 (Icecreamsandwich) during this work.

Hardware

The basic parameter of the Nexus S hardware which are relevant for this work are the following:

- 1 GHz ARM Cortex-A8 Hummingbird Single-Core Multimedia Applications Processor
- 512 MB total RAM
- 16 GB internal storage
- NXP PN65 Nfc chip

NFC Chip

As described in section 2.1.3 the PN65 is a package of SmartMX and PN544 which is designed for use in mobile phones and portable devices. Some if its key features are [NXP10b]:

- Support for variety of RF protocols
- Integrated power management unit
- Simultaneous multi cards management (ISO14443-A,B,B', MIFARE)
- Multiple interfaces (Serial UART, I2C, SPI, SWP, NFC-WI)

Operating System

Android is a Linux kernel based operating system for mobile devices as well a software platform for applications. Each of this application runs on a separate Virtual Machine (VM) called Dalvik VM which supports Just In Time (JIT) compiling since version 2.2. The NFC support was introduced in version 2.3.

4.2 Used Developing Tools

For implementing the AES authentication protocol on the different platforms a number of developing tools were required. It starts with the LPCXpresso IDE to do the C Implementation on the LPC1343. Then it goes on with the Eclipse IDE that provides together with the Android Source Developing Kit (SDK) and the Android Native Developing Kit (NDK) an environment to do the Java and JNI implementations on the actual phone. To do the changes on the Android source code a virtual Linux machine was required and for the functional and FDT measurement tests Microsoft Virtual Studio with the Testbench framework was used. Therefore these programs and the use versions are described briefly.

4.2.1 LPCXpresso IDE

The LPCXpresso IDE is specially designed for the NXP LPC microcontroller series and is powered by Code Red Technologies. It bases on an adapted Eclipse platform and provides a C programming environment that uses an "industry-standard GNU tool chain with an optimized C library" [NXP11a].
All other necessary libraries and examples can be downloaded on the NXP homepage.
The effort to set up this environment keeps low which is as well a reason for taking the LPC1343 Board. The only thing to do is to run the setup, connect the device via an USB cable and run the compiled code.

4.2.2 Eclipse

Eclipse is an open source IDE developed by the Eclipse Foundation. It is widely used for Java programming and also provides plug-ins for other programming languages. To be able to do Android application programming an additional plugin called Android Development Tools (ADT) is required .This allows Eclipse to create Android project and make use of the Android Source Developing Kit (SDK) and Android Application Framework API.

Android SDK

The Android Source Developing Kid is a set of software developing tools. It provides all tools for a successful developing of applications e.g. libraries, debugger, sample code and emulators. It was developed by Google and is available for free.

Android NDK

The Android Native Developing Kit allows in combination with the SDK to develop in C or C++. It contains all required components to build this code for sev-

eral instruction sets including ARMv5TE, ARMv7-A, ix86 instructions and MIPS [Goo11b] and it is available for free as well.

4.2.3 Visual Studio

Visual Studio is an IDE from Microsoft. It can be used for developing in languages based on the .net framework and supports for example C#, C, C++ and Visual Basic. Visual studio was mainly used to write the FDT measurement test cases in the Testbench framework.

4.2.4 Testbench

For controlling the ISO Setup as described in section 5.1.1 a software framework was used called Testbench. This built up on the NXP Reader library and allows flexible definition of automated test cases in C# to verify RFID Cards . The execution of these tests is then controlled by NUnit, a unit-testing framework for .Net languages.

4.2.5 Android System Build Environment

To make a build of the Android source file Linux or Mac OS is required. To avoid the effort of setting up an extra Linux system can be bypassed by using a virtual machine (VM). The VMWare^{TM} Player of VMWare Inc. is able to host a Linux system image. There already exist system images where the Linux environment is configured correctly for doing an Android source built. Such an image is provided by the company Marakana and is usually used for training purposes. The actual Linux VMWare image used in this work contain a 64Bit Ubuntu distribution. To be able to do a full system built it is recommended to equip the VM with at least 4 GB RAM.

If the virtual machine set up is done a number of steps have to been taken to come to a success running system on the phone:

1. **Download Android Source Code**
   The source code of all existing android versions can be downloaded from a git repository.
2. **Built System**
   (a) **Select make configuration**
   For the Nexus S the required configuration is “Crespo user debug“.
   (b) **Do make**
   A full build takes several hours
   (c) **Copy binaries to build output**
   Some libraries are not part of the open source code and have to be added after the compilation.

3. **Copy System to device**
   (a) **Unlock boot loader**
   To be able to flash the system the boot loader has to be unlocked as described on [Goo11b].
   (b) **Restart to boot loader** In the boot loader environment the built code can be sent to the device.
   (c) **Write the new system to the flash memory**

After a restart the phone is booting the new system.

It is recommended to do all of these steps before making changes in the source code to have a working configuration. Additionally a rebuilt after modifications are done will than take just some minutes.

For doing the adjustments in the Android source code the Eclipse editor was used as well.

4.2.6 Used Versions of Developing Tools

The used program versions of the development tools are the following:

LPCXpresso V 4.0.6

Eclipse Version 3.7 Indigo , ADT 17.0.0

SDK Revision 17

NDK Revision 7b

Visual Studio 2010

NUnit V 2.5.9

VM VMWare^{TM} Player 4.0.2 and Ubuntu 11.04 64Bit

4.3 C Implementation

The C implementation was designed to show the basic performance of the protocol implemented in software. It gives early information if such an implementation would make sense at all. This implementation comes together with the decision for a certain AES implementation. As described in section 2.2.2 there are already some existing implementations. A short researches about the availabilities of the source code of these implementations showed the code of Gladmann [Gla11] seemed to be the perfect choice. It fulfils the requirement of availability, is well documented and is one of the fastest implementations according to [Sha06].

4.3.1 Gladman Implementation

The AES implementation by Gladman [Gla11] is a good performing C implemtation. In [Sha06] this is described as followed:

Efficient use of macros and interweaving of instructions for the decryption round key setup allow for the excellent performance of this implementation.[Sha06]

It is designed to operate on 32bit words and has a very modular setup that allows optimizing for the target environment and is controlled by a number of compiler options. These options give the possibility to e.g. enable assembler support for a number of targets, unroll the round loops or use lookup tables for the round functions.

4.3.2 Structure of Implementation

The structure of the actual protocol implementation was kept very simple. It contains an initialising part where encryption and decryption structures were set up. For reasons of simplicity the implementation does not deal with more than one stored key and therefore the key schedule can be done in this part as well. The other computation blocks of reader and card are grouped in methods and get called by the main routine. In order to get the optimal performance, different compiler settings and build options of the Gladman implementation were used. These are the only optimization measures because the option to write code parts in assembler would not be available for the JNI implementation on the phone.

4.4 Application Level Implementations

4.4.1 User Interface

The interface bases on an xml file design this Android XMLdesigner. As you see in figure 4.2 it will not win a design price but it is sufficient for the purpose of this work and fulfils the requirements of section 3.6.1. Additionally the input field for the use case implementation is appended.

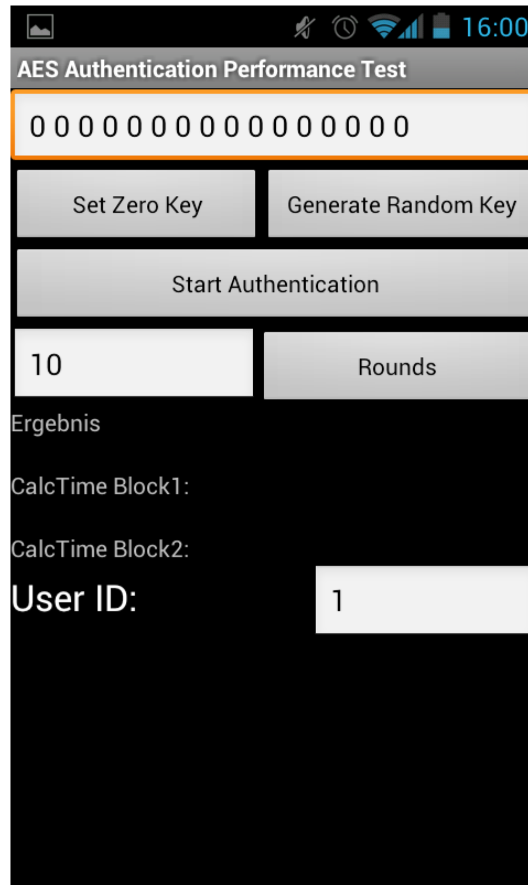


Figure 4.2: Screenshot Graphical User Interface

The event handling of the interface elements is done in the Main Application Class.

4.4.2 Java Implementation

Used Java Methods

Java provides a very rich portfolio of crypto functions. So it was obvious to use library methods for the AES de- and encryption. It has to be mentioned that this holds an uncertainty because there is no information about the actual implementation of this library function available and so they might be implemented in C as well.

Structure

As described in section 3.6.2 the Java implementation is structured quite simple. An actual class diagram can be seen in figure 4.3

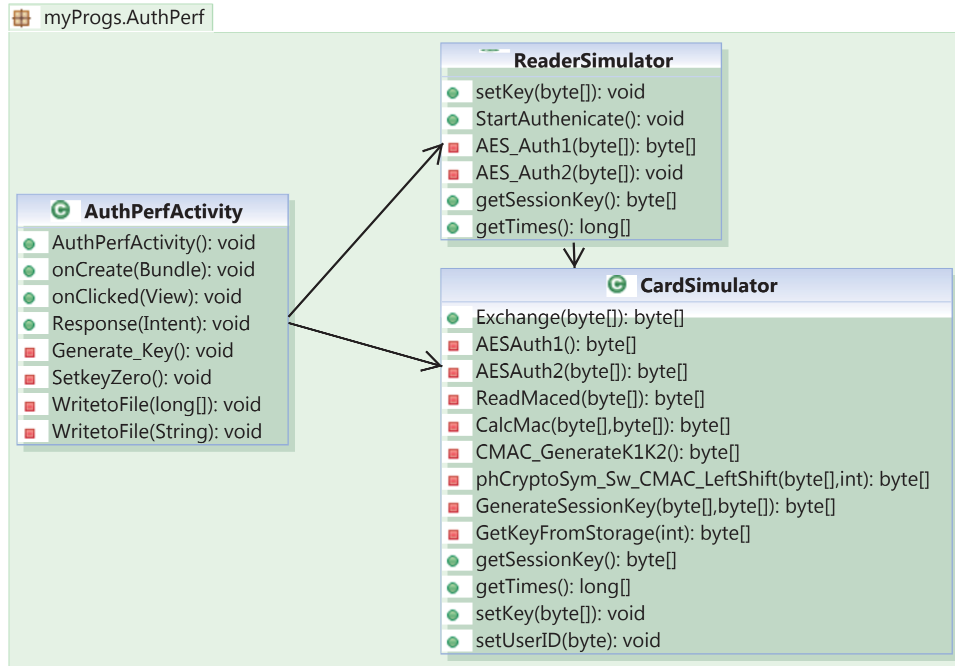


Figure 4.3: Application Class Diagram of Java Implementation

The authentication is then performed on a button click as seen in figure 4.4. Additionally you see in this figure that a method to write the measurement result to a file is added. This was done in order to make the evaluation easier.

Figure 4.4: Sequence Diagram of Authentication with Java Implementation

4.4.3 JNI Implementation

For the implementation with the Java Native Interface the C implementation could be totally reused. The only things to add were the profiling functions and the JNI calls from the Java code. The rest of the application structure is quite similar to the Java implementation with the exception that no separate card emulation class was used any more. With respect to simplicity the JNI functions were called directly from the application class as the class diagram in figure 4.5 shows. The authentication

flow is similar to the Java implementation.

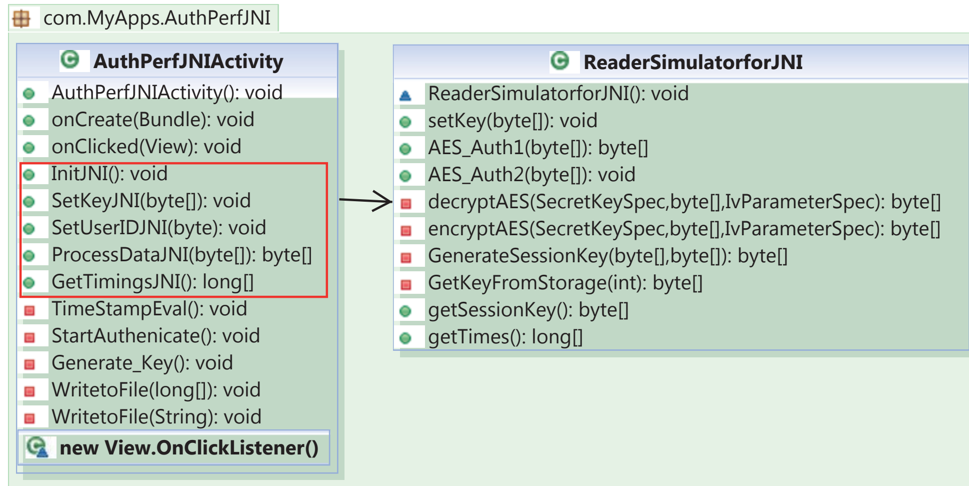


Figure 4.5: Application Class Diagram of JNI Implementation

4.5 Proof of Concept Implementation

4.5.1 Stack Adaptation

The essential point of the proof of concept is the enabling of the host emulation feature. This requires changes on certain places in the Android source code:

\external\libnfc-nxp
This is the location where the NFC stack is implemented.

\device\samsung\crespo\nfc
Contains the hardware setting files for the Nexus S (code name Crespo).

\framework\base\core\java\android\nfc
Here the relevant interface definition files are located.

\packages\apps\Nfc\java
Hosts the NFC service files with the interface implementations in Java and C++.

The first point on the list is to enable the Host RFGate A on the PN544. Therefore an additional EEPROM setting has to be added to the nfc_hw.c file in the hardware setting directory. These settings are written to the controller during the initialization routine as described in section 3.7.1

Next a pipe between the host RFGate and the PN544 RFGate has to be established. As mentioned in section 3.7.1 the entrance point for this is the com_android_nfc_NativeNfcManager.cpp file where the highest level of the initialization routine is implemented. During this routine a list of the available Secure Elements is requested from the library to get a handle on them. Now to activate the host emulation a virtual Secure Element type called phLibNfc_SE_Type_HOST including a handle on it was created in the NFC stack (phLibNfc_SE.c). With this handle the phLibNfc_SE_SetMode() method can be used to route the initialization as seen in figure 4.6.

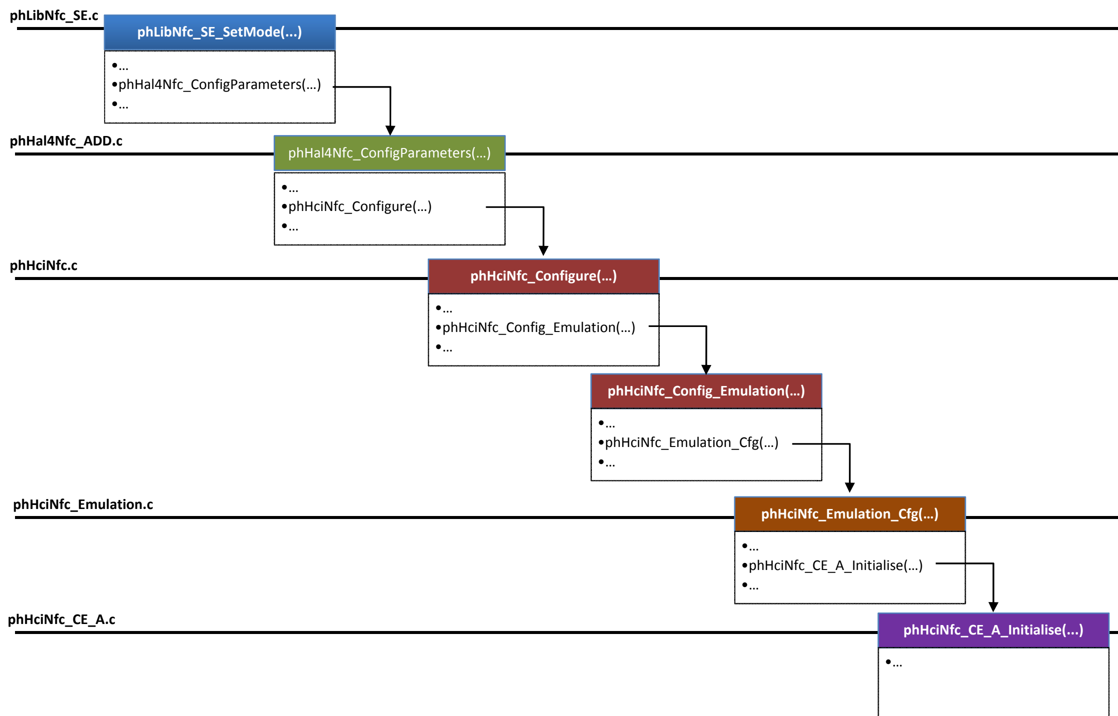


Figure 4.6: Call Trace for Initializing Host Emulation

The phHciNfc_CE_A class already contains a framework to do the initialization. This was extended according to the sequence described in section 3.7.2. The actual gate register parameter for this can be seen in table 4.1.

Register	Value	Comment
UID	-	The PN544 doesn't support the definition of an UID over the Host Controller Interface. Therefore a random UID is used.
SAK	0x39	Allows the reader to identify the card type. This value is taken to differ from other PICCs.
ATQA	0x0004	Default value
FWI,SFGI	0xE8	The first half byte is the FWI and the second the SFGi. FWI is set to max. value, SFGI to default

Table 4.1: Card Emulation Type A Register Values

The Card Emulation Type A Gate allows even more parameters to be set e.g. CID_Support but as they are not important for this implementation they are left at the default value.

After this initialization the PN544 should have been able to response on a Request command of a reader. Unfortunately this was not the case because the NFC controller did not accept the enabling of the pipe. So an additional step had to be taken.

With the NP544 it is not necessary to do a pipe configuration at every start up. Usually it just has to be done once and then this configuration is stored at the chip. Therefore it was not done for all the other pipes at a usual initialization of the NFC controller and this blocked the enabling of the host emulation pipe. To solve this problem a full initialization was forced by the use of existing compiler switches in the library source code.

After this full initialization the Nexus S was able to successfully pass ISO 14443-4 activation with the configured parameters.

4.5.2 PoC Stack Level

Now that the host emulation is activated the actual functionality has to be implemented. Obviously therefore the easiest way is to use the C implementation and add this to the NFC stack. Additionally the received data from the NFC chip has to be routed to the implementation and has to be sent back to the chip after processing them. Fortunately the catching of the receive event is already implemented in the source code and it is only necessary to pass the data via a method call to the C

implementation where the command is identified and the according authentication part is called. Therefore the communication data blocks including the commands can be seen in figure 4.7 for the case of an successful authentication. After processing the data they can be sent back by raising an HCI SentData Event.

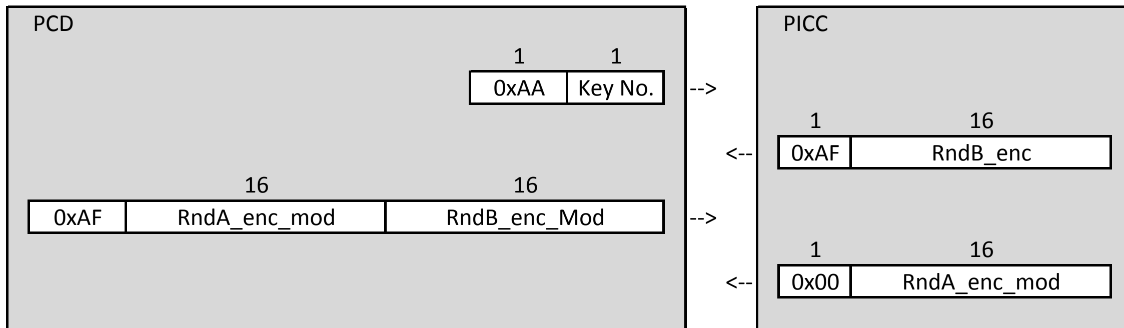


Figure 4.7: Structure AES Authentication Communication Blocks

4.5.3 Routing

For the PoC application level implementations it is required to pass the data from the HCI level to the application layer and back after the processing. How this is done is shown in figure 4.8. One important part of this work was to add the RawSendCE() methode to NfcAdapter interface to make it available from the application level. The other was to register the method signature of the C++ function manually to make it callable from the NativeNfcManager.java class and reverse. For the doRawSendCE() method this signature looks like this:

{“doRawSendCE“, “([B)V“, (void *)com_android_nfc_NfcManager_doRawSendCE}

The next point is to implement the notification of the receive event. As already explained there are two possibilities to implement this and because the easier method showed very bad performance and for comparison the second got implemented as well.

Intent Notification to Application Level

As mentioned earlier the first method used is the intent mechanism where the data are transmitted by broadcasting. An intent is an asynchronous messages that has an signature with which the application can identify and catch it. It is easy to implement because for doing this just a new Intent containing the data has to be created and be broadcast.

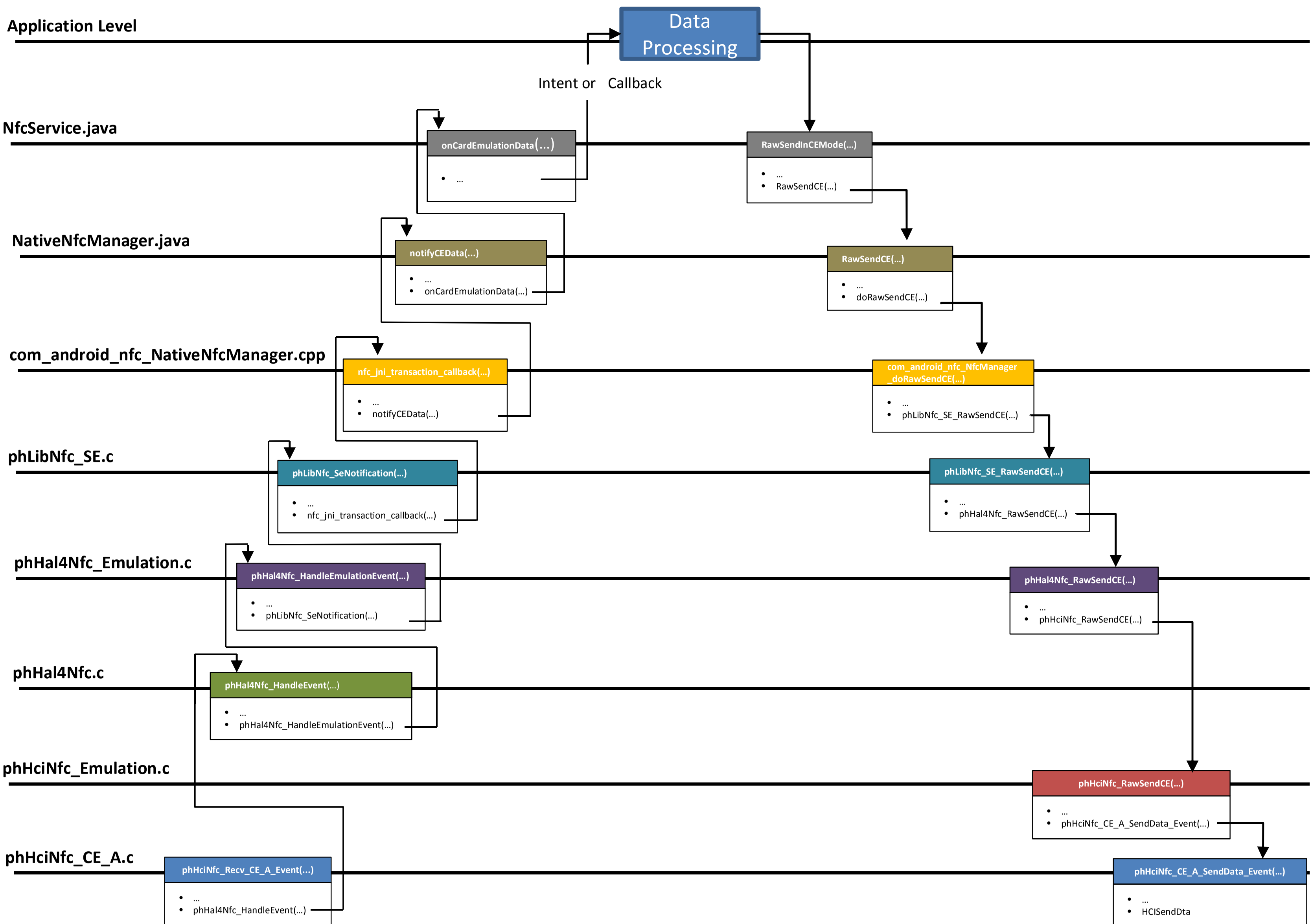


Figure 4.8: Data Flow after HCI Receive Event for PoC Application Level Implementations

Callback Notification to Application Level

Implementing a callback in a Java environment is more tricky because opposite to C and C++ it does not work with passing function pointers. The designate way to do something comparable is to take the indirection over an interface. For this work an interface called INfcCENotifier was generated. It provides a declaration for a ProcessData() method which than can be called from the NfcService.java class. The actual implementation of this method is then done in the application.

The final implementation was done in a way that if the callback is not implemented in the application the intent mechanism is used. This allows an easy switching between them without recompiling the android source code.

4.5.4 SDK Adoption

The next step is to add the newly designed API to the Android SDK in order to make it accessible from the Eclipse built environment. The Android SDK Eclipse plug-in contains information about the available API from the android.jar file of each platform. In order to not destroy a platform it is better to copy the desired platform files and create a new one. In this case the desired one is called platforms/android-15 (for Android 4.0.3) in the SDK source directory. This step is similar to the procedure to uncover hidden Android API methods.

To get the interface information in the needed format the class files that containing the new API can be taken from the Android source compiler output which is located in /out/target/common/obj/Java_LIBRARIES/...
.../android_stubs_current_intermediates/classes/android/nfc/ in the Android source base directory. Here you can find the files INfcCENotifier.class,
INfcCENotifier$Stub.class and NfcAdapter.class with the modified and new API. These files have to be copied to the android.jar of the new platform. More precise: to the subdirectory /android/nfc. To allow Eclipse to distinguish between the new and the original platform the file "build.prop" in the platform directory has to be modified as well. Changing the line ro.build.version.sdk=15 to ro.build.version.sdk=-15 and ro.build.version.release =4.0.3 to ro.build.version.release=4.0.3.mod will do this. Now the target platform can be changed in Eclipse to the new customized platform and the newly created API is accessible.

4.5.5 PoC Application Level

After the received data is routed up to the application API and has been made accessible it can be used from an application. For the functional part the previous Java and JNI implementations could be totally reused but need some add-ons. For the intent method a class is required that extends the "BroadcastReceiver" class. And with the Callback approach the newly created interface has to be implemented.

As described earlier both methods are implemented simultaneously to make a fast switching possible. This is reflected in the class diagram for the PoC Java implementation as seen in figure 4.9.

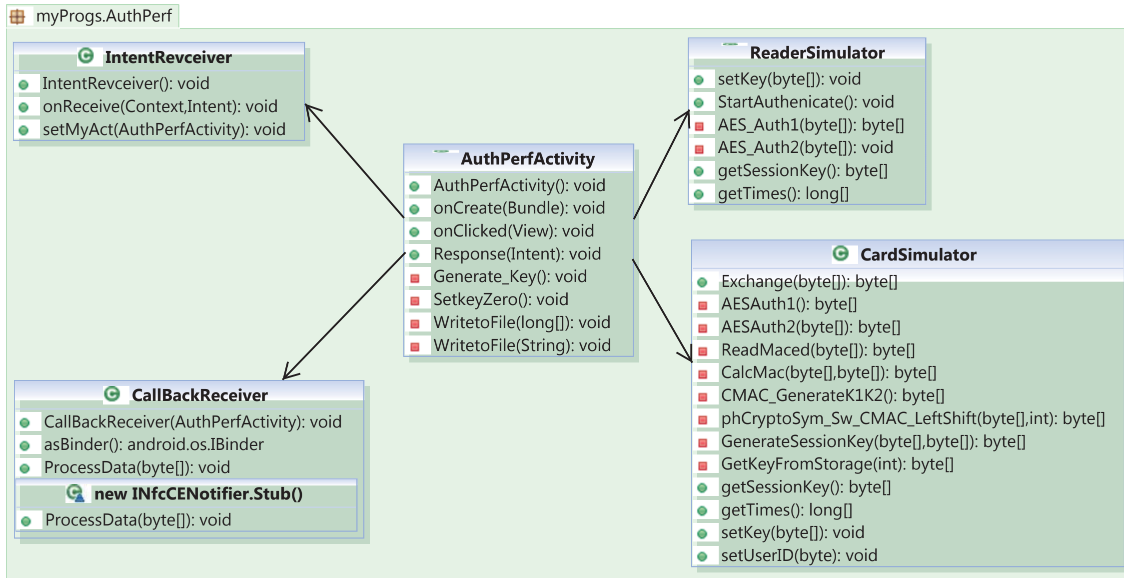


Figure 4.9: Class Diagram of PoC Java

The resulting execution sequence for the PoC JNI implementation on application level is visualised in figure 4.10.

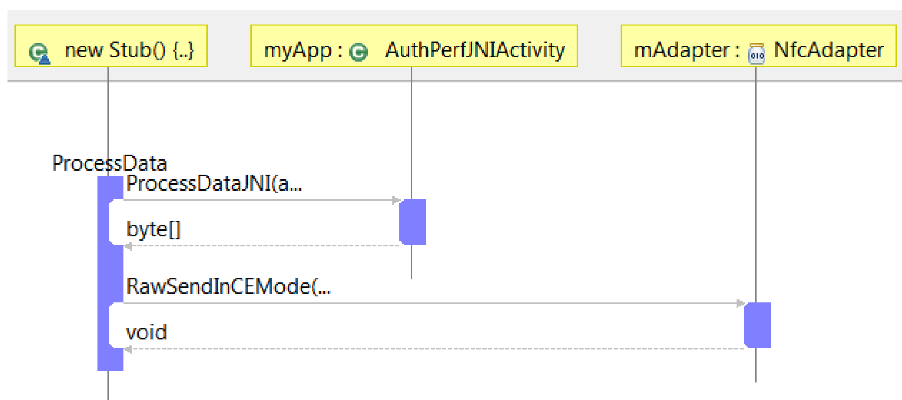


Figure 4.10: Sequence Diagram for PoC JNI with Callback

4.5.6 PoC Use Case Implementation

Based on the previously PoC implementations the extension for the access management use case is done quite straightforward as the basic framework is already set up. The additional functions are implemented as described in section 3.7.5.It is important to mention that the response on a Select Application Command is always 0x00 which stands for success as it can be seen in figure 4.11.

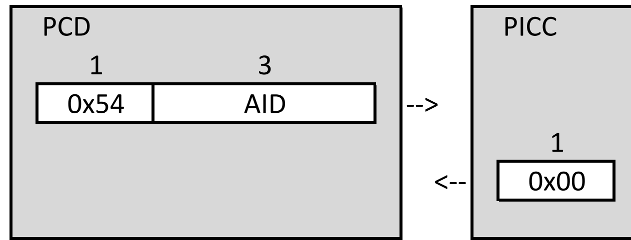


Figure 4.11: Structure of Selcect Application Command and Response

For securing the message authenticity for the read operation a 8 CMAC with the session key has to be calculated. Thereby the padding is done according to ISO/IEC 9797 section 1, padding method 2 and the CMAC is calculated after NIST special Publication 800-38D. For this block cypher method the CMAC of the received command has to be calculated as well to work as an initial vector for the CMAC calculation for the actual data to response.

The implemented response on a read command is a 32 byte data block that contains a user ID and has space for other informations e.g. a "card personification" timestamps as it could be used in a real access management system.The structure of the response is illustrated in figure 4.12. As mentioned for the PoC application level implementations the actual user ID is changeable on runtime via the user interface. This would be possible for the PoC Stack version as well but the additional effort would not bring any additional findings.

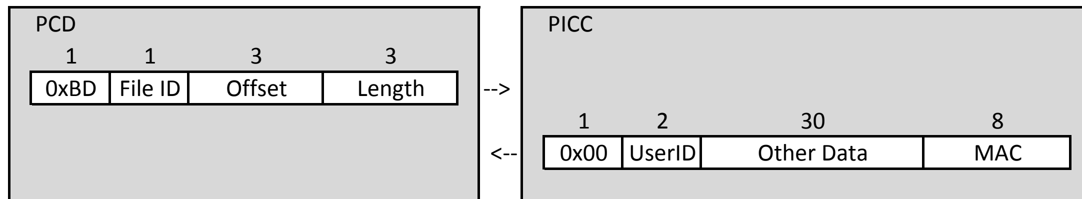


Figure 4.12: Structure of Read Command and Response

Chapter 5

Experimental Results

This chapter starts with a description of the used measurement equipment. It continues with the used profiling functions and measurement corrections. Then presents the actual measured values of the comparison devices and of the previously created implementations. Finally the security impact of this result is highlighted.

5.1 Equipment and Tools

For this work two kinds of measurement tools were used. An external device in form of the ISO Setup and for the internal measurement the Android Debug Bridge (ADB) including the LogCat viewer for observing the output.

5.1.1 ISO Setup

The ISO Setup is a "*modular test bench allowing to combine higher layer protocol tests with analogue parameter variation in the contactless antenna arrangement as specified in the ISO/IEC 10373-6 test standard.*" [GBBM08]

Its principle components can be seen in figure 5.1. The Device Under Test (DUT) it placed on a special antenna arrangement. This consists of a reader (PCD) Antenna that is surround by two sense coils in Helmholtz configuration which allow an exact measurement of the electromagnetic field.

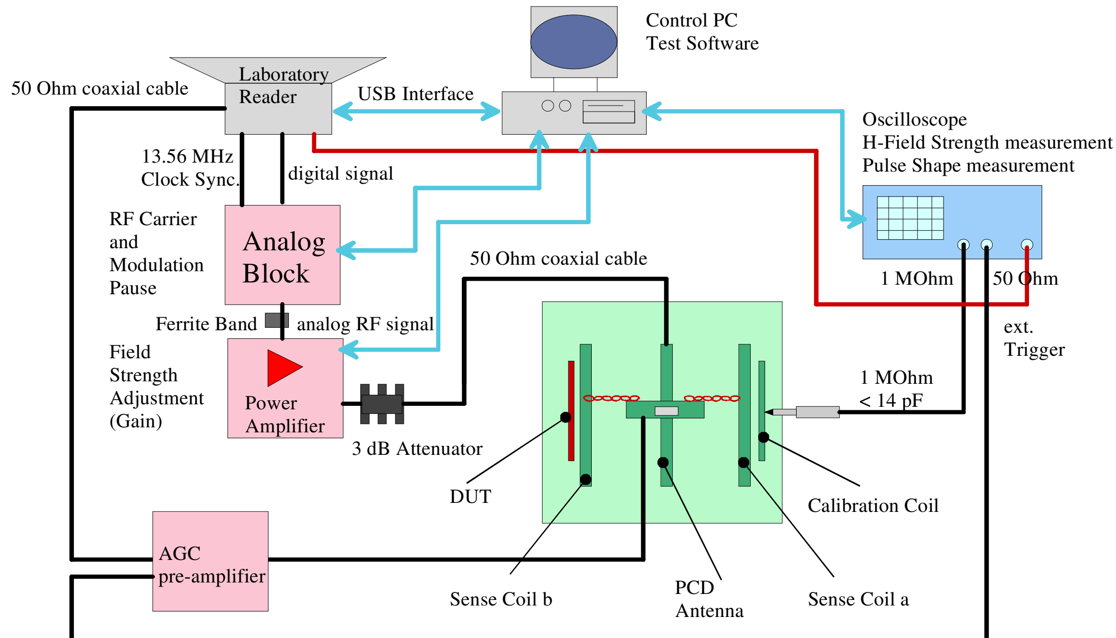


Figure 5.1: Complete Test Bench in Detail [GBBM08, p. 5]

For the measurements in this work a device that combines the Laboratory Reader and the Analog Block was used. This so called "PFGA Box" allows in combination with the Oscilloscope and the controlling PC with the test bench software an exact and automated measurement of the FDT. Beside this automated measurement the mean reason for using the ISO Setup is that it is nearly impossible to measure a period of several ms in the required resolution just on an Oscilloscope.
As mention in section 4.2.3 the used Testbench builds on a .dll written in C# which is used by the NUnit test environment.

5.1.2 Android Debug Bridge and LogCat

The Android Debug Bridge (ADB) is a client-server program that contains three components [Goo11a, c.f.]:

- A client, which runs on the development machine.
- A server, which runs as a background process on the development machine
- A daemon, which runs as a background process on the device instance [Goo11a, c.f.]:

Such a client is available in the ADT Eclipse plug-in and can receive messages from the debug log output. These messages can be collected and viewed with LogCat.

5.1.3 Time Measurement

To get the actual computation and communication durations three principles were used.

The first one is the use of an external device like the ISO Setup. This requires the existence of physical signals and in this case this is just useful for the overall time consumption from end of communication of the reader device to the start of the response sent by the card or NFC phone, which means to measure the total Frame Delay Time (FDT).
The second principle is used to gain performance information of the C Implementation running on the LPC1343. As the attached JTAG controller allows direct access to the CPUs clock register the differences of this value before and after the computation can be multiplied by the CPU clock to derive the duration. Thereby the CPU clock is assumed to be constant. These values can be gathered by the use of debugging brake points in the code.
The third principle is for application level and proof of concept implementation where profiling functions provided by the operating system were used. They were introduced into the code according to the timing points described in section 3.7.4. As in the proof of concept implementation some of these points are located in software levels which are not accessible through the Eclipse IDE they have to read by the use of the ADB servers log output. To minimize the influence of the time measurement on the computation time the profiling values were stored to static local variables and were handed over to the ADB daemon after the end of communication.

Under the assumption that the average durations of these functions from their start to the fixation of the profiling value (t_{start}) and the return of this value (t_{end}) are constant the time consumed by this profiling function (t_{pro}) only has to be subtracted from the difference of this two profiling points. In the following part the notation t'_x stands for the time returned by the profiling function and t_x for the corrected time value.

$$t_{prof} = t_{end} + t_{start} \tag{5.1}$$

$$\begin{aligned} T_{diff} &= t_2 - t_1 = t'_2 - t_{end} - (t'_1 + t_{start}) \\ &= t'_2 - t'_1 - (t_{end} + t_{start}t) = t'_2 - t'_1 - t_{prof} \end{aligned} \tag{5.2}$$

The outputs of two profiling function calls in sequence give the profiling duration.

$$\begin{aligned} t_2 - t_1 &= 0 = t'_2 - t'_1 - t_{prof} \\ \rightarrow t_{prof} &= t'_2 - t'_1 \end{aligned} \tag{5.3}$$

	Stack	JAVA	JNI
$Funcion:$	clock_gettime()	nanotime()	clock_gettime()
$t_p =$	2.4 us	4.9 us	7.8 us

Table 5.1: Profiling Time Consumption

Table 5.1 shows the time consumption of the profiling functions on the different software levels. These times have to be discounted for each profiling point. The table also indicates that for code parts written in C and Java different profiling functions were used:

- On Java level the system function nanotime().
- On C Level function clock_gettime(CLOCK_REALTIME, ...) .
- On the JNI the clock_gettime() function was used as well but it turned out that it ticks four times faster than it should. The reason for this could not be found but it could be corrected easily be dividing the measured values by 4.

The values in table 5.1 are the average values of fifty measurements and used to do a systematic correct of the values measured during the authentication. As the used profiling function for Java and C code provided different timestamps they had to be harmonised. Therefore the time difference at the C/Java border is equally distributed to the C/Java call and the Java/C call. The same is done for the Java/JNI border. The time of I2C communication is segmented according to the datareceived (B_r) and sent (B_s).
The whole formulas that were used to correct the measurements are the following:

$$t_0 = 0 \tag{5.4}$$

$$t_1 = (FDT - (t'_{10} - t'_1 + t_{p,c}))\frac{B_r}{B_s + B_r} \tag{5.5}$$

$$t_i = t_{i-1} + (t'_i - t'_{i-1} - \Delta_i)\ for\ i = 2, 4, 6, 8, 10 \tag{5.6}$$

$$t_i = t_{11-i} + (t'_i - t'_{11-i} - \Delta_i)\ for\ i = 7, 9 \tag{5.7}$$

$$t_3 = t_2 + \frac{(t'_7 - t'_2 - t_{p,c}) - (t'_6 - t'_3 + t_{p,j})}{2} \tag{5.8}$$

$$t_5 = t_4 + \frac{(t'_7 - t'_4 - t_{p,j}) - (t'_6 - t'_5 + t_{p,JNI})}{2} \tag{5.9}$$

$$t_{11} = FDT - \Delta_{11} \tag{5.10}$$

Delta	PoC JNI	PoC JAVA	PoC Stack
$\Delta_2 = \Delta_{10} =$	$t_{p,c}$		
$\Delta_4 = \Delta_8 =$	$t_{p,j}$		-
$\Delta_6 =$	$t_{p,JNI}$	-	-
$\Delta_7 =$	$t_{p,j} + 2t_{p,JNI}$	$t_{p,j}$	-
$\Delta_9 =$	$t_{p,c} + 4t_{p,j} + 2t_{p,JNI}$	$t_{p,c} + 4t_{p,j}$	-
$\Delta_{11} =$	$4t_{p,c} + 4t_{p,j} + 2t_{p,JNI}$	$4t_{p,c} + 4t_{p,j}$	$4t_{p,c}$

Table 5.2: Correction Values for Profiling

5.2 Devices for Comparison

Procedure

These devices were measured at the ISO Setup and the average of 50 measurements is presented in table 5.3.

Results

Device	PICCBlock1	PICCBlock2
	[us]	[us]
Reference PICC	2567	2640
JCOP+SmartMX	6221	7992
PN544 Demo board + Gemalto SIM	37293	372*
Turkcell T20 + Gemalto SIM	37389	367*

* Not total FDT

Table 5.3: Measured FDT of Comparison Devices

As expected the referent PICC is the fastest solution. If looking at the table 5.3 you might negate this proposition because of the * marked values. For example in case of the SIM solution on the PN544 Demo board where the PICCBlock2 a value of 372 us is indicated. This value does not represent the total FDT because the device requests a Waiting Time Extension (WTX). During the L4 Activation this device requests a Frame Waiting Time of ~77,33 ms which means that the really required FDT for the PICCBlock2 probably exceeds this value but cannot be determined more precisely because the reader has to acknowledge the WTX command and this depends on the performance of the reader. The marked values in the table therefore just indicate the measured time between the WTX acknowledge and the response.

All together you can see that the hardware implementation in form of the reference PICC is at least 20 times faster than the SIM solution.
Additionally it can be said that the FDT values of the repeating measurements did not differ much for the SIM solutions. The maximum standard deviation lies beneath 2 % for all devices. This does not hold for the SmartMX on JCOP where the deviation is much higher.

5.3 C Implementation

Procedure

The LPC1343 allows direct access to the ARM processors core registers. Therefore the computation time can be easily extracted from the clock count register. As GNU tool chain of the PLCXpresso Studio provides a set of compiler optimization settings and the used Gladman implementation offers a variety of compiler switches to optimize the performance the computation time of the different settings is presented in table 5.4 and table 5.5.

Results

Table 5.4 shows the computation afford with the different compiler settings. They go from 0 for “no optimization“ to s for “optimized for size“. It can be seen the optimization level 1 could improve the performance dramatically compared to “no optimization“ but higher optimization levels even shrink this benefit. Therefore this setting is used for the further measurements.

Compiler Setting	PICCBlock1	PICCBlock1
	[cycles]	[cycles]
0	12664	42603
1	2515	10182
2	2810	12174
3	2923	24362
s	2700	10811

Table 5.4: Computation Cycles for Different Compiler Settings

In table 5.5 the influence of the settings made with the Gladman implementation as described in section 4.3.1 are shown. The first column indicates if the code for the AES crypto rounds is unrolled and the second if pre calculated lookup tables were used. The figures stand for the number of tables in the following order: for normal encryption round, for last encryption round, for normal decryption round, for last decryption round.

Enabling all tables and do a full code unroll was not possible because this blows up the code size too much and it did not fit it the LPCs 32k flash memory any more. But this does not seem too be a problem as the best performance was achieved with one table each.

unroll code	Lookup Tables	PICCBlock1	PICCBlock1
		[cycles]	[cycles]
no unroll	4,4,4,4	2048	7215
part unroll	4,4,4,4	1939	6906
full unroll	1,1,1,1	1808	6537
full unroll	4,1,4,1	1908	6871

Table 5.5: Computation Cycles for Different Number of Lookup Tables and Code Unroll

5.4 Application Level Implementations

Phone state conditions

On the phone the computation performance does not only depend on the power of the processor but also on the system state and the running background processes. To make this state as defined as possible all measurements were performed on the same conditions:

- Only applications of the Android 4.0.3 source code as fetched from the repository are installed plus the particular implementation to be measured.
- The device is restarted before each measurement cycle and the measurement is started after a break of a least one minute to ensure that the boot process has been finished
- Wi-Fi, GPS, Bluetooth and Mobile data are switched off, NFC is on
- The setting of the Developer options different to default are as followed:
  USB Debugging: on;
  Stay awake: on;
  Don't keep activity: on

Procedure

Similar to the reference devices 50 samples were drawn in order to evaluate the performance of the application level implementations . In addition the findings on

the Gladman options with the C implementation are tested to show if they hold for the JNI implementation as well.

unroll code	Lookup Tables	PICCBlock1	PICCBlock1
		[us]	[us]
full unroll	1,1,1,1	47	76
full unroll	4,4,4,4	61	141

Table 5.6: Computation Cycles for Different Number of Lookup Tables with JNI Implementation

Results

The measurements presented in table 5.6 confirm that the configuration “full unrolled and one lookup table each“ is the fastest and therefore is used for the rest of the measurements.
The average calculation time for PICCBlock1 and PICCBlock2 with the Java and the JNI implementation are shown in figure 5.2. It can be seen easily that the JNI implementation is about 40 to 50 times faster than the Java correspondent, but it has to be mentioned that this just reflects the pure computation time and no calls from an upper layer.

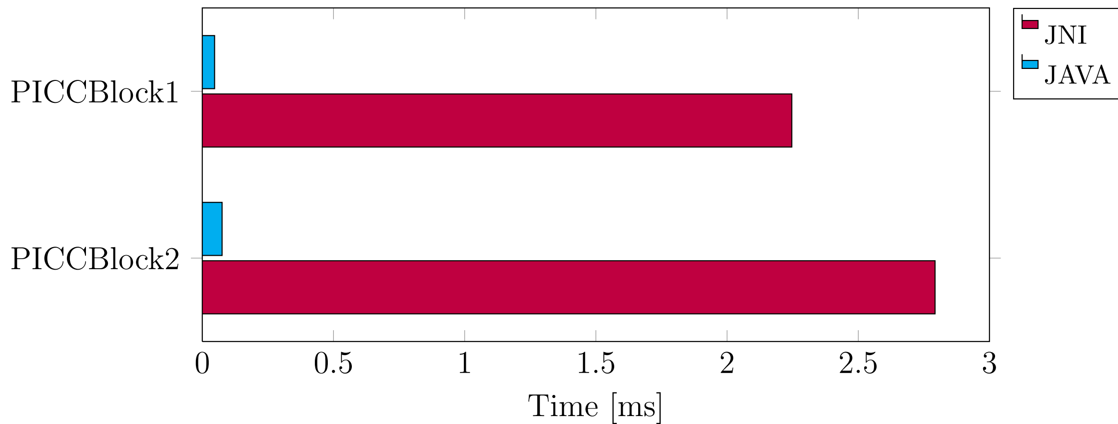


Figure 5.2: Average Measured Computation Times of JNI and Java Implementation

5.4.1 Multiple Repetitions

As mentioned the Dalvik VM is capable of JIT compiling. Therefore it would be interesting if this has an effect on the computation time if performing more

authentications in sequence.

Procedure

For visualising this effect the average computation times of various authentication round from 1 to 100000 were measured 10 times each.

Results

In figure 5.3 the average computation time over the number of authentication rounds ins shown. For having a better orientation the FDT of the reverence PICC has been added to the figure. With 10000 repetitions the calculation time for PICCBlock1 decreases to 255 us. As this is still 5 times higher than the average computation time achieved with the JNI implementation and as this kind of solution would have an extremely bad energy balance it will not be taken into account for further consideration.

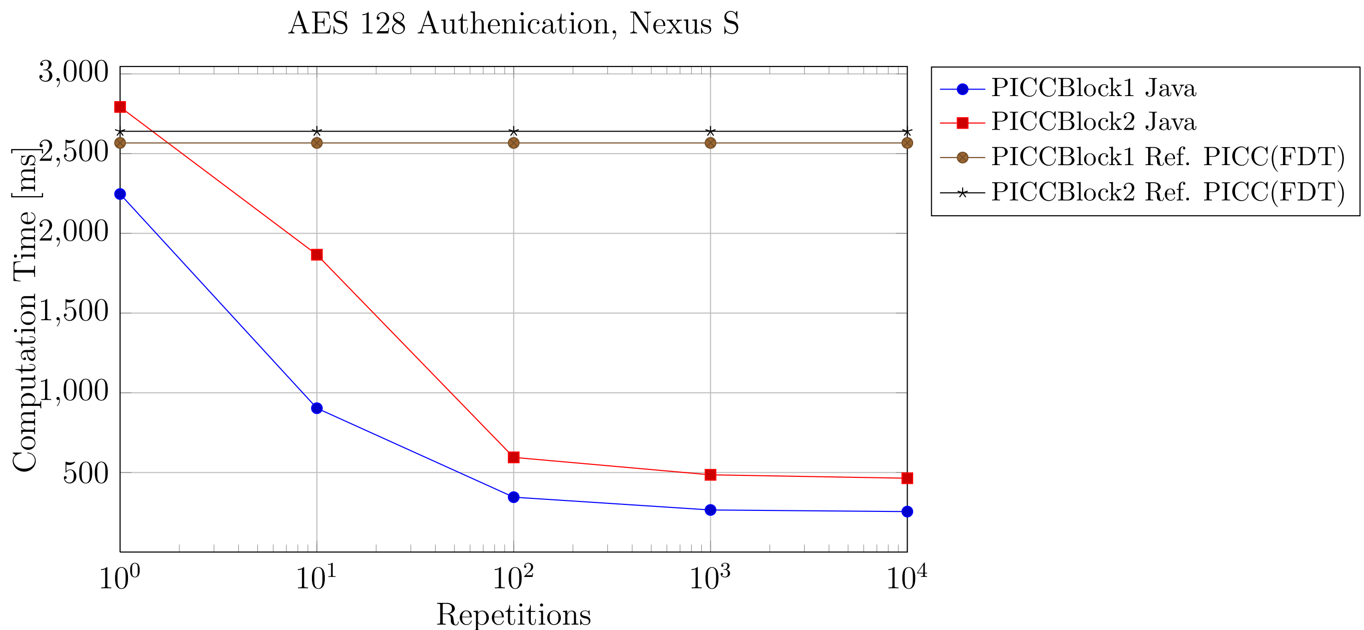


Figure 5.3: Repetition Effect on JAVA Implemetation

5.5 Proof of Concept Measurement

Procedure

The FDT of the PoC implementations was measured on the ISO Setup and the evaluation of the profiling timestamps was done with the help of the LogCAT. For each implementation the measurement was performed 50 times and was corrected after the formalism in section 5.1.3. These measurement results are completely displayed in Appendix B. The following section will only present an abstract of them.

5.5.1 PoC Stack

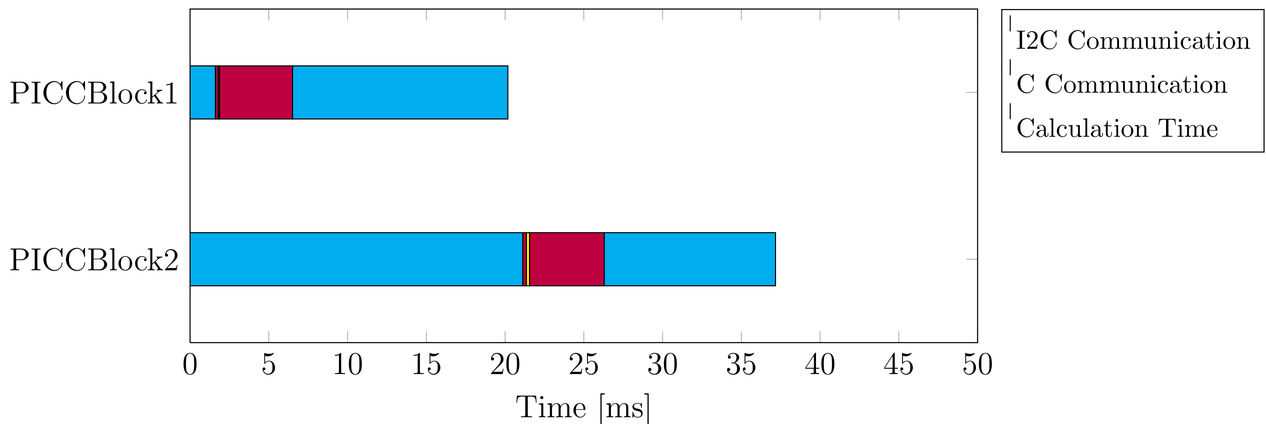


Figure 5.4: Measured FDT of PoC Stack

Results

Figure 5.4 shows the average FDT and how it is composed. It can be seen that the actual computation of the authentication (yellow) just consumes a very little part of the FDT and is hardly recognizable at all. The difference in I2C communication can be explained by the different amount of data to be transmitted.

5.5.2 PoC Java

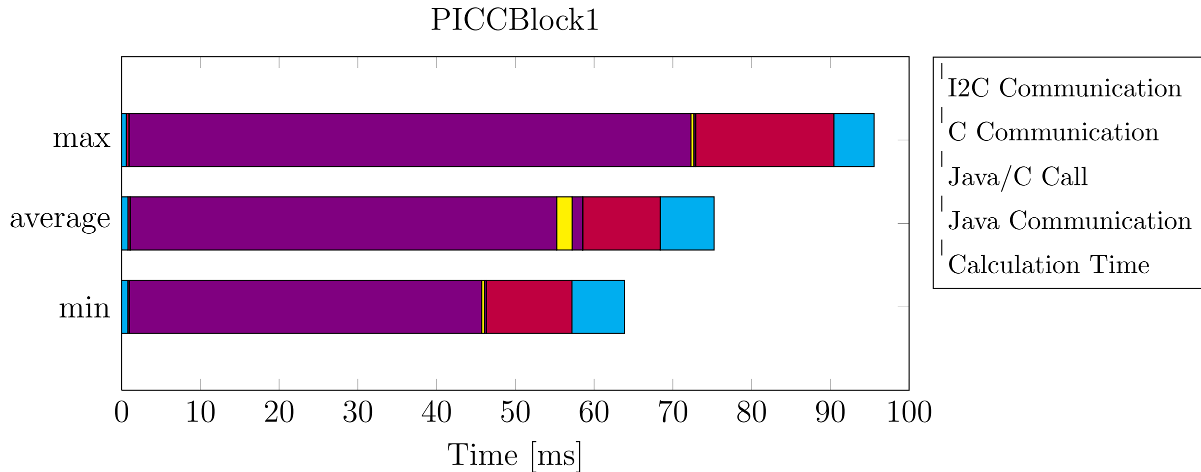


Figure 5.5: Measured FDT of PoC Java with Intent Notification

Results

In figure 5.5 the differences between the shortest, the longest and the average measured FDT of the PoC Java implementation with intent notification are illustrated. There is a quite big variance in these results which is manly driven by the intent notification mechanism (violet) and the time after entering the stack level again(purple). The marginal influence of the authentication computation also can be seen in figure 5.5 and as shown in table 5.7 just 2.6% of the total FDT are consumed by the pure calculation in the average case. With more than 70% the intent notification consumes the biggest part.

	I2C Communication 1	C Communication 1	C/Java Border	Java Communication 1	Calculation	Java Communication 2	Java/C Border	C Communication 2	I2C Communication 2	FDT
	[us]									
Min Average Max	786 804 602	190 268 329	17 24 29	44724 54150 71334	359 1967 356	234 1325 212	17 24 29	10855 9843 17546	6677 6833 5118	63858 75238 95556
	[% of FDT]									
Min Average Max	1,23 1,07 0,63	0,30 0,36 0,34	0,03 0,03 0,03	70,04 71,97 74,65	0,56 2,61 0,37	0,37 1,76 0,22	0,03 0,03 0,03	17,00 13,08 18,36	10,46 9,08 5,36	100,00

Table 5.7: Measured FDT of PoC Java with Intent Notification for PICCBlock1

5.5.3 PoC JNI

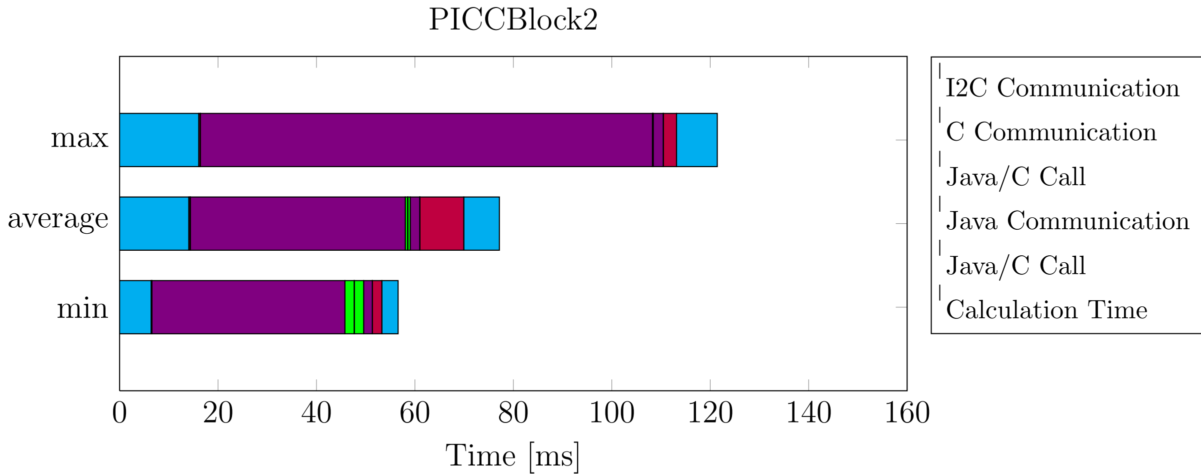


Figure 5.6: Measured FDT of PoC JNI with Intent Notification

Results

The measurement on the PoC JNI implementation with intent notification shown in figure 5.6 paint a similar picture as for the PoC Java implementation. Even if

the FDT PICCBlock2 which consumes more computation time it has hardly any impact on the total FDT.

5.6 Discussion of Measurement Results

C vs. JNI

The first point to take a closer look on is the comparison of the two implementations written in C, the first on the LPC board and the second on the Nexus S over the JNI. If just looking at the computation time in table 5.8 the two implementations seem to play in the same league but if you look at the performance per MHz you can see the influence of the OS that makes the computation about 10 to 25 times slower.

Implementation	PICCBlock1		PICCBlock2	
	$[us]$	$[\frac{us}{MHz}]$	$[us]$	$[\frac{us}{MHz}]$
C	25	1808	91	6537
JNI	47	47000	76	76000

Table 5.8: C vs JNI

Java vs. JNI

The influence of the programming language is quite high as well. As figure 5.2 showed the JNI implementation is on average 40 to 50 times faster than the Java implementation. Therefore it would be recommender in case of commercial use of the host emulation.

Intent Call vs. Callback

The next point to discuss is the influence of the notification mechanism to the application level. As already mentioned the easer to implement intent notification is quite slow. Therefore the Callback notification was introduced. This could improve the average performance as you can see in figure 5.7 on the example of the Poc JNI implementation. As expected the computation starts much earlier after the reception of data than it did with the intent mechanism but this is not totally reflected in the overall FDT because the time between end of computation and start of the I2C communication increased. To evaluate why this is the case a deeper analysis of the Android source code had to be made. It turned out that sending an HCI message does not directly trigger an I2C communication. Instead the massage is put into a message queue. Those massages in the queue are executed by a separate thread which is implemented in the NativeNfcManager class. These factors that influence

the processing of this thread could not be determined in detail, but it seems that it is somehow effected by the callback implementation. Additionally, the phone waits for an 500 us guard time before sending an I2C command which is recommended by NXP as written in a NFC stack source code comment.

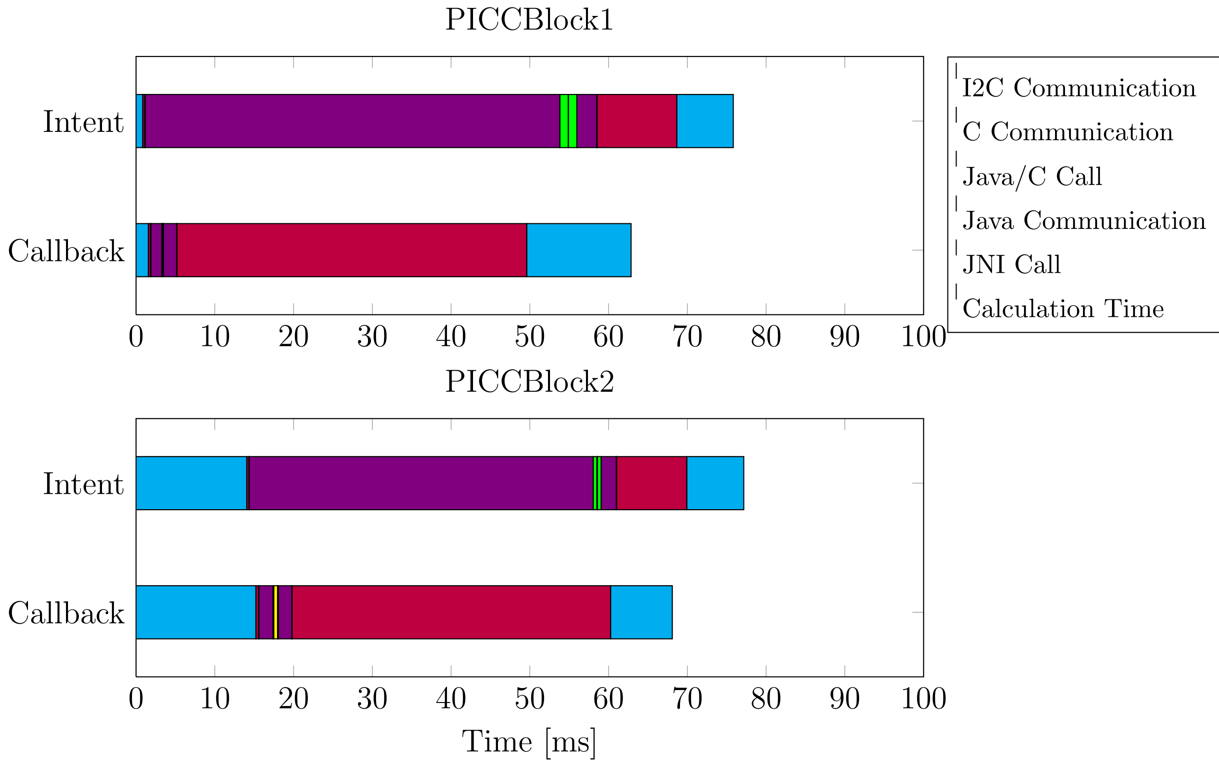


Figure 5.7: Callback and Intent Comparison for PoC JNI

Use Case Functions

If looking at the complete set of commands used for the access management use case it also shows that the computation time is marginal compared to the rest of the communication parts. The figure 5.8 for the PoC Stack implementation also shows that the communication time after the actual computation of a read command is much longer than for the other commands. This is the case because the 41 Bytes of data are split when they are sent over the I2C interface. Therefore additional time for the queue operating thread and guard time is consumed.

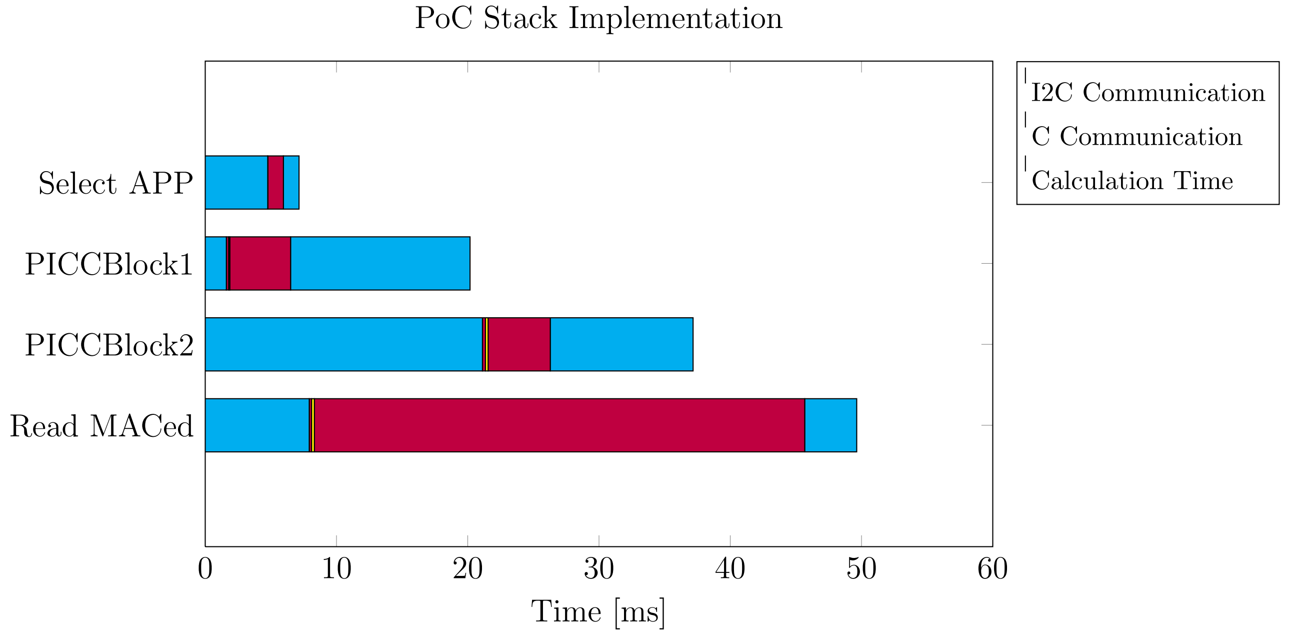


Figure 5.8: Measured FDT for PoC Use Case on PoC Stack

A summery of the average FDTs of all PoC implementations is provided in table 5.9 which also reflects the fact the PoC Stack implementation is the fastest of all.

Commands	POC Stack	POC Java Intent	POC Java Callback	POC JNI Intent	POC JNI Callback
	[us]				
Select Application	7146	52278	71237	50737	72306
PICCBlock1	20175	64265	75238	62858	75833
PICCBlock2	37168	69892	81550	68090	77164
Read MACed	49635	74741	95240	71608	93357

Table 5.9: FDTs of PoC Use Case Commands for Different PoC Implementations

Comparison to Reverence Devices

Finally the focus comes to the discussion of the over all performance of the PoC implementations compared to reference devices. Figure 5.9 shows the average and the shortest measured FDT for PICCBlock1 of the different solutions. In case of the PoC application level implementations the results for the callback implementations are shown because they are the faster solutions.

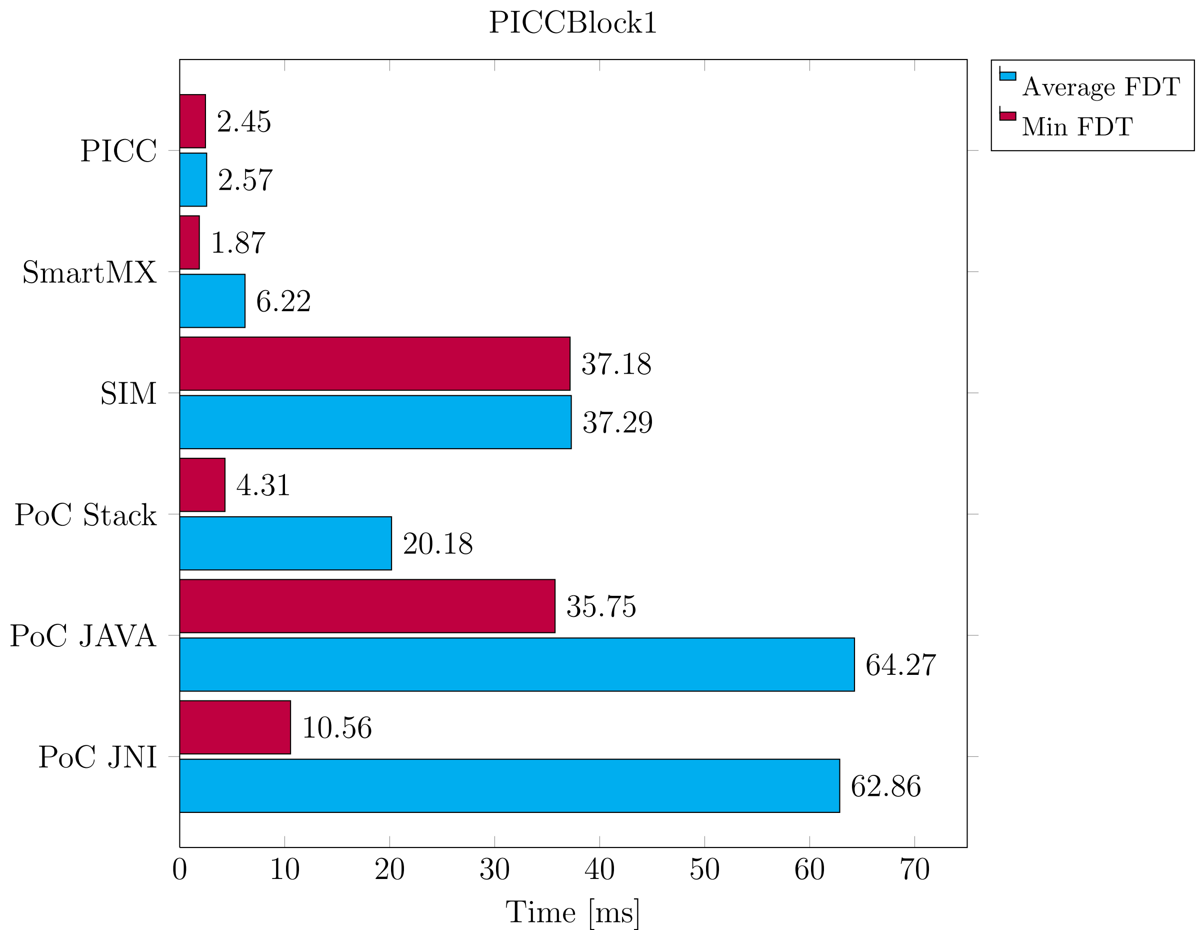


Figure 5.9: Comparison Devices and PoC Implemetations PICCBlock1

It can be seen that the PoC Stack implementation compete easily with the SIM based card emulation. In the best case it even comes in the area of the reference PICC. As expected the PoC Java solution is the slowest one. Even in ideal case it is just slightly faster than the Gemalto SIM. And the PoC implementation performed slightly better on average but could undercut the average PoC Stack performance at least once.

The comparison for the second PICCBlock in figure 5.10 shows a different picture. As no real FDT for the SIM solution could be measured the requested FWT is shown in the figure to illustrate the minimum duration. Due to the higher data exchange and higher computation effort the average FDT almost doubles compared to PICCBlock1 but the other PoC implementations are now faster than the SIM solution as well.

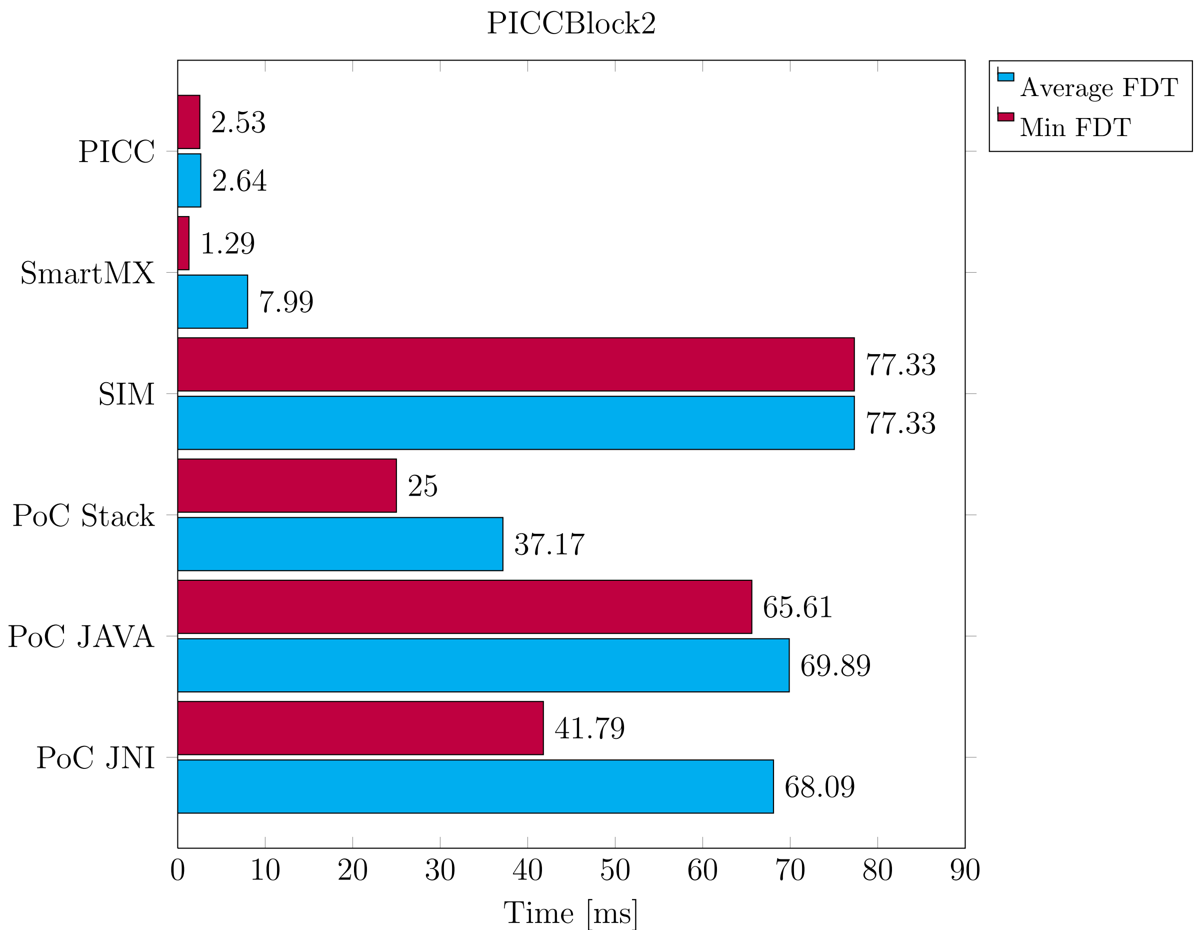


Figure 5.10: Comparison Devices and PoC Implemetations PICCBlock2

Measurement Conclusion

The measurements presented in this chapter show some very intriguing facts. As expected the C implementation is faster than the JNI implementation which again is faster than the Java implementation. But the PoC implementations showed that the influence of the actual computation time is marginal. The main time consumption is

produced by the communication on the I2C bus, the notification mechanism and the I2C massage queuing. At least one of these could be bypassed with the PoC Stack implementation. The disadvantage of this solution is that it is less flexible than the application level implementations but as the source code has to be changed any way this could be diminished by routing additional functions through the system stack. On the performance point of view it can be seen that the PoC stack implementation, which is the fastest PoC implementation, easily competes with SIM solutions whereas performance of the hardware accelerate reference PICC will not be cached.

5.7 Impact on the Integrity of Existing RFID System

The previous implementations and measurements show that a state of the art NFC mobile phone has no additional impact on the security of an existing RFID system. This has three main reasons.

Random UID only

The PN544 NFC chip does not allow to set a specific UID over the host controller interface. This disqualifies the device even for systems that just do a rudimentary UID check for authentication. In state of the art access system each card uses its own authentication key which is derived from this UID and therefore is not possible for the Nexus-S.

Caused by the leak of NFC enabled Android smart phones with other chips this assumption can not be made for NFC enabled smart phones in general because other manufacture might do not block this feature. Additionally it can be speculated that in case of the PN544 the disability for specific UIDs is caused by the firmware running on the chip because a SIM base solution would be able to set an UID. If so this might create a security thread in collaboration with a firmware hack.

Nevertheless even if it is possible to use a specific UID with host emulation it is not a problem per se as the other two reasons would have to kick in as well.

Need for key

Like all other card emulation device the host emulation solution has to use the right key for the authentication. As long as the other credential used in a system prevent a key extraction the system is save. A problem could arise for example if RFID systems use for reasons of old card stocks a mix credential population with e.g. the MF3ICD40 which could be hacked [OP11].

Slow interface

The major part of the frame delay time is caused by the host controller interface itself. For the second PICCBlock this delay is in the area of about 10 ms minimum. With such a performance the host emulation can be easily distinguished from a state of the art contacless smart card even if the authentication keys are known. This also disqualification the NFC enabled smart phone to be used for a relay attack as well as it wouldn't pass any proximity check.

5.8 Usage Possibility for the Host Emulation

Even if the implementations provided in this work are not designed to be securely used in a state of the art RFID System they show the potential of this solution. Especially the PoC use case implementation. If the host emulation is enabled once it allows a very flexible definition of the "cards" functionality and full control over the implementation. Additionally it allows in combination with the authentication protocol a secure data exchange fully backwards compatible to existing ISO 14443 reader devices even if they do not support the NFCIP-1 transmission protocol.

Nevertheless before using a host emulation implementation some problems have to be considered. The main problem is the untrusted execution environment. The other point is that the system has to deal with the Random UID and therefore its not possible to diversify the key for the phone with the UID. And of course a proximity check can not be used as well.

If the first problem is solved a host emulation implementation can in principle be used for the same areas of application as the SIM based solution is used if no fixed UID is required.

Chapter 6

Conclusion and Future Work

This work shows the ability of a state of the art NFC enabled smart phone to perform an AES authentication protocol by using the baseband processor and it evaluates the performance that could be achieved for the authentication. Therefore a number of implementations had to be provided. This started with a low level C Implementation to show the basic performance and continued with the Java and JNI implementation on application level to show the performance on a state of the art smart phone. By enabling the host emulation feature of the Nexus S the previous implementation were combined into a proof of concept. This shows different possible implementation levels and the delays introduced by the hosting platform. Additional the extension of this proof of concept to be able to be used for the use case of access management showed the potential for future use of the AES authentication one the mobile phones base band processor.

6.1 Conclusion

For doing a final conclusion we should recall the objective questions of this work and give an answer to them:

Question:	Is it possible to compute the AES based authentication protocol on a mobile phone's base band processor in comparable time to existing smart card and NFC solutions?
Answer:	On the one hand this can be answered with a clear "Yes" for the existing NFC solutions because the measurements in chapter 5 showed that the fastest PoC Implementation consumes on average even less time than the SIM card solution does. On the other hand it has to be answered with a "No" for the smart card solutions as compared to the reference PICC the host emulation is et least 10 times slower.

Question: How does the context of a mobile phone environment affect the computation time and total delays?

Answer: The main influence of the mobile phone environment are the system delays. With the fastest implantation only 0.5 % of the total FDT on average is caused by the actual calculations. If comparing this calculation time again to the basic C Implementation you can see that the mobile phone platform including the operating system causes a slowdown by the factor of 40.

Question: Can the computation on the base band processor affect the security of systems relying on this AES based authentication?

Answer: No additional impact on the security of an existing RFID system could be found if performing the AES authentication on the base band processor. This assumption is based on the slow host controller interface, the Random UID and the need for the correct authentication key as described in section 5.7.

On a final note it has to be mentioned that even if all technical issues of section 5.7 are solved the authentication key still will be the critical component to allow possible miss-use of card emulation. Therefore owner of those keys should be adviced to use cards and devices with state of the art protection against unauthorized key access. Especially this should be considered when using card emulation in a commercial context and when keys are processed in software on the emulation device.

6.2 Future Work

For using the host emulation mode including the AES authentication in a commercial RFID system there has to be put a lot of effort it the creation of a security aware software design. This is worth to do if a reasonable use case is found that takes advantage of the ability to have direct access to the computation power of the base band processor. New NFC enabled smart phones with multi core CPUs and NFC chips with faster host interface can increase this performance. Therefore this can make an implementation on application level more attractive as the delay caused by the system can be decreased. On the other side this speedup could introduce new security threads and therefore this development should be observed in the future.

Bibliography

[ARM11] ARM Ltd. Cortex-A Series. `http://mobile.arm.com/products/processors/cortex-a/index.php`, December 2011.

[BBF+03] Guido Bertoni, Luca Breveglieri, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin. Efficient Software Implementation of AES on 32-Bit Platforms. In Burton Kaliski, etin Ko, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 129–142. Springer Berlin / Heidelberg, 2003.

[BK09] Alex Biryukov and Dmitry Khovratovich. Related-Key Cryptanalysis of the Full AES-192 and AES-256. In *ASIACRYPT*, pages 1–18, 2009.

[BKG11] T.Ravichandra Babu, K.V.V.S.Murthy, and G.Sunil. AES Algorithm Implementation using ARM Processor. *IJCA Proceedings on International Conference and workshop on Emerging Trends in Technology (ICWET)*, (12):24–29, 2011. Published by Foundation of Computer Science.

[BKR11] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique Cryptanalysis of the Full AES. *IACR Cryptology ePrint Archive*, 2011:449, 2011.

[ETS08] ETSI TS 102 622: Smart Cards; UICC - Contactless Front-end (CLF) interface; Host Controller Interface (HCI). `http://ics.nxp.com/support/documents/microcontrollers/pdf/lpcxpresso.getting.started.pdf`, 2008.

[Fin10] Klaus Finkenzeller. *RFID Handbook: Fundamentals and Applications in Contactless Smart Cards and Identification.* John Wiley & Sons, Inc., New York, NY, USA, 3 edition, 2010.

[GBBM08] M. Gebhart, S. Birnstingl, J. Bruckbauer, and E. Merlin. Properties of a test bench to verify standard compliance of proximity transponders. In *Communication Systems, Networks and Digital Signal Processing, 2008. CNSDSP 2008. 6th International Symposium on*, pages 306 –310, july 2008.

[Gem10] Gemalto. Worlds First: Gemalto Integrates DESFire Transport Card into NFC Mobile Phone. `http://www.gemalto.com/php/pr_view.php?id=704`, February 2010.

[Gla07] Brian Gladman. A Specification for Rijndael, the AES Algorithm , v3.16. `http://gladman.plushost.co.uk/oldsite/cryptography_technology/rijndael/aes.spec.v316.pdf`, August 2007.

[Gla11] Brian Gladman. AES Implementation. `http://gladman.plushost.co.uk/oldsite/AES/aes-src-12-09-11.zip`, September 2011.

[Goo11a] Google. ANDROID developers. `http://developer.android.com/index.html`, December 2011.

[Goo11b] Google. ANDROID source. `http://source.android.com/`, December 2011.

[Goo11c] Google. What is Android? `http://developer.android.com/guide/basics/what-is-android.html`, December 2011.

[Han06] G.P. Hancke. Practical attacks on proximity identification systems. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6 pp. –333, may 2006.

[ifi11] ifixit. Nexus S Teardown. `http://www.ifixit.com/Teardown/Nexus-S-Teardown/4365/1`, December 2011.

[ISO08] ISO/IEC 14443 Identification cards – Contactless integrated circuit cards – Proximity cards, 2008.

[KvMOP11] Timo Kasper, Ingo von Maurich, David Oswald, and Christof Paar. Chameleon: A Versatile Emulator for Contactless Smartcards. In Kyung-Hyune Rhee and DaeHun Nyang, editors, *Information Security and Cryptology - ICISC 2010*, pages 189–206. Springer Berlin / Heidelberg, 2011.

[LR10] Josef Langer and Michael Roland. *Anwendungen und Technik von Near Field Communication(NFC)*. Springer, 2010.

[Mad11] Gerald Madlmayr. Uncovered: The hidden NFC potential of the Google Nexus S and the Nokia C7. http://www.nfcworld.com/2011/02/13/35913/uncovered-the-hidden-nfc-potential-of-the-google-nexus-s-and-the-nokia-c7/, February 2011.

[MDLS08] Gerald Madlmayr, Oliver Dillinger, Josef Langer, and Josef Scharinger. Management of Multiple Cards in NFC-Devices. In Gilles Grimaud and François-Xavier Standaert, editors, *CARDIS*, volume 5189 of *Lecture Notes in Computer Science*, pages 149–161. Springer, 2008.

[NFC11a] NFC Admin. NXP NFC Chips. `http://www.nfc.cc/technology/nxp-nfc-chips`, July 2011.

[NFC11b] NFC Forum. About NFC. `http://www.nfc-forum.org/aboutnfc/`, September 2011.

[NXP10a] NXP. MIFARE DESFire EV1 contactless multi-application IC Product short data sheet. `http://www.nxp.com/documents/short_data_sheet/MF3ICDX21_41_81_SDS.pdf`, December 2010.

[NXP10b] NXP. NXP NFC controller PN544 for mobile phones and portable equipment. `http://www.nxp.com/documents/leaflet/75016890.pdf`, February 2010.

[NXP11a] NXP. Getting started with NXP LPCXpresso Rev. 11.1. `http://ics.nxp.com/support/documents/microcontrollers/pdf/lpcxpresso.getting.started.pdf`, December 2011.

[NXP11b] NXP. LPC1311/13/42/43 User manual, Rev. 4. `http://www.nxp.com/documents/user_manual/UM10375.pdf`, September 2011.

[NXP11c] NXP. NFC Software in a Mobil System. `http://wenku.baidu.com/view/4c4c18202f60ddccda38a024.html`, September 2011.

[NXP11d] NXP. PN544 C2 User Manual Rev. 1.8. October 2011.

[OBSC10] Dag Arne Osvik, Joppe W. Bos, Deian Stefan, and David Canright. Fast software AES encryption. In *Proceedings of the 17th international conference on Fast software encryption*, FSE'10, pages 75–93, Berlin, Heidelberg, 2010. Springer-Verlag.

[OP11] David Oswald and Christof Paar. Breaking Mifare DESFire MF3ICD40: Power Analysis and Templates in the Real World. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 207–222. Springer Berlin / Heidelberg, 2011.

[RNP10] Andreas Rohr, Karsten Nohl, and Henryk Pltz. Establishing Security Best Practices in Access Control. `http://www.srlabs.de/pub/acs`, September 2010.

[Sha06] R. Shamsuddin. A comparative study of AES implementations on ARM processors. Master's thesis, 2006.

Appendix A

Abbreviations

AES	Advanced Encryption Standard
ARM	Advanced RISC Machine
ATQA	Answer To Request Type A
ATS	Answer To Select
CE	Card Emulation
DES	Data Encryption Standard
DUT	Device Under Test
FDT	Frame Delay Time
FRI	Forum Reference Implementation
FWI	Frame Waiting Integer
FWT	Frame Waiting Time
HAL	Hardware Abstraction Layer
HCE	Host Controller Em
HCI	Host Controller Interface
JNI	Java Native Interface
MAC	Message Authentication Code
NFC	Near Field Communication
OS	Operating System
OSAL	Operating System Abstraction Layer
PCD	Proximity Coupling Device
PICC	Proximity Integrated Circuit Card
PoC	Proof of Concept
PPS	Protocol and Parameter Selection
RATS	TRequest Answer to Select
REQA	Request Type A
RFID	Radio Frequency Identifcation
SAK	Select Acknollage
SAM	Secure Access Module
SDK	Source Developing Kid
SE	Secure Element

SFGI	Start Frame Guard Integer
SIM	Subscriber Identity Module
SWP	Single Wire Protocol
TEE	Trusted Execution Environment
UICC	Universal Integrated Circuit Card
UID	Unique Identification Number
VM	Virtual Machine
WTX	Waiting Time Extension

Appendix B

Measurement Tables

B.1 PoC Stack

	I2C Communication 1	C Communication 1	Calculation	C Communication 2	I2C Communication 2	FDT
Select App	[us]					
Min	4036	-	-	1084	1009	6129
Average	4760	-	-	1196	1190	7146
Max	6052	-	-	1321	1513	8886
PICCBlock1	[us]					
Min	377	22	18	690	3205	4313
Average	1608	178	84	4634	13671	20175
Max	2064	159	87	7074	17546	26930
PICCBlock2	[us]					
Min	12705	142	186	5418	6545	24995
Average	21121	231	199	4738	10880	37168
Max	35272	173	162	2055	18171	55833
Read	[us]					
Min	7865	150	203	24031	3932	36181
Average	7923	178	214	37359	3961	49635
Max	4286	159	214	53706	2143	60507

Table B.1: Total Measurement Results for PoC Stack

B.2 PoC JAVA

	I2C Communication 1	C Communication 1	C/Java Border	Java Communication 1	Calculation	Java Communication 2	Java/C Border	C Communication 2	I2C Communication 2	FDT
Select App	[us]									
Min	3221	52	10	41292	-	-	10	2629	805	48020
Average	4291	260	31	56075	-	-	31	9477	1073	71237
Max	3443	284	30	79909	-	-	30	16172	861	100729
PICCBlock1	[us]									
Min	786	190	17	44724	359	234	17	10855	6677	63858
Average	804	268	24	54150	1967	1325	24	9843	6833	75238
Max	602	329	29	71334	356	212	29	17546	5118	95556
PICCBlock2	[us]									
Min	7588	152	62	39169	584	257	62	13935	3909	65717
Average	13846	346	33	46971	3052	1041	33	9095	7133	81550
Max	28070	331	118	65598	662	819	118	868	14460	111046
Read MACed	[us]									
Min	2543	188	17	43927	17491	1584	17	7427	3814	77007
Average	3289	365	28	55437	4096	1531	28	25533	4933	95240
Max	3107	430	29	84728	894	254	29	33405	4660	127537

Table B.2: Total Measurement Results for PoC Java with Intent Notification

	I2C Communication 1	C Communication 1	C/Java Border	Java Communication 1	Calculation	Java Communication 2	Java/C Border	C Communication 2	I2C Communication 2	FDT
Select App	[us]									
Min	3083	185	21	1516	-	-	21	20245	771	25843
Average	6297	302	50	2888	-	-	50	41117	1574	52278
Max	3467	280	61	5663	-	-	61	71213	867	81612
PICCBlock1	[us]									
Min	991	191	22	728	1757	619	22	22994	8424	35747
Average	1408	285	41	1458	3816	1932	41	43316	11969	64265
Max	4048	325	39	1637	3027	2001	39	43855	34411	89382
PICCBlock2	[us]									
Min	17626	316	40	2015	6053	1658	40	28784	9080	65613
Average	15023	330	48	1995	11790	2459	48	30459	7739	69892
Max	19878	320	42	3540	9999	2095	42	34430	10240	80585
Read MACed	[us]									
Min	3665	45	5	208	508	3576	5	14552	5497	28063
Average	3056	236	40	1469	6635	2367	40	56315	4584	74741
Max	2778	187	26	1176	6415	2588	26	99517	4167	116880

Table B.3: Total Measurement Results for PoC Java with Callback Notification

B.3 PoC JNI

	I2C Communication 1	C Communication 1	C/Java Border	Java Communication 1	Java/JNI Border	Calculation	JNI/Java Border	Java Communication 2	Java/C Border	C Communication 2	I2C Communication 2	FDT
Select App	[us]											
Min	2309	57	4	4027	-	-	76	200	4	7313	577	14567
Average	3908	252	30	51126	-	-	2321	3538	30	10125	977	72306
Max	7428	287	43	82294	-	-	781	1760	43	25241	1857	119733
PICCBlock1	[us]											
Min	892	52	4	4843	66	18	66	258	4	6802	7585	20590
Average	844	256	50	52663	1064	26	1064	2520	50	10119	7177	75833
Max	1842	279	41	85014	493	18	493	4934	41	17865	15660	126681
PICCBlock2	[us]											
Min	6407	147	19	39205	1874	57	1874	1761	19	1921	3300	56583
Average	14055	272	36	43678	470	66	470	1931	36	8910	7240	77164
Max	16080	309	30	91840	49	48	49	2040	30	2672	8284	121432
Read MACed	[us]											
Min	5599	298	32	46519	49	67	49	1828	32	4644	2800	61916
Average	5881	285	67	56289	1353	508	1353	1563	67	23049	2941	93357
Max	8552	305	30	93364	43	66	43	207	30	36737	4276	143653

Table B.4: Total Measurement Results for PoC JNI with Intent Notification

	I2C Communication 1	C Communication 1	C/Java Border	Java Communication 1	Java/JNI Border	Calculation	JNI/Java Border	Java Communication 2	Java/C Border	C Communication 2	I2C Communication 2	FDT
Select App	[us]											
Min	4414	178	21	778	-	-	29	560	21	19357	1104	26461
Average	6391	244	39	1260	-	-	59	1378	39	39729	1598	50737
Max	3580	283	42	1502	-	-	88	1084	42	75810	895	83325
PICCBlock1	[us]											
Min	631	44	3	208	1	15	1	145	3	4139	5365	10556
Average	1559	270	36	1404	46	110	46	1681	36	44414	13255	62858
Max	3126	270	39	4736	78	115	78	1326	39	48300	26575	84684
PICCBlock2	[us]											
Min	11729	151	22	769	34	441	34	591	22	21957	6042	41791
Average	15210	334	42	1789	102	487	102	1701	42	40444	7836	68090
Max	21098	309	42	2152	57	279	57	1836	42	54597	10869	91339
Read MACed	[us]											
Min	6019	220	70	1505	40	380	40	732	70	31720	9028	49824
Average	3083	295	46	1421	72	572	72	1373	46	60005	4624	71608
Max	3449	286	70	1519	61	492	61	1996	70	88355	5174	101534

Table B.5: Total Measurement Results for PoC JNI with Callback Notification