Connecting NFC to the Cloud

Remote Updating of Smart Cards

Daniel Correia Andrade

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairperson:	Prof. João Emílio Segurado Pavão Martins
Supervisor:	Prof. Miguel Nuno Dias Alves Pupo Correia
Co-Supervisor:	Prof. Tuomas Aura
Member of the Committee:	Prof. Rui Miguel Soares Silva

November 2013

Resumo

Near Field Communication (NFC) é uma tecnologia sem fios com emparelhamento automático dos dispositivos intervenientes. Um dispositivo NFC comunica com outro dispositivo NFC ou com uma tag quando são colocados junto um do outro. No contexto de dispositivos móveis e no modo de operação leitura-escrita, um telemóvel equipado com NFC normalmente lê dados de uma tag mas não escreve para essa tag por razões de segurança. As modificações à tag são efectuadas localmente num ponto de venda do operador.

Esta tese propõe um novo modelo de utilização para o modo de operação leitura-escrita, no qual um servidor remoto consegue manipular a informação presente numa tag através de um telemóvel equipado com NFC. O servidor remoto envia comandos para a tag, e recebe as respectivas respostas, através do telemóvel. A tese apresenta um protocolo de comunicação que permite alcançar este objectivo tendo em consideração a confiabilidade e a segurança do sistema.

Começamos por estudar as funcionalidades e os mecanismos de segurança do cartão MIFARE DESFire EV1. Com base no conhecimento adquirido, desenhamos um protocolo de comunicação que permite a um servidor remoto modificar o conteúdo dos ficheiros de um cartão DESFire EV1 de forma segura. É também analisada a possibilidade de falhas e é apresentada uma solução para essas falhas resultando num protocolo mais robusto. Este protocolo é avaliado, tendo em conta uma lista de requisitos previamente elaborada para o sistema, através de um protótipo que é testado num operador público de computação em nuvem.

Palavras-chave: nfc
modo de operação leitura-escrita
mifare desfire ev1
protocolo de actualizações remotas
protocolo seguro

Abstract

The purpose of this thesis is to research a new usage model for the reader-writer operating mode, where the NFC-enabled mobile phone not only reads from but also writes into the tag, according with instructions dictated by a remote server. We start by studying the functionality of the MIFARE DESFire EV1 smart card. We then create a set of requirements for a system capable of remotely updating files on a card application of a DESFire EV1 and devise a remote update protocol that satisfies those requirements. This protocol is evaluated through a prototype we built as a proof of concept, which in turn is subject to experiments in a cloud provider.

This thesis demonstrates that it is feasible for a remote server to update the files contained in a card application of a DESFire EV1 via an NFC-enabled mobile phone, and that this can be accomplished in a reliable and secure way. We analyze the failures that may affect the remote update protocol and provide a solution to those failures leading to a robust protocol. Its security comes from the use of encrypted communication between server and mobile phone, from the use of the enciphered communication mode of DESFire EV1 between the mobile phone and the smart card, and from the fact that the mobile phone does not have access to the secret keys required to read and update the data stored on the card application.

Palavras-chave: nfc
reader-writer operating mode
remote update protocol
mifare desfire ev1 smart card
remote update protocol

Acknowledgments

This undertaking was feasible as a result of the support of my supervisors who motivated me in the best way possible and always found the time to help me when needed, and to whom I express my gratitude.

I thank Professor Tuomas Aura for his guidance, for his insightful remarks, and for providing me with all the necessary equipment and with a place at Aalto to work on this project. It was an invaluable experience to be able to learn from someone as knowledgeable. Thank you for lending me your smart phone during the evaluation phase!

I thank Professor Miguel Pupo Correia for his advice, for his careful review of the thesis, which resulted in a more coherent manuscript, and for helping me improve the presentation for the thesis defense at IST.

Thank you Professor Pavão Martins for presiding the thesis defense. Thank you Professor Rui Silva for the interesting questions and comments about the thesis; and for having traveled from far to be present at the defense.

There were others working next door whom I occasionally met with and who I would like to acknowledge: Sandeep, Anh and Robert. I particularly thank Sandeep Tamrakar for sharing his knowledge and answering my questions about the intricacies of NFC; and thank you for helping me process the thesis approval. Best of luck with the remainder of your doctoral studies!

When your brain is locked in the *thesis operating mode* from dawn to dusk, seven days a week, it is always nice to have someone around to chit-chat for a bit and make it wander off somewhere else. Incidentally, I thank the many I met in Otaniemi and around and who made my stay in Finland more enjoyable. Thank you Alice.

Last but not least I thank my grandparents—João and Maria—without whom I would not have made it this far.

Lisbon, November 2013

Daniel Andrade

Abbreviations and Acronyms

AID	Application IDentifier
APDU	Application Protocol Data Unit
CAR	Change Access Rights
CBC-MAC	Cipher-Block Chaining MAC
CICC	Close-coupled Integrated Circuit Card
CMAC	Cipher-based MAC
DES	Data Encryption Standard
2K3DES	Triple DES, keying option 2 ($K1 \neq K2 \wedge K1 = K3$)
3K3DES	Triple DES, keying option 1 ($K1 \neq K2 \neq K3$)
FID	File IDentifier
MAC	Message Authentication Code
MF0ICU2	MIFARE Ultralight C
MF3ICD40	MIFARE DESFire (predecessor of MF3ICD41)
MF3ICD41	MIFARE DESFire EV1
NFC	Near Field Communication
OTP	One-Time Programmable
PCD	Proximity Coupling Device
PICC	Proximity Integrated Circuit Card
POS	Point of Sale
IV	Initialization Vector
RF	Radio Frequency
Triple DES	Triple Data Encryption Algorithm (TDEA)
UID	Unique IDentifier
VICC	Vicinity Integrated Circuit Card

Symbols

$\{0,1\}^s$	An s-byte long byte string consisting of random zeros and ones.
$x \oplus y$	The bitwise exclusive OR of bit strings x and y.
$E_K(y)$	Symmetric encryption of y using secret key K.
$D_K(y)$	Symmetric decryption of y using secret key K.
$RotLeft_s(x)$	Rotate the byte string x, s bytes to the left.
$Z_{s\text{–}t}$	The byte string consisting of, and including, bytes s to t of the byte string Z.

Contents

Abbreviations and Acronyms **9**

Symbols **11**

1 Introduction **17**
1.1 Problem statement and methodology 17
1.2 Structure of the thesis 19

2 Background and related work **21**
2.1 Smart cards 22
2.2 Near field communication 24
2.2.1 Operating modes 25
2.2.2 Standards 26
2.2.3 Security 27
2.3 Related projects 29

3 MIFARE DESFire EV1 **31**
3.1 Commands 32
3.2 File system 32
3.3 Security 34
3.3.1 Cryptographic primitives 35
3.3.2 Authentication protocol 37
3.3.3 Keys 40
3.3.4 Data transmission 42
3.3.4.1 Communication settings 42
3.3.4.2 Access rights 45
3.4 Trace 46

4 Remote card update **55**
4.1 Requirements 55
4.2 Solution 57

4.2.1 Architecture . . . 57
4.2.2 Transactions . . . 59
4.2.3 Protocol . . . 61
4.2.3.1 Failures . . . 64
4.2.3.2 Early-commit attack . . . 66

5 Implementation **69**
5.1 Tools and technologies . . . 69
5.2 Architecture . . . 71
5.2.1 Communication between the phone and the server . . . 72
5.2.2 Communication between the phone and the card . . . 73
5.2.3 Class diagrams and dependencies . . . 73
5.3 Flow of operations . . . 79
5.3.1 Acquiring a new update . . . 79
5.3.2 Updating the card . . . 80
5.4 Implementation issues . . . 82

6 Evaluation **87**
6.1 Experimental setup . . . 87
6.2 Experiments . . . 88
6.2.1 Experiment 1—functional requirements . . . 88
6.2.2 Experiment 2—reliability . . . 91
6.2.3 Experiment 3—security . . . 94
6.3 Analysis of requirements . . . 95

7 Discussion **99**
7.1 Rethinking the NFC Java library . . . 99
7.2 Improving the system . . . 100
7.2.1 UID and RID . . . 100
7.2.2 Token-based authentication . . . 101
7.2.3 Other enhancements . . . 102
7.3 Insights and applications . . . 103

8 Conclusions **105**

A MIFARE Ultralight C **111**
A.1 Memory organization . . . 112
A.2 Security . . . 113

B New NFC Java library **117**
B.1 Evaluation of the MDF implementation . . . 118
B.2 Evaluation of the MUC implementation . . . 124

C Creation of digital certificates **127**

Chapter 1

Introduction

Near Field Communication (NFC) is a recent but promising technology, that enables two NFC-enabled devices, or an NFC-enabled device and an NFC tag, to wirelessly exchange data. NFC-enabled devices include NFC readers, such as contactless Point of Sale (POS) terminals, and NFC-enabled mobile phones. NFC devices interact with one another when brought close together.

NFC devices can operate in three different modes: peer-to-peer mode, card emulation mode and reader-writer mode. In *peer-to-peer mode*, two NFC-enabled devices exchange data, for example, business cards. In *card emulation mode*, an NFC-enabled device acts as a tag and communication takes place with another NFC-enabled device. This mode makes it possible for the mobile phone to replace cards and keys, such as credit cards, travel cards, coupons, car keys and house keys. This means that the already ubiquitous mobile phone will have an even greater impact in our daily lives. In *reader-writer mode*, an NFC-enabled device reads from and writes into a tag or another NFC-enabled device operating in card emulation mode. This thesis focuses on this popular [32] last operating mode.

1.1 Problem statement and methodology

> *Alice uploads her completed assignment to the system and prepares to leave the department. It is nearly midnight. She walks towards the bus stop and realizes she forgot to top-up her travel card at the station earlier that day. Alice uses her NFC-enabled phone to pay for a top-up online and update her travel card.*

To update a smart card, e.g. a travel card, it is necessary to go to a point of sale of the service provider, i.e. the place where a retail purchase is completed. Alternatively, some service providers make available point-of-sale terminals,

allowing customers to autonomously—and locally—update their cards. Our objective is to enable the remote update of cards with the help of an NFC-enabled mobile phone and of a remote server.

The typical architecture of the reader-writer operating mode, in the context of mobile devices, consists of an NFC-enabled mobile phone and an NFC tag. The mobile phone touches the tag to acquire a small amount of data that can be displayed on the screen of the device or trigger an action such as downloading additional content from the Internet, opening an application, calling a number or sending a message. The mobile phone receives data from the tag but does not write into it. In fact, it is reasonable to assume the tag to be locked to prevent accidental or malicious data alteration.

This thesis presents a new usage model for the reader-writer operating mode, where the mobile phone reads and updates the tag according with instructions dictated by a remote server. In this architecture, the commands sent to the smart card come from the remote server instead of being produced by the mobile phone. The mobile phone effectively acts as a proxy, enabling the remote server to reach the card and manipulate its contents. A range of new possibilities such as topping-up travel cards and loading vouchers into loyalty cards while on the move becomes possible in this novel architecture.

We start by studying and reverse engineering the functionality of DESFire EV1, using the specification of its predecessor and other information found online, since we do not have access to its technical documentation. We create a set of requirements for a system capable of remotely updating files on a card application of DESFire EV1. Then we base ourselves on the knowledge acquired about this smart card to devise a remote update protocol that satisfies the requirements previously set. This protocol is evaluated through a prototype we built as a proof of concept, which in turn is subject to experiments in a public cloud provider.

These experiments test the prototype to find if the functional, reliability and security requirements previously set for the system are met. In addition, we test the network latency and server congestion effect on the card, and measure the time taken to perform a card operation to verify that it completes in a timely manner. The experiments are carried in the pay-per-use public cloud provider Jelastic [14], which offers Platform as a Service (PaaS) cloud computing hosting for Java-based applications. We chose to host the remote server in the cloud due to its benefits [1], namely the provisioning of resources on demand and only paying for the resources used.

This thesis demonstrates that it is feasible for a remote server hosted in a cloud provider to update the files contained in a card application of a DESFire EV1 via an NFC-enabled mobile phone, and that this can be accomplished in a reliable and secure way. We analyze the failures that may

affect the remote update protocol and provide a solution to those failures leading to a robust protocol. Its security comes from the use of encrypted communication between server and mobile phone, from the use of the enciphered communication mode of DESFire EV1 between the mobile phone and the smart card, and from the fact that the mobile phone does not have access to the secret keys required to read and update the data stored on the card application. The remote update protocol is optimized by minimizing the number of round trips on the communication between server and mobile phone. Finally, and as a result of the laborious task of studying the functionality of DESFire EV1, a library that eases the manipulation of DESFire EV1 and of Ultralight C smarts cards is produced a part of this work.

1.2 Structure of the thesis

The thesis is organized in eight chapters. Chapter 2 presents background work related to the topic, namely about smart cards and near field communication. Chapter 3 provides a comprehensive study of the MIFARE DESFire EV1 smart card, which is necessary to understand the following chapters. The focus of this study is on the security mechanisms of DESFire EV1. Chapter 4 proposes a solution to the problem. It achieves this by creating a set of requirements for a system capable of solving the problem stated and conceiving a remote update protocol that meets those requirements. Chapter 5 details the design of the proof of concept prototype. Chapter 6 evaluates the proposed solution against the requirements previously set for the system. Chapter 7 discusses potential improvements for the prototype and provides insights into the researched topic. Finally, Chapter 8 concludes the thesis.

In addition to the chapters previously mentioned, the thesis includes three appendices. Appendix A provides a study of MIFARE Ultralight C, that was carried after studying MIFARE Ultralight and before studying the more complex MIFARE DESFire EV1. MIFARE Ultralight is quite similar to MIFARE Ultralight C and is not included in the thesis for that reason. Appendix B presents a Java library to manipulate both MIFARE Ultralight C and MIFARE DESFire EV1 smart cards. This library is a working proof of the card functionalities and was key in understanding how MIFARE DESFire EV1 operates. The resulting knowledge assisted in developing the proposed solution, taking into account the behaviors of the card. Appendix C shows the commands used to created the digital certificates used by the prototype.

Chapter 2

Background and related work

The ubiquity and increasing processing, memory, network and battery capacities of smart phones made them a platform of choice for multiple services. Long gone are the days when mobile phones were used just to make calls. From instant messaging and browsing the web to multiplayer games and movies, these mobile devices pack an assortment of applications and multimedia capabilities. With the rise of near field communication, the mobile world is yet again taking another leap forward. Among many other applications, NFC-enabled systems can replace house and car keys, credit and debit cards, travel cards, boarding passes, cinema and concert tickets, and they can be used to track and monitor both persons and objects and adjust the surrounding environment to our preferences.

Mobile devices, smart phones in particular, are central in the NFC world. The integration of personal and private information, such as credit cards, into mobile phones is the main motivation for NFC. However, while mobile phones are intrinsically connected to NFC, by no means are smart phones the only way to take advantage of this technology. An NFC-enabled wristwatch can be used to open a door. A new kind of NFC-enabled device can be created to work only as an electronic card wallet and key ring. This new device would hold credit and debit cards, identification and social security number cards, and the house and car keys. Should the mobile phone, which is often connected to untrusted networks, be subject to malicious software, lost or stolen, the cards and the keys would still be secure on the other dedicated device since it is independent from the compromised device. This offers an additional sense of security to the owner, at the expense of yet another device to carry.

This chapter contains three sections. Section 2.1 introduces smart cards. Section 2.2 presents the NFC technology including the operation and communication modes, the relevant standards, and a listing of possible attacks

to tags. Section 2.3 provides examples of research papers related to each of the three operating modes of NFC and aims to illustrate the potential of the technology.

2.1 Smart cards

Nearly 7.7 billion microcontroller-based smart secure devices will be shipped in 2013 according to Eurosmart [5]. Smart cards are portable computational devices—containing a memory or a microprocessor chip—used in a variety of sectors, such as banking, retail and transportation, for identification, authentication, authorization and data storage purposes. The motivation behind smart cards is the need for a convenient and efficient way to store portable records, namely personal and private information. Consider a mass transit operator using a ticketing system based on smart cards. Customers swipe the contactless smart card through a reader to validate their ticket, a practical and easy to use solution. A personal travel card is easily updated or reloaded through automated processes and canceled if lost or stolen. While the initial system setup is costly, smart cards by themselves are affordable. The operator also gains valuable data, which can be used to optimize the mass transit system. For example, the amount of customers that used a given station during a period in time and their transfers and itinerary. As the data is in electronic form, it can be efficiently processed and transferred.

Capability-wise, smart cards can be grouped into memory-based cards and into microprocessor-based cards. Memory-based cards have no data processing capacity and data is read from and written to fixed memory addresses. Intelligent memory cards have an additional access control logic imposing restrictions on which memory sectors can be read or written. Microprocessor-based cards comprise a microprocessor, a memory and an operating system. In addition to data processing capabilities, an elaborate access control mechanism providing a higher level of security in comparison to memory-based cards is usually included. These cards may have a file system capable of containing several coexisting applications. Smart card operating systems (SCOS) eased the implementation of multiple applications on the same card.

Until the mid-nineties it was a challenge to develop even a single application for smart cards, mainly due to memory constraints. Multi-application smart card operating systems enabled the implementation of multiple applications on a single card and to dynamically load new ones during its lifecycle. The two main multi-application smart card operating systems are Java Card and MULTOS [4, 37]. In its simplest form, the architecture of a multi-application SCOS consists of the hardware, that is the smart card chip, the

operating system and one or more applications.

Java Card enables multiple applications written in the Java Card language to run on the same smart card. The applications are known as applets and the Java Card language is a subset of the Java programming language. The applets are interpreted by the Java Card Virtual Machine (JCVM), ensuring their independence from the underlying hardware and promoting interoperability. The JCVM is divided into the byte code interpreter and the converter. The byte code interpreter runs on the card and the converter runs outside of the card, handling tasks such as class loading, byte code verification and optimization. The Java Card Runtime Environment (JCRE) provides a firewall, which keeps each applet in its own context, enforcing application isolation.

MULTOS uses the MULTOS Executable Language (MEL) for application development, running on a virtual machine. MEL is an optimized language for smart cards based on Pascal P-codes. Applications can be written in C or Java, among other programming languages, and then translated into MEL. In comparison with Java Card, MULTOS smart card manufacturers are restricted to use the well-defined provided API and are not authorized to create their own APIs. The aim of this policy is to achieve real interoperability between smart card manufacturers.

Smart cards are divided into contact smart cards and contactless smart cards. Contact smart cards are inserted into a reader and the connectivity is established through a contact pad on the surface of the card. Contactless smart cards are held in close proximity to the reader and connectivity is established through inductive coupling. Readers are usually connected to a computer which handles most of the data processing. An alternative to smart card readers are terminals. Terminals can be thought of as a self-contained computer, e.g. a Point of Sale, used to interact with the card, and that communicate with the outside world via a network connection. Further classifications of smart cards include hybrid smart cards and dual-interface cards. Hybrid smart cards have two independent chips, each with its own interface. Dual-interface smart cards have two interfaces but a single chip.

Contactless smart cards can be grouped in three different types according with the operating distance from the card reader. These are proximity cards (PICC), vicinity cards (VICC) and close-coupled cards (CICC). The operating range is up to 10 cm for PICCs, up to 1 m for VICCs and up to 1 cm for CICCs. CICCs operation may be designated as slot or surface operation because the card is either inserted into a slot or laid on a marked surface on the reader. In practice, CICCs do not offer significant advantages in relation to contact smart cards, which partly explains their low adoption by the market [34].

Popular contactless smart card systems include FeliCa and MIFARE. *FeliCa* is a trademark of Sony and offers a line of contactless integrated circuit products popular in Japan. *MIFARE* is a trademark of NXP Semiconductors for a line of contactless integrated circuit products, including card and reader components, that follow ISO/IEC 14443. This standard is used in more than 80% of all contactless smart cards according to Coşkun, Ok, and Özdenizci [2, ch. 1].

Two smart cards of the MIFARE line of products, the MIFARE Ultralight C and the MIFARE DESFire EV1, are analyzed in this thesis in Appendix A and Chapter 3 respectively. The remote update protocol detailed in Section 4.2 and the implemented prototype detailed in Chapter 5 are based on and use DESFire EV1.

2.2 Near field communication

Near field communication (NFC) is a technology relying on inductive coupling for low bandwidth short-range—commonly 4 to 10 cm—wireless connectivity. It operates at a frequency of 13.56 MHz enabling data transfers at up to 424 Kbit/s. Deloitte predicts that up to 300 million NFC-enabled smart phones, tablets and e-readers will be sold in 2013 [3].

Central to NFC is the touch paradigm. An NFC-enabled device—for example, an NFC-enabled mobile phone—touches another NFC-enabled device or an NFC tag to initiate an action. If the NFC-enabled device touches a target with the expected form of data, this intuitive mechanism prompts NFC applications to start. The result of this process ranges from paying for a service using one of the credit or debit cards stored in the NFC-enabled device or validating the presence of the carrier at the workplace to sharing data between two devices such as business cards. The NFC Forum created the N-Mark, which is the universal symbol for NFC. The N-Mark indicates that a device or object is NFC compliant and functions as a touch point, showing consumers where to touch to initiate NFC actions. Touch, swipe and tap are terms found in the literature that essentially have the same meaning. The range of operation of NFC is rather small. To trigger an action, the target must come within the electromagnetic field of the initiator, which implies close proximity. However, it is not necessary for the initiator to physically touch the target.

Jaring et al. [13] state that usability studies show the touch paradigm as easy to learn and intuitive to use, and through six test pilots focused on work time management, logging, and electronic reporting, demonstrate that NFC-based systems can enhance mobile and remote workflow and improve

productivity. Textual input in the field is pointed as a disadvantage given the small-sized keyboard of mobile devices and should be avoided. This disadvantage, however, is not restricted to NFC but is part of the mobile ecosystem. Franssila [6] explores user experiences—ease of use, usability and performance—in NFC through a pilot in three units of a company offering security and guarding services. The author positively concludes that the usability and reliability of NFC applications and NFC-enabled phones are important determinants for the success of the technology and that NFC must beat the alternative older technologies in user experience and performance for NFC to succeed, especially when both options offer nearly the same end-user functionality.

Coşkun et al. [2] provide a study of NFC covering both theory and practice. Özdenizci et al. [32] offer a review of NFC literature and propose a content-oriented NFC research framework by categorizing the NFC academic literature in NFC theory and development, NFC infrastructure, NFC applications and services, and NFC ecosystem.

2.2.1 Operating modes

Communication always takes place between two NFC devices: the initiator and the target. The initiator initiates the communication and the target waits for an initiator to start a communication session. The initiator is always an active device but the target can be an active device or a passive device.

An NFC-enabled device is an active device and can operate in reader-writer and peer-to-peer modes, may operate in card emulation mode and is able to generate its own electromagnetic field. A passive device such as an NFC tag does not produce its own electromagnetic field and uses the one produced by the active device instead. The initiator transfers data by directly modulating its electromagnetic field and the target responds to initiator commands through load modulation.

The three operating modes offered are reader-writer mode, card emulation mode and peer-to-peer mode. In *reader-writer mode*, the NFC device acts as the initiator and can read and write data from passive targets. A passive target can be an NFC-enabled device in card emulation mode or an NFC tag. In *card emulation mode*, the NFC device acts as a passive target. Because the NFC tag is being emulated, the NFC device can present itself to readers as different tags as appropriate. In *peer-to-peer mode*, two NFC devices establish a bidirectional communication channel to exchange data. While the initiator must be in active mode, the target can be in either active mode or passive mode. The latter has the advantage of saving energy on the target. Example

applications are respectively buying a movie ticket by taping an NFC-enabled smart phone against a movie smart poster, using an NFC device as both a travel card and a corporate identification card, and exchanging business cards between two NFC-enabled smart phones.

2.2.2 Standards

The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) defined multiple standards regarding NFC and smart cards. The most relevant ones are ISO/IEC 7816, ISO/IEC 14443, ISO/IEC 15693, ISO/IEC 18092 and ISO/IEC 21481.

ISO/IEC 7816 is composed by fourteen parts. The first three parts are specific to contact smart cards and define the physical characteristics of the card, the location and dimension of the contact pads, and the electrical interface and transmission protocols. The remaining parts define, among other topics, the logical structure of the card and the commands and responses and are independent of the physical interface therefore applying to both contact and contactless smart cards. ISO/IEC 14443 is composed by four parts and refers to proximity cards. It specifies the physical characteristics of the card, radio frequency power and signal interface between PCD (Proximity Coupling Device) and PICC, and initialization, anticollision and communication protocols. ISO/IEC 15693 is composed by three parts and is similar to ISO/IEC 14443 but applied to vicinity cards. ISO/IEC 18092 defines an interface and protocol for NFC. In particular, it defines the active and passive communication modes, modulation schemes, codings, transfer speeds and frame formats of the RF interface, and initialization, data collision and transport protocols. NFC devices comply with ISO/IEC 18092 for peer-to-peer mode and with ISO/IEC 14443 for reader-writer and card emulation modes. ISO/IEC 21481 defines a communication mode selection mechanism for devices implementing ISO/IEC 14443, ISO/IEC 15693 or ISO/IEC 18092. An older standard not considered by ISO/IEC 21481 is ISO/IEC 10536 for close-coupled cards.

Another relevant standard is the Japanese Industrial Standard (JIS) X6319-4. This standard, known as FeliCa, was proposed for ISO/IEC 14443 type C but was rejected.

In 2004, Nokia, Philips and Sony formed the NFC Forum [21], which currently has over 170 members. The objective of the forum is developing NFC standards, in order to maintain the interoperability of NFC devices and protocols, and to encourage the use of NFC technology in general. It is also a certification body for NFC devices. Among the standards specified by the NFC Forum, the NFC Data Exchange Format (NDEF) and the NFC Forum

Tag Types are of particular relevance.

The *NFC Data Exchange Format* specification [22] defines a common message format to exchange information between two NFC devices, or a device and a tag. It strictly defines a format to encapsulate application-defined payloads, independently and without making assumptions about the actual data transfer over the communication link.

The *NFC Forum Tag Types* defines four types of tags that are operable with NFC devices [23–26]. The tags differ from one another in the transmission protocol, memory size and organization, and security features. Type 1 and type 2 tags are based on ISO/IEC 14443A. Type 1 tags have 96 bytes of available memory and type 2 tags have 64 bytes of available memory, both expandable to 2 KB. Type 3 tags are based on JIS X6319-4 and have a theoretical memory limit of 1 MB. Type 4 tags are compatible with ISO/IEC 14443 and have a memory of up to 32 KB. Examples of tags are, respectively, Innovision Topaz, MIFARE Ultralight C, Sony FeliCa Lite, and MIFARE DESFire EV1.

2.2.3 Security

Incentives to attack an NFC system range from the challenge and reputation that a successful attack comes with to financial rewards and espionage. The following section lists possible attacks to tags and the respective solution or mitigation strategy.

Active attacks

Cloning. Cloning consists in copying the contents from one tag into another tag. This enables, for example, to reuse the credit on the first tag over and over again. The unique ID can be used to prevent tag cloning.

Denial of service. Touching an NFC reader with a tag keeps the device occupied. Leaving the tag in place prevents other users from interacting with the NFC reader.

Data corruption. This is similar to a DoS attack, but instead of keeping an NFC reader active by touching it with a forged tag, the objective is to corrupt the data being transmitted by two legitimate parties. This attack can be detected by monitoring the RF field while transmitting data, because the power required to corrupt the data is significantly large [12].

Forging/alteration. An attacker modifies the contents of, or replaces, a legitimate tag. Some devices are configured to take actions automatically when a tag is touched, for instance, to open a URL present on the tag. When the user touches a forged tag, she may become the victim of a phishing or spoofing attack.

In a **phishing** attack, the victim is redirected to a forged website and lead to enter personal information such as access credentials, which are captured by the attacker.

In a **spoofing** attack, the victim may unknowingly trigger an action such as making a call to a premium number or sending a message to a third party.

In addition, the legitimate tag may be modified to do nothing or replaced with an empty tag, causing a **denial of service** since the victim is unable to access the intended service.

This attack can be overcome by using signed tags [16].

Relay attack. This attack requires a card reader and a card emulator, respectively the *mole* and the *proxy*, and a low latency communication channel between the two because of timing constraints. The attacker places the mole next to the tag of the victim and relays the data to the proxy. The proxy then touches a legitimate card reader, which believes to be communicating with the tag of the victim. This is feasible because the mole does not have to touch the tag of the victim and because the tag of the victim activates autonomously when it is in the field of the mole [11].

A similar attack, known as software-based relay attack, exists for NFC devices working in card emulation mode. In this scenario, the mole is a malicious application installed in the NFC device of the victim. The communication with the Proxy takes place, for instance, through the cellular network [35].

The first version of the attack can be prevented by physically shielding the card, when not in operation, using a Faraday cage and by using two-factor authentication. In both cases, the attack can be mitigated by introducing distance bounding protocols.

Passive attacks

Eavesdropping. The close range at which NFC operates confers it some security. It is still possible, however, to eavesdrop the communication between two NFC devices or between an NFC device and a tag by using a high-powered antenna. Eavesdropping can

be prevented by establishing a secure channel between the two communicating parties, commonly by means of encryption [12].

2.3 Related projects

This section describes relevant research projects related to each of the three operating modes of NFC and innovative ideas found in the literature. It aims to illustrate the potential of NFC in connection with mobile phones. The following paragraphs give a brief description of the projects and their relevance to this thesis is established at the end of the section.

Ghìron et al. [9] develop an NFC ticketing application to allow users to buy tickets for public transportation by touching a smart poster. The NFC-enabled mobile phone is used in card emulation mode and the smart poster relies on a passive MIFARE tag. Usability plays an important part in this work, which attemps to achieve a good balance between the cognitive load imposed to the user and the functionality of the system.

Saminger et al. [36] present an approach for ticketing systems using an inverse reader mode, which consists in using the NFC-enabled mobile phone in reader-writer mode and the NFC readers of the service provider in card emulation mode. The objective is to bypass access restrictions imposed to the secure element of the phone, which are usually under the control of the handset manufacturer, the mobile network operator (MNO), or a trusted service manager (TSM).

Siira et al. [38] use the reader-writer mode to create an NFC-based wiki system. Tags placed in areas of interest return an identifier when tapped, which is used to download contextual information from a server. A tag at a bus stop, for instance, may return information about the bus timetables and nearby restaurants, tourist attractions and events.

Fressancourt et al. [7] integrate NFC with messaging systems and social networks, to overcome the tedious process related to keeping an up-to-date presence across all platforms. Touching a tag placed in a key location brings up a menu to choose one of two predefined moods, which is then sent to the platforms. The tag may include elements of contextual information such as its location, which raises privacy concerns. The authors of the paper consider the act of touching the tag as an agreement by the user to reveal her location to selected peers.

Nandwani et al. [20] research the use of the peer-to-peer operating mode of NFC for social mobile games. Two games, inspired by parlor games, are developed and tested in different social areas. The usability tests confirm the ease of use of NFC by users and the interacting parties felt more connected

to each other due to the shift of focus from the screen to the player area. It is also argued that the general public awareness of NFC can be increased through gaming.

Matos et al. [17] rely on NFC to address security issues of WiFi configuration and access in public locations. The user touches a passive tag or an active NFC device to receive network parameters and a public key certificate, which are used to establish a secure connection with a local wireless network and confirm the legitimacy of the AP. This method avoids the authentication via captive portals, subject to evil twin and man-in-the-middle attacks, and prevents the eavesdropping associated with using an open network.

The approach followed by Ghìron et al. [9] to create a ticketing application is the obvious one, i.e. using the mobile phone in card emulation mode to replace the travel card. However, this operating mode requires an authorization by the issuer of the secure element in order to gain access to it and this may also imply paying a fee to that entity. Saminger et al. [36] use a different approach to achieve a similar objective and bypass the restriction of the secure element. This is achieved by using the phone in reader-writer mode and the ticketing terminal in card emulation mode. In this thesis we also use the mobile phone in reader-writer mode. However, while Saminger et al. [36] simulate the card using a terminal which is connected to the back-end system, effectively resulting in having both ends of the communication connected to the server, we are interacting directly with an isolated NFC tag which is unable to reach the remote server on its own. These two papers focus on replacing the ticket itself by the mobile phone whereas in this thesis the objective is not to replace the tag but to enable the users of the system to use it remotely.

The papers by Siira et al. [38], by Fressancourt et al. [7], and by Matos et al. [17], use the mobile phone in reader-writer mode to achieve their respective objectives. These three projects illustrate the typical usage model for the reader-writer mode, where the mobile phone touches the tag to acquire a small amount of data that can be displayed on the screen of the device and trigger other actions. In our usage model, the mobile phone not only reads from but also writes into the tag and is able to do it in a reliable and secure way, following the instructions of a server hosted on the cloud.

The usability tests by Nandwani et al. [20] confirm the ease of use of NFC, which is an aspect of the technology that we should not neglect. The end users must feel comfortable using the technology in order to accept it. This thesis uses a different operating mode but the same usability principles apply.

Chapter 3

MIFARE DESFire EV1

MIFARE DESFire EV1 is the evolution of MIFARE DESFire (MF3ICD40). The security of MF3ICD40 was broken in 2011, when researchers were able to retrieve the secret key by means of a power analysis attack [31].

DESFire EV1 is a contactless smart card that holds up to 28 applications. Each application can have up to fourteen keys associated with it and contain up to 32 files. There are five different file types. Each file has its own file settings, which include communication mode and access control parameters. The communication modes are plain, mac'ed and enciphered and are discussed in Section 3.3.4. Each DESFire EV1 has a single PICC master key and supports DES, 2K3DES, 3K3DES and AES. In comparison, its predecessor, DESFire, only holds up to sixteen files per application and supports DES and 3DES. DESFire EV1 is backward compatible with DESFire [28].

This chapter discusses the operations and data storage of DESFire EV1 and its security mechanisms. It focuses on the cryptographic operations taking place, authentication protocol and data transmission modes between the reader and the card. The objective is to understand the functionality of DESFire EV1 before attempting to solve the problem statement from Section 1.1.

The specification of DESFire EV1 is not publicly available and other resources had to be used instead to reach a deep understanding of its functionality. An old specification [33] of its predecessor, DESFire, can be found online and Kasper, von Maurich, Oswald, and Paar [15, sec. 3.2] expose the AES authentication protocol for DESFire EV1; the libfreefare [19] project aims at providing a C library for the manipulation of MIFARE cards, but unfortunately the code is not documented and the card reader used for the thesis is incompatible with this library. Additionally, pieces of information regarding MIFARE smart cards is found in blogs and forums via search engines, albeit often incomplete.

3.1 Commands

Level	Commands
Security-related	`Authenticate`, `ChangeKeySettings`, `SetConfiguration`, `ChangeKey`, `GetKeyVersion`
PICC-level	`CreateApplication`, `DeleteApplication`, `GetApplicationsIDs`, `FreeMemory`, `GetDFNames`, `GetKeySettings`, `SelectApplication`, `FormatMF3ICD81`, `GetVersion`, `GetCardUID`
Application-level	`GetFileIDs`, `GetFileSettings`, `ChangeFileSettings`, `CreateStdDataFile`, `CreateBackupDataFile`, `CreateValueFile`, `CreateLinearRecordFile`, `CreateCyclicRecordFile`, `DeleteFile`
File-level	`ReadData`, `WriteData`, `GetValue`, `Credit`, `Debit`, `LimitedCredit`, `WriteRecord`, `ReadRecords`, `ClearRecordFile`, `CommitTransaction`, `AbortTransaction`

Table 3.1: List of commands of DESFire EV1, grouped by level.

DESFire EV1 commands are divided in four levels: security-related commands, PICC-level commands, application-level commands and data manipulation (file-level) commands. Table 3.1 lists the available commands for the manipulation of the contents of DESFire EV1. The commands were retrieved from the short data sheet of the card, made available by NXP Semiconductors [28].

Security-related commands include authentication and key-related operations. PICC-level commands include application and memory manipulation operations. Application-level commands include operations to manipulate files such as file creation and file deletion. File-level commands include operations that manipulate data such as reading from files and writing to files. See Table 3.1 for a list of commands organized by level.

3.2 File system

DESFire EV1 has a flexible file system capable of containing up to 28 applications and up to 32 files in each application. One can think of the file

system as directories containing files, where the directories are the applications. Each application is represented by a 3-byte application identifier (AID), set on application creation. Each file is represented by a 1-byte file number, set on file creation. Access to files may require a preceding authentication, depending on the file access rights. Keys and access rights are discussed in Section 3.3.3 and in Section 3.3.4.2 respectively.

File type	Description	Commands
Standard data file	Storage of unformatted user data	`ReadData` `WriteData`*

Backup data file	Storage of unformatted user data + integrated backup mechanism	`ReadData`
		`WriteData`*
		`CommitTransaction`
		`AbortTransaction`

Value file	Storage and manipulation of a 32-bit signed integer	`ReadData`
		`WriteData`*
		`GetValue`
		`Credit`*
		`Debit`*
		`LimitedCredit`*
		`CommitTransaction`
		`AbortTransaction`

Linear record file	Storage of structured user data
	(e.g. loyalty programs)
		`WriteRecord`*
		`ReadRecords`
		`ClearRecordFile`*
		`CommitTransaction`
		`AbortTransaction`

Cyclic record file	Storage of structured user data + automatically overwrite oldest record when full
	(e.g. logging transactions)
		`WriteRecord`*
		`ReadRecords`
		`ClearRecordFile`*
		`CommitTransaction`
		`AbortTransaction`

Table 3.2: DESFire EV1 file types and data manipulation operations. The starred commands require validation.

Table 3.2 summarizes the main points of the types of files available on the card. The files contained by an application can be of the same type or of different types. DESFire EV1 supports five different types of files, each one with a different purpose and with a set of operations that can be performed on it. The types of files are standard data files, backup data files, value files,

linear record files, and cyclic record files. Standard data files and backup data files are used for the storage of unformatted user data. Value files are used for the storage and manipulation of a 32-bit signed integer value. Linear record files and cyclic record files are used for the storage of structured user data. Structured user data are also called records. A record is a single piece of data organized in a predetermined way and with size, in bytes, set on file creation. A linear record file will become full of records at some point, after which it has to be cleared for further writing. A cyclic record file automatically overwrites the oldest record once it is full. Backup data files, value files, linear record files and cyclic record files include an integrated backup mechanism and require a `CommitTransaction` to validate commands that modify data stored on the card. Alternatively, an `AbortTransaction` can be used to abort the command and proceed with other operations on the card. Multiple commands may be issued, within the same application, with a single commit or abort command at the end. This applies to commands targeting the same file or different files, irrespective of the file type. If commands requiring validation are sent to the card and a `SelectApplication` takes place before a `CommitTransaction`, the data modifications requiring validation are discarded. The backup mechanism takes an additional portion of memory from the card. For backup data files, it consumes double the non-volatile memory when compared with standard data files with the same size. For cyclic record files, it consumes one extra record, which must be taken into account when defining the maximum number of records upon file creation.

3.3 Security

The security of DESFire EV1 relies on multiple mechanisms. It uses error-detecting codes to detect unintentional changes to the data, message authentication codes to ensure the data authenticity and integrity, and encryption algorithms to ensure data confidentiality. The error-detecting codes used are CRC16 and CRC32, implemented according with ISO/IEC 14443A and ITU-T Recommendation V.42 respectively. The message authentication codes are CBC-MAC and CMAC and the encryption algorithms are DES, 2K3DES, 3K3DES and AES. The cryptographic primitives of DESFire EV1 are further explained in Section 3.3.1.

Each DESFire EV1 can be identified by its unique UID, which is programmed into the device during production and cannot be altered. This is done by writing into a manufacturer reserved part of the non-volatile memory and write-protecting those bytes. MF3ICD81 conforms to the Smartcard IC

Platform Protection Profile and is certified by the German Federal Office for Information Security according with the Common Criteria for Information Technology Security Evaluation (CC) at level EAL 4 augmented [8, 29].

3.3.1 Cryptographic primitives

DESFire EV1 supports the following cryptographic primitives: CBC-MAC, CMAC and encryption and decryption using the ciphers Triple DES with 56/112/168-bit keys and AES. Each message authentication code is associated with a cipher. CBC-MAC is used in conjunction with DES and 2K3DES and CMAC is used in conjunction with 3K3DES and AES.

There is a notion of a *global IV*, where the input or output of an encryption, decryption or CMAC operation is used as initialization vector for the next cryptographic operation. This only applies to 3K3DES and AES. The global IV is defined as the last block of the ciphertext resulting from an encryption, as the last block of the ciphertext about to be decrypted and as the result of a CMAC operation. These cryptographic operations—encryption, decryption and CMAC calculation—use the session key created after a successful authentication as secret key. The global IV is set to zeros after the successful authentication and is only applied while the authenticated state is valid.

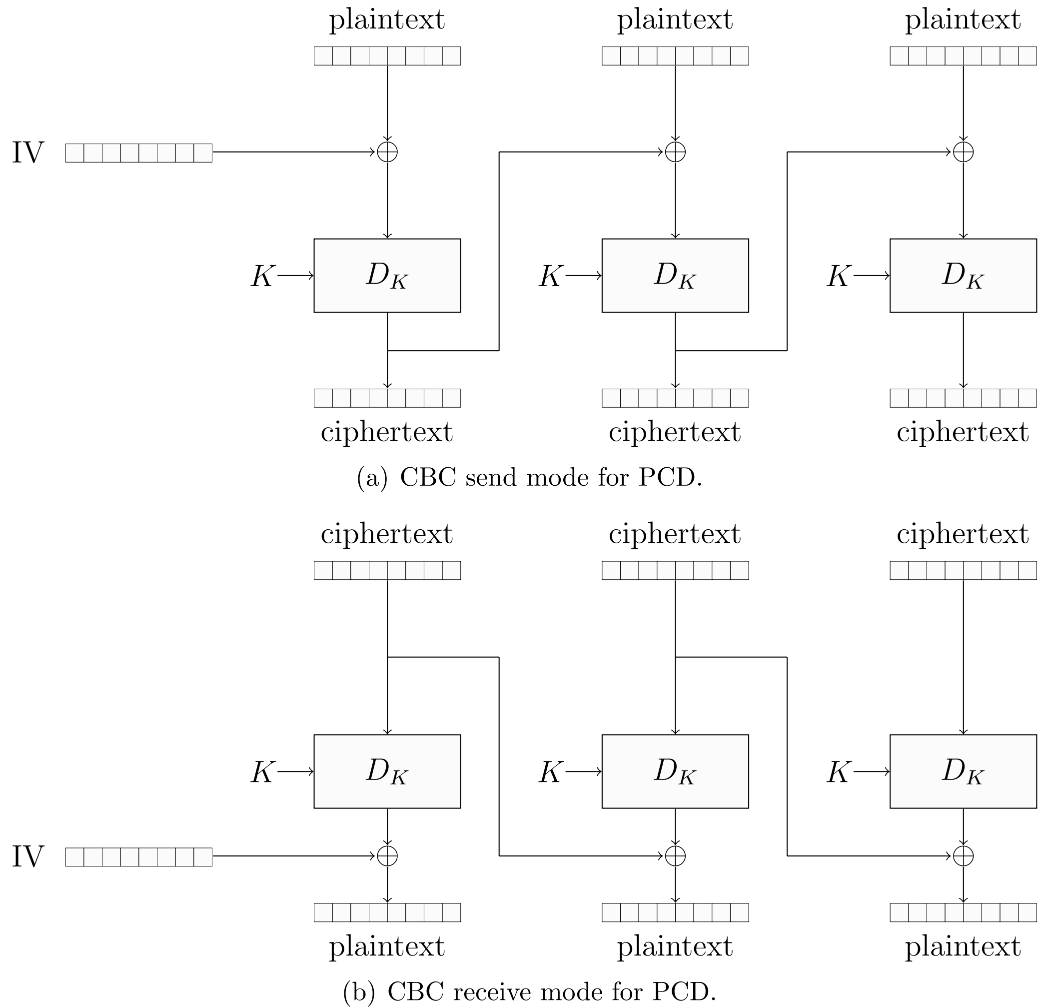
The CBC-MAC is calculated by applying Triple DES encryption in CBC mode to the data. It is used with DES and 2K3DES, which has a block size of 8 bytes. Since the data length must be multiple of 8 bytes it is padded with zeros if required. The MAC is defined as the first half of the last block, that is, the first 4 bytes of the last 8-byte block.

The CMAC is calculated according with the NIST Special Publication 800-38B, with the modification that the encryption function, during MAC generation, receives the global IV as its initialization vector instead of an IV composed by zeros. For 3K3DES, the resulting 8-byte CMAC is used as is. For AES, the first half of the resulting block is used as CMAC, that is, the first 8 bytes of the resulting 16-byte block.

The data length is included in the commands that read data and in the commands that write data. The padding of the CBC-MAC and the padding of the CMAC is only used for the computation and is not exchanged between PICC and PCD.

When using a DES or 2K3DES cipher, the PCD always decrypts the data and the PICC always encrypts the data.[1] The cryptographic operation

[1] The DESFire MF3ICD40 is only able to encrypt data in order to save on hardware implementation costs. The DES and 2K3DES related operations on DESFire EV1 are



(a) CBC send mode for PCD.

(b) CBC receive mode for PCD.

Figure 3.1: CBC send mode and CBC receive mode for PCD.

is done in either CBC send mode or CBC receive mode. In CBC send mode, the PCD performs the logical operation XOR before decrypting a block of data. The data block being decrypted is xor'ed before the decryption, with the previously decrypted data block, as seen in Figure 3.1(a). In CBC receive mode, the PCD performs the logical operation XOR after decrypting a block of data. The data block being decrypted is first decrypted and only then it is xor'ed, with the previous 8-byte block of ciphertext, as seen in Figure 3.1(b). In both cases and since there is no previous block, the first block being decrypted is xor'ed with an all-zero initialization vector.[2]

backward compatible with DESFire MF3ICD40.

[2]For all practical purposes, this returns the first block without any modification, making

When using 3K3DES or AES, the PCD encrypts when sending data and decrypts when receiving data. CBC mode is used in both cases. In comparison to DES and 2K3DES, there is an IV maintained between cryptographic operations: the global IV. The first block of data being encrypted or being decrypted is xor'ed with this IV and not with the all-zero IV. Like previously mentioned, this is the same IV used in the encryption operation that takes place during the MAC generation phase of CMAC.

Operation	DES/2K3DES	3K3DES/AES
CBC-MAC	DES/2K3DES encryption + CBC	
CMAC		CMAC + reuse of IV in MAC generation phase
Encryption	DES/2K3DES decryption + CBC send mode	3K3DES/AES encryption + CBC + reuse of IV
Decryption	DES/2K3DES decryption + CBC receive mode	3K3DES/AES decryption + CBC + reuse of IV

Table 3.3: Cryptographic operations in DESFire EV1.

The cryptographic primitives detailed in the previous paragraphs are outlined in Table 3.3. The operations for DES are essentially the same for 2K3DES and the same relation applies to 3K3DES and AES.

3.3.2 Authentication protocol

A mutual three-pass authentication may take place before transmitting data, which ensures that both the PCD and the PICC are in possession of a common secret. This secret, the secret key, is stored on the PICC and the PCD is required to have the correct key for the authentication to succeed. A successful mutual authentication results in a session key that can be used to protect subsequent data transmissions. The data transmission modes supported by DESFire EV1 are described in Section 3.3.4.

The authentication protocol is similar for all ciphers. The differences are on the length of the random numbers generated, the algorithms used for the encryption of data and for the decryption of data, and the session key generation algorithm. The length of the random numbers is 8 bytes for DES and 2K3DES, and 16 bytes for 3K3DES and AES. The encryption and decryption algorithms are presented in Section 3.3.1, and the session key

---

the XOR operation redundant.

Cipher	Session key
DES	$RndA_{0-3} \| RndB_{0-3}$
2K3DES	$RndA_{0-3} \| RndB_{0-3} \| RndA_{4-7} \| RndB_{4-7}$
3K3DES	$RndA_{0-3} \| RndB_{0-3} \| RndA_{6-9} \| RndB_{6-9} \| RndA_{12-15} \| RndB_{12-15}$
AES	$RndA_{0-3} \| RndB_{0-3} \| RndA_{12-15} \| RndB_{12-15}$

Table 3.4: Session key generation for DESFire EV1 using *RndA* and *RndB*.

generation algorithm is presented in Table 3.4. A diagram of the DES and 2K3DES authentication protocol is depicted in Figure 3.2, and a diagram of the 3K3DES and AES authentication protocol is depicted in Figure 3.3. The authentication protocol is always initiated by the PCD and it is composed by the following steps:

1. The PCD sends an authentication request to the PICC, along with a 1-byte key number. This key number references the secret key to be used during the authentication protocol. The secret key can be part of an application or it can be the PICC master key, depending on the selected AID. After powering up the PICC, the 3-byte AID `00 00 00`$_H$ is implicitly selected.

2. The PICC receives the authentication command and generates and encrypts a random number *RndB*. The length of the random number generated and the algorithm used for encryption are related to the cipher associated with the key number received from the PCD. The resulting ciphertext is sent to the PCD as a response.

3. The PCD receives and decrypts the response obtaining the random number *RndB* generated by the PICC. It then rotates *RndB* one byte to the left yielding $x_2 = RndB'$. The PCD generates its own random number *RndA* and concatenates $RndB'$ to it. $RndA \| RndB'$ is encrypted and sent to the PICC.

4. The PICC decrypts the received message, rotates its *RndB* to the left and compares it with the decrypted $RndB'$ sent by the PCD. If the match fails, the PICC returns an error code to the PCD. Otherwise, it rotates *RndA* one byte to the left, yielding $RndA'$ (x_8). $RndA'$ is encrypted and sent to the PCD.

5. The PCD decrypts the received message, rotates its *RndA* one byte to the left and compares it with the $RndA'$ received from the PICC. If the match fails ($x_{10} \neq x_{11}$), the PCD may abort the authentication

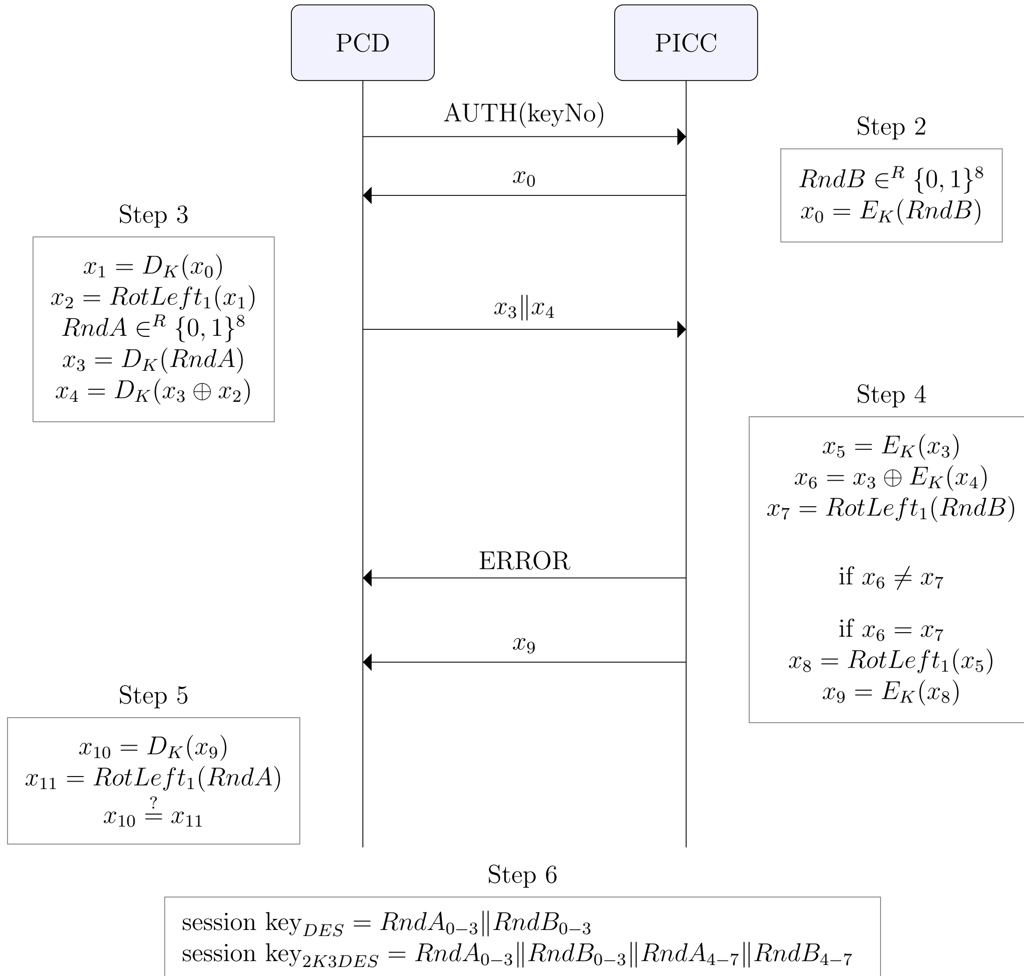


Figure 3.2: DES and 2K3DES authentication protocol for DESFire EV1.

protocol at this point. If the verification is successful ($x_{10} = x_{11}$), then the PCD is ready to generate a session key.

6. The session key is generated by the PCD and by the PICC using both *RndA* and *RndB* according with Table 3.4. For 3K3DES and for AES, the global IV is reset to zeros at this point and is reused between all subsequent cryptographic operations. For DES and for 2K3DES, cryptographic operations are independent from each other and always start with an IV set to zeros.

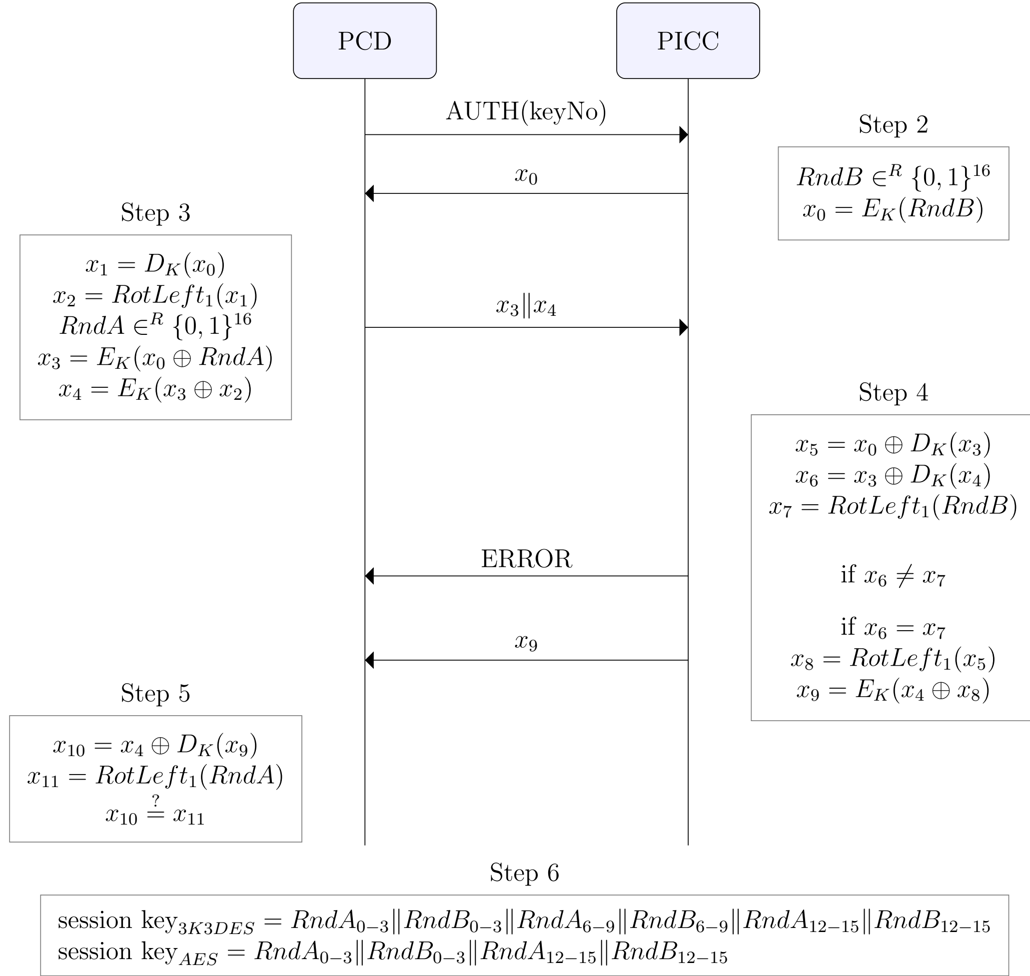


Figure 3.3: 3K3DES and AES authentication protocol for DESFire EV1.

3.3.3 Keys

There is a single PICC master key and up to fourteen keys per application. Each application always has at least one key: the application master key. The PICC master key and the application master key have the key number 00_H. For certain commands, a preceding authentication with a specific key is a requisite for their successful completion. For instance, a preceding authentication is compulsory when changing a key. For other commands, however, whether a preceding authentication is required for a successful completion or not depends on the access rights of files, explained in Section 3.3.4, and on the PICC master key settings or on the application master key settings.

The PICC master key settings apply to PICC level and each application

has its own application master key settings. In both cases, the size of the master key settings is 1 byte. These settings define if the master key is changeable and if the master key settings are changeable. In addition, the master key settings indicate if an authentication is required to perform certain operations such as getting the master key settings and creating applications and files. This is achieved by toggling bits on the master key settings byte. The MF3ICD40 specification [33, sec. 4.3.2] provides more information about the master key settings and about which commands will require a preceding authentication when bits are toggled.

The length of the keys, when interacting with the card, is 8 bytes for DES, 16 bytes for 2K3DES, 24 bytes for 3K3DES, and 16 bytes for AES. Triple DES keys use the least significant bit of each byte as a parity bit, which means that the actual size of the keys is respectively 56, 112 and 168 bits for DES, 2K3DES and 3K3DES. DESFire EV1 ignores the parity bits when handling Triple DES ciphers and these can be used for key versioning. It is advisable to set the parity bits to a default value, if not used for key versioning, because the PICC will reveal them with a `GetVersion` command. This would reduce the strength of the key if the bits were really used as parity bits. The key version is only one byte and for Triple DES it is taken from the first key. If using AES, there is an extra byte specifically for the purpose of key versioning.

Internally, DESFire EV1 handles both DES and 2K3DES keys as 16-byte keys. When the first half of the key is equal to the second half of the key, the key type is considered to be DES. When the first half of the key is different from the second half of the key, the key type is considered to be 2K3DES. For example, `20 00 00 00 20 00 00 00 20 00 00 00 20 00 00 00`$_{\mathrm{H}}$ is a DES key and `20 00 00 00 60 00 00 00 00 00 00 00 00 00 00 00`$_{\mathrm{H}}$ is a 2K3DES key. Note that a key such as `00 00 00 00 00 00 00 02 00 00 00 00 00 00 01 03`$_{\mathrm{H}}$ is a DES key because the key values only differ in the parity bits, i.e. key version. In the example, the version of the key would be `00`$_{\mathrm{H}}$ and clearing the parity bits results in the key `00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 02`$_{\mathrm{H}}$. In the authentication, differences in the parity bits have no effect in the access as failure.

The cipher used by an application is decided during its creation together with the number of keys. Three possibilities exist for this option: DES+2K3DES, 3K3DES and AES. The type of cipher cannot be changed for the application later on. The default secret keys, upon application creation, are 16 bytes set to zero for DES+2K3DES and AES and 24 bytes set to zero for 3K3DES. Since DES and 2K3DES are grouped, it is possible to have DES and 2K3DES keys mixed within the same application and it is possible to change a key from DES to 2K3DES and vice versa. When using

DES and 2K3DES as the application cipher, the default key is 16 bytes set to zero, which means it is a DES key. The cipher of the PICC master key can be altered during a key change by tweaking the key number.

To change a key, the PCD sends the message $C4\|keyNo\|ciphertext$ to the PICC. $C4$ is the command code. $keyNo$ is the number of the key to be changed and contains a value between `00` and `0D`. $ciphertext$ contains information about the new key and its generation is presented in Algorithm 3.1. When changing the PICC master key, the $keyNo$ is tweaked according with the cipher used by the new key. When the new key type is 3K3DES, the key number is set to $\texttt{40}_{\texttt{H}}$, and when the new key type is AES, the key number is set to $\texttt{80}_{\texttt{H}}$. For both DES and 2K3DES keys, the key number is set to $\texttt{00}_{\texttt{H}}$. For applications, the tweak is not required because the cipher chosen when the application is created is fixed.

3.3.4 Data transmission

The three communication modes supported by DESFire EV1 are plain, mac'ed and enciphered. Which communication mode is used during an operation depends on the file settings, if dealing with files, and on the command itself. A rule of thumb is that the plain communication mode is used for all commands, unless stated otherwise.

Each file is associated with its own file settings. The file settings indicate the file size for data files, the record size, boundaries and current number of records for record files, and the boundary values and state of the limited credit option for value files. Additionally, the file settings include the communication settings and the access rights for the file. The communication settings point out the communication mode to use when operating on this particular file. The access rights state whether a preceding authentication is required to interact with the file or not and, if so, which key number the PCD is to authenticate against.

3.3.4.1 Communication settings

The different communication settings are applied to the file-level commands `ReadData`, `WriteData`, `GetValue`, `Credit`, `Debit`, `WriteRecord`, `ReadRecords` and `LimitedCredit`. The application-level command `ChangeFileSettings` uses the enciphered communication mode and requires the preceding authentication to be done with the CAR key of the related file. If the CAR key of that file is set to free access, then the data is exchanged in cleartext. The commands `ChangeKeySettings` and `GetCardUID` use the enciphered communication mode. The commands `Authenticate` and `ChangeKey` use their own

keyNo	number of the key to be changed
new key	the new key
old key	the old key (only required if changing different keys)

```
if changing the PICC master key then
    if type of new key is 3K3DES then
        keyNo ← 40_h
    else if type of new key is AES then
        keyNo ← 80_h
    end
else if authentication key number ≠ number of key to change then
    temp ← new key ⊕ old key (concatenate multiple copies of old key if
    necessary)
end
if new key type is AES then
    plaintext ← temp + key version
else
    plaintext ← temp
end
switch type of key used for authentication do
    case DES or 2K3DES
        compute the CRC16 of plaintext
        append the resulting CRC16 to plaintext
        if authentication key number ≠ number of key to change then
            compute the CRC16 of new key
            append the resulting CRC16 to plaintext
        end
        ciphertext ← decryption of plaintext in CBC send mode
    endsw
    case 3K3DES or AES
        compute the CRC32 of cmd||keyNo||plaintext
        append the resulting CRC32 to plaintext
        if authentication key number ≠ number of key to change then
            compute the CRC32 of new key
            append the resulting CRC32 to plaintext
        end
        ciphertext ← encryption of plaintext in CBC mode using global IV
    endsw
endsw
```

DES or 2K3DES: $D_{CBC_{send}}(\mathsf{new} \oplus \mathsf{old} \| CRC16(\mathsf{new} \oplus \mathsf{old}) \| CRC16(\mathsf{new}))$
3K3DES or AES: $E_{CBC}(\mathsf{new} \oplus \mathsf{old} \| CRC32(\mathsf{cmd} \| \mathsf{keyNo} \| \mathsf{new} \oplus \mathsf{old}) \| CRC32(\mathsf{new}))$

Algorithm 3.1: Computation of the ciphertext for the change key command.

security mechanisms, which are detail in Section 3.3.2 and in Section 3.3.3 respectively. The remaining commands use the plain communication mode but some of these commands may require a preceding authentication to succeed depending on the PICC master key settings or on the application master key settings, and on the command itself.

A native DESFire EV1 command can be divided in a command code, headers and the data to be secured. The command code is one byte stating which operation is under action. The headers are pieces of data which are part of the payload of the command but which are not to be encrypted. These include the 1-byte key number referencing secret keys, and the 3-byte offset and 3-byte length fields when writing data to files. The data to be secured is the piece of data which is to be stored on the card. A native DESFire EV1 response can be divided in a status code and the data to be secured. The status code indicates whether the command is successful and the error code if it failed, and the data to be secured is the piece of data received from the card.

When calculating a cyclic redundancy check or message authentication code, or when ciphering data, it is necessary to know on which piece of data to perform the operation. The CRC16 and the CBC-MAC are calculated only over the data to be secured for both commands and responses. The CRC32 and the CMAC are calculated over the command code, headers and data for commands, and over the status code and data, if any, for responses. While a native response is presented as the status code followed by the data, the CRC32 and CMAC calculation is done over the data followed by the status code, i.e. the status code and the data swap for the computation. The encryption and decryption takes place only over the data to be secured and the CRC, leaving out the command code and headers and the response status code.

The objective of the communication modes is to protect the data exchanged between the two parties. In other words, while both the command and respective response are interlinked, the security is applied to the direction carrying the data. If writing data into the card, then the command is protected. If reading data from the card, then the response is protected. This is how DES and 2K3DES operate. For 3K3DES and for AES the system is different because all the operations are chained in order to improve the overall security of the system.

The chaining occurring for 3K3DES and for AES relies on a global initialization vector, an IV which is shared between the commands and responses. This IV is read and subsequently updated when computing CMACs and when encrypting and decrypting data. If authenticated, the IV is still kept up to date even when the plain communication mode is used, which means this

chaining is applied to nearly all commands. The exception are commands that cause the authentication state to be changed, such as a new authentication, or that cause the authentication state to be lost, such as an application selection operation.

In the plain communication mode, the data is transmitted between the PCD and PICC in cleartext for DES and for 2K3DES. This is also the behavior for 3K3DES and for AES if not authenticated. If authenticated then the CMAC is calculated over the commands and over the responses enabling all the operations to be chained. For commands, the CMAC is not appended but it is still calculated by the PCD in order to update the global IV. For responses, the CMAC is calculated and appended to the data by the PICC.

In the mac'ed communication mode, a CBC-MAC or CMAC is appended to the data. The message authentication code is appended to the command if modifying data on the card and is appended to the response if reading data from the card. When using 3K3DES or AES, the CMAC is always appended to the response.

In the enciphered communication mode, a CRC is appended to the data for integrity, and the payload—data to be secured and the CRC—is encrypted for confidentiality. For DES and for 2K3DES, the CRC16 is calculated over the data to be secured. For 3K3DES and for AES, the CRC32 is calculated over the command code, headers and data to be secured in the case of commands, or over the data to be secured and the status code in the case of responses. The encryption is done according with Table 3.3.

3.3.4.2 Access rights

Access rights are used to control the access to files. Each file has four access rights associated with it: read (R), write (W), read&write (RW) and change access rights (CAR). Each of these access rights is coded in 4 bits and references one of the keys associated with the file. A file can have between 1 and 14 keys associated with it and this value is set when the file is created. To reference a key, the access right will hold a value between `0` and `D`. The key referenced is required to exist. There are two special values, `E` and `F`, to respectively indicate free access and deny access. Free access means that access is always granted to the linked access right, with or without a preceding authentication. Deny access means that access is always denied to the linked access right, irrespective of the authentication state. Table 3.5 lists the access rights and the commands to which they apply. These are all file-level commands, with the exception of the application-level command `ChangeFileSettings`.

For a command to succeed, one of the associated access rights must be

Command	Access rights			
	R	W	RW	CAR
ChangeFileSettings				✗
GetValue	✗	✗	✗	
Debit	✗	✗	✗	
LimitedCredit		✗	✗	
Credit			✗	
ReadData	✗		✗	
WriteData		✗	✗	
ReadRecords	✗		✗	
WriteRecords		✗	✗	
ClearRecordFile			✗	

Table 3.5: DESFire EV1 access rights associated with commands.

satisfied. This means that a single positive acknowledgment suffices, independently of the amount of access rights associated with a given command. An access right is satisfied when a preceding successful authentication takes place using the referenced key. Alternatively, access is always granted when one of the access rights associated with the command is set to E_H.

If one of the access rights associated with a command references the key number used to reach an authenticated state, then the communication takes place according with the communication settings for that file. Otherwise, one of those access rights may contain the free access value E, in which case communication is done in plain mode, ignoring the communication settings of the file. In this case, the authentication state is irrelevant. If neither of the access rights associated with the command references the authenticated key number or contains the free access value E, the command fails with an authentication error status code.

3.4 Trace

The goal of the following traces is to illustrate the differences between the communication modes and between DES/2K3DES and 3K3DES/AES ciphers in DESFire EV1. The traces are generated by a sample application created for this purpose that uses the library developed during this thesis and presented in Appendix B. The communication between the reader and card is shown in command–response pairs with the native DESFire EV1 commands

and responses wrapped in ISO/IEC 7816-4 APDUs. Only DES and AES are used in the example, but the same line of thought used for DES applies to 2K3DES, and the same line of thought used for AES applies to 3K3DES. The flowchart of the sample application showing the commands sent to the card is depicted in Figure 3.4.

The sample application authenticates at PICC-level and formats the card, which deletes all the existing applications and frees the corresponding memory, and creates a new card application setting its cipher to either DES or AES. This new application is selected and another authentication takes place but at the application-level instead of PICC-level. Three value files are then created, one for each of the three possible communication settings—plain, mac'ed and enciphered. A value file consists of a four-digit counter, which is in this example set to the initial value of 32_H (50_{10}). A value of 7_H is credited twice in each file and the new value 40_H (64_{10}) is retrieved from the card.

```
1  PC/SC card in SCL011G Contactless Reader [SCL01x Contactless Reader]
       (21161044200765) 00 00, protocol T=1, state OK
2  >> 90 aa 00 00 01 00 00 (AUTHENTICATE_AES)
3  << c9 c1 a7 12 57 e5 2c b6 d7 c8 9c ea 9a 73 c1
       17 91 af (ADDITIONAL_FRAME)
4  >> 90 af 00 00 20 02 f4 cc 61 fb 8a b1 33 2c 48 87 cf 44 c5 bf c2 57 41 a7
       b1 8a 24 8b 14 0f 39 08 a4 96 35 c3 d6 00 (MORE)
5  << 18 e9 ba c5 7f 99 c1 3a d8 ba 5f d6 de 96 90 ce 91 00 (OPERATION_OK)
6  The random A is 76 69 06 3b d7 51 01 a8 0a 5a b8 35 2b 23 4d 5a
7  The random B is ad 2c a4 85 6d 7d f5 73 ae 87 0e 7f 07 6a 3c cc
8  The secret key is 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
9  The session key is 76 69 06 3b ad 2c a4 85 2b 23 4d 5a 07 6a 3c cc
10 >> 90 fc 00 00 00 (FORMAT_PICC)
11 << fd 21 05 eb 70 fa e9 62 91 00 (OPERATION_OK)
12 >> 90 ca 00 00 05 01 02 03 0f 05 00 (CREATE_APPLICATION)
13 << 2a fd d4 17 a4 21 34 73 91 00 (OPERATION_OK)
14 >> 90 5a 00 00 03 01 02 03 00 (SELECT_APPLICATION)
15 << 91 00 (OPERATION_OK)
16 >> 90 0a 00 00 01 03 00 (AUTHENTICATE_DES_2K3DES)
17 << da eb 40 1d c9 49 56 6a 91 af (ADDITIONAL_FRAME)
18 >> 90 af 00 00 10 ac d7 f0 80 f6 18 97 70 17 f3 74 76 75 8c e7
       54 00 (MORE)
19 << ca 92 61 78 15 31 b2 1e 91 00 (OPERATION_OK)
20 The random A is c5 a0 5c 2c 39 4c 91 42
21 The random B is d0 04 8c 5e 1a 2f 4b f0
22 The secret key is 00 00 00 00 00 00 00 00
23 The session key is c5 a0 5c 2c d0 04 8c 5e
24 >> 90 cc 00 00 11 04 00 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
       CREATE_VALUE_FILE)
25 << 91 00 (OPERATION_OK)
26 >> 90 cc 00 00 11 05 01 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
```

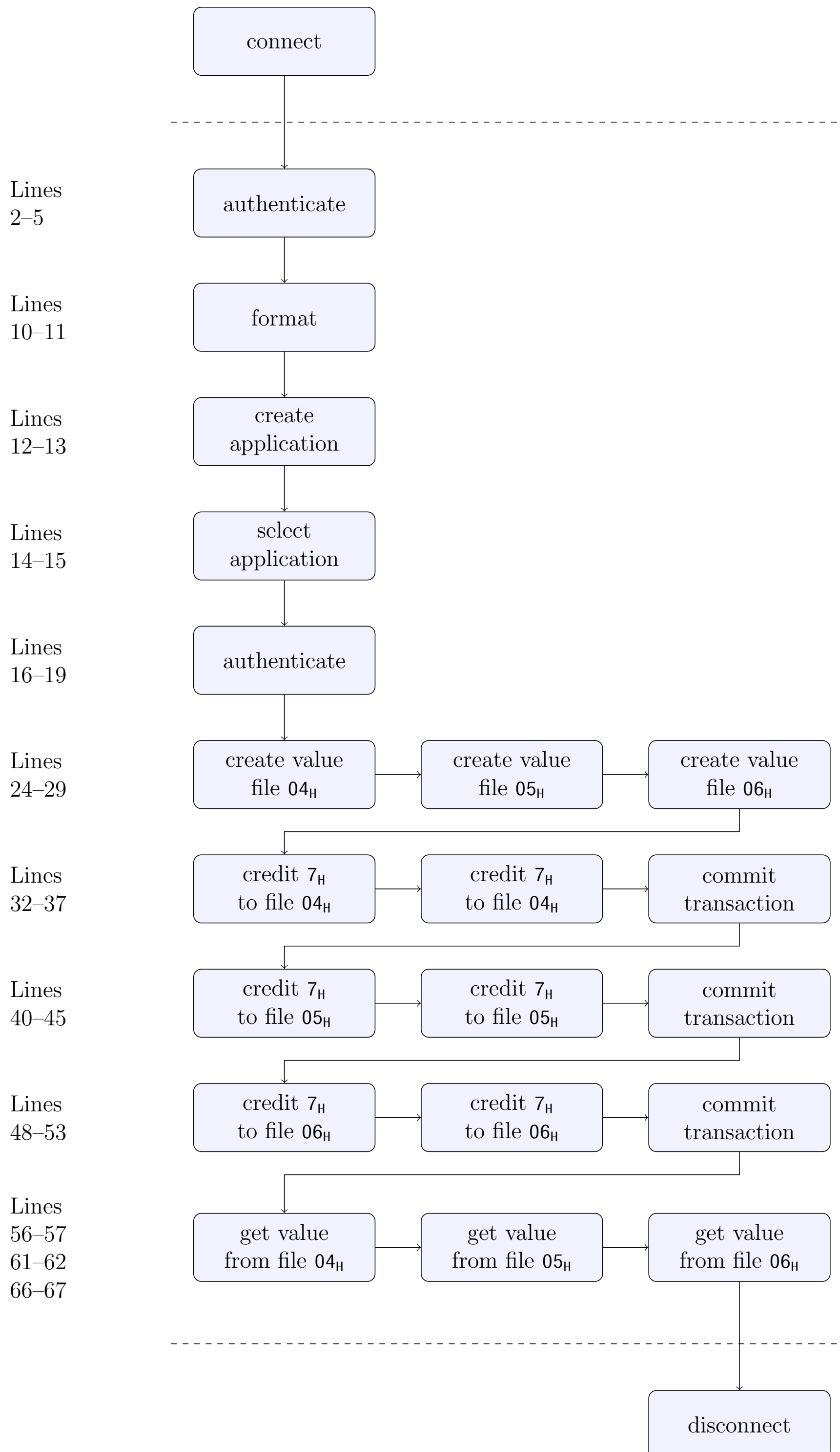


Figure 3.4: Flowchart of the sample application that generates the traces showing the commands sent to the card.

```
       CREATE_VALUE_FILE)
27  << 91 00 (OPERATION_OK)
28  >> 90 cc 00 00 11 06 03 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
       CREATE_VALUE_FILE)
29  << 91 00 (OPERATION_OK)
30  >> 90 f5 00 00 01 04 00 (GET_FILE_SETTINGS)
31  << 02 00 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 91 00 (OPERATION_OK)
32  >> 90 0c 00 00 05 04 07 00 00 00 00 (CREDIT)
33  << 91 00 (OPERATION_OK)
34  >> 90 0c 00 00 05 04 07 00 00 00 00 (CREDIT)
35  << 91 00 (OPERATION_OK)
36  >> 90 c7 00 00 00 (COMMIT_TRANSACTION)
37  << 91 00 (OPERATION_OK)
38  >> 90 f5 00 00 01 05 00 (GET_FILE_SETTINGS)
39  << 02 01 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 91 00 (OPERATION_OK)
40  >> 90 0c 00 00 09 05 07 00 00 00 e1 f6 48 e4 00 (CREDIT)
41  << 91 00 (OPERATION_OK)
42  >> 90 0c 00 00 09 05 07 00 00 00 e1 f6 48 e4 00 (CREDIT)
43  << 91 00 (OPERATION_OK)
44  >> 90 c7 00 00 00 (COMMIT_TRANSACTION)
45  << 91 00 (OPERATION_OK)
46  >> 90 f5 00 00 01 06 00 (GET_FILE_SETTINGS)
47  << 02 03 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 91 00 (OPERATION_OK)
48  >> 90 0c 00 00 09 06 5c ba af d0 96 5c d3 fc 00 (CREDIT)
49  << 91 00 (OPERATION_OK)
50  >> 90 0c 00 00 09 06 5c ba af d0 96 5c d3 fc 00 (CREDIT)
51  << 91 00 (OPERATION_OK)
52  >> 90 c7 00 00 00 (COMMIT_TRANSACTION)
53  << 91 00 (OPERATION_OK)
54  >> 90 f5 00 00 01 04 00 (GET_FILE_SETTINGS)
55  << 02 00 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 91 00 (OPERATION_OK)
56  >> 90 6c 00 00 01 04 00 (GET_VALUE)
57  << 40 00 00 00 91 00 (OPERATION_OK)
58  The stored value (fileNo=4, cs=0) is 64
59  >> 90 f5 00 00 01 05 00 (GET_FILE_SETTINGS)
60  << 02 01 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 91 00 (OPERATION_OK)
61  >> 90 6c 00 00 01 05 00 (GET_VALUE)
62  << 40 00 00 00 24 3a fa 5d 91 00 (OPERATION_OK)
63  The stored value (fileNo=5, cs=1) is 64
64  >> 90 f5 00 00 01 06 00 (GET_FILE_SETTINGS)
65  << 02 03 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 91 00 (OPERATION_OK)
66  >> 90 6c 00 00 01 06 00 (GET_VALUE)
67  << 93 a9 4b 99 61 fd 21 68 91 00 (OPERATION_OK)
68  The stored value (fileNo=6, cs=3) is 64
69  success.
```

Listing 3.1: Trace with DES.

```
1   PC/SC card in SCL011G Contactless Reader [SCL01x Contactless Reader]
```

```
    (21161044200765) 00 00, protocol T=1, state OK
2   >> 90 aa 00 00 01 00 00 (AUTHENTICATE_AES)
3   << 48 2f 40 ad eb f2 47 a6 e6 e3 fe fe 83 06 0c
       07 91 af (ADDITIONAL_FRAME)
4   >> 90 af 00 00 20 91 89 ac dc 04 37 67 fa 7d 25 ef 5f b3 ce 68 9d a7 cc 9e
       a8 a7 5b 2a 69 73 9c f0 ab 64 f0 8d 92 00 (MORE)
5   << 88 30 a2 33 db b8 d1 16 1d 28 fa 08 af f6 3e e4 91 00 (OPERATION_OK)
6   The random A is 95 6b 22 dc 89 f3 ae 21 ab 3c 5b d1 97 11 a3 e1
7   The random B is 14 43 ba 75 6c 21 84 5b 4c 30 a7 83 d0 d2 1b 8c
8   The secret key is 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
9   The session key is 95 6b 22 dc 14 43 ba 75 97 11 a3 e1 d0 d2 1b 8c
10  >> 90 fc 00 00 00 (FORMAT_PICC)
11  << 66 75 82 d7 7b 34 fc 64 91 00 (OPERATION_OK)
12  >> 90 ca 00 00 05 01 02 03 0f 85 00 (CREATE_APPLICATION)
13  << d9 6a c1 e7 7b 7b 43 76 91 00 (OPERATION_OK)
14  >> 90 5a 00 00 03 01 02 03 00 (SELECT_APPLICATION)
15  << 91 00 (OPERATION_OK)
16  >> 90 aa 00 00 01 03 00 (AUTHENTICATE_AES)
17  << 1f 56 4c 74 42 d0 d6 81 76 5a 29 92 7c d6 f6
       a4 91 af (ADDITIONAL_FRAME)
18  >> 90 af 00 00 20 76 1b fe 66 9c 55 d9 04 7a 5f ec c2 a8 68 02 8b 07 0b ec
       22 d2 ab d2 3b b6 87 63 bc e7 6f 06 f9 00 (MORE)
19  << 4f 62 62 2d f0 e8 a5 aa 97 46 22 5c 7e d2 ec 1f 91 00 (OPERATION_OK)
20  The random A is ab df 1b 16 60 7d 5c cd fe 74 97 35 c2 5e bf a4
21  The random B is 0f a9 a1 2c 31 4f 93 e4 85 8a 0c e7 b2 80 f9 a7
22  The secret key is 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
23  The session key is ab df 1b 16 0f a9 a1 2c c2 5e bf a4 b2 80 f9 a7
24  >> 90 cc 00 00 11 04 00 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
       CREATE_VALUE_FILE)
25  << 38 71 1c 80 dd b4 c9 99 91 00 (OPERATION_OK)
26  >> 90 cc 00 00 11 05 01 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
       CREATE_VALUE_FILE)
27  << 93 4f 97 85 4d 56 5c 6d 91 00 (OPERATION_OK)
28  >> 90 cc 00 00 11 06 03 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
       CREATE_VALUE_FILE)
29  << 6e d7 80 b7 b7 58 9f fe 91 00 (OPERATION_OK)
30  >> 90 f5 00 00 01 04 00 (GET_FILE_SETTINGS)
31  << 02 00 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 63 87 c3 00 59 1d 6d
       00 91 00 (OPERATION_OK)
32  >> 90 0c 00 00 05 04 07 00 00 00 00 (CREDIT)
33  << 20 bf 20 a0 6a 48 1b eb 91 00 (OPERATION_OK)
34  >> 90 0c 00 00 05 04 07 00 00 00 00 (CREDIT)
35  << a4 d1 20 40 48 42 cd 4b 91 00 (OPERATION_OK)
36  >> 90 c7 00 00 00 (COMMIT_TRANSACTION)
37  << be bf a5 86 0d 5a f3 2c 91 00 (OPERATION_OK)
38  >> 90 f5 00 00 01 05 00 (GET_FILE_SETTINGS)
39  << 02 01 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 5b 1f 61 37 d7 37 f2
       f9 91 00 (OPERATION_OK)
40  >> 90 0c 00 00 0d 05 07 00 00 00 1b b5 e6 91 77 50 d2 ca 00 (CREDIT)
```

```
41  << 0c b6 7f a8 12 69 62 4f 91 00 (OPERATION_OK)
42  >> 90 0c 00 00 0d 05 07 00 00 00 7b 36 d6 fe f0 66 15 7c 00 (CREDIT)
43  << 62 50 7a cc f4 15 54 0d 91 00 (OPERATION_OK)
44  >> 90 c7 00 00 00 (COMMIT_TRANSACTION)
45  << 13 7a af 32 5d e5 a3 38 91 00 (OPERATION_OK)
46  >> 90 f5 00 00 01 06 00 (GET_FILE_SETTINGS)
47  << 02 03 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 f4 f2 44 99 12 58 78
        e3 91 00 (OPERATION_OK)
48  >> 90 0c 00 00 11 06 c7 12 75 ba 6b 57 7f ec 92 91 3d 7c ef 4c 1a
        27 00 (CREDIT)
49  << c7 dd 3f 55 94 e3 b8 25 91 00 (OPERATION_OK)
50  >> 90 0c 00 00 11 06 3c 42 75 b5 c9 7e 08 94 c0 88 17 a1 c0 c2 f0
        29 00 (CREDIT)
51  << d8 1c 31 e4 72 7c 57 fb 91 00 (OPERATION_OK)
52  >> 90 c7 00 00 00 (COMMIT_TRANSACTION)
53  << e2 ba 9a e7 7d 63 aa 77 91 00 (OPERATION_OK)
54  >> 90 f5 00 00 01 04 00 (GET_FILE_SETTINGS)
55  << 02 00 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 f1 0d 4e 2d 33 31 2e
        3d 91 00 (OPERATION_OK)
56  >> 90 6c 00 00 01 04 00 (GET_VALUE)
57  << 40 00 00 00 c6 ed 8f 47 49 4b 83 0d 91 00 (OPERATION_OK)
58  The stored value (fileNo=4, cs=0) is 64
59  >> 90 f5 00 00 01 05 00 (GET_FILE_SETTINGS)
60  << 02 01 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 ac 3e bc 34 09 c5 de
        89 91 00 (OPERATION_OK)
61  >> 90 6c 00 00 01 05 00 (GET_VALUE)
62  << 40 00 00 00 81 b2 95 31 ac bf d9 bb 91 00 (OPERATION_OK)
63  The stored value (fileNo=5, cs=1) is 64
64  >> 90 f5 00 00 01 06 00 (GET_FILE_SETTINGS)
65  << 02 03 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 67 9e b6 25 d4 3f c9
        4c 91 00 (OPERATION_OK)
66  >> 90 6c 00 00 01 06 00 (GET_VALUE)
67  << 99 ff 1c 08 9f 2b 33 8a d4 67 d0 94 74 3d 08 2e 91 00 (OPERATION_OK)
68  The stored value (fileNo=6, cs=3) is 64
69  success.
```

Listing 3.2: Trace with AES.

```
1   PC/SC card in SCL011G Contactless Reader [SCL01x Contactless Reader]
        (21161044200765) 00 00, protocol T=1, state OK
2   >> 90 aa 00 00 01 00 00 (AUTHENTICATE_AES)
3   << ec 42 de 2d c8 0a 67 4b 94 9e 52 e9 ad 6c 69
        ba 91 af (ADDITIONAL_FRAME)
4   >> 90 af 00 00 20 b3 b0 a5 26 49 5c 97 72 a6 9b 88 cb c9 f7 b3 aa 2c f9 ba
        08 fd c6 86 99 05 84 64 73 8a 55 77 26 00 (MORE)
5   << 72 8c 5c 09 6f f8 ba f8 de 8a 00 99 b2 27 2b ec 91 00 (OPERATION_OK)
6   The random A is 4c a1 76 1b c4 c9 b5 5d ad ee 29 09 17 d7 a6 4f
7   The random B is df a3 28 c7 3e 68 e5 88 99 a5 3a 65 03 1a 80 b4
8   The secret key is 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
9   The session key is 4c a1 76 1b df a3 28 c7 17 d7 a6 4f 03 1a 80 b4
10  >> 90 fc 00 00 00 (FORMAT_PICC)
11  << c8 18 0c 83 ae f2 8c e6 91 00 (OPERATION_OK)
12  >> 90 ca 00 00 05 01 02 03 0f 85 00 (CREATE_APPLICATION)
13  << 43 c3 2e 35 64 5d d5 bc 91 00 (OPERATION_OK)
14  >> 90 5a 00 00 03 01 02 03 00 (SELECT_APPLICATION)
15  << 91 00 (OPERATION_OK)
16
17
18
19                 // No authentication
20                 // inside the application.
21
22
23
24  >> 90 cc 00 00 11 04 00 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
        CREATE_VALUE_FILE)
25  << 91 00 (OPERATION_OK)
26  >> 90 cc 00 00 11 05 01 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
        CREATE_VALUE_FILE)
27  << 91 00 (OPERATION_OK)
28  >> 90 cc 00 00 11 06 03 30 00 0a 00 00 00 5a 00 00 00 32 00 00 00 00 00 (
        CREATE_VALUE_FILE)
29  << 91 00 (OPERATION_OK)
30  >> 90 f5 00 00 01 04 00 (GET_FILE_SETTINGS)
31  << 02 00 30 00 0a 00 00 00 5a 00 00 00 00 00 00 00 00 91 00 (OPERATION_OK)
```

Listing 3.3: Trace with AES and no authentication at the application-level.

The same piece of code is used for the three traces with minor differences. The security of the operations, after selecting the application, is based on CBC-MAC and DES for the first trace, and it is based on CMAC and AES for the second and third traces. For the third trace, however, there is no authentication inside the application, which results in the last steps, `Credit` and `GetValue`, not being executed. In this case, the application terminates before the first `Credit` because it verifies the settings of the target file, in lines 30–31, and realizes that it does not have the necessary permissions. The traces are presented in Listing 3.1, Listing 3.2 and Listing 3.3.

AES is always used for the PICC-level authentication in lines 2–5. The application-level authentication uses DES or AES, according with the parameters used on application creation. The `CreateApplication` command in line 12 contains either $\texttt{05}_\texttt{H}$ or $\texttt{85}_\texttt{H}$ as the penultimate byte. The first digit is associated with the cipher used by the application, which is DES for the first case and AES for the second case. The second digit is the number of keys, which is five in this example.

File	FileNo	CommSett	ARs
Value file 1	04_H	00_H–plain	$30\ 00_H$
Value file 2	05_H	01_H–mac'ed	$30\ 00_H$
Value file 3	06_H	03_H–enciphered	$30\ 00_H$

Table 3.6: Settings of the value files.

The three value files are created with the same access rights but with different communication settings and the initial value 32_H. The access rights, explained in Section 3.3.4.2, control the access to the file. The communication settings, explained in Section 3.3.4.1, define the level of security for the communication between the reader and the card. The communication settings for the first, second and third value files are respectively plain, mac'ed and enciphered. The communication settings are represented by the seventh byte in lines 24, 26 and 28, and the access rights are represented by the eigth and ninth bytes of the same lines. The settings for each value file are summed up in Table 3.6.

The access rights of the value files are $30\ 00_H$. This indicates that an authentication with key number 3_H grants RW access and that an authentication with key number 0_H grants CAR, R and W access. RW access allows the use of `Credit` and `GetValue`. R and W also grant access to `GetValue`. The first two traces are complete because the authentication done inside the card application is against the third key, granting the program access to the required commands. However, the third trace does not authenticate inside the card application; hence the `Credit` and `GetValue` commands are not executed and the program terminates before reaching the end. This is the reason why Listing 3.3 ends in line 31, just before the `Credit` command.

Command	Plain	Mac'ed	Enciphered
`Credit`	32–35	40–43	48–51
`GetValue`	56–57	61–62	66–67

Table 3.7: Lines of the `Credit` and `GetValue` commands per communication mode.

The `Credit` and the `GetValue` commands are affected by the communication settings of the target file. The `Credit` command writes data into the card and the `GetValue` command reads data from the card. When a secure communication mode is used, it applies to the command APDU for `Credit` and it applies to the response APDU for `GetValue`. The objective is to add

the security mechanism to the messages carrying data. Table 3.7 shows the lines where these two commands occur and the respective communication modes.

In Listing 3.1, since the application relies on DES for security, each command-response pair is an independent operation when calculating the CBC-MAC or enciphering data. In Listing 3.2, the application relies on AES instead of DES for security, which reuses the IV between operations making the end operation dependent from the previous one. As long as the preceding authentication is done successfully, a CMAC is appended to all the responses, unless the response data is enciphered. When creating the value files in lines 24–29 of Listing 3.2, the CMAC is appended to the response received from the PICC but in Listing 3.3 the CMAC is not appended. This happens because the authenticated state at PICC-level is lost in lines 14–15, when the application is selected, and in Listing 3.3 there is not a successful authentication inside the application. Without the authenticated state it is not possible to calculate the CMAC, because the cryptographic functions require the session key generated after a successful authentication to operate.

For the mac'ed communication mode, a MAC is appended to the command when crediting and a MAC is appended to the response when retrieving data. For the enciphered communication mode, the body of the command is enciphered when crediting and the body of the response is enciphered when retrieving data. Additionally, if AES is used and a preceding successful authentication took place inside the application, a CMAC is also appended to the responses when crediting. Since with DES the pairs of operations are independent from each other, it is likely that operations such as `Credit` can be successfully replayed by a third party. For example, in Listing 3.1, repeating lines 42–43 after line 43 or lines 50–51 after line 51 appears to be feasible and would result in additional credit being added to the target files. However, in the latter case, with an enciphered command-response pair, the amount being credited is not visible. The application purposely contains `Credit` operations in pairs to illustrate their similarities when using DES and their differences when using AES. For Listing 3.1, the command-response pair in lines 32–33 is exactly the same as in lines 34–35, and likewise when comparing lines 40–41 with lines 42–43 and lines 48–49 with lines 50–51. This does not happen in Listing 3.2, where all the `Credit` command-response pairs are similar, yet with a different CMAC. Equally important, the security mechanisms when relying on the 2K3DES cipher operate the same way as in DES, hence suffering from the same weakness. In conclusion, an attacker relaying messages between the PCD and the PICC and with the ability to inject new ones can take advantage of this protocol flaw, effectively producing a man-in-the-middle attack.

Chapter 4

Remote card update

The goal of the system presented in this thesis is to enable secure remote manipulation of data on a DESFire EV1 smart card. This includes both reading data and modifying data. Data is read to check for a particular piece of information stored on the smart card and data is modified to update that information. Possible uses include checking the balance on the smart card, performing a remote top-up and updating personal information fields.

This chapter collects a set of requirements for the successful operation of the system and presents a solution for the problem. The communication protocol detailed in this section is at the core of the proposed solution.

4.1 Requirements

The foremost requirement of the system—and the problem we are trying to solve—is enabling reliable and secure remote manipulation of data on a DESFire EV1 smart card, a clear requisite, but too general nevertheless. This requirement can be divided into smaller parts. The constituting parts can then be defined, subdivided and grouped into functional and non-functional requirements, resulting in a better picture of what the system should be.

The stakeholders of the system are defined in order to understand the requirements that follow. Only two stakeholders are considered, for the sake of simplicity. These are the service provider and the user of the service. The service provider is the entity responsible for the service. The responsibilities of the service provider include managing the application on the smart card and handling the respective servers. The service provider may or may not be the issuer of the smart card. The role of issuing smart cards may be taken by a third party, trusted by the other entities of the ecosystem. The initialization of the card application, however, lies with the service provider.

The user of the service is the entity consuming the service. The user is in possession of and uses the smart card to access the service.

The problem of enabling reliable and secure remote manipulation of data on a DESFire EV1 smart card is subdivided into the functional requirement to enable remote manipulation of data and into the non-functional requirements reliability and security. *Enabling remote manipulation of data* is defined as the ability to remotely update data on the card and as the ability to read data from the card. *Reliability* is defined as the ability to resume an interrupted operation, at a later time, with loss for neither the service provider nor the user. *Security* is defined as the user being only able to read and modify the content of the application on the smart card with the consent of the service provider.

A further subdivision of these high-level requirements yields the following list of functional and non-functional requirements:

Functional requirements

REQ1. The service provider can update the data stored on the card application remotely.

REQ2. The service provider can read data from the card application remotely.

Non-functional requirements

REQ3. Interrupted operations can be resumed without loss. (*reliability*)

REQ4. Only the service provider can update the data stored on the card application. (*authorization*)

REQ5. Only the service provider can read the data stored on the card application. (*authorization*)

REQ6. Only authenticated operations may perform data updates on the card application. (*authentication*)

REQ7. The confidentiality of the data stored on the card application is at the discretion of the service provider. (*confidentiality*)

REQ8. The integrity of the data stored on the card application is enforced. (*integrity*)

REQ9. Users cannot deny having used the smart card. (*non-repudiation*)

4.2 Solution

The focus of the proposed solution is on the ability to manipulate the DESFire EV1 smart card remotely. A *remote operation* is defined as an operation on the smart card using an NFC-enabled mobile device as card reader. The NFC-enabled mobile device is connected to the server, possibly through the cellular network, and is a personal device of the user, for example, an NFC-enabled smart phone.

In addition to remote operations, the solution takes into account that smart cards are likely to be operated in a traditional fashion. A *traditional operation* is defined as an operation on the smart card using a card reader of the service provider, for example, a POS terminal. This type of operation is not explored in detail but is defined in order to better understand the architecture of the system.

The card manipulation is based on transactions. A *transaction* is an action to be performed on the card such as reading or updating one or more files. In the context of the proposed solution, the server applies the transactions to the card through remote operations which are done via smart phone.

This section presents the architecture of the proposed solution, how transactions are managed, and the communication protocol. The protocol section details how remote operations should be carried in order to ensure their reliability and security.

4.2.1 Architecture

The main components of the architecture of the proposed solution are the DESFire EV1 smart card, the NFC-enabled smart phone, and the server. The smart card is the secure device storing the data. The smart phone relays messages between the smart card and the server. The server is able to generate the secret keys of all the smart cards for a specific card application and manipulate the data present on that application.

The architecture of the proposed solution includes a POS terminal to illustrate the traditional mode of operation. In this case, it is the POS that relays messages between smart card and server instead of the smart phone. The POS terminal may be able, on its own, to manipulate the data on the smart card and operate without being permanently connected to the server. The POS is only included in the architecture to have a point of comparison with the remote type of operation and is not part of the communication protocol detailed in Section 4.2.3.

The smart card and the server are both assumed to be trusted components

of the system. The POS terminal may be trusted, depending on the design of the system. The NFC-enabled smart phone cannot be trusted with complete access to the contents of the files on the card application, as it may attempt to corrupt the system for the benefit of its user, for instance, by acquiring extra credit without paying for it.

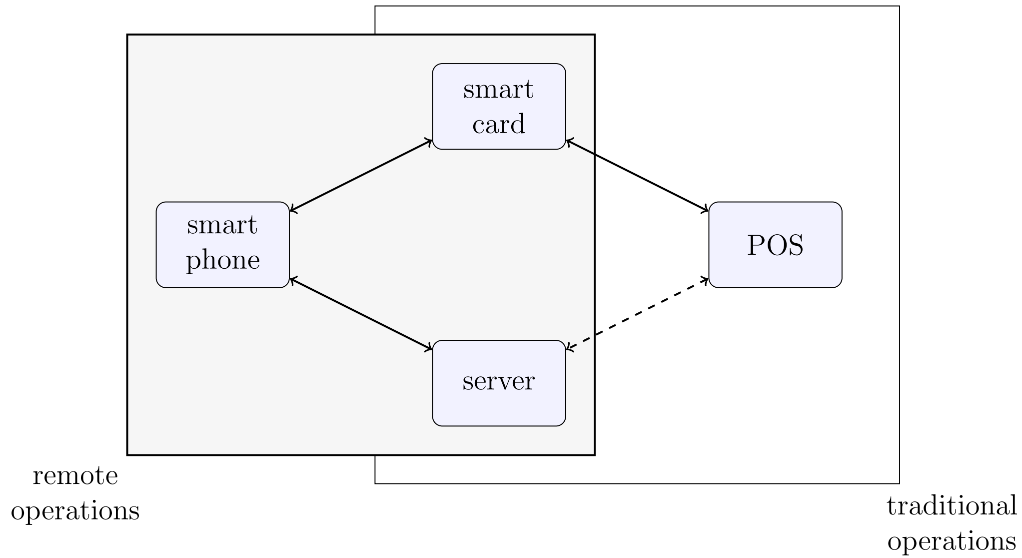


Figure 4.1: Architecture of the proposed solution.

The architecture is presented in Figure 4.1. To perform remote operations, the NFC-enabled smart phone reads from and writes into the DESFire EV1 smart card as indicated by the server. The smart phone acts as a proxy, enabling the server to reach the smart card and manipulate its data. In traditional operations, the POS terminal reads from and writes into the DESFire EV1 smart card. This may be done without permanent access to the server, depending on how the system is designed and implemented.

A permanent link must exist between the smart phone and the server while remote operations are taking place. To establish a secure session with the smart card, a preceding authentication is required. If the smart card is torn apart from the reader, the authentication protocol must be restarted. This means that the smart card has to be kept inductively coupled to the smart phone from the beginning to the end of the session. During the authentication protocol and the manipulation of data, messages are exchanged with the server. It is assumed that the phone has a continuous connection to the server. In order to support offline use of the system, that is, to perform remote operations without a permanent link between the smart phone and the server, the smart card would have to be kept near the smart phone from

the beginning of the session until the end of it, including the periods when the connection to the server is lost. Aside from the added complexity, this is impractical from a usability perspective. Therefore, in our system, if the communication link between smart phone and server is broken, a new session is started once it is resumed.

A POS terminal may operate on smart cards without a permanent connection to the central server because it is a trusted component. For that reason, it can contain the necessary keys enabling it to read from and write into selected files of the application on the smart card. This means it can operate offline and update the server with the new transactions when it goes online.

4.2.2 Transactions

The actions to be performed on the card, for instance, reading and updating files, are grouped in transactions. When a user logs into the server, the server checks whether there are pending transactions for the smart card associated with this user. If there are pending transactions, these are applied to the card.

The server uses the `Transaction`, `StartedTx` and `FileUpdate` tables to manage transactions. These tables are presented in Figure 4.3, next to the sequence diagram of the communication protocol which is detailed on the following section. The `Transaction` table associates transactions to cards and keeps the state of each transaction. The `StartedTx` table contains temporary entries used when a transaction is in progress, i.e. being applied to the card. Each of these entries store temporary data for a transaction in progress such as the session key and the final ACK. The session key grants access to the card application by the server to carry the necessary operations and the final ACK is the response of the card to the `CommitTransaction`, the last command sent to the card. Since all the operations are chained, the final ACK is used as proof that all preceding operations have been executed on the card as indicated by the server. The `FileUpdate` table contains individual updates to files which are linked to an existing transaction. A transaction can have multiple file updates but a file update can only be associated with a single transaction.

Each entry in the `Transaction` table can be in one of three different states. These are `WAITING`, `STARTED` and `COMPLETE`. The `WAITING` state is for new transactions. The `STARTED` state is for transactions in progress, that is, transactions that are currently being applied to the card. The `COMPLETE` state is for transactions that have been applied to the card and for which the server successfully received an acknowledgment.

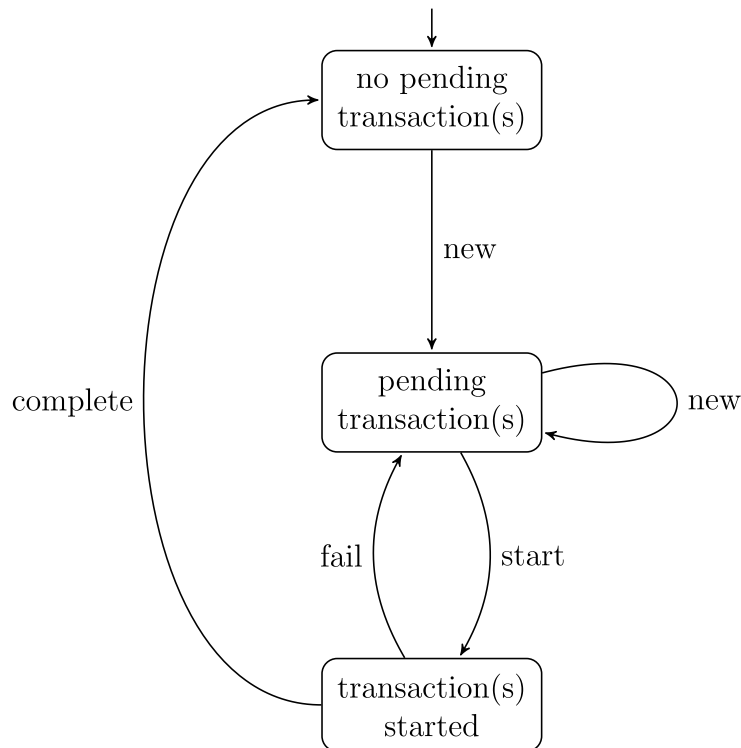


Figure 4.2: State machine of the state of transactions for a single smart card, from the perspective of the server.

Figure 4.2 presents the lifecycle of transactions, for a single card, on the server side. The state *no pending transaction(s)* indicates that there are no transactions associated with this card or that all the existing transactions are in the `COMPLETE` state. The state *pending transaction(s)* indicates that pending transactions exist. Transactions in states `WAITING` and `STARTED` are both pending transactions, but at this stage, the latter implies one or more failed attempts to update the smart card and inform the server. The state *transaction(s) started* indicates that one or more transactions are in progress. A transaction in progress is set to the `STARTED` state.

A transaction in the third state, *transaction(s) started*, can either succeed or fail. If the transaction succeeds, there is a move to the first stage of the lifecycle and the state of the transaction is updated to `COMPLETE` in the `Transaction` table. In this case, the entry in the `StartedTx` table associated with the transaction in progress and holding temporary data is deleted. If the transaction fails, it returns to the previous lifecycle stage and the `STARTED` state remains unchanged.

4.2.3 Protocol

A smart phone will typically produce itself the commands that it uses to interact with a smart card. In the proposed solution, the smart phone receives those commands from a remote server. The communication interface between the smart phone and the smart card is still the same, but we take advantage of the security features of DESFire EV1 to achieve our goal. The proposed remote update protocol connects the smart card of the user to the server of the service provider. This is done via an NFC-enabled smart phone, which relays messages between the smart card and the server.

The protocol is presented in Figure 4.3, which depicts the communication between the parties and the required tables for data storage and user and card management. The communication between the server and the smart phone relies on the SSL/TLS protocol to enforce the confidentiality and the integrity of data traveling over the Internet. The server authenticates itself before the client using a certificate and the client authenticates itself before the server using a username and a password. Once the SSL/TLS handshake is completed, the server and the smart phone are ready to exchange messages through the secure channel. Since the user logged into the server, the server is able to check on the database whether there are any pending transactions for the smart card associated with this username. If there are no pending transactions, the server informs the smart phone that there are no pending transactions for the smart card of this user. Otherwise, the server instructs the user to touch the smart phone to the smart card in order to proceed with the update. A successful run of the communication protocol, performing an update, is composed by the following steps:

1–6: Phone⇄Card. The smart phone reads the UID from the smart card and selects the application. It initiates the authentication protocol, described in Section 3.3.2, and receives the random number B from the smart card, ciphered with the secret key $K1$ shared between the smart card and the server.

7–8: Server⇆Phone. The smart phone sends the UID and the ciphered random number B to the server. The server calculates the session key and creates an entry in the `StartedTx` table to store it. The transaction state changes from `WAITING` into `STARTED`. The server replies with the next message to be sent to the smart card to conclude the authentication protocol and the expected response to that command.

9–16: Phone⇄Card. The smart phone is able to conclude the authentication protocol with the smart card, checking that the $\{r'_A\}_{K1}$ received

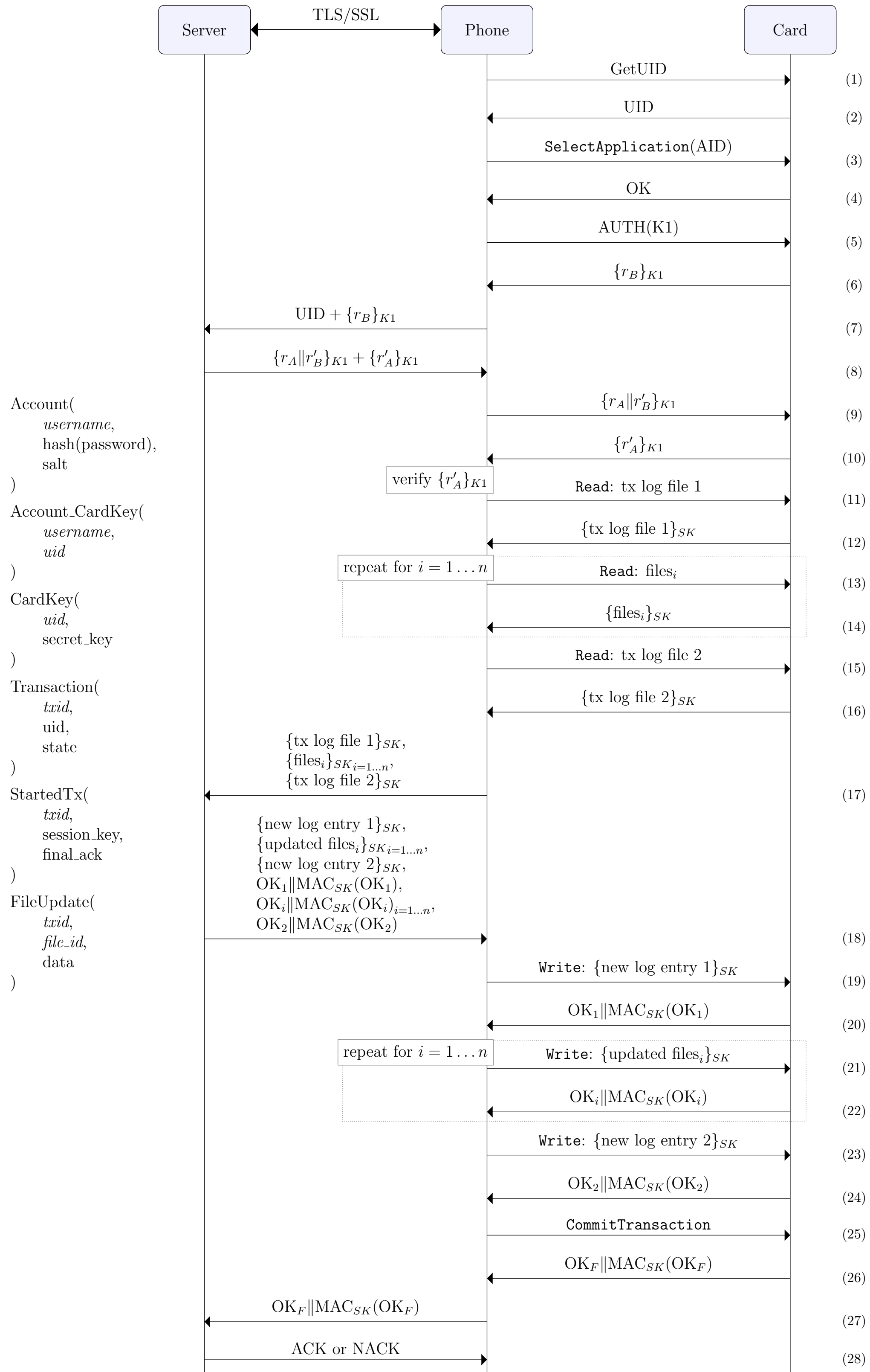


Figure 4.3: Sequence diagram of remote update to DESFire EV1.

from the smart card matches the one received from the server. This check is only for error detection and is insecure because it is done on the smart phone side. At this point, the smart phone reads the files within the application on the smart card, namely the log files containing the identifiers of the last completed transactions.

17–18: Server⇆Phone. The smart phone sends all the files read from the smart card to the server. The server replies with the updated version of the files, ready to be written to the smart card. The reply also includes the confirmation messages that the card is expected to reply with at each write.

19–26: Phone⇄Card. The smart phone writes the updated files into the smart card, verifying that the responses received from the smart card match the ones received from the server. A `CommitTransaction` is then issued to make the previous changes to the files permanent. The smart card replies with the confirmation message $\mathrm{OK}_F \| \mathrm{MAC}_{SK}(\mathrm{OK}_F)$.

27–28: Server⇆Phone. The smart phone sends the confirmation message of the `CommitTransaction` operation, $\mathrm{OK}_F \| \mathrm{MAC}_{SK}(\mathrm{OK}_F)$, to the server. The server compares the message with the one stored and replies with either ACK if the messages are equal or NACK if the messages are different. If the confirmation message received by the server matches the one stored in the database, the transaction state is set to `COMPLETE` and the corresponding entry is removed from the `StartedTx` table.

The remote update protocol has been optimized in order to minimize the number of round trips of the communication with the server. In a traditional communication setting with the card, the initiator would send a request to the card, wait for the associated response, and only then send a new request to the card. The protocol presented takes advantage of the fact that some commands sent to the card are always the same. These commands are sent by the phone without being instructed by the server. The only commands and responses exchanged with the server are the ones which are different in between sessions with the card. The messages are also ordered in a way that minimizes the total number of messages required to update the card in a secure and reliable fashion. In addition, the SSL/TLS connection with the server is established at the beginning of the protocol in order to reduce the latency of the subsequent message exchanges.

These optimizations are important since the protocol is aimed at mobile devices. In comparison to wired networks, wireless networks are more likely

to suffer from sudden drops in throughput or increased latency, and from loss of connectivity. Minimizing the amount of round trips and grouping the messages to be exchanged with the server also saves battery power by reducing the usage of the radio interface of the mobile device.

4.2.3.1 Failures

Failures may occur at any stage of the protocol. The network connection linking the smart phone to the server may be closed or the smart card may be torn apart from the smart phone, preventing messages from reaching their destination. Whatever the cause leading to the loss of messages, the system is expected to eventually resume and complete the interrupted operation.

When the objective is to read data from the smart card, failures are harmless except for the inconvenience to the user. This case does not require data to be changed on the system. When the objective is to update data on the smart card, failures become a problem. The following paragraphs explain where failures happen and how the protocol handles those failures in order to keep a consistent state in the system, and an improvement to the protocol regarding the recovery from failures.

The messages exchanged between the server, the smart phone and the smart card are numbered, as shown in Figure 4.3. When the message lost is up to and including the `CommitTransaction` operation numbered (25), the failure is considered an *update failure* because the data on the smart card is not modified. When the failure happens after the `CommitTransaction`, the failure is considered an *acknowledgment failure* because the data on the smart card is modified but at least one of the parties is not informed. For all practical purposes, the update is done when an acknowledgment failure takes place and the data on the smart card is modified. However, one of three problems arise:

- Message (26) is lost: neither the smart phone nor the server is aware that the update is done;
- Message (27) is lost: the smart phone knows the update was successful but the server does not and is therefore unable to deem the transaction complete; or
- Message (28) is lost: both the smart phone and the server are aware that the update is done but the smart phone does not know whether the server has been notified.

Recall that transactions can be in one of three states: `WAITING`, `STARTED` and `COMPLETE`. Transactions in state `WAITING` have definitely not been applied

to the card since these are new transactions. However, the server does not know whether a transaction in state STARTED is applied to the card and the server failed to receive the acknowledgment message, or the transaction completely failed and the card data did not change.

Recovering from a failed operation is achieved by restarting the protocol. The log files sent to the server in message (17) contain the txid of the last transaction successfully applied to the card. Transaction identifiers are sequential and only the transactions that made changes to the card are logged by the smart cards logs. For these reasons, all transactions with txids up to the txid from the logs can be considered complete. The server compares the txids of the pending transactions in state STARTED with the txid from the logs. If the txid of an entry in the `Transaction` table is less than or equal to the txid from the card logs, then this transaction has been applied to the card, although the server was not aware of it completing. The state of these transactions can safely be set to COMPLETE and the associated StartedTx entry deleted.

An improvement to the presented solution to failures can be made, when the lost message is the acknowledgment sent to the server or the reply to that message by the server. That is, the lost message is message (27) or message (28). For these particular cases, the smart phone stores the $\text{OK}_F \| \text{MAC}_{SK}(\text{OK}_F)$ received from the smart card. Upon the next attempt at updating the smart card, $\text{OK}_F \| \text{MAC}_{SK}(\text{OK}_F)$ is sent to the server right after establishing the secure tunnel before message (1) in Figure 4.3. The server verifies if an entry exists in the StartedTx table related to this user. If an entry related to the user does not exist in the StartedTx table, the server safely assumes that message (28) was lost and informs the smart phone that the transaction is complete. If an entry related to the user does exist in the StartedTx table, the message lost was message (27). In this case, the server compares the received $\text{OK}_F \| \text{MAC}_{SK}(\text{OK}_F)$ from the smart phone with the one stored in the database. If there is a match, the transaction is complete, the entry removed and an acknowledgment sent to the smart phone. Otherwise, the protocol proceeds as normal.

For the last improvement to work, the NFC-enabled smart phone must be the same one that was used on the previous failed attempt at updating the smart card. A different smart phone will not have the required $\text{OK}_F \| \text{MAC}_{SK}(\text{OK}_F)$ stored in it. The major benefit of this improvement relates to usability, since the user does not have to touch the smart card with the smart phone. Additionally, since less messages are exchanged in the system, the entire update process is faster.

4.2.3.2 Early-commit attack

This section discusses an attack on the remote update protocol where a malicious user sends a premature `CommitTransaction` command to the card thus breaking the protocol. The following solution does not prevent the attack from happening but can successfully detect it giving the service provider an opportunity to take action.

Attacking the protocol. The transaction mechanism of DESFire EV1 ensures the atomicity of operations, but it only does so for accidental failures. A malicious user with the capacity to inject commands into a transaction in progress can submit a premature `CommitTransaction` command causing only some of the files to be permanently written into the card. In the proposed communication protocol, the server reaches the card via an NFC-enabled mobile phone. The phone is in a privileged position that enables it to perform this attack.

The data exchanged between the server and the card, via the phone, is enciphered using AES. In addition, all the commands and responses are cryptographically chained. DESFire EV1, however, does not validate the commands received unless they change data on the card. A write command is sent enciphered and a CRC32 is concatenated to the payload. When it reaches the card, the data is validated before being written. A read command, which does not change data on the card, is sent without a MAC appended. A malicious user would still be unable to understand the contents of the file read because he does not have access to the session key. Nonetheless, the command sent by the malicious user is executed by the card.

This problem happens because DESFire EV1 does not validate all commands. The respective MAC is still calculated in order to keep the IV up to date for future commands and responses, but it is not appended to the command sent to the card. Therefore, the card cannot tell whether the `CommitTransaction` command is legitimate or not.

Mitigation strategy. The protocol relies on two log entries to wrap the file updates in order to detect the early-commit attack. The backup mechanism of DESFire EV1 is explained first since it is necessary to understand the solution to detect the attack.

The backup mechanism of DESFire EV1 works by using extra space for each file. When writing into these files, the backup area is used to store the new data. The `CommitTransaction` command basically flips a bit that instructs the card to turn the backup space into the main content

and the previous main content area into the next backup space of that file. This is the reason why a backup data file consumes twice as much space than a regular data file and why with record files it is necessary to allocate one more entry than the file size. The card supports writing into multiple files and committing all changes only at the end. There is the restriction, however, that only one update can be submitted for each record file during one transaction. Otherwise, the previous write into that file, yet to be committed, is overwritten.

The remote update protocol writes two log entries for every card update, one entry at the beginning of the card update and another entry after writing into all the other files. After writing to all the necessary files on the card application, a `CommitTransaction` is sent to make all the changes permanent. If a failure occurs during the writing process, nothing is written because the changes are still in the backup area. Since the `CommitTransaction` command is not sent to the card because of the failure, the updates to the files are abandoned. Two cyclic record files are used to keep the two log entries because, with a single log file, the last entry would overwrite the first.

```
Write  {<new log entry 1>||CRC32}sk   // start log
...
Credit {<value file 5>||CRC32}sk
Update {<bdata file 6>||CRC32}sk
...
Write  {<new log entry 2>||CRC32}sk   // end log
CommitTransaction                     // make changes permanent,
                                      // not validated
```

Figure 4.4: A card update with two logs wrapping the remaining file changes.

The protocol starts by writing an entry into the first log file, then writes to all other files it needs to update, and finally writes an entry into the second log file, as shown in Figure 4.4. Basically, the main files to be updated are wrapped in two log files. This can somewhat be thought of as a transaction, where the first log file marks the beginning of the transaction, and the second log file marks the end. The card validates these commands because all of them modify the content of files. Since the commands and response are cryptographically chained, the attacker cannot choose in which order to write these files into the card. The only possibility is to issue an early `CommitTransaction`. By

doing so, at least the last log file is not written into. On a subsequent access to the card, the server can compare both log files and find the mismatch in the log entries. The incoherent log entries give grounds to the server to believe that the card has been tampered with. So while it does not solve the problem entirely, this mechanism with two log files allows the server to successfully detect the attack and subsequently blacklist the card.

Chapter 5

Implementation

This chapter describes the implementation of a prototype enabling the remote update of DESFire EV1 smart cards. This prototype works as a proof of concept to demonstrate the feasibility of the solution presented in Section 4.2.

The prototype consists of four related components, in addition to the NFC Java library previously developed. The four projects are a client, a server, a set of classes common to the previous two projects, used to exchange data between them, and a last project used to initialize the smart card application. The client and the server form the bulk of the system. The client is an Android application that serves as a proxy between a DESFire EV1 smart card and the server and provides an interface for the users of the system. The server includes a web application that provides REST-compliant web services to the client.

5.1 Tools and technologies

Multiple tools and technologies are required to develop and successfully run the implemented system. The following paragraphs describe which tools and technologies are used and why these are needed.

The development machine runs the desktop version of the Linux distribution Ubuntu 12.04 LTS. The client application is developed in Android Studio 0.2.x and the remaining development is done in Eclipse IDE for Java EE Developers 4.3. All projects use Java 7 but the client and projects imported into the client require compliance with Java 6. The reason behind this is the Android platform not supporting some features of Java 7 that were initially used, such as multi-catch exceptions and try-with-resources. OpenJDK is the chosen Java implementation both because it is the reference implementation for Java 7 and because it is the default implementation of

the Linux distribution.

The web service is hosted in Apache Tomcat 7. GlassFish 4 was used at the beginning of the development but was dropped because of the high resource consumption when compared to Apache Tomcat 7. The extra resource consumption is not a problem in the development machine but the cloud provider chosen to host the server application, Jelastic [14], charges according with the resources consumed. Hosting the web service on a cloud provider results in a more reliable evaluation of the system in comparison to running it only on a local development machine. In addition to the extra cost for GlassFish, the cloud provider does not support GlassFish 4 but GlassFish 3.1. Changing from GlassFish 4 to Apache Tomcat 7 required setting up the new servlet container and modifying configuration files on the web service, but there was no change in code.

The communication between the client and the server uses the wireless local area network eduroam [18]. The data is exchanged, in XML, through a secure tunnel. Java objects are used on the client and on the server to encapsulate the data to be exchanged, since it would be cumbersome to interact directly with XML. To marshal classes into XML and unmarshal XML into classes, the server uses the Java Architecture for XML Binding (JAXB) API, which is part of Java SE. JAXB is not supported on Android. XML Pull Parser is used to parse XML instead, which has the disadvantage of requiring a lot of additional work since the entire XML string is manually parsed. Although XML Pull Parser is not part of Java SE, it is natively supported by Android.

The web application uses JAX-RS to provide resources to clients. JAX-RS is a framework that supports the development of RESTful web services in Java. Jersey 2.1, the reference implementation of JAX-RS, was used initially. When the change from GlassFish 4 to Apache Tomcat 7 took place, Jersey was downgraded to version 1.17 because Apache Tomcat 7 does not support version 2.1. Despite the downgrade of Jersey, no modifications were required on the server. The resources provided to clients use the REST architectural style.

The secure connection between the phone and the server requires the creation of public-private key pairs and the corresponding certificates. These are created using keytool and OpenSSL. The Bouncy Castle Crypto APIs, and later on Spongy Castle, are used to handle the BKS storetype used in Android.

The persistent data storage uses the SQLite database engine. This database is created using a Python script and is accessed from the server using the SQLite JDBC Driver.

5.2 Architecture

The architecture of the system is composed by three main entities in addition to the user. The three entities are the server, the phone and the card, which directly relate to the components of the architecture of the proposed solution for remote operations presented in Figure 4.1. The user interacts directly with the phone and is responsible for triggering actions on the system. When an operation is selected on the phone by the user, the phone interacts with the server and, depending on the operation, may also interact with the card. The architecture of the system is shown in Figure 5.1.

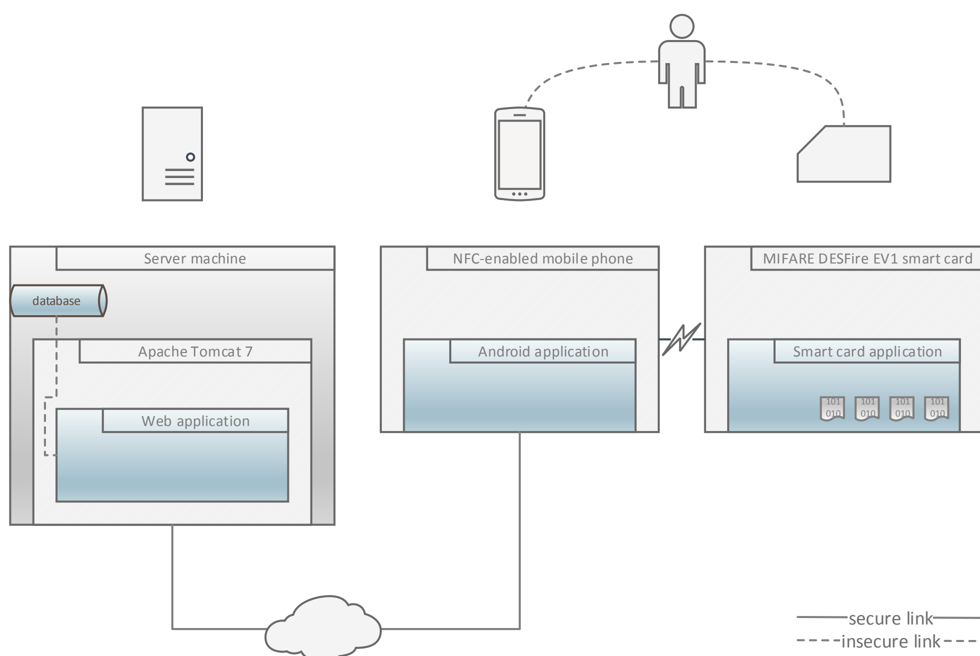


Figure 5.1: The architecture of the prototype.

The server is composed by the SQLite database and by the Apache Tomcat 7 servlet container, which in turn contains the web application. The database is stored on the server data storage device and is composed by a single file. Since the database does not run in its own separate process, it can be considered part of the web application. Nonetheless, an SQLite database allows access from, and can be opened by, multiple applications at the same time. For this reason, it is depicted as a separate component inside the server

instead of an integral part of the web application.

The client is an NFC-enabled Android mobile phone and it holds the client application of the system. In addition to NFC, the phone also has network capabilities used to interact with the server.

The card is a MIFARE DESFire EV1 smart card. It contains the smart card application, which is required to be initialized by a separate application prior to being used in the system.

The communication between the phone and the server is done through an SSL/TLS tunnel and it is detailed in Section 5.2.1. The communication between the phone and the card uses its own security mechanism and is detailed in Section 5.2.2.

5.2.1 Communication between the phone and the server

The phone and the server exchange data through an SSL/TLS tunnel to enforce data confidentiality and integrity during transit. The server authenticates itself to the phone using a digital certificate. The phone authenticates itself to the server using a username and a password.

The secure connection is established before data is exchanged between the two parties. The digital certificate, sent to the phone by the server, is signed by a root certification authority (Root CA). The digital certificate of the Root CA is installed, by default, on the client application. When the client application receives the digital certificate from the server, it validates the certificate by verifying that it is signed by the Root CA and by checking that the address of the resource matches the address stored on the certificate. This enables the server to change its digital certificate by having the Root CA sign the new one and without having to reinstall it in all the client applications. The digital certificates, including the certificate from the Root CA, are created using keytool. Appendix C demonstrates the creation process of the certificates and Section 5.4 discusses issues that occurred during the setup of the secure connection.

The system uses HTTP Basic Authentication to authenticate the user. The client application establishes the secure connection with Apache Tomcat 7 and not with the web application directly. The servlet container is also responsible for the authentication of the users. The credentials of the user cannot be eavesdropped during transport because they are sent to the server over HTTPS.

5.2.2 Communication between the phone and the card

All the files in the card application use the enciphered communication mode and none of the access rights of those files is set to free access (see Section 3.3.4). This means that a previous successful authentication is required to manipulate those files. It also implies that the card returns enciphered data when reading from this card application and expects to received enciphered data when writing into this card application. The phone serves as a proxy between the card and the server but does not have access to the contents of the files because it does not have access to the session key. For the same reason, and because a CRC is appended to the data before it is enciphered, the card and the server will know if the data has been tampered with. The commands sent to the application in the card and the respective responses are cryptographically chained to prevent replay attacks. This chaining is achieved by using the session key to both encipher files and to calculate CMACs, and by reusing the initialization vector.

The card application and the web application share a secret key. The application is created with five keys, with key number $\texttt{03}_{\texttt{H}}$ being the shared secret key. If this specific key number is not used outside of the system, then the only entities that know the secret key are the card, the server, and the card reader that initialized the card application. Since the card reader that initialized the card application had, at some point in time, access to the secret key, the card cannot be absolutely certain that it is talking with the server. It is expected that the provider of the system manages both the card reader that initialized the card application and the web application. For this reason, the card reader that initialized the card application and the server are considered to be the same entity, and the secrecy of the shared secret key is maintained. Following steps 1–10 in Figure 4.3, the card and the server can mutually authenticate themselves and reach a shared session key.

5.2.3 Class diagrams and dependencies

The following figures depict the class diagrams of the client application and of the web application. The class diagrams do not include all implementation details in order to keep them clear and concise.

Figure 5.2 represents the class diagram for the client application and its dependencies. The client application is composed by a `MainActivity` that uses the classes `AcquireProduct` and `CardUpdate`. The class `AcquireProduct` is used when the user buys a credit top-up or updates her personal information. The class `CardUpdate` is used to update the files on the card application. In both cases, Android `AsyncTasks` are used to perform network operations

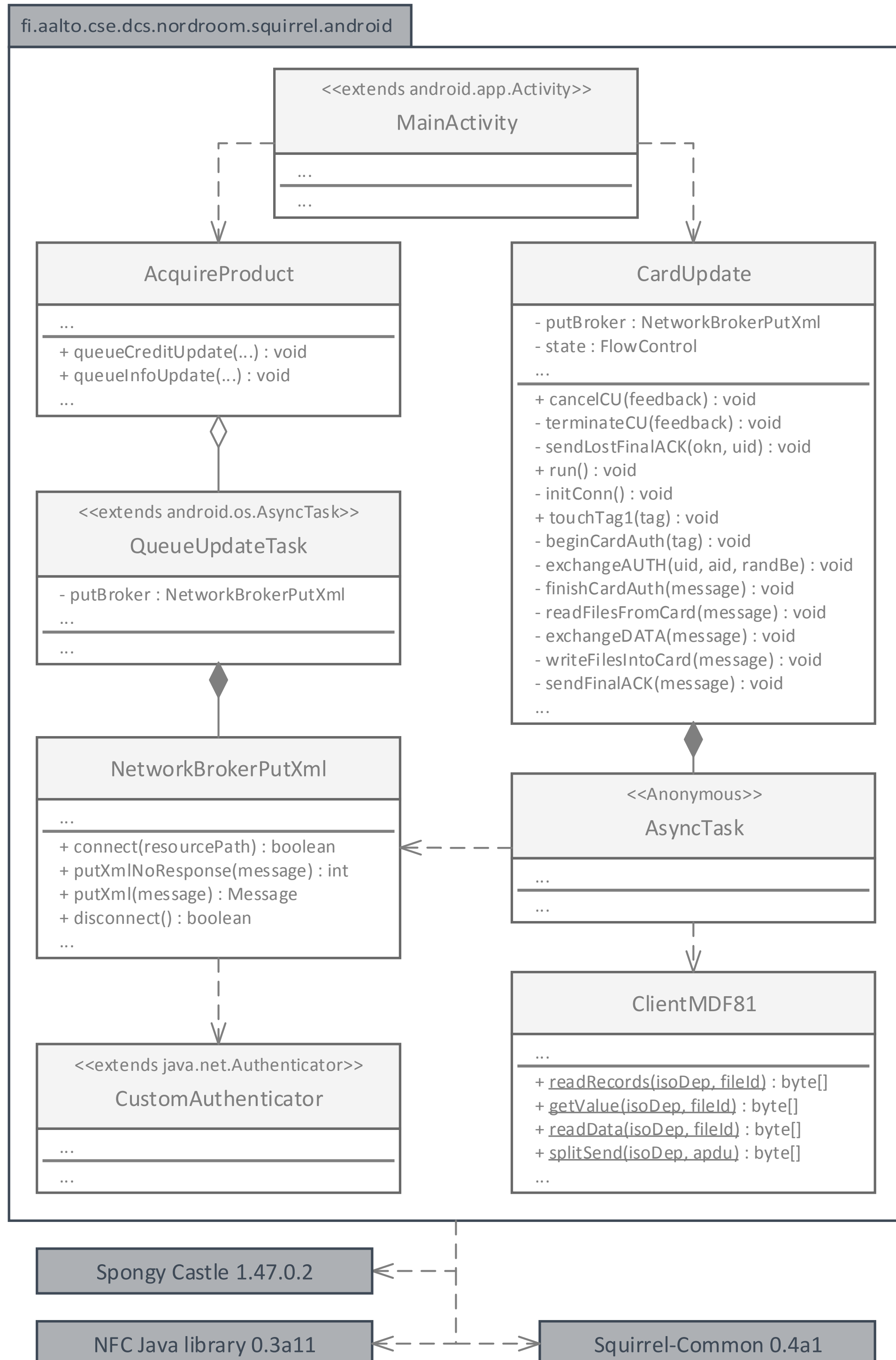


Figure 5.2: Class diagram and dependencies of the client application.

and manipulate the tag. Network operations are done through the class `NetworkBrokerPutXml`. These are required when contacting the server to acquire new updates and to update the card. The `CustomAuthenticator` is called to authenticate the user to the server. Since the credentials are stored on the phone, the user is only required to introduce the credentials if the credentials stored are wrong or non-existent. Tag manipulations are done through the class `ClientMDF81`, although some of the less complex commands are done inline. The external dependencies of the client application include the Spongy Castle core lightweight API and JCE provider, the NFC Java library and the Squirrel Common project. The Squirrel Common project contains classes to encapsulate the data exchanged between the web service and the client application and is further explained on the last paragraph of this section.

Figure 5.3 represents the class diagram for the web application and its dependencies. The class `AppBoot` tells the environment, the Apache Tomcat 7 servlet container, which JAX-RS services should be registered. In this web application, only the class `CardappResourceService` is registered. When a request for one of the methods served by this registered class arrives, the JAX-RS vendor implementation, Jersey in this particular case, creates a new instance for the duration of the request. This follows a per-request model. The alternative would be to reuse the same instance, by overriding the `getSingletons` method instead of the `getClasses` method. `CardappResourceService` implements the `CardappResource` interface, which contains the methods to be provided to the clients in the context of the prototype. One of those methods is used to acquire new updates and the other to apply the updates to the card. The class `CardappResourceService` provides additional resources, which are not part of the implemented interface. The objective of those resources is to debug the system and these would not be present on a production environment. `getHelloUnprotected` and `getHelloProtected` both return text to test the access to the protected area of the application. The difference is that the latter requires authentication, while the former does not. The method `dumpSquirrelDB` returns the content of some of the database tables. The external dependencies of the web application include Jersey, which is the reference implementation of JAX-RS, the SQLite JDBC Driver, the NFC Java library and the Squirrel-Common project. It also depends on JAXB, which is now part of Java SE.

Figure 5.4 represents the class diagram for the `persistence` package of the web service. This package serves as an interface for the SQLite database. It contains a `ConnectionFactory` that provides `Connections` to the database, data access objects (DAO), that implement CRUD functions to access the persistent storage, and the respective Java beans.

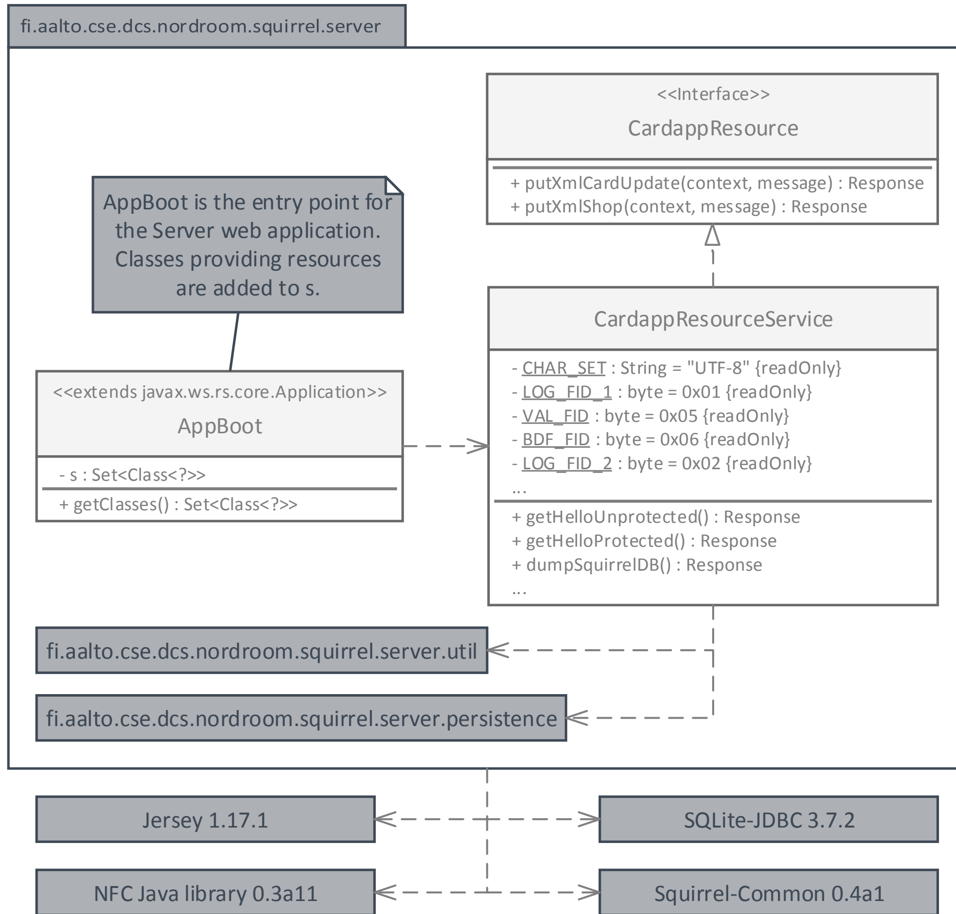


Figure 5.3: Class diagram and dependencies of the web application.

Figure 5.5 represents the class diagram for the utilities package of the web service. The class `ServerMDF81` provides methods to logically manipulate APDUs. Logically, because the server does not have direct access to a physical device. These methods allow to create and encipher command APDUs and respective response APDUs to be sent to client application, and to decipher response APDUs received from the client application. These APDUs wrap the updated files to be written to the card and the files read from the card. The class `SecretKey` enables the generation of secret keys for the card application, based on the smart card UID and a secret key known only to the entity providing the services. Since the cards are initialized by a different application and the web service can obtain the already diversified keys from the database, this class is included only for the purpose of completeness.

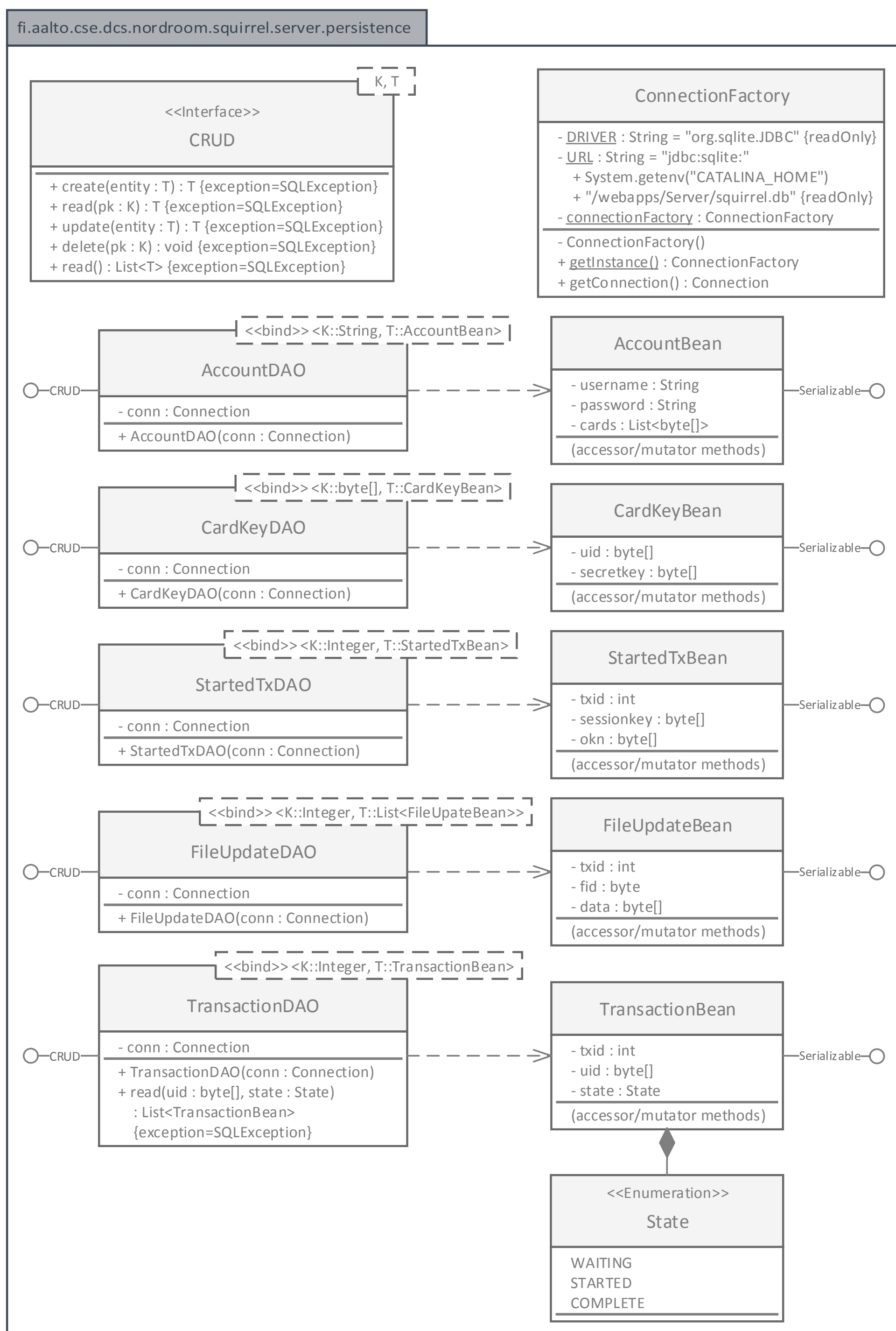


Figure 5.4: Class diagram of the `persistence` package of the web service.

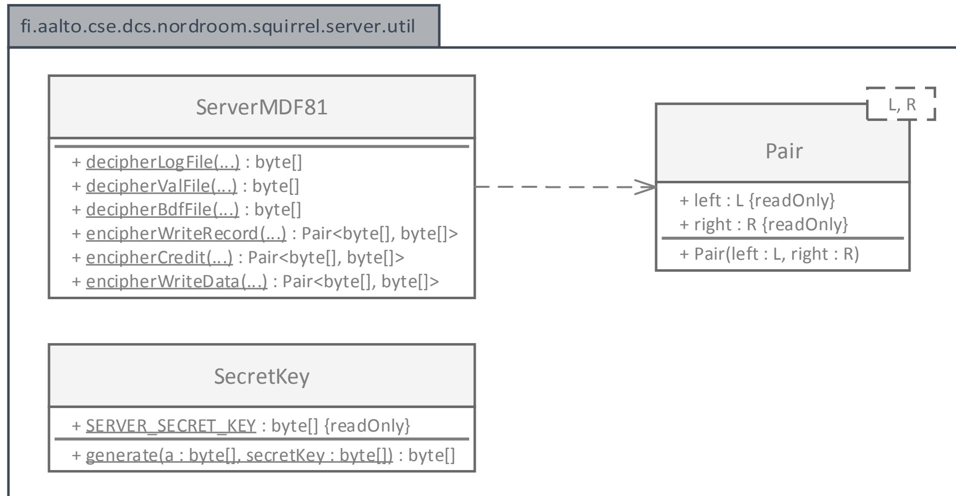


Figure 5.5: Class diagram of the utilities package of the web service.

The client application and the web service have two common dependencies, the NFC Java library and the Squirrel-Common project. The NFC Java library supports the manipulation of DESFire EV1 and the role of the Squirrel-Common project is described on the following paragraph.

The Squirrel-Common project contains classes to encapsulate the data exchanged between the web service and the client application. When the client application makes a request to the server, it sends a marshalled `Message` object and, depending on the operation, it may also receive a marshalled `Message` object. The object is marshalled and unmarshalled on the web service using JAXB. On the client application this parsing is done by hand, with the support of the XML Pull Parser library for unmarshalling. The `Message` class contains an `Intent`, that is, the purpose of the request or response message, and the AID of the application. These are the only attributes required. Optionally, it may contain the UID of the card, the enciphered and already manipulated random numbers to generate the session key, an array of files and the final ACK, that is, the response received from the card when the `CommitTransaction` command is sent. Each file contains its FID and space to store the enciphered data read from the card. Alternatively, instead of the enciphered data read from the card, it can hold an already prepared command to send to the card and the expected response to that command. When marshalling the `Message` class, the unused fields are omitted from the XML exchanged between the client application and the web application. Listing 5.1 exemplifies a marshalled `Message` object. All the attributes of

`Message` are present, but the second file does not have the `ack` set.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<message intent="NOT_SET">
    <uid>172582</uid>
    <aid>340595</aid>
    <session-key-data>
        <randBe>1a1b1c1d1e1f</randBe>
        <randABre>2a2b2c2d2e</randABre>
        <randAre>3a3b3c3d</randAre>
    </session-key-data>
    <files>
        <file fid="01">
            <data>010203040e</data>
            <ack>1a2b</ack>
        </file>
        <file fid="05">
            <data>050607080f</data>
        </file>
    </files>
    <commit-response>4a4b4d4e4f5a5b</commit-response>
</message>
```

Listing 5.1: Example of a marshalled `Message`.

5.3 Flow of operations

The operation to be performed by the system dictates whether the phone only needs to contact the server or if it needs to contact both the server and the card. A full purchase is a two-step process. First, there is the acquisition of the new update(s) to be performed on the card. Second, any pending transactions are applied to the card.

A new update can be a credit purchase or an update to the personal information of the user. In both cases, the phone only needs to communicate with the server. A new update is stored by the server, in the form of a pending transaction, and waits until a card update is requested by the phone.

A card update is requested by the phone to apply pending transactions to the card. In this case, the phone needs to communicate with both the card and the server. The card update follows the protocol detailed in Section 4.2.3.

5.3.1 Acquiring a new update

The user can either acquire a credit top-up or a personal information update. To acquire a credit top-up, the user selects an amount on the phone and

presses Buy.[1] This action creates a new Message with the NEW_FILE_UPDATE Intent, the AID of the card application, and the amount to top-up and respective value file number. To update her personal information, the user fills in the required fields and presses Update. This action creates a new Message with the Intent NEW_FILE_UPDATE, the AID of the card application, and the updated information and respective backup data file number.

When the web service receives the Message, it finds the card associated to this user to retrieve the UID of the card. A new Transaction and a new associated FileUpdate are then created on the database containing the update information. The state of the Transaction is set to WAITING.

At the end of this process, the HTTP status code 200 is returned to the client application. If the client application sends an invalid request, it receives a 400 HTTP status code. If a valid request is received but the web service is unable to process it, the client application receives the HTTP status code 500.

There is a caveat regarding the insertion of new transactions on the database: there can never be, at any point in time, more than five pending transactions. Pending transactions are transactions in the WAITING or in the STARTED state. If multiple pending transactions exist, these are applied to the card in a single card update, even though only one log entry is created for each of the two log files. Each entry contains the txids of the transactions applied to the card. If there are many txids, it is possible that this will generate a larger log entry than what is allowed by the card. The limit of five pending transactions prevents the overflow.

5.3.2 Updating the card

To update the card, the user selects the Update option on the client application. The card update follows the protocol detailed in Section 4.2.3, but the next paragraphs provide more details in relation to this implementation.

Multiple steps are taken on both the client application and on the web service to perform a card update. The user also participates actively in the update process because she is asked to, at some point, hold the card next to the phone. The steps are, in order:

Phone: initiate connection; touch card. The first action taken by the client application is to establish a secure connection to Apache Tomcat 7 and authenticate the user. This contributes to a reduced latency on the following message exchanges with the web service. The next step is

[1] *Buying* additional credit is done *for free* and without limits since setting up an online payment system is outside the scope of this thesis.

for the user to hold the card next to the phone. The client application waits until this action is taken.

Phone: initiate authentication. Once the user touches the phone, the UID of the card is read. Then the card application created for the prototype is selected and the authentication protocol is initiated for key number 03_H. The client application stores the enciphered random number received from the phone in a `Message` and sets the `Intent` to `AUTHENTICATE`. It also includes the UID and the AID before sending the `Message` to the web service. This `Message` will be the one exchanged between the client application and the web service until the end of the card update process, although its attributes are updated or deleted at different steps.

Server: process random numbers. The web service uses the diversified secret key associated with this card, which is stored on the database, to calculate the session key. It also finds all `Transactions` for this card in the `WAITING` or `STARTED` state. These are pending transactions in both cases, but a card update was already attempted for the latter. The state of the pending transactions found is set to `STARTED` and a `StartedTx` entry is added to the database, or updated if it already exists, with the new session key. If no pending transactions are found, the update is complete at this point and the client application is informed. Otherwise, the next command APDU to send to the card, in order to proceed with the authentication protocol, and the respective response APDU, are created and sent to the client application.

Phone: conclude authentication; read files. According with the `Intent` received, the client application either terminates the update or proceeds to a new step. The new step consists in sending the command APDU, received from the web service, to the card. The card replies with a new enciphered and rotated random number. The client application validates the response APDU against the one received from the server. The authentication protocol is now complete and the client application reads the files from the card application. The files read are enciphered with the session key, which the client application does not have access to. The data read is stored in the `Message` and sent to the web service using the `EXCHANGE_FILES Intent`.

Server: update files. The web service deciphers the files using the session key previously stored in a `StartedTx` entry. The log files read are validated. In practice, the last entry of the first log file is compared

with the last entry of the last log file. The timestamp and the identifiers of the transactions have to be the same in both log entries, otherwise it is likely that the system has been tampered with.

The files read are updated using the contents of `FileUpdate`s associated with `Transaction`s in `STARTED` state. Updates for the same file are aggregated in a single update for that file. The files are finally enciphered and added to `Message`, which is then sent to the client application.

Phone: write files. The client application writes the updated files into the card application and concludes the operation by sending the command `CommitTransaction` to the card. This makes the changes on the files of the card application permanent. The response APDU to this last command is sent to the web service.

Server: conclude update. The web service compares the received response APDU with the one pre-calculated and stored in `StartedTx`. If there is a match, it replies to the client application with an `UPDATE_COMPLETE Intent`. Otherwise, it replies with a `FINAL_ACK_MISMATCH Intent`. On success, the web service deletes the respective entries from the `StartedTx` table and sets the state of the `Transaction`s to `COMPLETE`.

Phone: conclude update. Whether there is a final ACK mismatch or not, the files have been updated and written to the card, or this step would have not been reached. The client application informs the user that the card has been updated. If there was a final ACK mismatch, the user is also informed.

Even though the updated files have been written to the card application, it is possible that the final ACK is not delivered. Attempting a new update will solve this situation, as discussed in Section 4.2.3.1.

On every request the web service receives, it validates not only the contents of `Message` but also makes sure that the username and the UID are associated with each other. This is possible because users cannot have multiple cards.

5.4 Implementation issues

The following section lists and describes issues with the implementation of the system and provides additional details regarding its functioning.

Initialization of the card application. The card application is initialized by an additional Java project. This involves the creation of the application itself and of the files in the application, the initialization of those files to default values and changing the secret keys of the application.

The application is created with master key settings $\texttt{0B}_{\texttt{H}}$ instead of the default of $\texttt{0F}_{\texttt{H}}$. This forces a previous authentication with the application master key to create and delete files, to change keys within the application and to modify the master key settings.

Two cyclic record files, a value file and a backup data file are created inside the card application. The two cyclic record files, with FIDs $\texttt{01}_{\texttt{H}}$ and $\texttt{02}_{\texttt{H}}$, constitute the start log file and the end log file. Each of these record files is created with five entries, which means that only four entries are usable since one of these entries is required for the backup mechanism. The capacity of each entry is 64 bytes. The start log file is initialized with the entry "201307070800;S;0" and the end log file is initialized with the entry "201307070800;E;0". The value file is created with zero credit, a minimum value of zero and a maximum value of 1024. The backup data file is created with a size of 128 bytes. It takes double of this memory on the card because of the backup data mechanism. A regular data file, without the backup mechanism, would not work, since the backup mechanism is needed to ensure atomic transactions. The backup data file is initialized with "Jane Doe;Female;Adult;Espoo".

The access rights of the files are set to $\texttt{3000}_{\texttt{H}}$, which means that key number $\texttt{03}_{\texttt{H}}$ has read and write access rights. The remaining access rights, given to key number $\texttt{00}_{\texttt{H}}$, are read, write and change access rights. The key number $\texttt{03}_{\texttt{H}}$ is the one used by the phone, although the phone does not know the secret key. This key is diversified by hashing the card UID concatenated to a secret string. The first half of the SHA256 hash is used as the diversified key. The other keys of the application are changed to new diversified keys, which follow the same process. The secret string is different for the other keys, hence the other keys are also different.

The card application cannot be created remotely by an untrusted device, such as the phone, because it requires access to the secret keys. The keys could be generated by the web service and sent to the card application enciphered. However, the phone knows the default keys and can gain access to the new ones using that information.

Digital certificates. The digital certificates were created using keytool. When creating a key pair, keytool stores it in a keystore. The store-

type, by default, is JKS. This is fine for the server keystore, but Android does not support the JKS storetype. Instead it uses BKS from Bouncy Castle, which is natively supported by the Android platform.

For keytool to support the BKS storetype, the Bouncy Castle collection of APIs must be included in the keytool path. This enables the creation of a keystore that Android is expected to work with. Even so, Android ships with an outdated version of Bouncy Castle and it is not trivial to import the updated version into the application because of classloader conflicts.

The solution is to use Spongy Castle, a modified version of Bouncy Castle. It contains the same libraries as Bouncy Castle but renames all packages and the security provider to avoid conflicts. By using Spongy Castle with keytool and importing it into Android, it is possible to successfully use the BKS storetype and have an updated version of the libraries in Android.

Root CA from industry. The system works with its own Root CA to sign the server digital certificate. The initial idea was to also support the use of certificates signed by a real CA from industry, and have the client application seamlessly working with both. Unfortunately, this was not possible because CAs require validation of the domain by email when using the free trial and there is no possibility of receiving an email on the subdomain assigned by the cloud provider.

A single company signed the digital certificate of the server. However, since this was a free trial for testing purposes only, the aforementioned company also provided the corresponding root certificate to be installed on the client machine. In the end, this amounted to what is already present when using the custom Root CA of this system.

Apache Tomcat 7 and SSL/TLS. To setup a secure connection to the resources of the web application, it is necessary to modify the configuration file `server.xml` of Tomcat and the `web.xml` configuration file of the web application. The `server.xml` configuration file needs a new `Connector` customized with the details of the keystore the server will use to provide a digital certificate to clients. The `web.xml` configuration file is updated to state which resources are to be protected and which users and/or realms have access to those resources.

This setup works for the development machine but not for Jelastic. Instead of setting up a new `Connector` in the `server.xml` configuration file, the customer is required to upload the signed domain certificate,

the intermediate certificate of the Root CA and the private key of the server. In the current system setup, the two certificates can be obtained using keytool but not the server private key. To extract the server private key from a keystore, the keystore if first converted to the PKCS12 storetype and then the key is extracted using OpenSSL.

Apache Tomcat 7 and SQLite. SQLite is a library that implements a self-contained transactional SQL database engine. It is used in the system given its simplicity. Apache Tomcat 7 allows the use of an external database to manage user accounts but does not support SQLite. The tables to manage users were created in the database of the system, but given this restriction the information had to be replicated on Apache Tomcat 7.

This is not a problem for the prototype. In a production system either Apache Tomcat 7 would have to be extended to support SQLite or the system would have to use a different database engine to avoid redundant information. GlassFish 4 does not support SQLite either.

Android tag dispatch system. Android creates an NFC intent when a tag is touched and sends it to interested applications. The client application is using `ACTION_TECH_DISCOVERED`, a rather general intent, which may cause it to open when an unrecognized type of tag is touched.

Chapter 6

Evaluation

This chapter evaluates the proposed solution and the implemented prototype against the requirements previously set for the system, provides a network latency and server congestion simulation to find how the implementation behaves under these conditions, and measures the time taken to perform a *card update* operation to verify that it completes in a timely manner. The evaluation is carried by means of experiments and in particular cases of a textual analysis. Recall that Section 4.2 describes the proposed solution, Chapter 5 describes the implemented prototype and the requirements previously set can be found in Section 4.1.

The experimental evaluation involves a remote server, an NFC-enabled smart phone and a smart card. The roles of these entities are introduced in Section 5.2 and depicted in Figure 5.1. Section 6.1 describes the test environment before proceeding to the analysis of the requirements in Section 6.3. Some of the requirements are proved by means of experiments, detailed in Section 6.2.

6.1 Experimental setup

The test environment is composed by three distinct entities. A web service contained in Apache Tomcat 7 and hosted by the Jelastic cloud provider, a Samsung Galaxy S3 GT-I9300 smart phone running Android 4.1.2, and a MIFARE DESFire EV1 smart card. The DESFire EV1 storage size is 4096 bytes, the hardware version is 1.0 and the software version is 1.4.

The web service is uploaded to Jelastic and started just before the tests and the server database file is recreated using a Python script. This script creates the required tables and inserts a few test accounts and associated cards, namely the Alice account used throughout the evaluation process.

The previously created digital certificates are uploaded to Jelastic, enabling SSL/TLS communication with the Apache Tomcat 7 servlet container. The secure communication between the phone and the server is discussed in Section 5.2.1.

The client application is uploaded to the smart phone several times throughout the evaluation process, since some of the tests require code modifications on the client side. However, the application preferences are kept. These include the user credentials and the web service base address. The smart phone has also been recently reset to its factory default settings and has been used exclusively to develop the application since that time. For that reason, it only contains a few test applications and the implemented prototype. The phone is connected to the eduroam WLAN, through which is able to reach the server.

The smart card is formatted before the first test and the card application is recreated. The card default master key is of type AES and it is maintained since it is not used. The application master key and remaining keys are modified according with the card initialization application. Section 5.4 contains additional information regarding the initialization of the card application.

6.2 Experiments

The following experiments aim to test the functionality of the system. The results are analysed in Section 6.3, taking into account the requirements set in Section 4.1.

There are three experiments. Experiment 1 tests the functional requirements of the system and the network latency and server congestion effect on the card, and measures the time taken to perform a *card update* operation. Experiment 2 tests the reliability of the system. Experiment 3 tests the authentication and authorization security properties.

6.2.1 Experiment 1—functional requirements

Experiment 1 involves remotely reading and updating the data stored on the card. It is composed by the following steps:

1. Attempt an update without having any pending transactions.

2. Acquire a credit top-up and apply this update to the card.

3. Acquire a personal information update and five credit top-ups. Apply these updates to the card.

4. Acquire a personal information update and apply this update to the card. This test includes a simulation of network latency and server congestion.

```
StartedTx       // no entries
Transaction     // no entries
FileUpdate      // no entries

Log 1 (Start):  201307070800;S;0
                201307070800;S;0
                201307070800;S;0
                201307070800;S;0
Value (0x5):    0
BData (0x6):    Jane Doe;Female;Adult;Espoo
Log 2 (End):    201307070800;E;0
                201307070800;E;0
                201307070800;E;0
                201307070800;E;0
```

Figure 6.1: The empty `StartedTx`, `Transaction` and `FileUpdate` tables and default values of the files on the card application.

Figure 6.1 shows the content of the relevant tables on the database and files on the card application, before attempting an update without having any pending transactions. The tables and the files do not change for step 1. These are also the default contents when database and card application are recreated.

Figure 6.2 shows the content of the same tables and files after executing steps 2–4. Transaction 1 is added by step 2, transactions 2–6 are added by step 3 and transaction 7 is added by step 4. When the updates are performed at each of these steps, the files are also updated to reflect the new credit value, personal information and log entries. Two interesting things happen here on step 2 and on step 4.

On step 2, one personal information update and five credit top-ups are acquired and applied to the card. However, only five entries are shown and not six. This happens because the last credit top-up was not acquired, since is not possible to have more than five pending transactions for reasons explained in Section 5.3.1. This is the expected behavior.

On step 4 and in order to simulate network latency and server congestion, the server includes a 60-second `Thread.sleep` on the method providing card resources. Network latency and server congestion are likely to be less than 60 seconds in a real usage scenario but testing the system with extreme

```
StartedTx                                 // no entries

Transaction
--------------------------------
(1, 04 2f 19 c2 80 26 80, COMPLETE)       // from step 2
(2, 04 2f 19 c2 80 26 80, COMPLETE)       // from step 3
(3, 04 2f 19 c2 80 26 80, COMPLETE)       // from step 3
(4, 04 2f 19 c2 80 26 80, COMPLETE)       // from step 3
(5, 04 2f 19 c2 80 26 80, COMPLETE)       // from step 3
(6, 04 2f 19 c2 80 26 80, COMPLETE)       // from step 3
(7, 04 2f 19 c2 80 26 80, COMPLETE)       // from step 4
--------------------------------

FileUpdate
--------------------------------
(1, 05, 05 00 00 00)
(2, 06, 41 6c 69 63 65 3b 46 65 6d 61 6c 65
        3b 41 64 75 6c 74 3b 45 73 70 6f 6f)
(3, 05, 0a 00 00 00)
(4, 05, 0a 00 00 00)
(5, 05, 0a 00 00 00)
(6, 05, 0a 00 00 00)
(7, 06, 41 6c 69 63 65 3b 46 65 6d 61 6c 65 3b
        41 64 75 6c 74 3b 48 65 6c 73 69 6e 6b 69)
--------------------------------

Log 1 (Start):  201307070800;S;0
                201308220851;S;1          // from step 2
                201308220855;S;2;3;4;5;6  // from step 3
                201308220900;S;7          // from step 4
Value (0x5):    45
BData (0x6):    Alice;Female;Adult;Helsinki
Log 2 (End):    201307070800;E;0
                201308220851;E;1          // from step 2
                201308220855;E;2;3;4;5;6  // from step 3
                201308220900;E;7          // from step 4
```

Figure 6.2: The contents of the tables and of the files after the updates.

values ensures the common cases are covered. The client application is set to timeout after 30 seconds when connecting to the server and 90 seconds when performing a request-response cycle. Since the 60-second delay and processing time on the server side and the current network latency between the phone and the server are less than 90 seconds, the client application does not timeout when requesting a resource, and the card update succeeds. The smart card is also active and touching the phone during the card update.

From the perspective of the smart card, this period is higher than 120 seconds because that is the sleep time from the first two request-response cycles between the client and the server. The card can safely be pulled apart from the phone after writing and committing the files, which happens just before the third and last data exchange to inform the server of the successful card update.

The experiment demonstrates that it is possible to perform a remote update to the files on the card application. Implicitly, it also demonstrates that the files on the card application can be remotely read. This happens during the card update because the files are read before being sent to the server to be updated.

In addition to the above tests, time measurements of the *card update* operation are taken to verify that it completes in a timely manner. Each round creates a transaction containing two updates, one for a credit file and one for a data file, and then executes the card update operation to apply the pending transaction to the card. The results are based on the execution of one hundred rounds and the time is measured only for the card update operation. The completion time for a card update ($N = 100$) operation averaged 1979.5 ms (sample standard deviation $s = 153.8$ ms).

6.2.2 Experiment 2—reliability

Experiment 2 tests the reliability of the system. The objective of this experiment is to observe how the data in the database and the card application change, when the update protocol is interrupted in key points. The protocol is interrupted at the following stages:

(1) Before initiating the secure connection with the server.

(2) When waiting for the user to touch the tag.

(3) Before initiating the authentication protocol with the card.

(4) Before exchanging the authentication data with the server.

(5) Before completing the authentication protocol with the card.

(6) Before reading the files from the card application.

(7) Before exchanging the files with the server.

(8a) Before writing the updated files into the card application.

(8b) After writing the files and before the `CommitTransaction`.

(9a) Before sending the final ACK to the server.

(9b) The final ACK is sent, but the server does not reply.

(10) The protocol runs from the beginning to the end.

For each stage and for each type of interruption, the tables `StartedTx`, `Transaction` and `FileUpdate` on the database and the content of the files on the card application are analyzed.

The six different forms of interrupting the system at these stages are: introducing a `System.exit`; pressing the Android *Home* button; pressing the Android *Back* button; switching off the wireless connection to *eduroam*; pulling the card away from the phone—card tear; and switching off the server. A `System.exit` terminates the card application immediately and proves to be the best way to test the protocol itself, since the exact exit point is known. For the other interruption types, a `Thread.sleep` is used to provide enough time for the interruption method to be carried.

In some cases, the client application does not leave at the `Thread.sleep`. This happens because either the interruption method does not apply immediately to the following instructions or the client application is still able to process additional instructions while the interruption method is being carried. Examples for the first case include switching off the wireless connection or the server, or pulling the phone and the card apart. The client application will only be unable to proceed when the required resource is not available. Examples for the second case include pressing the Android *Home* or *Back* buttons. The *Home* button sends the application to background and the *Back* button closes the application. In both cases, operations in progress are canceled, but the key press effect is not immediate.

Figure 6.1 presents the results of the experiment. Rows indicate the stage at which the client application is interrupted and columns indicate the interruption method. The ε symbol indicates an update failure, the ς symbol indicates an acknowledgment failure and the φ symbol indicates that the update protocol ran to its completion. See Section 4.2.3.1 for additional information on update and acknowledgment failures. The subscripts provide information on the data changed on the database and on the card application:

ε No data is changed.

ε_1 The transaction is started on the server. An entry is created in `StartedTx` with the session key set and the corresponding `Transaction` entry is updated from `WAITING` to `STARTED`.

	System.exit	Home	Back	Wireless	Tear	Server
(1)	ε	ε	ε	ε	–	ε
(2)	ε	ε	ε	ε	–	ε
(3)	ε	ε	ε_1	ε	ε	ε
(4)	ε	ε	ε_1	ε	ε_1	ε
(5)	ε_1	ε_1	ε_1	ε_1	ε_1	ε_1
(6)	ε_1	ε_1	ε_1	ε_1	ε_1	ε_1
(7)	ε_1	ε_1	ε_2	ε_1	ε_2	ε_1
(8a)	ε_2	ς	φ	ς	ε_2	ς
(8b)	ε_2	ς	ς	ς	ε_2	ς
(9a)	ς	ς	φ	ς	φ	ς
(9b)	ς_1	ς_1	ς_1	ς_1	ς_1	–
(10)	φ	φ	φ	φ	φ	φ

ε: update failure
ς: acknowledgment failure
φ: protocol completed

Table 6.1: The results of the reliability experiment.

ε_2 This implies ε_1, and in addition the `StartedTx` is later on updated with the final ACK during the file exchange.

ς The files on the card application are successfully updated but the server is not informed. The database state is the same as in ε_2.

ς_1 The files on the card application are successfully updated and the server is informed, but the client application is not aware that the acknowledgment message reached the server. The `StartedTx` entry is deleted and the `Transaction` is updated from `STARTED` to `COMPLETE`.

φ The protocol runs in its entirety from the beginning to the end.

In all stages marked with ε, and independently of changes happening on the database, the protocol is always restarted on a subsequent *card update* operation. In ε_1 and ε_2, since the state of the `Transaction` is already updated to `STARTED` and the entry in `StartedTx` is already created, the subsequent card update does not recreate the entry but updates the session key and the final ACK on the current entry instead.

In all stages marked with ς, the files on the card application are successfully updated but the server is not informed. When this happens, the client

application stores the final ACK and on a following card update attempts to send this information to the server before beginning the update protocol. If the final ACK sent matches the one stored by the server, the card update is complete without having to touch the card. Otherwise, the protocol restarts and the server can deduce from the logs read from the card application which transactions were applied. ς_1 is similar to ς, but the server fails to inform the client that it is aware of the finished state of the transaction. For that reason, the client application still stores the final ACK. When it is sent to the server, it is ignored because the entry in the `StartedTx` table has already been deleted. The client application then deletes the final ACK from its persistent storage.

To summarize, for all tested interruptions at these stages, the card, the phone and the server are eventually brought to a *consistent state*. Even when the files on the card application are updated but the final acknowledgment is lost, a consistent state between all entities is achieved.

6.2.3 Experiment 3—security

Experiment 3 tests the authentication and authorization properties of the system. Four different experiments are conducted:

(a) Attempt a card update without entering any credentials.

(b) Attempt a card update with a non-existent user.

(c) Attempt a card update with incorrect credentials.

(d) Attempt to update a card associated with a different user.

For the first test, a card update attempt is made without entering the credentials. For the second test, the user is set to *Mallory* in the client application settings. This user does not exist in the system. For the third test, the user is set to *Alice* in the client application settings. *Alice* is a legitimate user of the system, but the password entered is incorrect. A card update can be performed even without previously acquiring an update, resulting in an *update complete, no pending transactions* message. An authentication is still needed, but when the server finds that no pending transactions exist the operation is terminated. When a card update is attempted, the user is presented with a sign in screen in all three tests. The system does not inform the user whether the username does not exit or if it is only the password that is incorrect. The sign in screen disappears after entering valid credentials or after selecting a new option in the application, thus canceling the operation in progress.

For the fourth test, the user is set to *Alice* in the client application settings and the correct password is entered. A credit top-up is acquired, which does not require using the card, and stored as a pending transaction for *Alice*. The test itself lies in attempting to apply this new update to a card belonging to *Bob*, who is a legitimate user of the system. This card update fails because the card is not associated with Alice. The system only allows users to perform operations on their cards.

6.3 Analysis of requirements

This section argues that the prototype satisfies the requirements set for the system in Section 4.1. This claim is supported by the experiments carried in the public cloud provider Jelastic and described in Section 6.2.

The tables `StartedTx`, `Transaction` and `FileUpdate` are cleared before each experiment. `StartedTx` contains transactions in progress, `Transaction` contains all the transactions and respective state and `FileUpdate` contains the data to apply to files. The card is also formatted and the card application is recreated. The content of the tables and the files on the card application, before initiating the tests, are shown in Figure 6.1.

Functional requirements

REQ1. *Users can update the data stored on the card application remotely.*

REQ2. *Users can read data from the card application remotely.*

Non-functional requirements

REQ3. *Interrupted operations can be resumed without loss.*

REQ4. *Users can only update the data stored on the card application with the consent of the service provider.*

REQ5. *Users can only read the data stored on the card application with the consent of the service provider.*

REQ6. *Only authenticated operations may perform data updates on the card application.*

REQ7. *The confidentiality of the data stored on the card application is at the discretion of the service provider.*

REQ8. *The integrity of the data stored on the card application is enforced.*

REQ9. *Users cannot deny having used the smart card.*

REQ1 and **REQ2** are substantiated in Section 6.2.1 by Experiment 1. The card is successfully updated remotely. During a card update, the files on the card application are both read and updated.

REQ3 is substantiated in Section 6.2.2 by Experiment 2. A card update operation is interrupted at different stages of the protocol and using different methods. The system successfully recovers from failures.

REQ4 and **REQ5** are substantiated in Section 6.2.3 by Experiment 3 (d). Alice authenticates successfully with the server but is denied permission to update the card, because the card belongs to a different user. Furthermore, a rogue user attempting to read or update the files on the card application, without contacting the server, is unable to do so since the secret key required to read and update files is known only to the service provider. During a card update, the files on the card application are both read and updated.

REQ6 is substantiated in Section 6.2.3 by Experiment 3 (a–c). The user is unable to access the resources provided by the web service, which are required to perform operations, without being authenticated with Apache Tomcat 7.

REQ7 relates to the communication settings and the access rights of files, which are establish by the server provider. In the prototype and for all files, the communication settings are set to $\texttt{03}_\texttt{H}$ and none of the access rights is set to $\texttt{E}_\texttt{H}$. This implies that a previous authentication is always required to read and update files and that files can only be received from the card and sent to the card enciphered. It is possible for the service provider to relax these settings and allow, for instance, reading a specific file without a previous authentication. This would enable everyone to read that file in any card holding the same card application, which may raise privacy concerns.

REQ8 is accomplished by having the communication settings of files on the card application set to $\texttt{03}_\texttt{H}$. This implies that updates to files on the card can only be sent enciphered. Before the enciphering operation, a CRC is calculated and appended to the data. When the card receives a file, it validates the CRC and only proceeds with writing the file if it is correct.

REQ9 is fulfilled by logging every card update. The log files have limited but enough space to record the most recent updates applied to the card application. The server may include a logging mechanism, which can keep a longer history than what can be stored on the card.

The implementation passes the requirements set for the system, but with a notice. One special case related to Experiment 2, which proves **REQ3**, is not tested. When the client application sends a `CommitTransaction` to the card, it is theoretically possible that the card commits the transaction but the response to that command is lost. This means that the client application is unable to know whether the card update succeeded or not. The response to

this command is known in the previous sections and chapters as the *final ACK* and corresponds to message (26) in Figure 4.3. Its presence and correctness proves that the files on the card application were successfully updated.

In theory, if this response is lost the system is still able to reach a consistent state on a subsequent card update attempt. From the perspective of the server, it is as if the final exchange between client and server, which informs the server of the success of the operation and corresponds to message (27), is lost. The failure of this final exchange request is successfully tested in Experiment 2. It is therefore assumed that if the `CommitTransaction` response is lost the system can equally recover successfully and reach a consistent state.

Chapter 7

Discussion

The prototype enables a user to acquire updates for a card online and to rely on an NFC-enabled mobile phone and on a remote web service to apply those updates to the files on the card application. The card update operation is done in a reliable and secure way as proved by the evaluative tests.

This chapter discusses potential improvements to the designed and developed system and restrictions of the NFC Java library created during the study of DESFire EV1. It concludes with possible applications of, and some insights into, the researched subject.

7.1 Rethinking the NFC Java library

The NFC Java library assumes that the entity communicating with the smart card has all the required parameters for the manipulation taking place. This is not the case on the prototype, since the device interacting with the card, the NFC-enabled Android mobile phone, does not know the secret keys required to manipulate the files on the card application on its own. The phone only knows so much and mostly acts as a proxy, enabling the server to read and update files on the card application. In addition to creating command APDUs, wrapping native commands, validating responses, and updating the initialization vector autonomously, the library also applies the correct security mechanisms—CRC, MAC and encryption and decryption according with the selected cipher—to command APDUs and to response APDUs, shielding the application developer from tedious tasks prone to errors.

During the creation of the library, the priority was to understand how the card worked. The filesystem, its security mechanisms, the operations available and how to use them. A use case like the one of the prototype was not foreseen when developing the architecture of the library, in part

because of the lack of experience on this subject. This lead to the creation of additional methods specifically for this implementation. Some of these methods, the more general ones, were included in the library, while others, more specific, were added to the client application and to the web application.

7.2 Improving the system

The prototype developed as part of the thesis meets the requirements previously set for the system. There are, however, improvements to be made.

These improvements include using a database engine with a higher level of support for concurrency, such as MySQL [39]; and allowing more than five pending transactions for the system, which can be achieved by applying five transactions at a time, when more than five pending transactions are available. Additional enhancements to the system are discussed in the sections that follow.

7.2.1 UID and RID

A card update requires the card UID to be read before the first message exchange with the server. This raises a problem because DESFire EV1 requires a previous authentication to use the `GetCardUID` command, which returns the card UID, and the authentication protocol with the card only happens after this particular message exchange.

The implementation solves this problem by asking for the UID using the Android `getId` method for tags, which does not require a previous authentication. However, this method may return a random ID instead or no ID at all. The assumption is that the `getId` method uses the lower level ISO/IEC 14443-3 ANTICOLLISION and SELECT commands to acquired the tag ID, because the method header states "The tag identifier is a low level serial number, used for anti-collision and identification." [10]. This would be in agreement with the specification for handling of UIDs from NXP [27], where it is said that when using ISO/IEC 14443-3 identification and anticollision procedures, DESFire EV1 returns the RID instead of the UID if the RID option is enabled.

The cards used in the prototype return the UID using `getId`, because the random ID option is not enabled. Since each user only has one card and is authenticated with Apache Tomcat 7 before a card update, the system could also fetch the UID associated with the username from the database. Assuming a user with multiple cards, the system could ask the user to select the card she wants to update before starting the operation.

Another solution is for the service provider to write a unique custom identifier into a file and allow the file to be freely read, i.e. without being authenticated. The server would first read that file to acquire the custom card identifier and then proceed with the authentication protocol to gain access to the other files. Since this file can be freely read, the service provider can store the custom ID encrypted to prevent others from gaining access to the plaintext value. The disadvantage of this method is that the secret key used to decipher the contents of the file is the same across all cards.

Alternatively, instead of ciphering the contents of the file the service provider can simply protect the file with a secret key different from the one that grants access to the remaining files of the card application. This is possible since each card application can hold multiple secret keys. If the communication settings of the file are set to enciphered then only the service provider can read the contents of the file unless it shares the secret key with a third party. This option would require a higher number of messages to be exchanged with the card in comparison with the previous solution because the authentication protocol would have to be executed twice instead of only once. A first execution of the authentication protocol to retrieve the custom ID, which is used to generate the diversified secret key, and a second execution of the authentication protocol with the diversified secret key to gain access to the remaining files in the card application. This is a generic solution that could be applied to other systems.

7.2.2 Token-based authentication

The client application is required to authenticate against the Apache Tomcat 7 servlet container when performing requests to the web application. The client application stores the user credentials in cleartext for future use. This enables the application to only request new credentials from the user when the ones stored on the phone are either incorrect or non-existent. It is a useful feature from a usability perspective but it is also relatively insecure.

A possible approach to solve this security hole would be to use token-based authentication, that is, to store a token on the phone instead of the password of the user. When the user signs in, the server validates the credentials and creates an entry with the username, a creation timestamp and a token on a table. The token is sent to the client application. This token can be used by the client application to authenticate itself on future requests and can be safely stored on the phone.

There are multiple possibilities on how to create the token. For instance, the server can create a single secret key using a cryptographically secure PRNG. The token can then be generated by computing a HMAC-SHA256

over the username and the timestamp. If the client application sends the username, timestamp and token to the server, the server is able to compute the token again and check if it is forged. Alternatively, it can look for the token in the database and only grant access if a corresponding entry is present.

The user can sign out from the system by deleting the token in the client application and the corresponding entry in the server. Should the phone be lost or stolen, it should also be possible for the user to access the server, for example, through a browser, and instruct the server to sign out from a particular device or to sign out from all devices. In practice, this would respectively delete a specific token or all tokens, associated with the user, from the table. If a malicious user had access to the device, while it would be possible to access the server until a password change occurs or the token expires, it would be impossible to change the password without knowing the current one. In addition, since the password is not stored on the phone, the malicous user cannot recover it from the device.

7.2.3 Other enhancements

When performing a card update the system requires the user to be authenticated and to hold the phone next to the card. This happens because the proposed protocol for updating the card is followed to the letter in the implementation. In order to improve the usability of the system, the client application can first query the web service to find if there are pending transactions. If there are pending transactions, then the server asks the user to touch the card, which may imply, for instance, having to fetch the card out of the wallet. Otherwise, the card update completes without requiring the user to touch the card.

A constraint of allowing only one card per user was decided in order to lighten the complexity of the system. The database is designed to support multiple cards per user, but changes would have to be made in both web service and client application to support this new feature. In relation to the last paragraph, the system would then inform the user about which card to touch. When acquiring an update, the user could choose for which card the update is intended.

A further optimization is to allow users to buy updates to other users, for example, by entering the username of the target user. In addition, the system could also enable anyone with the client application installed to update any card with updates associated to that card. This is feasible since, from a design perspective, an authentication is not strictly required to execute the update protocol, although the current implementation enforces a previous

authentication by the user. If so decided by the service provider, this feature could be restricted by having a list of authorized users associated with each account.

Updating the cards of other users would not work with RID enabled. Since there is no authentication procedure with the server, it cannot find the UID in the database through the username. A possibility would be for the user to indicate a username when touching the card, but without providing a password. The server then relies on that username to find the associated cards and continue the authentication protocol with the card, which would fail if the username and the card are not associated, as a result of the diversified secret key being incorrect.

7.3 Insights and applications

The capability of applying updates to a tag following the orders of a remote server brings new and exciting possibilities. The service provider of a mass transit system, for instance, can develop an application to enable users to top-up their travel cards without the hassle of going to a service point, and to do so at their convenience; a supermarket chain can issue coupons for their customers, who are able to load them to their loyalty cards at home; a company can distribute discount vouchers for their employees over the Internet, and the employees can update the discount cards themselves. In all cases and from the perspective of the service provider, it is possible to apply the changes to the card securely and to save resources since less equipment and staff are required.

Consider a company that stores an identification number, name, gender and birth date in each of their employee cards. At some point later in time it is decided that an employee card should contain neither gender nor birth date. An update to the file containing that information would solve the issue and could be applied by the employees themselves via an NFC-enabled device. This would avoid having to call back all cards to the department responsible for them to perform the update.

Two optimizations proposed in Section 7.2.3, allowing the acquisition of new updates for other users and updating the cards of other users, enable a new interesting use case for the system. *Dave went to school and forgot to top-up his travel card. His mom, Carol, uses her mobile phone to acquire a new credit top-up for the travel card of Dave. Since Dave does not have an NFC-enabled mobile phone, he asks his friend Erin to apply the pending updates to his card. Erin touches the travel card of Dave with her NFC-enabled device and the system automatically finds and applies all pending*

updates for that card. Two interesting things happen in this scenario, a user acquires an update not to herself but to a third party, and the receiver of the update uses the NFC-enabled mobile phone of his friend to update the travel card, overcoming the lack of an appropriate device of his own.

Chapter 8

Conclusions

The main benefits brought by NFC and typical usage scenarios are related to the operating mode [30]. Card emulation mode allows to store multiple objects in an NFC-enabled mobile phone, removing the need for its physical counterpart. Examples include credit cards, tickets and keys. Peer-to-peer mode enables to easily exchange data and to pair devices. Reader-writer mode allows to transfer data from a tag, which can either be displayed to the user or trigger an action such as downloading additional content from the Internet, calling a number or sending an SMS.

In the traditional usage model of the reader-writer mode and in the context of mobile devices, NFC-enabled mobile phones mostly receive information from a tag. This thesis researched a new usage model that enables the mobile phone not only to read data from a tag but also to update it in a secure and reliable fashion with data supplied by a remote server. The functionality of DESFire EV1 was studied and the possibility of applying this novel model to the tag was analyzed. A communication protocol for remotely updating the files on the tag was devised and its practicability was successfully evaluated through a prototype built as a proof of concept. This prototype was subject to experiments in a public cloud provider to attest the correctness of the remote update protocol that it implements.

In the new usage model for the reader-writer mode, the NFC-enabled mobile phone acts as a proxy relaying data between a server and a tag, providing the remote server with the means to indirectly update the NFC tag. The mobile phone is a central piece of the system as a result of being responsible for reading from, and writing into, the tag and establishing the communication channel between the server and the tag. However, at any point in time is the mobile phone allowed to have access to the diversified secret keys shared between the server and the card as this would compromise the security of the system.

The paradigm of remotely updating a tag via an NFC-enabled mobile phone is as yet and to our knowledge not been studied in academic research; no references to such approach were found in the literature. However, it is known that several companies are working at the moment with NFC, e.g. for transport-ticket top-up. Benefits include updating a smart card without having to go to a specific location and at the convenience of the user, avoiding long queues and the associated waiting time, and to reduce the need for staff, which would otherwise be needed to serve customers, and automatic service points, used to update the smart cards of customers.

In addition to this document and to the implemented prototype, this work produced a Java library that eases the manipulation of Ultralight C and DESFire EV1 smart cards by shielding the user from tedious tasks prone to errors. The library code and respective documentation can be used by others interested in learning about these two types of cards since their specification is not publicly available.

Future work would involve restructuring the prototype taking into account the improvements discussed in Chapter 7, applying the proposed solution to a concrete real-world scenario, finding if the model is suitable for other types of smart cards or how it can be changed to fit the new card structure, and collecting and discussing possible applications for the remote update protocol. It would also be interesting to formally verify the optimized remote update protocol to prove its correctness.

Bibliography

[1] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. A view of cloud computing. *Commun. ACM 53*, 4 (Apr. 2010), 50–58.

[2] Coskun, V., Ok, K., and Ozdenizci, B. *Near Field Communication: From Theory to Practice*. Wiley, December 2011.

[3] Deloitte. NFC and mobile devices: payments and more! [Press release], 2012. http://www.deloitte.com/assets/Dcom-Global/LocalContent/Articles/TMT/TMTPredictions2012/16470ANFClb1.pdf. Accessed 8.4.2013.

[4] Elliott, J. The MAOS trap [smart card platforms]. *Computing Control Engineering Journal 12*, 1 (2001), 4–10.

[5] Eurosmart. Eurosmart forecasts over 7 billion smart secure devices to be shipped in 2012. [Press release], November 2012. http://www.eurosmart.com/images/doc/EurosmartPR/eurosmart_-_pr_nov_2012.pdf. Accessed 8.4.2013.

[6] Franssila, H. User experiences and acceptance scenarios of NFC applications in security service field work. In *Near Field Communication (NFC), 2010 Second International Workshop on* (2010), pp. 39–44.

[7] Fressancourt, A., Herault, C., and Ptak, E. NFCSocial: Social networking in mobility through IMS and NFC. In *Near Field Communication, 2009. NFC '09. First International Workshop on* (2009), pp. 24–29.

[8] German Federal Office for Information Security. *MIFARE DESFire EV1 MF3ICD81–Certification Report*, July 2011. Certification Report BSI-DSZ-CC-0712-2011.

[9] Ghiron, S., Sposato, S., Medaglia, C., and Moroni, A. NFC Ticketing: A prototype and usability test of an NFC-based virtual ticketing application. In *Near Field Communication, 2009. NFC '09. First International Workshop on* (2009), pp. 45–50.

[10] Google. `getId` method. [Website]. `https://developer.android.com/reference/android/nfc/Tag.html#getId()`. Accessed 24.8.2013.

[11] Hancke, G. A practical relay attack on ISO 14443 proximity cards. [Online], January 2005. `http://www.rfidblog.org.uk/hancke-rfidrelay.pdf`. Accessed 25.8.2013.

[12] Haselsteiner, E., and Breitfuss, K. Security in near field communication (NFC). In *Workshop on RFID Security 2006* (2006).

[13] Jaring, P., Törmänen, V., Siira, E., and Matinmikko, T. Improving mobile solution workflows and usability using near field communication technology. In *Proceedings of the 2007 European conference on Ambient intelligence* (Berlin, Heidelberg, 2007), AmI'07, Springer-Verlag, pp. 358–373.

[14] Jelastic. Jelastic java and php web cloud hosting with the best service providers. [Website]. `https://jelastic.com/`. Accessed 24.8.2013.

[15] Kasper, T., von Maurich, I., Oswald, D., and Paar, C. Chameleon: a versatile emulator for contactless smartcards. In *Proceedings of the 13th international conference on Information security and cryptology* (Berlin, Heidelberg, 2011), ICISC'10, Springer-Verlag, pp. 189–206.

[16] Madlmayr, G., Langer, J., Kantner, C., and Scharinger, J. NFC devices: Security and privacy. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on* (2008), pp. 642–647.

[17] Matos, A., Romao, D., and Trezentos, P. Secure hotspot authentication through a near field communication side-channel. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2012 IEEE 8th International Conference on* (2012), pp. 807–814.

[18] n.a. education roaming. [Website]. `https://www.eduroam.org/`. Accessed 24.8.2013.

[19] N.A. libfreefare. [Website]. https://code.google.com/p/libfreefare/. Accessed 23.8.2013.

[20] Nandwani, A., Coulton, P., and Edwards, R. NFC mobile parlor games enabling direct player to player interaction. In *Near Field Communication (NFC), 2011 3rd International Workshop on* (2011), pp. 21–25.

[21] NFC Forum. Near Field Communication Forum. [Online]. http://www.nfc-forum.org/. Accessed 9.9.2013.

[22] NFC Forum. *NFC Data Exchange Format (NDEF)*, July 2006.

[23] NFC Forum. *Type 1 Tag Operation Specification*, April 2011.

[24] NFC Forum. *Type 2 Tag Operation Specification*, May 2011.

[25] NFC Forum. *Type 3 Tag Operation Specification*, June 2011.

[26] NFC Forum. *Type 4 Tag Operation Specification*, June 2011.

[27] NXP. *MIFARE and handling of UIDs (AN10927)*, August 2011. http://www.mifare.net/files/6213/2453/8738/AN10927.pdf. Accessed 24.8.2013.

[28] NXP Semiconductors. *MF3ICDx21_41_81–MIFARE DESFire EV1 contactless multi-application IC–Product short data sheet–Rev. 3.1*, December 2010. http://www.nxp.com/documents/short_data_sheet/MF3ICDX21_41_81_SDS.pdf. Accessed 25.8.2013.

[29] NXP Semiconductors. *Security Target Lite–MIFARE DESFire EV1 MF3ICD81–Rev. 1.5*, May 2011. Security Target BSI-DSZ-CC-0712-2011.

[30] Ok, K., Coskun, V., Aydin, M., and Ozdenizci, B. Current benefits and future directions of NFC services. In *Education and Management Technology (ICEMT), 2010 International Conference on* (2010), pp. 334–338.

[31] Oswald, D., and Paar, C. Breaking mifare desfire mf3icd40: Power analysis and templates in the real world. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, B. Preneel and T. Takagi, Eds., vol. 6917 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 207–222.

[32] Ozdenizci, B., Aydin, M., Coskun, V., and Ok, K. NFC research framework: A literature review and future research directions. In *14th IBIMA conference on Global Business Transformation through Innovation and Knowledge Management* (June 2010), pp. 2672–2685.

[33] Philips Semiconductors. *mifare DESFire: Contactless Multi-Application IC with DES and 3DES Security–MF3 IC D40–Product specification–Rev. 3.1*, April 2004.

[34] Rankl, W., and Effing, W. *Smart Card Handbook*, fourth ed. Wiley, June 2010, ch. 10.

[35] Roland, M., Langer, J., and Scharinger, J. Applying relay attacks to google wallet. In *Near Field Communication (NFC), 2013 5th International Workshop on* (2013), pp. 1–6.

[36] Saminger, C., Grunberger, S., and Langer, J. An NFC ticketing system with a new approach of an inverse reader mode. In *Near Field Communication (NFC), 2013 5th International Workshop on* (2013), pp. 1–5.

[37] Sauveron, D. Multiapplication smart card: Towards an open smart card? *Information Security Technical Report 14*, 2 (2009), 70–78.

[38] Siira, E., Tuikka, T., and Tormanen, V. Location-based mobile wiki using NFC tag infrastructure. In *Near Field Communication, 2009. NFC '09. First International Workshop on* (2009), pp. 56–60.

[39] SQLite. Can multiple applications or multiple instances of the same application access a single database file at the same time? [Website]. `http://www.sqlite.org/faq.html#q5`. Accessed 25.8.2013.

Appendix A

MIFARE Ultralight C

This appendix provides a study of MIFARE Ultralight C, that was carried after studying MIFARE Ultralight and before studying the more complex MIFARE DESFire EV1. MIFARE Ultralight is quite similar to MIFARE Ultralight C and is not included in the thesis for that reason.

MIFARE Ultralight C (MF0ICU2) is a low-cost memory-based smart card for limited-use applications. For instance, event ticketing, loyalty schemes and public transportation. It uses a page-based memory structure like that of MIFARE Ultralight (MF0ICU1), but it has a larger amount of memory and it comprises additional features such as the 16-bit counter and the 2K3DES authentication mechanism. Ultralight C can be considered an intelligent memory card because it offers an access control system, imposing restrictions on which memory pages can be read and written.

Manipulation of the content of an Ultralight C smart card is done using the `Read` and the `Write` commands. `Read` allows to read user data and memory-mapped security configurations. `Write` allows to update user data and modify the security configurations. In both cases, the access is done one page at a time. Exceptions to this guideline include modification of the locking mechanism and of the OTP bits, where only the bits to be modified are set. Table A.1 lists the commands available for the manipulation of Ultralight C.

Commands
`Authenticate`, `Read`, `Write`

Table A.1: List of commands of Ultralight C.

A.1 Memory organization

Ultralight C uses a page-based memory structure. Each of the 48 pages is 4 bytes in size, for a total of 192 bytes of EEPROM memory.

00_H	UID	UID	UID	BCC0	Page 0
01_H	UID	UID	UID	UID	Page 1
02_H	BCC1	internal	LOCK0	LOCK1	Page 2
03_H	OTP	OTP	OTP	OTP	Page 3
04_H	user data	user data	user data	user data	Page 4
...	...	...	...	...	...
27_H	user data	user data	user data	user data	Page 39
28_H	LOCK2	LOCK3			Page 40
29_H	counter	counter			Page 41
$2A_H$	AUTH0				Page 42
$2B_H$	AUTH1				Page 43
$2C_H$	K1/0	K1/1	K1/2	K1/3	Page 44
$2D_H$	K1/4	K1/5	K1/6	K1/7	Page 45
$2E_H$	K2/0	K2/1	K2/2	K2/3	Page 46
$2F_H$	K2/4	K2/5	K2/6	K2/7	Page 47

Figure A.1: Memory layout of MIFARE Ultralight C.

The memory layout of Ultralight C is presented in Figure A.1. The first nine bytes of memory contain the unique read-only 7-byte serial number (SN/UID) and two block check character bytes (BCC). The last two bytes of page 2 contain the lock mechanism for pages 3–15. Page 3 contains the one time programmable (OTP) bits. Pages 4–39 are the user pages, that is, the pages available for the application to store data. This means that 144-bytes of application data can be stored on the card. The first two bytes of page 40 contain the lock mechanism for pages 16–47. The first two bytes of page 41 contain the 16-bit one-way counter. The first byte of page 42 and the first byte of page 43 store the authentication configuration, used to restrict access to pages. Pages 44–47 contain the 2K3DES secret key.

A.2 Security

The security features provided by Ultralight C include the unique UID, the page locking mechanism, the OTP bits, the one-way counter and the authentication configuration. The authentication configuration, allied with the 2K3DES authentication mechanism, enables access restrictions to pages.

The unique 7-byte UID of Ultralight C is programmed after production by the IC manufacturer. These seven bytes, along with the two BCC bytes, are write-protected to prevent later modification. The BCC calculation follows ISO/IEC 14443-3 and is defined as $CT \oplus SN0 \oplus SN1 \oplus SN2$ for BCC0 and as $SN3 \oplus SN4 \oplus SN5 \oplus SN6$ for BCC1. CT stands for cascade tag byte and is defined as 88_H.

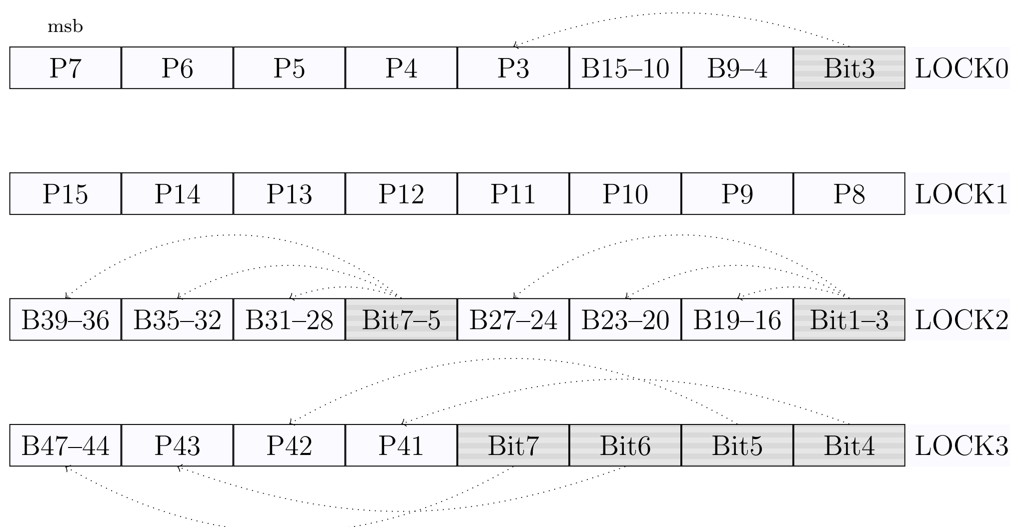


Figure A.2: Layout of the lock bytes of MIFARE Ultralight C.

The page locking mechanism enables the write-protection of pages. However, it does not prevent pages from being read. The locking mechanism is 4 bytes in size. LOCK0–1 are the last two bytes of page 2 and LOCK2–3 are the first two bytes of page 40. The layout of the lock bytes of Ultralight C is presented in Figure A.2. It is possible to lock individual pages, blocks of pages and individual lock bits. These are represented respectively as PX, B$X{-}Y$ and Bit$Z$. Pages 3–15 and pages 41–43 can be individually locked. Pages 4–39, that is, the user data pages, can be locked in blocks. Pages 44–47, which contain the 2K3DES secret key and are not readable, can also be locked as a single block. Some individual lock bits can be set to prevent other lock bits from being accidentally or intentionally set. For example, if

the most significant bit of LOCK3 is set, the secret key stored in pages 44–47 is frozen. To prevent this bit from being set, the fourth bit of LOCK3 can be set. If this is done while the eighth bit is cleared, then the secret key will always be changeable.[1]

Page 3 contains 32 OTP bits. These bits are cleared by default and can be individually set. Each of these bits can only be set once.

The 16-bit one-way counter, in page 41, is used to keep track of an always incrementing value. The default value is $\texttt{0000}_{\texttt{H}}$.

The authentication configuration bytes, AUTH0 and AUTH1, can be used to restrict either write or read and write access to pages. AUTH0 contains a page number, in hexadecimal, marking the page from which the settings in AUTH1 are applied. The effect extends from that page to the end of the memory. If AUTH0 is set to $\texttt{30}_{\texttt{H}}$, then there are no restricted pages because there are only $\texttt{2F}_{\texttt{H}}$ pages in total. The configuration in AUTH1 is either $\texttt{01}_{\texttt{H}}$ or $\texttt{00}_{\texttt{H}}$, for respectively restricted write access and restricted read and write access. A successful authentication enables full access to restricted pages.

The 2K3DES authentication protocol of Ultralight C ensures that both parties share a common 16-byte secret key. The authentication protocol, depicted in Figure A.3, is composed by the following steps:

1. The PCD sends an authentication request to the PICC. The APDU wrapped authentication request command is $\texttt{FF EF 00 00 02 1A 00}_{\texttt{H}}$.

2. The PICC receives the authentication command and generates and encrypts an 8-byte random number $RndB$. The resulting ciphertext is sent to the PCD as a response.

3. The PCD receives and decrypts the response obtaining the random number $RndB$ generated by the PICC. It then rotates $RndB$ one byte to the left yielding $x_2 = RndB'$. The PCD generates its own 8-byte random number $RndA$ and concatenates $RndB'$ to it. $RndA \| RndB'$ is encrypted in CBC mode and sent to the PICC.

4. The PICC decrypts the received message, rotates its $RndB$ to the left and compares it with the decrypted $RndB'$ sent by the PCD. If the match fails, the PICC returns an error code to the PCD. Otherwise, it rotates $RndA$ one byte to the left, yielding $RndA'$ (x_8). $RndA'$ is encrypted and sent to the PCD.

[1]Authentication with the current secret key may be required, depending on the authentication configuration in pages 42–43.

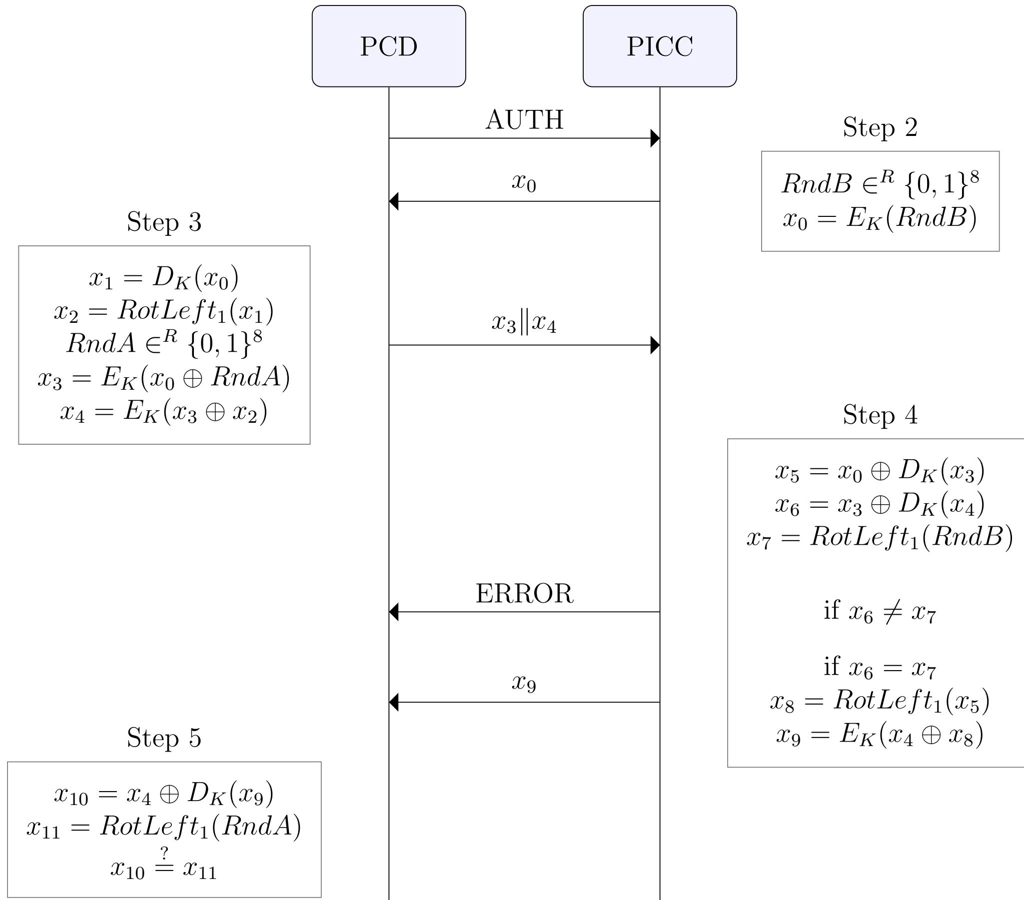


Figure A.3: 2K3DES authentication protocol for Ultralight C.

5. The PCD decrypts the received message, rotates its $RndA$ one byte to the left and compares it with the $RndA'$ received from the PICC. If the rotated $RndA$ and $RndA'$ match, the authentication is successful.

The parity bits of the 2K3DES secret key are ignored by the card. This means that a key with the parity bits set is equal to the same key with the parity bits cleared.

To change the secret key, it is enough to write the new key to pages 44–47. During the authentication protocol, the encryption and the decryption of data is done with the 2K3DES secret key in big-endian form. However, the secret key is stored in pages 44–47 in little-endian form, which is also the adopted form when changing the secret key. The first part of the key is written, in little-endian, to pages 44–45. The second part of the key is written, in little-endian, to pages 46–47. Consider the secret key `A7 A6 A5`

A4 A3 A2 A1 A0 B7 B6 B5 B4 B3 B2 B1 $B0_H$. To authenticate, the key is used as is. On the other hand, to change the secret key to this key, it is written as A0 A1 A2 A3 A4 A5 A6 A7 B0 B1 B2 B3 B4 B5 B6 $B7_H$. Page 44 stores A0 A1 A2 $A3_H$, page 45 stores A4 A5 A6 $A7_H$, page 46 store B0 B1 B2 $B3_H$ and page 47 stores B4 B5 B6 $B7_H$. After changing the secret key, the new key becomes active after reconnecting to the card. This implies that if the secret key is changed and an authentication is attempted before halting the card, the old secret key must be used.

Appendix B

New NFC Java library

The NFC Java library (nfcjlib) produced in this project eases the development of applications for MIFARE smart cards. The objective is to accelerate the development of applications and to increase their reliability by hiding the complexity of the native commands and cryptographic operations and to offer a less error-prone alternative to manually handle these repetitive tasks. The library is used to manipulate both MIFARE DESFire EV1 and MIFARE Ultralight C.

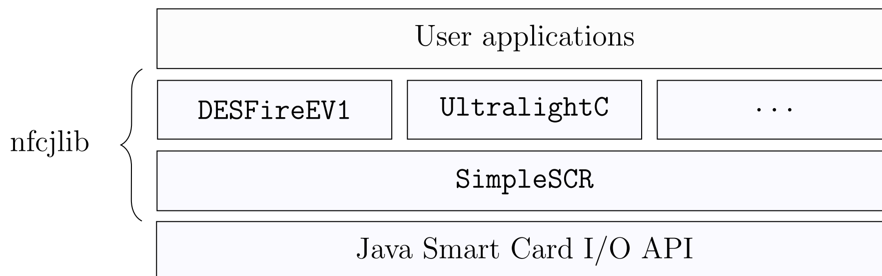


Figure B.1: The NFC Java library architecture.

The library was designed to be simple and expandable. The ecosystem is composed by three layers: the standard Java Smart Card I/O API, the nfcjlib library itself, and the applications of the end user. The Java Smart Card I/O API communicates with the smart card via a smart card reader and provides functions to nfcjlib to interact with the smart card. Nfcjlib consumes those functions and provides services to the end user. The end user manipulates the smart card through the services provided by nfcjlib. The architecture of the library is presented in Figure B.1 and is formed of two layers: the reader module and the smart card modules. The reader module interacts with the Java Smart Card I/O API and provides a service to the upper layer to connect, disconnect and transmit APDUs. This upper layer is divided into

modules, one for each smart card type. These modules, take `DESFireEV1` for instance, interact with the reader to establish and tear down connections and exchange information with the smart card, and to provide services to the end users on the upper layer.

The evaluation of the implementation relies on JUnit 4 for automated testing.[1] It aims to answer two questions:

Validation Are all the commands implemented for both smart cards?

Verification Are those commands implemented correctly?

To validate the library, the total number of commands is counted for both smart cards and compared with the number of commands implemented. To verify the correct implementation of those commands, a test plan is produced and JUnit 4 is used for the creation of test cases. The evaluation of DESFire EV1 is presented in Section B.1 and the evaluation of Ultralight C is presented in Section B.2.

In JUnit 4, both `Assume` and `Assert` methods are heavily used. Assumptions are conditions expected to be met for the test to be considered valid. Assertions are predicates related to the feature under test. For instance, suppose feature X requires a preceding authentication to be successfully tested. In such case, an assumption that a preceding successful authentication took place, before testing the feature, is made. Only then, can assertions regarding feature X take place. When an assertion fails, the test case fails. However, when an assumption fails, the test case succeeds. This happens because the conditions imposed for testing a feature are not met. This subtlety is important for a proper understanding of the test cases. It could be argued that some features may never be tested because the assumptions for those features are never met, and that it is impossible to tell whether a test is successful because the assertions were correct or because the assumptions failed. However, the assumptions are themselves tested. This ensures that if a test suite completes without errors, then all the test cases and respective assertions completed successfully.

B.1 Evaluation of the MDF implementation

The number of commands for the evaluation of the implementation of DESFire EV1 is taken from Table 3.1. There are five security-related commands, ten PICC-level commands, nine application-level commands and eleven data manipulation commands, for a total of 35 commands. Of the 35 commands,

[1] JUnit 4 is a framework to write repeatable tests for Java. See `http://junit.org/`.

only two commands, `SetConfiguration` and `GetDFNames`, are not implemented. This means that 33 out of 35 commands are implemented for DESFire EV1.

The DESFire EV1 smart card used during the evaluation is required to meet certain conditions. The PICC master key is set to 0^{16} and the type of key is AES. The PICC master key settings are set to $\texttt{0F}_{\texttt{H}}$. The following test plan details the tests taking place. The results of the test cases implemented in JUnit 4 are presented in Table B.1. All the test cases are done using all the different ciphers available—DES, 2K3DES, 3K3DES and AES.

100/Security Authenticate at PICC-level (key with version bits cleared).

101/Security Authenticate at PICC-level (key with version bits set).

102/Security Authenticate at application-level with key number $\texttt{00}_{\texttt{H}}$ (key with version bits cleared).

103/Security Authenticate at application-level with key number $\texttt{00}_{\texttt{H}}$ (key with version bits set).

104/Security Authenticate at application-level with key number different from $\texttt{00}_{\texttt{H}}$ (key with version bits cleared).

105/Security Authenticate at application-level with key number different from $\texttt{00}_{\texttt{H}}$ (key with version bits set).

106/Security Change PICC master key: from key with all version bits cleared to key with all version bits set and vice versa.

107/Security Change application master key: from key with all version bits cleared to key with all version bits set and vice versa.

108/Security Change application key different from $\texttt{00}_{\texttt{H}}$: from key with all version bits cleared to key with all version bits set and vice versa.

109/Security Change the PICC master key using X as version. Verify if the key version returned by the PICC is X.

110/Security Change the PICC master key settings to $\texttt{0F}_{\texttt{H}}$. Verify if the key settings are correct.

111/Security Change the key settings of an application to $\texttt{0D}_{\texttt{H}}$. Verify if the key settings are correct.

112/Security Successfully retrieve the manufacturing related data of the card.

113/Security Successfully retrieve the card UID.

114/Security Authenticate and format the card.

115/Security Attempt to format the card without a preceding authentication. The command fails.

116/Security Retrieve the free memory on the card.

200/Application Successfully select the AID $\texttt{00 00 00}_{\texttt{H}}$.

201/Application Attempt to select the AID of a non-existing application. The command fails.

202/Application Create an application.

203/Application Attempt to create an application with an existing AID. The command fails.

204/Application Create an application and then delete it successfully.

205/Application Attempt to delete a non-existing application. The command fails.

206/Application Create file X. Delete file X. File X does not exist after deletion.

207/Application Attempt to delete a non-existent file. The operation fails.

208/Application Create multiple files from 0 to 31 and verify if the amount of file identifiers is correct.

209/Application Create file X. Get and verify the properties of file X.

210/Application Attempt to get the properties of a non-existent file. The operation fails.

211/Application Create a file and successfully modify its properties.

300/SDF Create a standard data file.

301/SDF Create a standard data file and write X with offset Y. Read the entire file and verify if X is written at offset Y.

302/SDF Create a standard data file with size X and write data with size $> X$ to it. The writing fails.

400/BDF Create a backup data file.

401/BDF Create a backup data file with size > 512. To test the correct message separation in frames, write the full file length and read the entire file. Verify if the length and contents read are correct.

500/VF Create a value file with initial value X. A `GetValue` returns X.

501/VF Create a value file with initial value X. Increase the stored value by Y. The new value stored in the file is $X + Y$.

502/VF Create a value file with initial value X. Decrease the stored value by Y and by Z with $Y \neq Z$. The new value stored in the file is $X - Y - Z$.

503/VF Create a value file with initial value X and with `LimitedCredit` enabled. Decrease the stored value by Y. Successfully increase the value by Y using `LimitedCredit`.

504/VF Create a value file with initial value X and maximum value M. Increase the initial value by Y such that $X + Y > M$. The `Credit` command fails.

505/VF Create a value file with initial value X and minimum value M. Decrease the initial value by Y such that $X - Y < M$. The `Debit` command fails.

506/VF Create a value file with initial value X and with `LimitedCredit` enabled. Decrease the stored value by 5. Now increase the value by 1 using `LimitedCredit`. Repeat the previous command after committing. `LimitedCredit` fails. It can only be done once after a transaction involving a `Debit`.

507/VF Create a value file with initial value X and with `LimitedCredit` enabled. Decrease the stored value by Y. Now increase the value by Z using `LimitedCredit`, such that $Z > Y$. `LimitedCredit` fails.

508/VF Create a value file with initial value X and with `LimitedCredit` disabled. Decrease the stored value by Y. Now increase the value by Y using `LimitedCredit`. `LimitedCredit` fails.

600/LRF Create a linear record file.

601/LRF Create a linear record file with record size X and number of records Y. Create Y records. A full read returns $X \times Y$ bytes.

602/LRF Create a linear record file with two records. A read with offset 1 and number of records 0 returns 1 record.

603/LRF Create a linear record file with three records. A read with offset 0 and number of records 2 returns 2 records.

604/LRF Create a linear record file with record size 2. Create a record containing `41 42`$_H$. A read of the created record returns `41 42`$_H$.

605/LRF Create a linear record file and write a record. Clear the record file and attempt to read it. Reading fails because there are no records.

700/CRF Create a cyclic record file.

701/CRF Create a cyclic record file with record size 2 and number of records 3. Create $3 - 1$ records with contents `41 42`$_H$ and `51 52`$_H$ respectively for the first and for the second records. A full read returns `41 42 51 52`$_H$.

702/CRF Create a cyclic record file with record size 1 and number of records 3. Create 3 records with contents `1A`$_H$, `1B`$_H$ and `1C`$_H$. A full read returns `1B 1C`$_H$.

703/CRF Create a cyclic record file with record size 1. An attempt to write a record with size > 1 fails.

800/File Create a value file with initial value X, credit Y and commit the transaction. The value stored on the file is $X + Y$.

801/File Create a value file with initial value X, credit Y and abort the transaction. The value stored on the file is X.

Id/context	Key operations performed	Test for	Result
100/Security	`Authenticate`	success	✔
101/Security	`Authenticate`, `ChangeKey`	success	✔
102/Security	`Authenticate`	success	✔
103/Security	`Authenticate`, `ChangeKey`	success	✔
104/Security	`Authenticate`	success	✔
105/Security	`Authenticate`, `ChangeKey`	success	✔

106/Security	Authenticate, ChangeKey	success	✔
107/Security	Authenticate, ChangeKey	success	✔
108/Security	Authenticate, ChangeKey	success	✔
109/Security	GetKeyVersion, ChangeKey	success	✔
110/Security	ChangeKeySettings, GetKeySettings	success	✔
111/Security	ChangeKeySettings, GetKeySettings	success	✔
112/Security	GetVersion	success	✔
113/Security	GetCardUID	success	✔
114/Security	Authenticate, FormatPICC	success	✔
115/Security	FormatPICC	failure	✔
116/Security	FreeMemory	success	✔
200/Application	SelectApplication	success	✔
201/Application	SelectApplication	failure	✔
202/Application	CreateApplication	success	✔
203/Application	CreateApplication	failure	✔
204/Application	DeleteApplication	success	✔
205/Application	DeleteApplication	failure	✔
206/Application	DeleteFile, GetFileIds	success	✔
207/Application	DeleteFile	failure	✔
208/Application	GetFileIds	success	✔
209/Application	GetFileSettings	success	✔
210/Application	GetFileSettings	failure	✔
211/Application	ChangeFileSettings	success	✔
300/SDF	CreateStdDataFile	success	✔
301/SDF	WriteData, ReadData	success	✔
302/SDF	WriteData	failure	✔
400/BDF	CreateBackupDataFile	success	✔
401/BDF	WriteData, ReadData	success	✔
500/VF	CreateValueFile, GetValue	success	✔
501/VF	Credit, GetValue	success	✔
502/VF	Debit, GetValue	success	✔
503/VF	LimitedCredit, GetValue	success	✔
504/VF	Credit	failure	✔

505/VF	`Debit`	failure	✔
506/VF	`LimitedCredit`	failure	✔
507/VF	`LimitedCredit`	failure	✔
508/VF	`LimitedCredit`	failure	✔
600/LRF	`CreateLinearRecordFile`	success	✔
601/LRF	`WriteRecord`, `ReadRecords`	success	✔
602/LRF	`WriteRecord`, `ReadRecords`	success	✔
603/LRF	`WriteRecord`, `ReadRecords`	success	✔
604/LRF	`WriteRecord`, `ReadRecords`	success	✔
605/LRF	`ClearRecordFile`, `ReadRecords`	failure	✔
700/CRF	`CreateCyclicRecordFile`	success	✔
701/CRF	`WriteRecord`, `ReadRecords`	success	✔
702/CRF	`WriteRecord`, `ReadRecords`	success	✔
703/CRF	`WriteRecord`	failure	✔
800/File	`CommitTransaction`	success	✔
801/File	`AbortTransaction`	success	✔

Table B.1: Test case results of nfcjlib for DESFire EV1.

B.2 Evaluation of the MUC implementation

The number of commands, for the evaluation of the implementation of Ultralight C, is taken from Table A.1. There are three commands, all of them implemented in nfcjlib.

The Ultralight C smart card used during the evaluation is required to meet certain conditions. The secret key is set to 0^{16}, the lock bytes are all set to zero and AUTH0 is greater than or equal to $\texttt{30}_\texttt{H}$. The following test plan details the tests taking place for the evaluation of Ultralight C. The results of the test cases, implemented in JUnit 4, are presented in Table B.2.

900 Authenticate using the default secret key (0^{16}) with all parity bits cleared. The authentication succeeds.

901 Authenticate using the default secret key with some parity bits set. The authentication succeeds.

902 Change the secret key to X, where X is different from the default secret

key. Authenticate using X with all parity bits cleared. The authentication succeeds.

903 Change the secret key to X, where X is different from the default secret key. Authenticate using X with some parity bits set. The authentication succeeds.

904 Authenticate with an incorrect secret key. The authentication fails.

905 Change the secret key. The command succeeds.

906 Update a user page. The commands succeeds.

907 Read a user page. The commands succeeds.

908 Write X into a user page Y. Reading page Y returns X.

909 Attempt to update a non-existent user page. The command fails.

910 Attempt to read a non-existent user page. The command fails.

Id	Key operations performed	Test for	Result
900	`Authenticate`	success	✔
901	`Authenticate`	success	✔
902	`Authenticate`, `ChangeSecretKey`	success	✔
903	`Authenticate`, `ChangeSecretKey`	success	✔
904	`Authenticate`	failure	✔
905	`ChangeSecretKey`	success	✔
906	`Update`	success	✔
907	`Read`	success	✔
908	`Update`, `Read`	success	✔
909	`Update`	failure	✔
910	`Read`	failure	✔

Table B.2: Test case results of nfcjlib for Ultralight C.

Appendix C

Creation of digital certificates

This appendix shows the commands used to create the digital certificates and to extract the private key from the keystore. Both keytool and OpenSSL are used. The Spongy Castle security provider is added to the path when handling the keystore of the client application. The certificate of the server is tied to the Jelastic hostname *squirrel.jelastic.planeetta.net*.

Generate a key pair for the Root CA, export the self-signed certificate and import it into the keystore of the client application.

```
$ keytool -genkeypair \
    -alias myrootca -keyalg RSA -keysize 2048 -validity 365 \
    -dname "cn=myrootca, ou=CSE, o=Aalto, l=Espoo, s=Uusimaa, c=FI" \
    -ext bc:c -keystore ca.jks -storetype JKS \
    -storepass rootpass -keypass rootpass
$ keytool -exportcert \
    -alias myrootca \
    -keystore ca.jks -storetype JKS -storepass rootpass \
    -rfc -file myrootca.pem
$ keytool -importcert \
    -alias myrootca -file myrootca.pem -keystore android.bks \
    -storetype BKS -storepass changeme -noprompt \
    -provider org.bouncycastle.jce.provider.BouncyCastleProvider \
    -providerpath android-bks/scprov-jdk15on-1.47.0.2.jar
```

Generate a key pair for the server and import the self-signed certificate of the Root CA into the keystore of the server.

```
$ keytool -genkeypair \
    -alias squirrel.jelastic.planeetta.net -keyalg RSA \
    -keysize 2048 -validity 365 \
    -dname "cn=squirrel.jelastic.planeetta.net, ou=CSE, \
        o=Aalto, l=Espoo, s=Uusimaa, c=FI" \
    -keystore server.jks -storetype JKS \
    -storepass changeit -keypass changeit
```

```
$ keytool -importcert \
    -alias myrootca -file myrootca.pem -keystore server.jks \
    -storetype JKS -storepass changeit -noprompt
```

Export the keystore of the server into the PKCS12 storetype format and retrieve the private key of the server using OpenSSL.

```
$ keytool -importkeystore \
    -srcalias squirrel.jelastic.planeetta.net \
    -destalias squirrel.jelastic.planeetta.net \
    -srckeystore server.jks \
    -destkeystore intermediate.squirrel.jelastic.planeetta.net.pkcs12 \
    -srcstoretype JKS -deststoretype PKCS12
$ openssl pkcs12 \
    -in intermediate.squirrel.jelastic.planeetta.net.pkcs12 \
    -out squirrel.jelastic.planeetta.net.key -nocerts -nodes
```

Generate a CSR for the server.

```
$ keytool -certreq \
    -alias squirrel.jelastic.planeetta.net \
    -file squirrel.jelastic.planeetta.net.csr \
    -keystore server.jks -storetype JKS \
    -storepass changeit -keypass changeit
```

The Root CA uses the previously generated CSR to produce a signed certificate for the server and the server imports the signed certificate into its keystore. This keystore is accessed by the web server to retrieve the digitally signed certificate, which is sent to clients during the setup of the secure connection.

```
$ keytool -gencert -alias myrootca \
    -ext san=dns:"squirrel.jelastic.planeetta.net" \
    -infile squirrel.jelastic.planeetta.net.csr -rfc \
    -outfile squirrel.jelastic.planeetta.net.pem \
    -keystore ca.jks -storetype JKS -storepass rootpass
$ keytool -importcert \
    -alias squirrel.jelastic.planeetta.net \
    -file squirrel.jelastic.planeetta.net.pem -keystore server.jks \
    -storetype JKS -storepass changeit -noprompt
```

List the contents of the keystores.

```
$ keytool -list -keystore ca.jks -storetype JKS -storepass rootpass
$ keytool -list -keystore server.jks -storetype JKS -storepass changeit
$ keytool -list \
    -keystore android.bks -storetype BKS -storepass changeme \
    -provider org.bouncycastle.jce.provider.BouncyCastleProvider \
    -providerpath android-bks/scprov-jdk15on-1.47.0.2.jar
```