Robert Gerd Pleyer, BSc

Design and Implementation of an Encrypted Secure Channel for NFC Systems based on Android

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Institute for Technical Informatics

Advisors
Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger
Dipl.-Ing. Dr.techn. Manuel Menghin (Infineon Technologies Austria AG)

Graz, March 2015

Abstract

Integrated into nearly every new smartphone, Near Field Communication (NFC) is gaining momentum all over the world. This technology allows for communication with actively and passively powered Radio-Frequency Identification (RFID) tags. In combination with intelligent backend systems which process the gathered information, new applications for NFC arise: as a part of the META[:SEC:] research project, an NFC system with a backend server for RFID systems has been implemented. On the tag side a security cryptocontroller prototype allows for sophisticated security operations. An NFC-capable smartphone with Android Operating System forms the bridge technology to communicate with both sides via NFC and Wi-Fi, respectively. The system consisting of tag, central backend and mobile device offers a lot of possible attack vectors. Thus, the main focus of this thesis is to create and use a secure channel between the backend system and the tag by encrypting all sensitive communication. Therefore, the smartphone itself is not able to gather any information from the sent messages. An initial unencrypted tag authentication protocol features Elliptic Curve Cryptography (ECC) and Montgomery Multiplication in $\mathbb{F}_p$. The Advanced Encryption Standard (AES) is used in Offset Codebook (OCB) mode to encrypt further messages as well as to get hash values to provide message integrity. Using the secure channel, tag data related to the ePedigree standard is read from and written to the tag, while the data is signed using the Elliptic Curve Digital Signature Algorithm (ECDSA). Any communication between the smartphone and the backend is additionally encrypted using Transport Layer Security (TLS).

Keywords: RFID Security, Advanced Encryption Standard, Offset Codebook Mode, Authenticated Encryption, Near Field Communication, NFC, Android OS, Authentication, Transport Layer Security, Elliptic Curve Cryptography, METASEC

Kurzfassung

Durch die Integration in fast jedem neuen Smartphone, gewinnt Near Field Communication (NFC) weltweit an Bedeutung. Die Technologie ermöglicht Kommunikation mit aktiv oder passiv versorgten Radio-Frequency Identification (RFID) Tags. In Kombination mit intelligenten Backend Systemen, welche die gesammelten Daten auch verarbeiten können, ergeben sich neue Applikationsmöglichkeiten: Als Teil des META[:SEC:] Forschungsprojektes wurde ein NFC-System mit einem Backend Server für RFID-Systeme implementiert. Der Prototyp eines Sicherheits-Cryptocontrollers als RFID Tag ermöglicht hierbei das Durchführen anspruchsvoller Security-Operationen. Ein NFC-fähiges Smartphone, mit dem Betriebssystem Android, wird als Brückentechnologie verwendet um mit beiden Seiten mittels NFC bzw. Wi-Fi zu kommunizieren. Das System, bestehend aus Tag, Backend und Smartphone, hat potentiell aber auch eine Vielzahl von Angriffspunkten. Daher wurde in dieser Arbeit der Focus auf die Absicherung des Kanals zwischen Backend und Tag mithilfe von Verschlüsselung der sensiblen Daten gelegt. Dadurch kann die Smartphone Applikation die Informationen, in den von ihr gesendeten Nachrichten, selbst nicht einsehen. Ein initiales, unverschlüsseltes, Authentifizierungsverfahren verwendet Montgomery Multiplikationen in $\mathbb{F}_p$. Weitere Nachrichten werden mit dem Advanced Encryption Standard (AES) im Offset Codebook (OCB) Mode verschlüsselt, wobei auch ein Hash Tag für die jeweilige Nachricht erzeugt wird, um Datenintegrität zu erreichen. Über den sicheren Kanal werden außerdem Tag-Daten, in Form des ePedigree Standards, ausgelesen und wieder auf den Tag geschrieben, wobei die Daten mit dem Elliptic Curve Digital Signature Algorithm (ECDSA) signiert werden. Die Kommunikation zwischen Smartphone und Backend ist zusätzlich mit Transport Layer Security (TLS) gesichert.

Keywords: RFID Security, Advanced Encryption Standard, Offset Codebook Mode, Authenticated Encryption, Near Field Communication, NFC, Android OS, Authentication, Transport Layer Security, Elliptic Curve Cryptography, METASEC

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

....	....
Date	Signature

Acknowledgements

First I would like to thank my supervisor Christian Steger at the Institute for Technical Informatics and my advisors Manuel Menghin and Christian Lesjak at Infineon Technologies Austria AG for their aspiring guidance, invaluably constructive criticism and friendly advice during this thesis and the preceding master's project.

Next, I have to say many thanks to Matthias Weitlaner for giving me the opportunity to do this thesis in cooperation with Enso Detego. I would like to gratefully thank the entire staff at Enso Detego GmbH for their support and for being the most awesome colleagues I ever had.

I would like to thank my QA team, especially Thomas Kempter and Michael Goller, who took it upon themselves to proofread this thesis, therefore making it presentable to a public audience.

I have to say thanks to my security consultant Christoph Dobraunig for providing me with the insight and knowledge to carry out this thesis. I want to thank Peter, Alex, Dan, Michi, Manuel, Jürgen and all my other friends for their support and for saving me from finishing too early, by providing me with much needed distractions.

Thanks to my partner in life Heidi for being the best and most beautiful girlfriend during the whole thesis, and beyond.

Finally, I want to thank my whole family, my parents Gabriele and Alfred, my sister Martina and especially my grandfather Gerd, for supporting me throughout my entire academical career. Without them, I would have never been able to become the person I am today.

Graz, February 2015 Robert Gerd Pleyer

Contents

1 Introduction 23

2 Related Work 25

2.1 Cryptography Fundamentals . . . 25
2.1.1 Symmetric Cryptography . . . 25
2.1.2 Asymmetric Cryptography . . . 26
2.1.3 Authenticated Encryption . . . 28
2.1.4 Offset Codebook . . . 28
2.1.5 Montgomery Multiplication . . . 31
2.2 Secure RFID . . . 32
2.2.1 RFID . . . 32
2.2.2 NFC . . . 32
2.2.3 ECC . . . 33
2.2.4 RFID-Tag Prototype . . . 33
2.3 Secure Backend . . . 34
2.3.1 Backend-Connection with TLS . . . 35
2.4 Android Operating System . . . 36
2.5 ePedigree Standard . . . 37
2.6 Similar Systems . . . 38

3 Design 41

3.1 The Secure Channel System . . . 41
3.1.1 Concept . . . 42
3.1.2 Anti-Counterfeit Use Case . . . 42
3.1.3 Authentication . . . 43
3.1.4 Tag Data and Operations . . . 44
3.2 Backend Design . . . 46
3.2.1 Database . . . 47
3.2.2 Web Service . . . 49
3.3 RFID-Tag Prototype . . . 50
3.3.1 Protocol . . . 50
3.3.2 Tag Data . . . 53
3.4 Android Application Design . . . 54
3.4.1 Android User Interface . . . 54
3.5 Workflow . . . 57

4 Implementation **63**
4.1 Software Simulation . . . 63
4.1.1 Key Generation . . . 63
4.1.2 Authentication (Simulation) . . . 65
4.1.3 OCB Simulation . . . 67
4.2 Backend Server . . . 67
4.2.1 Command Builder . . . 68
4.2.2 Pedigree Handler . . . 69
4.2.3 Secure Channel Commands . . . 70
4.3 Android Application . . . 79
4.3.1 Class ConnectionHTTPS . . . 80
4.3.2 Class CommandHandler . . . 83
4.3.3 Class HomeActivity . . . 83

5 Results **87**
5.1 Android Application . . . 88
5.2 Timing Analysis . . . 90

6 Conclusion **95**

Bibliography **97**

A Figures **105**

B Listings **109**

List of Figures

1.1 Overview of the Secure Channel System 24

2.1 Secure Communication 26
2.2 X.509 Standard for Generating Digital Certificates 27
2.3 OCB AEAD Scheme 29
2.4 OCB with Message Padding 30
2.5 OCB: *Auth* Generation 30
2.6 Components of an RFID System 33
2.7 Block Diagram of a 16-bit Security Controller 34
2.8 TLS Protocols 36
2.9 Cryptographic Overview of TLS 37
2.10 ePedigree Example 39

3.1 The Secure Channel System 41
3.2 Authentication Protocol 44
3.3 detego SURVEYOR 46
3.4 Web Service Invocation 49
3.5 Physical and Command Frame of the Protocol 51
3.6 DATA Frames for Authentication 51
3.7 DATA Frames for Read and Write Operations 52
3.8 Android Application: Start-Up Screen 55
3.9 Android Application: Main Screen prior to Authentication 55
3.10 Android Application: Main Screen after Successful Authentication 55
3.11 Android Application: Loading Dialog 56
3.12 Android Application: Main Screen after Successful Pedigree-Read 56
3.13 Sequence Diagram of the Authentication Process 58
3.14 Sequence Diagram of the Pedigree-Read Process 60
3.15 Detailed Sequence Diagram of the Read Operation 61
3.16 Detailed Sequence Diagram of the Write Operation 62

4.1 Class Diagram of the Secure Channel Android Application 80
4.2 Methods and Tasks Executed During Authentication 84
4.3 Methods and Tasks Executed During Pedigree-Read 84

5.1 The Secure Channel System 87
5.2 Screenshots of the Secure Channel Application (1/2) 88
5.3 Screenshots of the Secure Channel Application (2/2) 89

5.4 240 Iterations of Successful Authentication Operations 91
5.5 2 000 Iterations of Successful Pedigree-Read Operations 92
5.6 Timings and CPU Clock of Successful Pedigree-Read Operations 93
5.7 Execution of Pedigree-Read 93
5.8 Detailed Execution of Pedigree-Read 94

A.1 TLS Handshake Protocol 105
A.2 Database Relation Diagram for Relevant Entities 106
A.3 Calling Structure of Android Application Tasks and Service Methods 107

List of Tables

3.1 Relevant Database Entities of the detego SURVEYOR Database 47
4.1 Overview of the Methods and Tasks of the CommandHandler Class 83
5.1 Timing Analysis of Authentication and Pedigree-Read 90

List of Listings

3.1 Example SQL Login . . . 48
3.2 Executed SQL Query . . . 48
3.3 Example Injection 1: <PasswordHash> Input: ' OR 1=1 . . . 48
3.4 Example Injection 2: <CommonName> Input: admin'– . . . 48
3.5 Item JSON object . . . 49
4.1 Keygen.cs: GenerateEcKeyPair . . . 64
4.2 Keygen.cs: GenerateRsaKeyPair . . . 64
4.3 Keygen.cs: GetKeysFromFiles . . . 65
4.4 Authentication.cs: ComputeSignatureOfPublicEcKey . . . 66
4.5 Authentication Simulation (Unit Test) . . . 66
4.6 Command Builder.cs: Build . . . 68
4.7 IDataService.cs: Declaration of the Secure Channel Commands . . . 71
4.8 DataService.scv.cs: ScReset . . . 72
4.9 DataService.scv.cs: ScCommandToPropVal . . . 72
4.10 DataService.scv.cs: ScServiceStatus . . . 73
4.11 DataService.scv.cs: ScGetChallenge . . . 73
4.12 DataService.scv.cs: ScVerifyResponse . . . 74
4.13 DataService.scv.cs: ScGetReadCommand . . . 76
4.14 DataService.scv.cs: ScGetNextReadCommand . . . 76
4.15 DataService.scv.cs: ScWrite . . . 78
4.16 DataService.scv.cs: ScFinalize . . . 79
4.17 Overview of the class ConnectionHTTPS . . . 81
4.18 AcceptCert . . . 81
4.19 PostToHttpsWithSelfSignedCertificate . . . 82
B.1 Keygen.cs . . . 109
B.2 Authentication.cs . . . 110
B.3 Montgomery.cs . . . 112
B.4 CommandBuilder.cs . . . 114
B.5 EPedigreeHandler.cs . . . 117
B.6 DataService.scv.cs: ScRead . . . 121
B.7 DataService.scv.cs: ScCheckWrite . . . 124
B.8 Store Keys to File . . . 124
B.9 CommandHandler.java . . . 125

List of Abbreviations

AE	Authenticated Encryption
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
AES-OCB	AES in Offset Codebook Mode
API	Application Programming Interface
CA	Certificate Authority
CBC	Cipher-Block Chaining
DDL	Data Definition Language
DES	Data Encryption Standard
DH	Diffie-Hellman Key Exchange
DHE	Diffie-Hellman Ephemeral
DML	Data Management Language
ECC	Elliptic Curve Cryptography
ECDHE	Elliptic Curve Diffie-Hellman Ephemeral
ECDH	Elliptic Curve Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
EPC	Electronic Product Code
GPS	Global Positioning System
GTIN	Global Trade Item Number
GUI	Graphical User Interface
HTTPS	Hypertext Transfer Protocol Secure
HTTP	Hypertext Transfer Protocol

IIS	Internet Information Services
IMEI	International Mobile Station Equipment Identity
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
LFSR	Linear Feedback Shift Register
MDM	Mobile Device Management
META[:SEC:	]Mobile Energy-Efficient Trustworthy Authentication Systems with Elliptic Curve based Security
NDEF	NFC Data Exchange Format
NIST	National Institute of Standards and Technology
NFC	Near Field Communication
OCB	Offset Codebook
OData	Open Data Protocol
OSI	Open Systems Interconnection
OS	Operating System
PFS	Perfect Forward Secrecy
PKC	Public-Key Cryptography
PKI	Public-Key Infrastructure
RC4	Rivest Cipher 4
RDBMS	Relational Database Management System
RFID	Radio-Frequency Identification
RRP	Recommended Retail Price
RSA	Rivest-Shamir-Adleman
SHA	Secure Hash Algorithm
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSL	Secure Sockets Layer
TLS	Transport Layer Security
URL	Uniform Resource Locator

VPN	Virtual Private Network
WCF	Windows Communication Foundation
WSDL	Web Services Description Language
XML	Extensible Markup Language
XSS	Cross-Site Scripting

Chapter 1

Introduction

In today's economy, every strong brand sees itself confronted with product counterfeit. In every line of business, may it be electronics, fashion industry or even consumer goods, products are replicated and sold as originals by third parties. There are obvious counterfeits, sold in backstreet market stands that are cheap compared to the original products and show quality flaws. These are bought on purpose, since the customer knows that he is getting faked goods. On the other hand, there are also well produced forgeries, placed in real stores that barely differ from the original product. Customers hardly have any chance to identify those counterfeits and therefore may pay the original price for fake goods. Fortunately, with new technologies, new solutions arise: This thesis proposes a way of giving customers the chance to distinguish original products from counterfeits, while giving the producer extended information on his product distribution and possible counterfeit hot spots.

In the last few years, Near Field Communication (NFC) is gaining momentum all over the world. This technology allows communication with actively and passively powered Radio-Frequency Identification (RFID) tags. A wide range of consumer focused applications, like payment, data transfer and device pairing brought many smartphone producers to integrate NFC into their flagship models. Using these NFC-capabilities, the proposed system can be used to identify passive RFID tags that are attached to original products. Therefore, authenticating the tag means authenticating the product. In combination with an intelligent backend system, to process gathered information, the proposed system allows for secure authentication and communication with RFID tags.

As part of the *META[:SEC:]*[1] research project (Mobile Energy-efficient Trustworthy Authentication Systems with Elliptic Curve based Security), this thesis demonstrates a secure communication between a backend server and a secure cryptocontroller on an RFID tag, using the wireless technologies of a smartphone as bridge technology. Figure 1.1 depicts an overview of the proposed *Secure Channel System*. As RFID tag, a security-enhanced cryptocontroller prototype from *Infineon Technologies* is used. The smartphone is operating on *Android* OS and communicating with the tag over NFC. The backend server is based on an existing backend solution from *Enso Detego* and communicates with the

[1]"META[:SEC:] : Mobile Energy-efficient Trustworthy Authentication Systems with Elliptic Curve based SECurity", funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 829586. The META[:SEC:] consortium consists of TU-Graz, Infineon Technologies Austria AG and Enso Detego GmbH.

smartphone over mobile internet or Wi-Fi. With attention to an open Android application, which users can download and install on their smartphones, the application itself may be counterfeited, and modified or malicious versions may be distributed. Therefore, it is important to minimize possible information leakage on the smartphone, which means that the application should not be able to leak any sensitive data from the communication between backend and tag. Thus, all sensitive data sent over the communication channel is encrypted. This secure channel enables private data transmission between backend and tag, while the Android application is not able to gather any information from the encrypted messages. The backend connection is additionally secured using HTTPS, which is the secure version of HTTP together with Transport Layer Security (TLS).

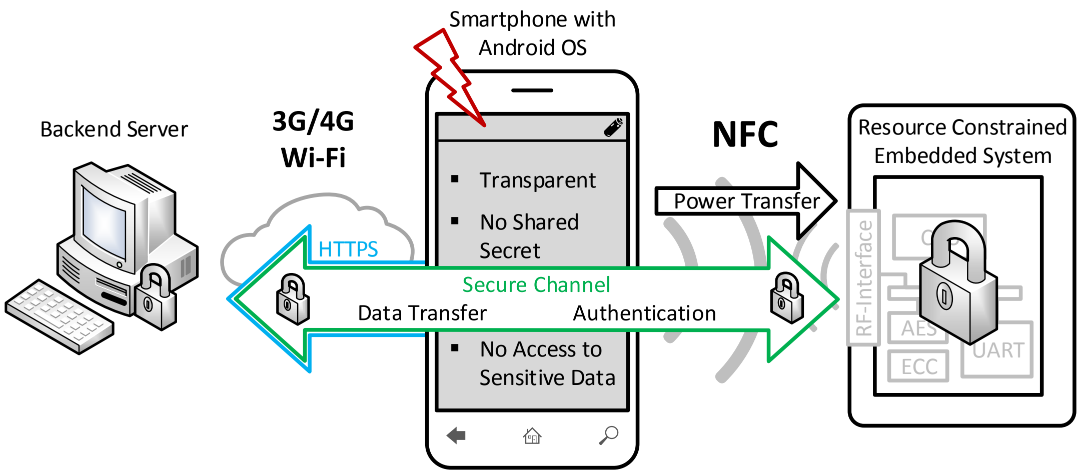


Figure 1.1: Overview of the Secure Channel System

The goal of this thesis is to create a secure communication between a backend server and an RFID tag, using a smartphone as bridge technology. With focus on encrypting sensitive information sent over the secure channel, tag authentication as well as encrypted tag-data transmission is achieved. The smartphone user can authenticate the tag to a secure backend and read data that is stored on the tag. Only insensitive data is presented to the smartphone, therefore mitigating possible attacks on the application. Authentication is realized using Montgomery Multiplication and signatures based on Elliptic Curve Cryptography (ECC). Sensitive data is encrypted with AES in Offset Codebook Mode (AES-OCB). This Authenticated Encryption (AE) scheme allows for symmetric encryption of data with an additional hash tag to detect modified or erroneous messages upon decryption.

Overview: Chapter 2 discusses related work. In Chapter 3 the design of the project is described: This chapter is divided into a first conceptual overview, the design of the three main parts and details on the workflow of the entire system. Followed by the implementation in Chapter 4, where the discussed implementation is split into a software simulation, a backend server and a smartphone section. The results are detailed in Chapter 5. Finally, a conclusion ans possible future work can be found in Chapter 6.

Chapter 2

Related Work

This chapter gives background information on the involved technologies, related standards and systems similar to this thesis.

2.1 Cryptography Fundamentals

The basic problem behind a secure communication between two parties is to prevent a third party to intercept, eavesdrop or modify data related to the communication. In the literature the two communicating parties are often called Alice and Bob, while the malicious third party is called Eve. In a modern cryptosystem, the message is encrypted by Alice with the encryption key before sending, and decrypted by Bob with the corresponding decryption key upon receiving. This way the plaintext P becomes ciphertext C during the communication and gets decrypted to P again (see Figure 2.1). Kerckhoff's principle states that the security level of a particular system is mainly determined by the used key. The entire system, except for the key, can be public knowledge [Ker83]. Modern cryptography can be divided into two major families: symmetric and asymmetric cryptography. Symmetric cryptography refers to cryptosystems where all keys are known by Alice and Bob alone. Thus, the sending and receiving sides can be interchanged. Asymmetric cryptography, however, refers to systems that use public and private (secret) keys. Every party has its own public-private keypair. The private key is only known by its owner. The public key is distributed to possible communication parties publicly, where every communication entity uses different keys. Alice encrypts the message with Bobs public key, Bob decrypts the message with his private key.

2.1.1 Symmetric Cryptography

In symmetric cryptography the communication is secured by encrypting and decrypting messages with the same key. This key has to be known by all communicating parties, thus representing a shared secret. Symmetric ciphers can be divided into two main fractions: stream ciphers that encrypt the message bit-wise one after another, and block ciphers that take a number of bits (block) and encrypt them as a single unit.

A stream cipher can be interpreted as a finite state machine, where the key determines the initial state. This internal state is stored in an Linear Feedback Shift Register (LFSR). The message is then encrypted bit-by-bit and the internal state changes depending on the

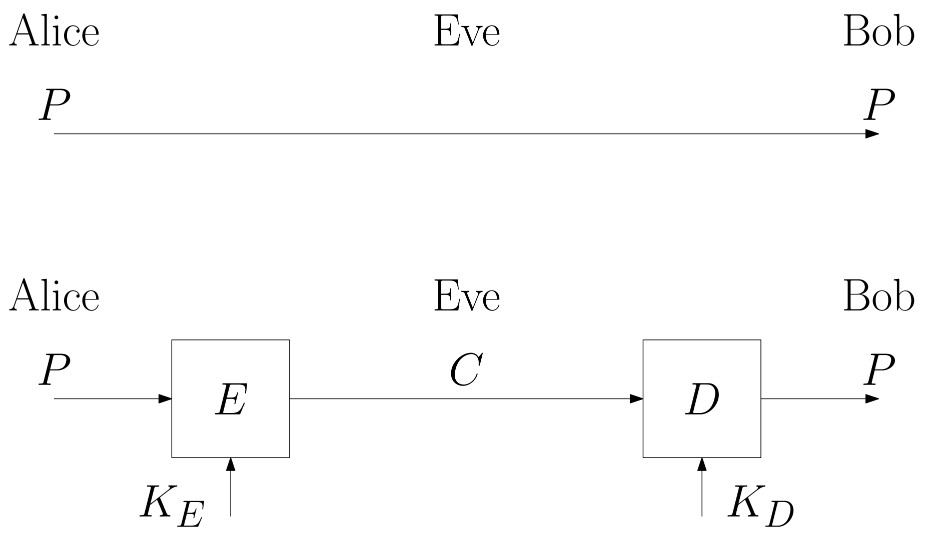


Figure 2.1: Secure Communication Realized With Encryption
(adapted from [Rij11])

encrypted message, directly affecting the subsequent parts. Already encrypted bits can be sent to the communicating party and can be decrypted using the same key. Thus, one of the key features of a stream cipher is the fast communication. Additionally, stream ciphers have a low implementation complexity, as they can perform well on low-energy constrained systems. Commonly known stream ciphers include RC4 [KT99], AC5/1 [BSW01] and SNOW 3G [EJ03].

Block ciphers always operate on a fixed length of bits called blocks. The plaintext block gets encrypted to a ciphertext block in multiple rounds of bit substitutions. Small non-linear functions, called S-boxes, substitute parts of the block, typically by using a lookup table. Additionally, the substituted parts are mixed and parts of the symmetric key are changing the intermediate values in every round. Already encrypted blocks can be sent to the communicating party, where they can be decrypted by using the same block cipher with the same key. For decryption, only the order of the used key parts changes. Well known block ciphers include the Data Encryption Standard (DES) [Sta99] and any derived form like 3-DES and DESX. The most prominent block cipher today is the Advanced Encryption Standard (AES) [DR98]. A block cipher by itself allows only encryption of a single plaintext block. For a variable message-length, the plaintext has to be partitioned into separate blocks. So called *Modes of Operation* describe how the separate blocks are encrypted and how the encryption of one block affects the next block.

2.1.2 Asymmetric Cryptography

Asymmetric cryptography, also called Public-Key Cryptography (PKC), refers to the use of public and private keys in a cryptosystem. The public-private keypair is different for every communication party. Messages are encrypted with the public key of the recipient, whereas the private key is only known by the owner. The receiving entity then decrypts the ciphertext with the corresponding private key. The notion of PKC was proposed by Diffie and Hellman in 1976 [DH76]. In a public key system, it is computationally infeasible to calculate the private key (also called secret key) from the public key. In their paper they proposed the Diffie-Hellman Key Exchange (DH) protocol, a widely used solution for key exchange and key establishment. In 1978 the first practicable public key system, the RSA algorithm, was invented by Rivest, Shamir and Adleman. Public-key algorithms are based on mathematical problems which currently have no efficient solution, such as integer factorisation, discrete logarithm or elliptic curve relationships. For RSA the difficulty is based on factoring the product of two large prime numbers. As computational performance increases, this factoring gets feasible for small keys. Therefore the bit length of RSA keys is significant for the security of the cryptosystem. The largest RSA modulus that has been factored so far was 768 bit by Kleinjung *et al.* [KAF+10]. Current recommendations pledge for bit sizes of 2048 bit and above [fSidIG14]. A popular algorithm, based on DH and the discrete logarithm problem, is the ElGamal encryption system, proposed by Elgamal in 1985 [ElG85].

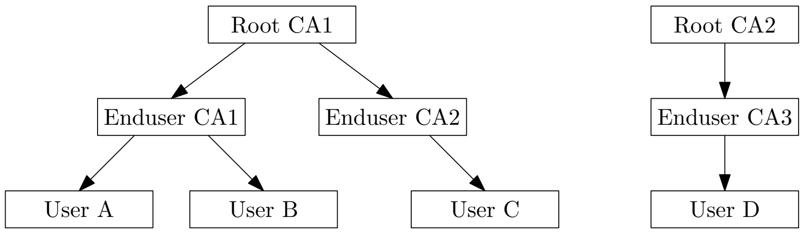


Figure 2.2: Structure of the X.509 Standard for Generating Digital Certificates
(adapted from [Lam10, pp. 15])

Digital Certificates: Digital certificates bind an identity to a certain public key [OL10, pp. 55–63]. Certificates contain the name and the public key of the subject, the name of the certificate issuer and an expiration date, along with additional information. The certificate is digitally signed by a trusted third party (issuer). This third party is called the *Certificate Authority (CA)*. Each CA also has a certificate signed by another CA. At the top of the certificate-tree is the root CA, which has a self-signed certificate. This self-signed certificate can be generated by everyone. There is no third party to validate the self-signed certificate, therefore it is untrusted. The whole system relies on trusting certain root CAs. Figure 2.2 shows the tree structure of the X.509 standard [AFKM05] Public-Key Infrastructure (PKI) for generating digital certificates.

PFS: Perfect Forward Secrecy (PFS) is a property of key-establishment protocols. The key establishment is the phase where a shared secret becomes available to the communicating parties [Rij11]. Some protocols generate unique session keys for every communication established. These session keys can be derived from not-changing long-term keys. PFS requires that leakage of such a long-term key does not compromise session keys used in the past [Rij11]. In relation to SSL/TLS, many big companies use long-term keys, also called *master keys*, to generate session keys for TLS connections with users. Not all master-key systems provide PFS. To offer PFS with certainty, the connection handshake has to use completely different keys for every session, which must not be derived from the same master key.

Hash Function: A hash function is an algorithm that maps an input message of arbitrary length to an output (hash-value or simply hash) of fixed length [MVOV10]. Identical inputs always have the same hash-value. The basic structure of a hash function shows that there have to be different inputs resulting in the same hash-value. These *collisions* are the main weakness of every hash function. Therefore commonly used hash functions have properties that make such collisions *hard to find*. The hash itself is a representative fingerprint of the input message. Slight changes in the input result in completely different hash-values. If the hash-value is sent together with the input message, the recipient can hash the message again (with the same hash function) and compare the new value with the sent hash-value. If the message has been modified, the values differ.

2.1.3 Authenticated Encryption

Security systems often need more than encryption. In the encrypted state the message content follows no clear schema. The decrypting side has to verify the integrity of the sent message before decrypting it. Thus, appending a hash value to the message makes a quick validation possible. Encryption as well as generating a hash value are both performance and time consuming operations. Processing both operations one after another is not the best solution. Thus, schemes that calculate the hash value while encrypting the original message at the same time are called *authenticated encryption* schemes. If an authenticated-encryption scheme allows for the additional authentication of unencrypted data, the scheme is an Authenticated Encryption with Associated Data (AEAD) scheme. As there is no standard for an authenticated encryption scheme, an ongoing cryptographic competition called CAESAR [D. 14] tries to determine standards for authenticated ciphers. One the CAESAR submissions, AES in Offset Codebook Mode (AES-OCB) [Phi14], is used as AEAD in this thesis, which provides the security of AES encryption while generating a hash tag to confirm the message integrity as well.

2.1.4 Offset Codebook

Offset Codebook (OCB) is a block-cipher mode of operation and an Authenticated Encryption with Associated Data (AEAD) scheme invented by Rogaway [Phi14] and specified in 2014 [KR14]. Rogaway first filed a patent for OCB in 2000, therefore, any commercial use will need a licence. OCB currently competes against other authenticated encryption schemes in the CAESAR cryptographic competition, where the goal is to define standards for authenticated ciphers.

OCB provides two security properties: confidentiality and authenticity. OCB outputs are confidential, or private, as an adversary is unable to distinguish them from an equal number of random bits. If an adversary is unable to create any nonce-ciphertext pair that he has not already acquired, one speaks of authenticity.

Figure 2.3 depicts the basic structure of an OCB encryption. A blockcipher E with an n-bit block-length and the key K are used. The message M is split into m blocks of n bits each. A unique nonce is denoted by N. The Δ value is different for each message block and generated with an increment function. The initial Δ is generated from N via an initialisation function. The value *Checksum* is the XOR product of all message blocks ($M_1 \oplus M_2 \oplus M_3 \oplus .. \oplus M_m$).

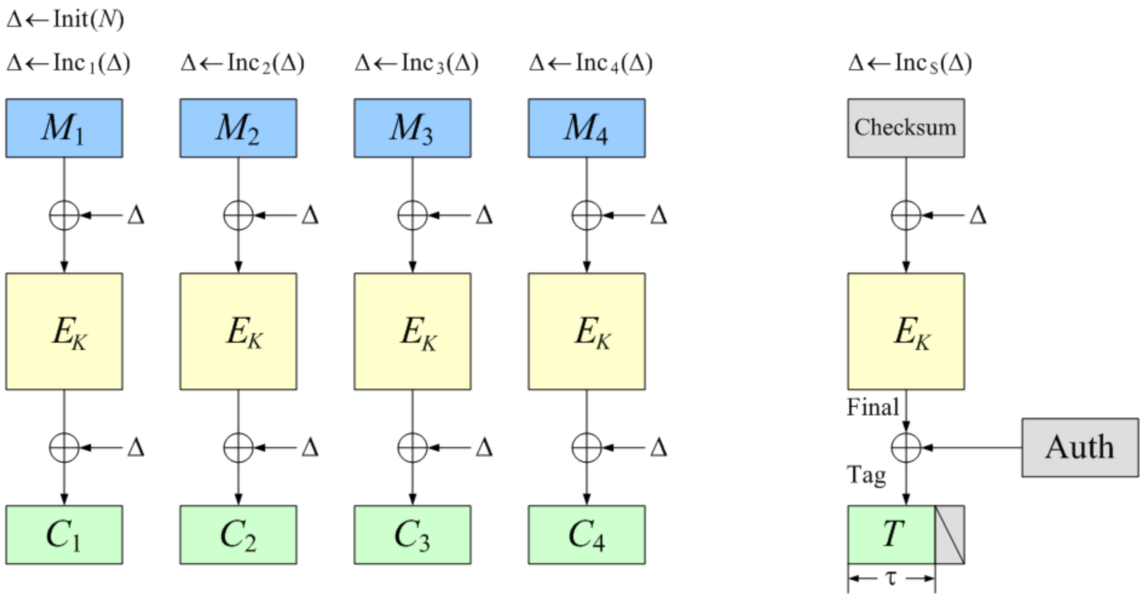


Figure 2.3: Offset Codebook (OCB) AEAD Scheme
(adapted from [Phi14])

During encryption, each message block M_i gets XORed with Δ, encrypted by the blockcipher E_K and XORed again with Δ to generate a cipher block C_i. The *Checksum* is also XORed with Δ and encrypted with E_K, but then XORed with *Auth*, to generate an internal tag of n bits. The first τ bits of the internal tag are appended to the cipher-text blocks as tag, or hash tag, T, thus τ is a number between zero and 128. Like the blockcipher, τ is a parameter of the mode.

If the message length is not a multiple of the block length n, the last message block M_* is zero-padded and the ciphertext is generated by XORing an encrypted Δ to the message block (see Figure 2.4).

The generation of the *Auth* value is shown in Figure 2.5. Here, the associated data is split into blocks A_i of n-bit length. Each block is XORed with Δ and encrypted with the blockcipher. The blocks are then XORed together, in order to generate the *Auth* value. Associated data is padded to be a multiple of the block length n by adding a "1"-bit followed by zero-padding.

The input of an OCB encryption is the message $(M_1||M_2||M_3||..||M_*)$, whereas the output ciphertext consists of the ciphertext blocks and the tag $(C_1||C_2||C_3||..||C_*||T)$. Therefore, the output is τ bits longer then the input. Decryption works similar to encryption, except for the blockcipher that is now used to decrypt the messages (D_K).

For each OCB encryption, a distinct nonce of up to 120 bits has to be used. For decryption, the same nonce is needed, thus creating a nonce-ciphertext pair that is only valid together. This nonce must not be repeated for multiple encryptions, otherwise the confidentiality and authenticity properties are undermined, as an attacker will be able to infer relationships between observed ciphertexts. Therefore, nonces need to be unique for each individual encryption. However, the nonces need not to be secret, which enables the use of a counter or timestamp.

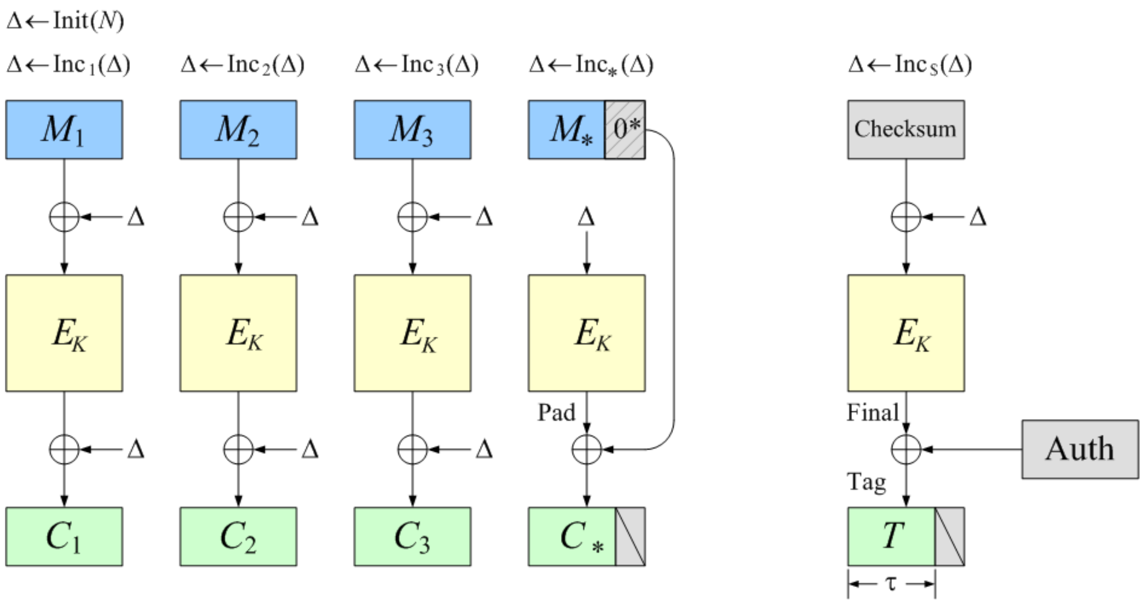


Figure 2.4: Offset Codebook (OCB) AEAD Scheme with Message Padding (adapted from [Phi14])

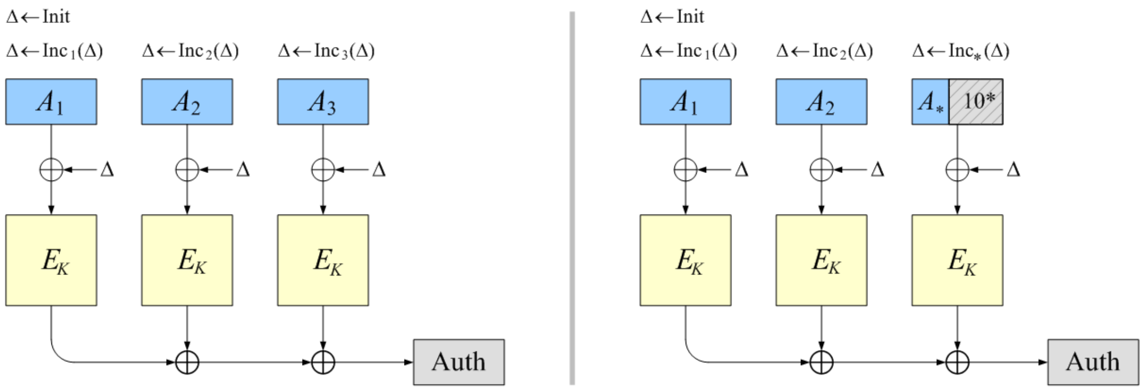


Figure 2.5: Offset Codebook (OCB) AEAD Scheme: Generation of *Auth* Value (adapted from [Phi14])

The used OCB global parameter set is *AEAD_AES_256_OCB_TAGLEN128*, which means AES with a 256-bit key is used as the blockcipher and the tag length is the full 128 bit of the internal tag. The current timestamp is used as associated data and as nonce.

2.1.5 Montgomery Multiplication

Many public-key cryptosystems are based on modular arithmetic: RSA requires exponentiation mod n, Diffie-Hellman and ElGamal are based on exponentiation modulus a prime and ECC can be implemented on a prime field. Introduced in 1985 by Peter Montgomery [Mon85], Montgomery multiplication is a method for computing modular multiplication efficiently (*i.e.* $a \cdot b \bmod p$) [HVM04]. In a modular multiplication, the computationally most expensive part is the reduction mod p, which is equivalent to a division. This is rather inefficient when working with large moduli like 2048 bit RSA. Montgomery multiplication works by transforming the factors a and b to a so called *Montgomery domain*, simplifying the division to a subtraction, at the cost of additional multiplications for the transformation.

Thus, the basic structure of calculating $c = a \cdot b \bmod p$ using the Montgomery multiplication is [WEW12]:

1. transform a and b to the Montgomery domain: $\overline{a}$ and $\overline{b}$
2. calculate $\overline{c} = \overline{a} \star \overline{b}$, where $\star$ denotes the Montgomery multiplication step
3. transform $\overline{c}$ back to c

To transform a, with a modulus p, a constant value $R = 2^k$ ($R{>}p$, $\gcd(R, p) = 1$) is needed, where k is the number of bits of p. Modular reduction by R can now be implemented by truncating (or masking) to k bits, and a division by R is a right-shift by k bits.

$$\overline{a} \equiv a \cdot R \; mod \; p$$

As the $\star$ operation is used to calculate $\overline{c} = \overline{a} \star \overline{b}$, this means

$$(a \cdot R) \star (b \cdot R) \equiv c \cdot R \equiv (a \cdot b) \cdot R \; mod \; p$$

$$\overline{a} \star \overline{b} := \overline{a} \cdot \overline{b} \cdot R^{-1} \equiv a \cdot R \cdot b \cdot R \cdot R^{-1} = a \cdot b \cdot R \equiv \overline{c} \; mod \; p$$

Whereas the value $d = \overline{a} \cdot \overline{b}$ is easy to calculate, the result $d \cdot R^{-1} \; mod \; p$ is somewhat trickier. This step is referred to as the *Montgomery reduction.* The Montgomery reduction of $r = d \cdot R^{-1} \; mod \; p$ is calculated using the following formula:

$$s = (d + (d \cdot p' \; mod \; R) \cdot p)/R, \qquad r = \begin{cases} s - p & s \geq p \\ s & \text{else} \end{cases}$$

where $p' = -p^{-1} \; mod \; R$. Additionally, the $\star$ operation can also be used to simplify the Montgomery transformation and back-transformation:

$$\overline{a} = a \cdot R \equiv a \cdot R \cdot R \cdot R^{-1} = a \cdot R^2 \cdot R^{-1} = a \star R^2 \; mod \; p$$

$$c = \overline{c} \cdot R^{-1} = \overline{c} \cdot 1 \cdot R^{-1} = \overline{c} \star 1 \; mod \; p$$

Therefore, the two implemented algorithms for calculating the Montgomery multiplication are:

Algorithm 1 MONT(x, y) to calculate $x \star y = x \cdot y \cdot R^{-1}\ mod\ p$

```
1: $d = x \cdot y$
2: $r = (d + (d \cdot p'\ mod\ R) \cdot p)/R$
3: if $r \geq p$ then
4:     $r = r - p$
```

Algorithm 2 Calculate $c = a \cdot b\ mod\ p$ using Montgomery multiplication

```
1: $\overline{a} = \text{MONT}(a, R^2)$
2: $\overline{b} = \text{MONT}(b, R^2)$
3: $\overline{c} = \text{MONT}(\overline{a}, \overline{b})$
4: $c = \text{MONT}(\overline{c}, 1)$
```

2.2 Secure RFID

This section describes the two used technologies Radio-Frequency Identification (RFID) and Near Field Communication (NFC) and how such systems can be secured[1].

2.2.1 RFID

RFID uses electromagnetic and magnetic fields for wireless data transmission. A small electronic device (chip) equipped with an antenna (together: RFID tag) is powered by an electromagnetic field from an RFID reader (see Figure 2.6). The range for such an over-the-air communication with passive tags is limited to a few meters. In contrast, active tags are equipped with a dedicated energy source, which enables communication over higher distances. There exists a wide variety of different RFID systems. One possible categorization can be performed according to the utilized frequency range. The HF-band (high frequency, 13.56 MHz) can operate at distances up to one meter. HF tags are used in smart cards and for NFC. The UHF-band (ultra high frequency, around 850 to 950 MHz) offers reading distances up to twelve meters [Wei11]. Passive UHF tags are widely used for item-level tagging and thus support or replace barcode systems.

Passive RFID tags use modulation of the reflected power of the tag antenna to communicate with the reader [Dob07]. This reflection of waves is called backscattering. Basically, current flowing in the reader antenna is inducing a voltage on the tag antenna. If the tag antenna is connected to a load, this leads to current flowing in the tag antenna which then radiates back to the reader antenna. This radiated wave induces a voltage on the reader antenna and therefore a backscattered signal can be detected. This backscattered signal is depending on the load on the tag antenna.

2.2.2 NFC

NFC typically refers to RFID communications between smartphones and passive HF tags or between two smartphones. Thus, the communication between a smartcard (HF tag) and a reader,*e.g.* for payment, is called NFC. The technology enables every NFC-capable smartphone to work as an HF RFID reader. As it is used more and more as a payment

[1]Parts of this section are adapted from a preceding master's project [Ple14]

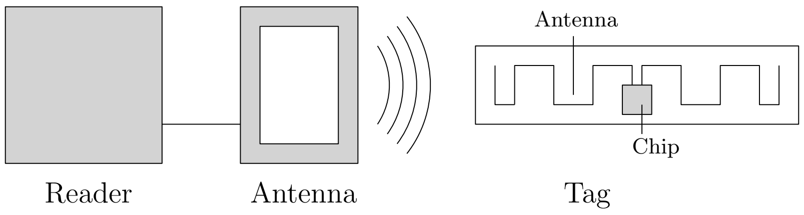


Figure 2.6: Components of an RFID System

option, ensuring a secure communication is very important. Moreover, NFC technology is also used to setup other wireless communications, such as Wireless LAN or Bluetooth.

2.2.3 ECC

Elliptic Curve Cryptography (ECC) is very popular nowadays. In comparison to other algorithms, ECC offers the same level of security with considerably shorter key lengths. In the last years extended effort has been taken to bring ECC to low-cost RFID devices: Wolkerstorfer [Wol04] presented one of the first low-area ECC implementations with $\mathbb{F}_{2^m}$ and $\mathbb{F}_p$ standardized curves. Further implementations [KP06, BGK$^+$06, FW07] showed ECC processors suitable for RFID devices. Ahamed *et al.* [ARH08] described an offline mutual authentication protocol based on ECC. Protocols resistant to currently known attacks and threats have been presented in [Chi07, SM08]. A rather simple technique for inexpensive untraceable identification for RFID is presented by Tsudik [Tsu06]. Existing protocols are evaluated for RFID suitability in [BGK$^+$07, LW07]. Feldhofer *et al.* presented a low-resource design and implementation of ECDSA for RFID [HFP10, WFF11, KF10]. Urien and Piramuthu [UP13] showed a framework and authentication protocol for smartphones, NFC and RFID for retail transactions.

A description of an implementation of elliptic curves over finite fields for RFID has been presented by Braun *et al.* [BHM08]. Based on the presented implementation and authentication protocol, Bock *et al.* [BBD$^+$08] showed an energy efficient hardware implementation resistant against side-channel attacks. The energy consumption of their 4-bit architecture, during ECC computation, is stated to be 7.5 μJ with an area size of 13 kGE for the ECC hardware module.

2.2.4 RFID-Tag Prototype

The RFID tag used in this thesis is a prototype security cryptocontroller developed by Infineon Technologies [Inf14]. The prototype is similar to the 16-bit Security Controller that is shown in Figure 2.7. However, the used version is modified to fit the purposes of this thesis. The tag communicates with the smartphone over NFC, using the NFC Data Exchange Format (NDEF). Multiple security layers prevent execution of unauthorized commands, while encrypting and decrypting all messages with the chosen AES-OCB mode.

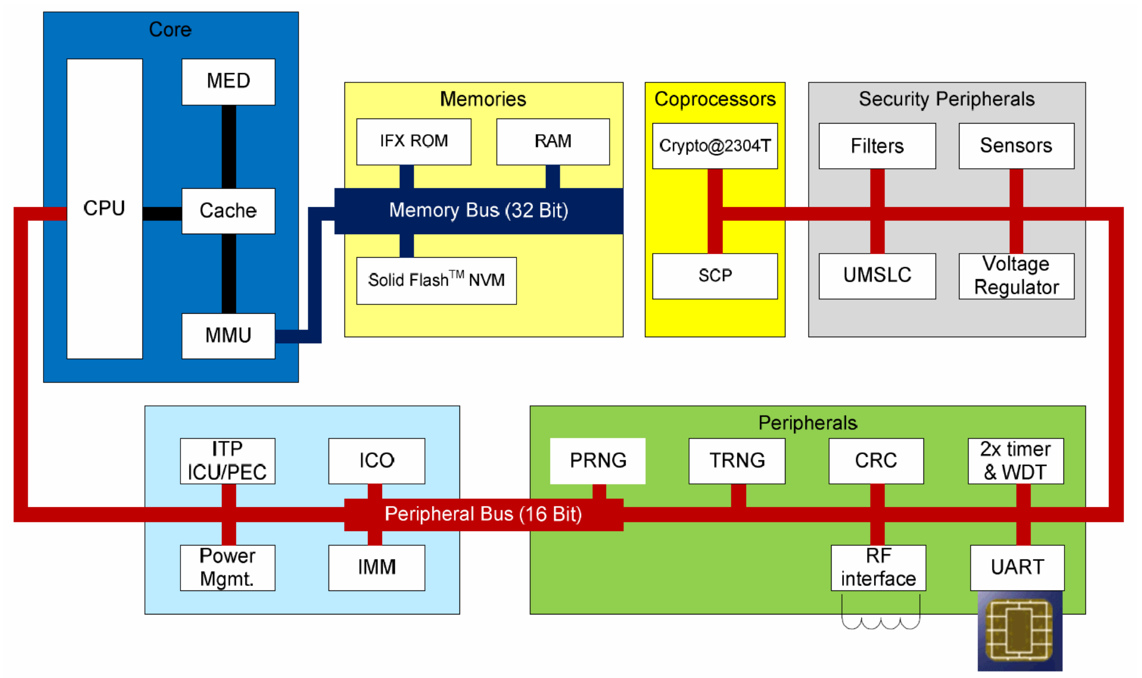


Figure 2.7: Block Diagram of a 16-bit Security Controller

2.3 Secure Backend

To evaluate the server security there are several security critical components such as the web applications, the web server itself and the database, whereas the underlying technologies are Microsoft Internet Information Services (IIS) [Mic13] as the web server and Microsoft SQL Server 2012 [Mic12a] as database server, respectively[2].

IIS: IIS is Microsoft's web server for hosting web applications in a Windows environment. A survey from Netcraft in September 2013 [Neta] showed that the IIS is the second most popular web server in the world. Versions earlier to 6.0 had many security vulnerabilities [SPW$^+$02], where one of them resulted into the so called Code Red Worm [CER02]. With the current global surveillance disclosures [Wik13] the focus on PFS increases constantly. With PFS, a session key derived from a long term key is not retrievable, even if the long term key gets compromised. As stated by Netcraft [Netb], only the use of Diffie-Hellman Ephemeral (DHE) or Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) ensures PFS.

Although the IIS became more secure with the latest releases, misconfiguration can still lead to critical exposure. Microsoft TechNet gives a deeper look into configuration details and best practices for securing the IIS [Tecc, Teca].

SQL Server: Microsoft SQL Server 2012 is a relational database management platform. The primary query language is Transact-SQL [Tecb] which is a extension to the Structured Query Language (SQL).

[2]Parts of this section are adapted from a preceding master's project [Ple14]

Every SQL Server is potentially vulnerable to SQL Injections [Mor06]. If SQL code uses unchecked input from external sources such as web forms, command line inputs or cookies, it is possible to inject malicious code by changing the interpreted code [SG10, pp. 15]. This can lead to possible unauthorized access to the database, manipulation of data in the database or denial-of-service from deleted data. Kumar presents a *"survey on SQL injection attacks, detection and prevention techniques"* [KP12]. SQL injections in RFID backend-systems are analysed in [JJ11].

XSS: Cross-Site Scripting (XSS) is the injection of code or malicious scripts into the victims content over browser input [Hei11]. There are two basic types, reflective and persistent XSS. Reflective XSS is based on a request-response strategy, where the attacker injects code and gets the desired response. Persistent XSS, however, defines permanent injection of code on the server itself. The main target here are site users that log on to the server.

2.3.1 Backend-Connection with TLS

There are several ways to secure a connection between a smartphone and a backend server. The best practice is to use Transport Layer Security (TLS).

SSL/TLS: Secure Sockets Layer (SSL) and its successor TLS are cryptographic protocols on the transport layer level of the Open Systems Interconnection (OSI) model [Sta96]. The SSL protocol [HE95] was developed by Netscape in 1994. The first version has never been released to the public. In 1995, SSL Version 2.0 [Hic95] was released, which still contained some security flaws, and therefore has been replaced with a complete redesign to SSL Version 3.0 [FKK11]. TLS 1.0 [DA99] has been designed in 1999 as an upgrade to SSL 3.0. Both protocols are similar, however not interoperable. An optional downgrade from TLS to SSL was possible. In 2006 TLS 1.1 [DR06] has been introduced, followed by the current version TLS 1.2 [DR08]. In 2011 the TLS protocols were refined to not support downgrading to SSL 2.0 any longer [TP11]. The result of layering HTTP on top of SSL/TLS is Hypertext Transfer Protocol Secure (HTTPS).

TLS provides privacy and data integrity between two communicating applications. It consists of two main parts: the *TLS Record Protocol* and the *TLS Handshake Protocol*. Figure 2.8 shows the TLS protocols in detail. The TLS Record Protocol provides privacy and integrity for the connection using symmetric cryptography to encrypt the data, where the corresponding keys are generated uniquely for each connection and are hence negotiated during the handshake. To establish a reliable connection, the TLS Record Protocol also adds a hash value to the message. The TLS Handshake Protocol is used for authentication on each side and key negotiation for encryption in the Record Protocol.

During the establishment of a TLS connection, both sides have to negotiate about the used *cipher suite*. This cipher suite defines how the key exchange will be executed and which signature algorithm, encryption and hash algorithm is going to be used. For instance the cipher suite denoted by `TLS_RSA_WITH_RC4_128_SHA` will use RSA [RSA78] for the key exchange, 128-bit RC4 [KT99] for encryption and SHA [rJ01] as a hash function [Int]. A cryptographic overview of TLS is given in Figure 2.9.

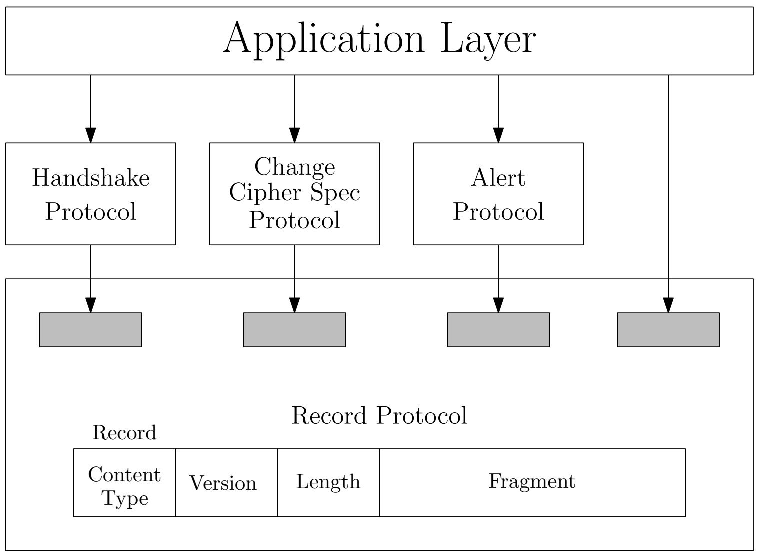


Figure 2.8: A Detailed View of the TLS Protocols
(adapted from [JQ13])

To authenticate server and client, certificates are exchanged. Each side has to validate the opposites certificate on their own. The TLS Handshake Protocol covers the following five steps [DR08]:

1. Exchange hello messages and agree on a specific cipher suite.

2. Exchange cryptographic parameters necessary for the specified cipher suites.

3. Exchange certificates to check authentication.

4. Generate a master key with specified key exchange.

5. Allow each side to check for modifications by adversaries.

If any of these steps fails, the handshake fails and the connection is dropped. A more detailed view of the handshake communication is depicted in Figure A.1. The actual session begins after the initial handshake. During the session, both sides encrypt, decrypt and validate sent messages with the session key. At any time a renegotiation can be started, which initiates the handshake again.

2.4 Android Operating System

In this thesis, smartphones are used as bridging technology between passive RFID tags and a database oriented backend system. From this viewpoint, two secure connections are required: the connection between smartphone and RFID tag and the connection between smartphone and the backend server.

SSL/TLS			
Handshake		Record	
Authentication, Certificates		Privacy	Integrity
Asymmetric Cryptography	Other	Symmetric Cryptography	Hash Algortihms
RSA	PSK	DES	MD5
DH	Kerberos	AES	SHA
DHE	SRP	IDEA	SHA256
ECDH	NULL	RC4	SHA386
ECDHE		NULL	NULL

Figure 2.9: Cryptographic Overview of TLS

NFC for Android: Nowadays many new smartphones provide NFC features. For Android devices with NFC capabilities there are three modes of operation: reader mode, peer-to-peer mode and card emulation mode [Goo13a]. In reader mode, devices can read and write data from and to an NFC tag. The peer-to-peer mode allows two Android devices to communicate with each other. Finally, the card emulation mode enables the device to act as an NFC tag, which then can be read by an NFC reader. For writing data to a tag, most Android frameworks use the NDEF[Nok13].

SSL and Android: It seems that while enabling SSL for an Android application is quite achievable, having a secure configuration is not trivial. Fahl *et al.* analysed the SSL security situation of popular Android applications [FHM$^+$12]. Most of the configuration problems found are related to certificate handling. Trusting all certificates or allowing all hostnames (not just the address the certificate was issued for) are the most common problems. Android Developers provide tips and strategies to avoid such pitfalls in certificate handling [Goo13b].

2.5 ePedigree Standard

The ePedigree, or electronic pedigree, is a certified record that contains the history of a pharmaceutical drug [EPC]. It holds information about each distribution of a prescription drug. It records the initial sale by a manufacturer, any acquisitions or sales by wholesalers or repackagers, and the final sale to a pharmacy or other entity administering or dispensing the drug. It also contains information concerning the product, transaction, distributor, recipient and signatures to verify the individual entities. The pedigree, as it is is described here, is required by law in the United States. The certificates used for signing pedigrees have to be X.509 certificates [AFKM05].

A simplified pedigree process contains the following steps:

- Create pedigree
- Add information to pedigree
- Digitally sign pedigree
- Send pedigrees for products in shipments to customer
- Receive pedigrees
- Authenticate pedigrees
- Verify products against authenticated pedigrees
- Sign pedigree for receipt and authentication

The ePedigree format is typically an XML envelope schema, where the document holds all current and previous pedigree versions. Any new data is simply appended to the existing document, and everything is signed. Figure 2.10 shows an example of such a nested ePedigree. Further details on the pedigree standard are given in [EPC].

2.6 Similar Systems

In Section 2.2 many RFID authentication-protocols are described. These protocols secure communication between tag and reader. The connection between reader and backend is assumed to be secure most of the time. An RFID system that authenticates with a untrustworthy backend server in a secure way is shown in [XZZT13], which is claimed to be the first to address the unsecured-backend problem.

There exist systems that are very similar to this thesis in therms of the chosen counterfeit use case, described in more detail in Section 3.1.2. The internet security company WISeKey [WIS14] offers a product called WISeAuthentic that provides product authentication and sales management as well as marketing possibilities. Similar to the use case presented in this work, the RFID reader is an NFC-capable smartphone with a provided application, where tag / product authentication is done by simply is done by simply reading the tag.

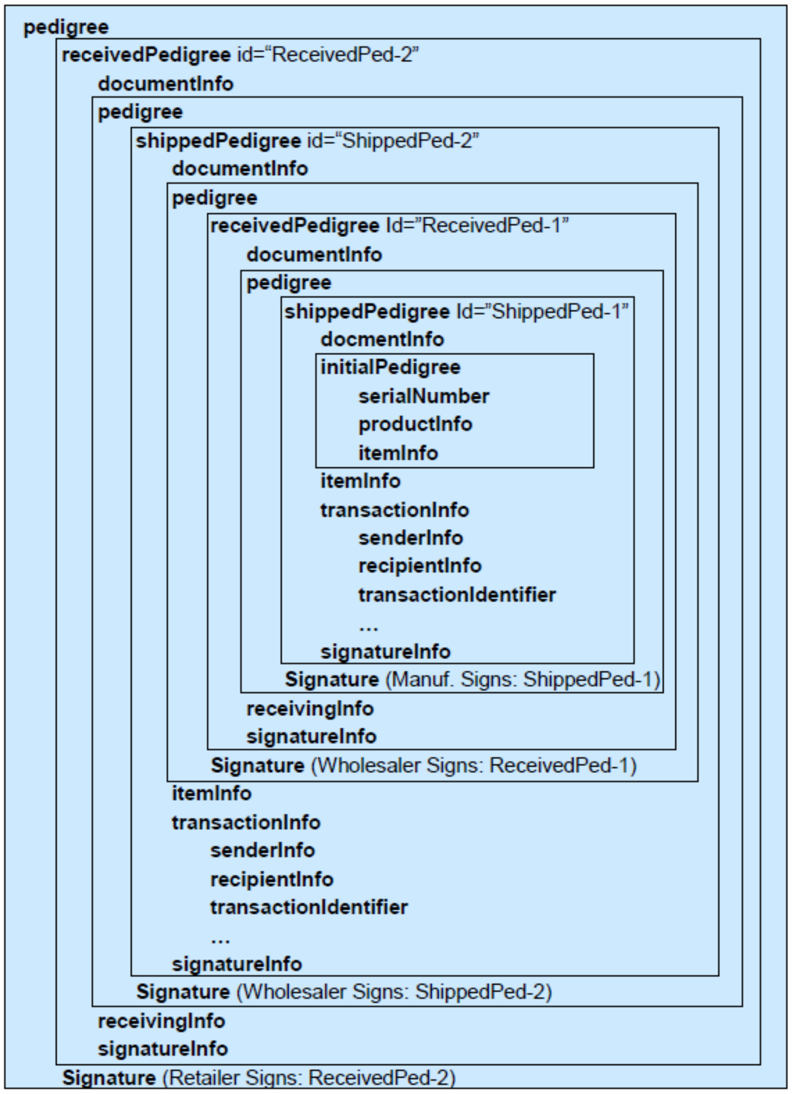


Figure 2.10: Example of an ePedigree
(adapted from [EPC])

Chapter 3

Design

3.1 The Secure Channel System

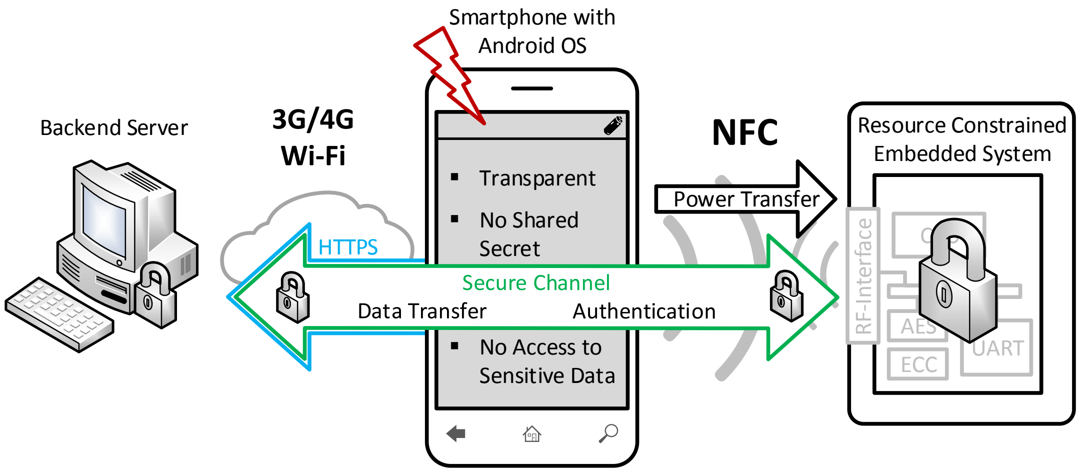


Figure 3.1: Overview of the Secure Channel System

The *Secure Channel System* consists of three main parts:

1. **Tag**
   A security-enhanced RFID-tag prototype from Infineon Technologies [Inf14]. The tag is the entity that gets authenticated by the system. In the anti-counterfeit use case, the product is equipped with an RFID tag to enable a secured authentication. In addition, the tag stores additional information about the product in a separate memory bank.

2. **Backend**
   A backend system that provides data through a database and handles calls over a secure web service. The backend *knows* the authentic tags.

3. **Smartphone**
   An NFC-enabled mobile device running Android OS. The smartphone is used as a bridge technology for communication between backend and tag. All communication between those two endpoints is encrypted and thus unreadable for the Android device. Throughout this thesis the therm "**application**" always refers to the Android application running on the smartphone.

The system supports two stand-alone features:

1. A one-way authentication of the tag (against backend).

2. Reading stored tag data and subsequently updating the stored data on the tag, both over an encrypted communication channel.

3.1.1 Concept

The main idea behind the Secure Channel System is to eliminate all risks of an open Android application by encrypting any private communication. The Android-running device resembles a communication bridge between backend and tag. The backend is hosted in a secure environment and administrated by the system owner. Communication between backend and smartphone is achieved over Wi-Fi or mobile internet and is secured with Transport Layer Security (TLS). The Android device connects with the tag over Near Field Communication (NFC). The tag contains functionality to get authenticated, on request, to the backend, over a secure one-way authentication protocol. Additionally, the tag holds information concerning the corresponding product. The format style of the data is derived from the *ePedigree standard* described in Section 2.5. Therefore, the tag's user data is referred to as *pedigree* throughout this thesis. A signature and timestamp, enables the use of the pedigree to reconstruct a product life cycle. One *pedigree-read* always consists of reading tag data, confirming the data with the backend and writing data with a new signature and timestamp back to the tag.

3.1.2 Anti-Counterfeit Use Case

This thesis focuses on the use of the Secure Channel System in an anti-counterfeit scenario, which helps to mitigate product counterfeit. Thus, the system enables customers and producers to identify product counterfeit and illegitimate distribution of such goods. Pedigree data history can also help the legitimate producer to identify trading routes and find stores that sell counterfeit products. Using a security-enhanced Android application, it is possible to distribute the application for free over digital distribution platforms like Google Play [Goo15]. Therefore, any customer can download and install the application to their personal smartphones, provided the phone is NFC-capable. If the product is equipped with a compatible RFID tag, the customer can scan the product's tag and get information about the product from the backend. The two main scenarios are as follows:

- A customer can verify that he is buying a legitimate product. Additional, displayed information about the recommended retail price might give him a better understanding of the current price margin. In addition, the producer gets even more useful data, such as the existence and possibly the location of the product, which is important information for any market analysis. Furthermore, information concerning the smartphone could be sent.

- The tag is not a legitimate tag and authentication fails. The customer then knows the product is a potential counterfeit. This of course implies that counterfeiters actually tried to resemble the original tag. The producer is informed about the possible counterfeit. Additional forms for the customer could give confirmation that he really tried to buy a legitimate product and what product he intended to buy originally. This additional information helps to distinguish between true negatives and false negatives. Moreover, location information could be of high value for the producer for future investigations.

If a customer installs the application on his phone, he may already have an initial buying intention for the tagged product. This intention is reinforced with every successful identification, which may also motivate the customer to use the application to scan other available goods. By means of that, the application can improve the customers shopping experience, which may lead to him buying more and returning to the same store again. Bonus programmes or special discounts can be used as incentive for the customer to send additional information concerning the goods. Privacy issues aside, the producer can gather valuable data from every pedigree-read. It enables live samples of the products in the field. The *work* itself is done by the potential customers. Actively reporting customers help to identify black markets faster. The scanned pedigree data helps to create and maintain an active product life cycle.

3.1.3 Authentication

In this thesis *authentication* is always used to describe the process of a one-way authentication between the backend and the tag, where the tag is authenticated to the backend. A protocol based on elliptic curve arithmetic proposed by Braun *et al.* [BHM08] is used, as depicted in Figure 3.2. The proposed system uses Montgomery Multiplication in $\mathbb{F}_p$ (see Section 2.1.5), whereas authentication is based on a master's thesis from Kuleta [Kul14]. The proposed system extends Kuleta's version to represent the proposed protocol by Braun accordingly. Specifically, signing of a tag's public key is included as well as the verification of the signature by the backend. Kuleta authenticated the tag to the smartphone, whereas the proposed system authenticates the tag to the backend. Therefore, all calculations previously done on the smartphone are shifted to the backend server. All communication between the backend and the tag during authentication, is thus insensitive per design. Hence there is no need to encrypt any communication between smartphone and backend during the authentication process.

The following section describes the authentication protocol in detail.

Tag Authentication Protocol

A sequence diagram of the used authentication protocol is shown in Figure 3.13. The tag gets authenticated to the backend with the smartphone as bridge in between. The protocol is very similar to the protocol proposed by Braun *et al.* [BHM08]. For details concerning the implementation of the authentication process refer to Section 4.1.2.

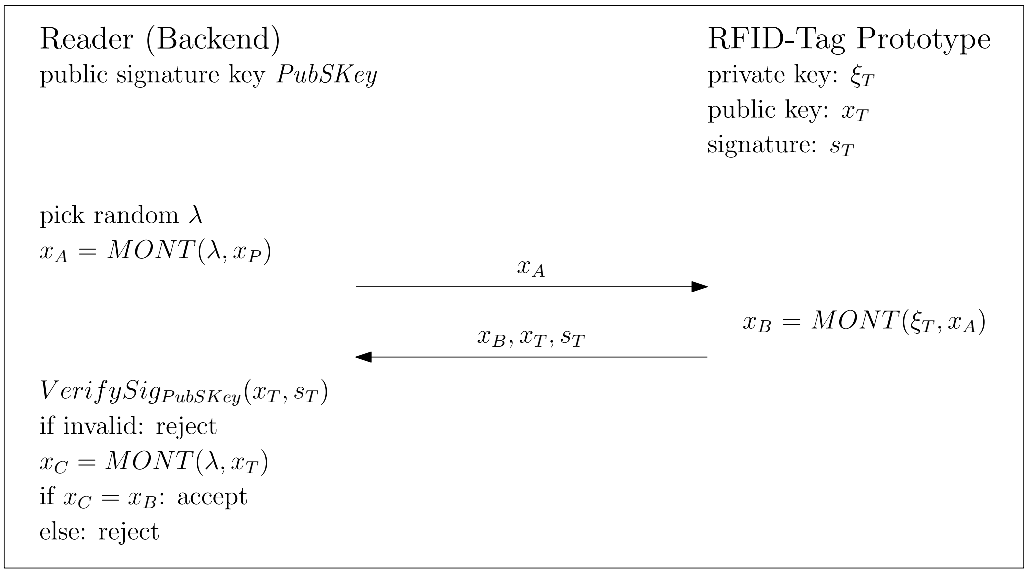


Figure 3.2: One-Way Authentication Protocol
(adapted from [BHM08])

Prerequisites: Let E be an elliptic curve over GF(p). Let $P = (x_P, y_P)$ be a point on E with order q where q is a prime. The NIST curve P-192 [Nat99] is used for E and the corresponding base point for P.

Tag Setup: The RFID-tag prototype is initialized with a randomly chosen private key $0 < \xi_T < q$ and a certificate (x_T,s_T) consisting of the corresponding public key x_T and the signature $s_T = GenSig_{PrivSKey}(x_T)$ using the private signature key $PrivSKey$.

Backend Setup: The backend is initialized with the public signature key $PubSKey$ to verify the tag signature of the pubic key x_T.

Authentication Process: The backend picks a random value λ and computes the challenge x_A with a Montgomery Multiplication $x_A = MONT(\lambda, x_P)$, where x_P is the x-coordinate of the base point P. This challenge is then sent to the tag. The tag now calculates the response $x_B = MONT(\xi_T, x_A)$. Together with the certificate (x_T, s_T) the response is returned to the backend. The backend first verifies the signature with $VerifySig_{PubSKey}(x_T, s_T)$. If the certificate is invalid, the tag is rejected. Otherwise the backend continues with the verification of the response. The check value $x_C = MONT(\lambda, x_T)$ is calculated and compared to x_B. If both values are identical, the authentication succeeds an the tag is accepted. If the two values differ, the tag is rejected.

3.1.4 Tag Data and Operations

The main operation visible to the smartphone user is the *pedigree-read* operation, which reads and writes tag data (in the background). A sequence diagram for a single pedigree-read is depicted in Figure 3.14. The following paragraphs describe the reading and writing procedure in detail. The communication is described on a higher level, as a backend-to-tag communication. The smartphone and its bridge functionality are not described. The content of the pedigree itself is described in Section 3.3.2. The pedigree has a size of up to 4 kB, whereas the data is sent in multiple packages of 100 B each. The communication is encrypted with AES-OCB. The protocol's details are discussed in Section 3.3.1, whereas the read and write operations operate as follows:

Reading Pedigree [0x43]

The basic `read` command consists of the command-byte 0x43 and the memory offset to be read from.

→ An initial `read` command is sent to the tag. The memory offset is reset to 0.

← As response, the first part of the pedigree is returned.

→ The next `read` command is sent. The memory offset is increased by 100.

← The next pedigree part is returned.

→ The next `read` command with increased offset is sent.

... (depending on the pedigree size)

← The final part of the pedigree is returned.

The backend assembles the pedigree parts and validates the included signature. After the reading is done and the signature is verified, the backend stores parts of the pedigree and calculates the new signature, depending on the new data for the pedigree. Then a new pedigree is going to be written to the tag.

Writing Pedigree [0x42]

The basic `write` command consists of the command-byte 0x42, the memory offset to be written to and up to 100 B of the pedigree.

→ An initial `write` command is sent to the tag. The memory offset is reset to 0.

← As response, a status byte is returned. This should be 0x00.

→ The next `write` command is sent. The memory offset is increased by 100.

← The next status byte is returned.

→ The next `write` command with increased offset is sent.

... (depending on the pedigree size)

→ The final `write` command with increased offset is sent.

← The final status byte is returned.

3.2 Backend Design

The backend server used is based on an existing backend solution called *detego SURVEYOR*. Developed by Enso Detego GmbH, detego SURVEYOR is a comprehensive platform which is used to configure and personalize software RFID solutions for a range of asset management and product life cycle management applications. It features a central data store, a web application, a reporting system and a set of applications for desktop, handheld and other purposes. An architectural overview of detego SURVEYOR can be seen in Figure 3.3.

The central backbone component is an MS SQL database. The communication between the database and connected clients is handled by a web service.

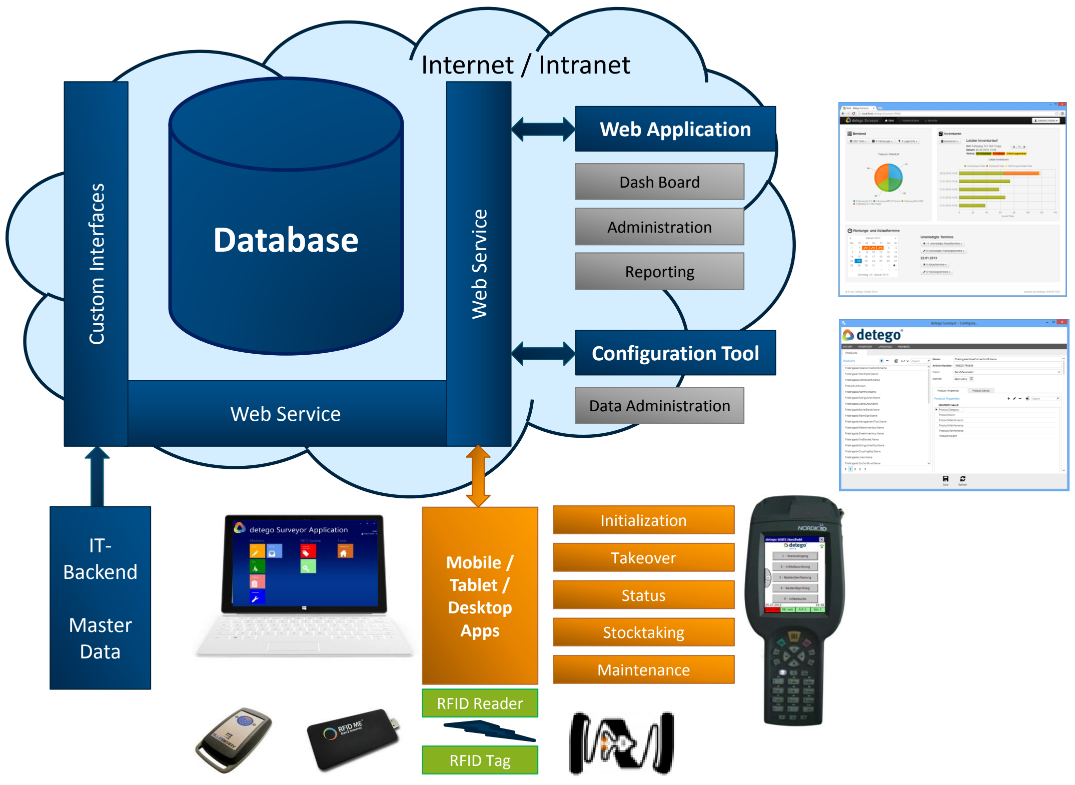


Figure 3.3: Architectural Overview of detego SURVEYOR (developed by Enso Detego GmbH)

3.2.1 Database

The basic storage for application relevant data is a database. A relational SQL database managed by Microsoft SQL Server is used. The impelemented data structure features a set of entities in order to provide a high degree of flexibility for different use cases. The major entities are listed in Table 3.1 and a database relation diagram can be found in the appendix (Figure A.2).

Entity Name	Description
Readpoint	The device running the reading application, *i.e.* a smartphone.
Site	The site roughly defines the location of a readpoint. This may be the city or a specific building.
Item	A unique entity with an Electronic Product Code (EPC), for identification. This is represented by a single tag.
Group	A collection of Items.
Product	The base of every Item. Used to define a product that then has multiple items.
Property	Any additional information for items, products and other entities. Properties are stored and linked to the corresponding entity. Examples for properties are scan times and GPS coordinates.
Person	Information concerning legitimate users, including login credentials.
Rule	Rules define the privilege escalation for each user.

Table 3.1: Relevant Database Entities of the detego SURVEYOR Database

The Structured Query Language (SQL) is a programming language for defining and manipulating data stored in a Relational Database Management System (RDBMS). SQL became an ISO standard in 1987, defined by its last version in [Sta11]. Different databases use modified versions of SQL. Microsoft SQL Server utilizes Transact-SQL, Oracle databases make use of PL/SQL and for MySQL there are many different front ends to choose from ([Cha14]). The language can be divided into two sub-languages: the Data Management Language (DML) and the Data Definition Language (DDL). While the DDL contains the creation and deletion commands for the database schema and tables, the DML consists of manipulation, updating, retrieving and inserting commands for the data sets. As the end-user does not always want to insert the SQL commands directly, the user-input is restricted to parts of a predefined SQL query. The end-user then only has to insert the remaining input-values. Although such predefined commands may look secure, an adversary can still do a lot of damage to a database, if the given user-input is not checked for malicious code by the backend structure. The following paragraph shows vulnerabilities in context with the database access.

Database Access: The database is queried with SQL statements. If the client could query the database directly, this would result in an easy way for so called *SQL Injections*. These SQL Injections enable the attacker to see and modify every entry of the database. Listing 3.1 shows an SQL query for a person entry with admin credentials.

Listing 3.1 Example SQL Login

```
SELECT PersonId, Firstname, Lastname, Email
FROM Person
WHERE CommonName=<input> AND PasswordHash=<input>
```

This statement is executed with a user given input. The AND-operator forces two valid inputs for a non-empty result. The statement is then executed with the given user input.

Listing 3.2 Executed SQL Query

```
SELECT PersonId, Firstname, Lastname, Email
FROM Person
WHERE CommonName='test.user' AND PasswordHash='80e..aa0'
```

Listing 3.2 returns the GUID for the entry, the user's first and last name as well as the email stored in the database. The attacker can now use the `<input>` fields for injecting additional commands to the query. The following examples show possible injections:

Listing 3.3 Example Injection 1: <PasswordHash> Input: ' OR 1=1

```
SELECT PersonId, Firstname, Lastname, Email
FROM Person
WHERE CommonName='admin' AND PasswordHash='' OR 1=1'
```

Here the attacker modified the query with an OR-clause that is always true. So the attacker does not need any password. The result includes all entries that match the `<CommonName>` input for `'admin'`.

Listing 3.4 Example Injection 2: <CommonName> Input: admin'–

```
SELECT PersonId, Firstname, Lastname, Email
FROM Person
WHERE CommonName='admin'--' AND PasswordHash='any'
```

SQL comments are started with "--". Therefore, by inserting the comment clause, the attacker successfully changed the query to not require the password hash at all. Everything after "--" is ignored during execution time.

As the attacker could modify the statement to execute an arbitrary command, the database is not secure at all. One approach to avoid these kind of vulnerabilities is to use a web service as the only entity to connect to the database. This sanitizes the input and filters out invalid input. It also manages user sessions, thus limiting the basic access to the database.

3.2.2 Web Service

A web service is a form of communication that provides machine-to-machine communication over a network [W3C04]. The interface is typically described in Web Services Description Language (WSDL), which is based on the Extensible Markup Language (XML). Other systems can interact with the web service using Simple Object Access Protocol (SOAP) messages. The SOAP protocol relies on XML for the message format and on HTTP for message transmission. Figure 3.4 shows a typical web service invocation on the Secure Channel structure. The web service basically protects the database from the outside world by acting as a proxy where security features can be implemented. Clients connect to the web service, and only the web service fetches information from the database. In the case of detego SURVEYOR, the web service is provided by Windows Communication Foundation (WCF). WCF provides a set of APIs, as well as a runtime, to build connected, service-oriented applications for the .NET Framework [Mic12b]. The provided web services are a part of the whole WCF structure.

The web service provided by the backend server is consumed by the smartphone. TLS is used to provide a secure access to this service.

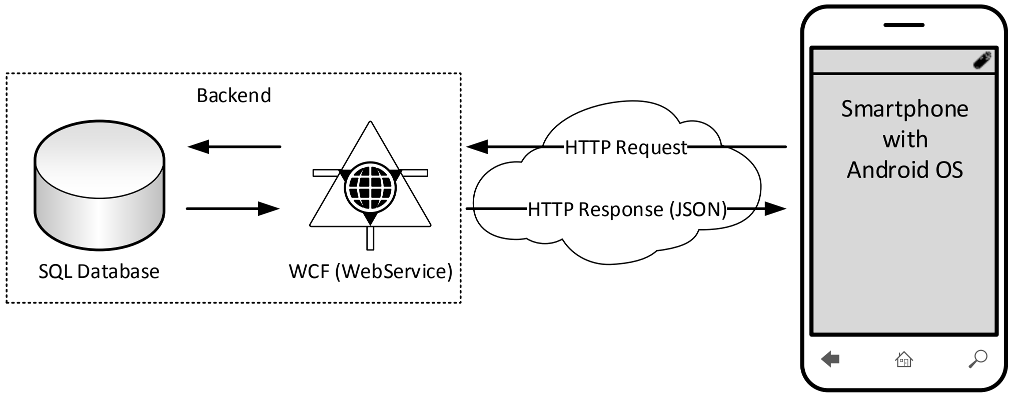


Figure 3.4: Web Service Invocation

JavaScript Object Notation: The web service used in this thesis uses the JavaScript Object Notation (JSON). JSON is an open standard human-readable text format that describes attribute-value pairs [ECM13]. Originally derived for use in JavaScript, it has a language-independent data format. Generating and processing JSON data is possible for a large variety of programming languages. Listing 3.5 shows parts of a typical JSON object, in this case the `Item` object.

Listing 3.5 Item JSON object

```
"value":[{
                "ItemId":"6f613bcc-0a0f-e311-be9f-001cc02ec8e3",
                "Identifier":"E00401000B5B93BB",
                "ProductId":"6e613bcc-0a0f-e311-be9f-001cc02ec8e3"
        }]
```

3.3 RFID-Tag Prototype

The used RFID tag is developed by Infineon Technologies [Inf14]. The following sections describe the protocol used for communicating with the tag and the data stored on the tag.

3.3.1 Protocol

Physical Frame (NDEF Message): The physical frame includes all parts that are needed according to the NDEF specification [Nok13]. The payload is further referred to as *command frame*. Table 3.5 shows the physical frame parts:

- **L1** (2 Bytes): The length of the whole NDEF message. This is 3 + length of the payload.
- **0xD0** (1 Byte): Constant
- **L2** (2 Bytes): The length of type and payload, 1 byte each.
- **PLD** (n Bytes): The command frame (payload), which is the actual message.

Command Frame: The command frame of the protocol is the message requested at the backend and delivered to the RFID tag by smartphone. The content of the command is listed in Figure 3.5. The command frame contains the following parts:

- **PID** (1 Byte): The protocol version. Used to implement different protocols for the same tag. Here the PID is always 0x02.
- **ENC** (1 Byte): The encryption type. Used types include not-encrypted (0x00), AES only (0x01) and AES-OCB (0x02).
- **LEN** (2 Bytes): The length of the unencrypted DATA frame.
- **TSP** (4 Bytes): The timestamp. This is used to mitigate replay attacks, where an attacker resends older packages. The timestamp is increased on every message. Additionally it is used as associated data and to calculate the nonce for the AES-OCB scheme. The 4 bytes can hold the standard Unix-timestamp [Wik14].
- ***CMD*** (1 Byte, *possibly encrypted*): The tag-command to execute.
- ***DATA*** (multiple Bytes, *possibly encrypted*): The actual data. The different DATA frames are explained in the following paragraphs. If the data is encrypted, the OCB hash tag is appended at the end. No nonce is included in the data package, as it is derived from the timestamp.

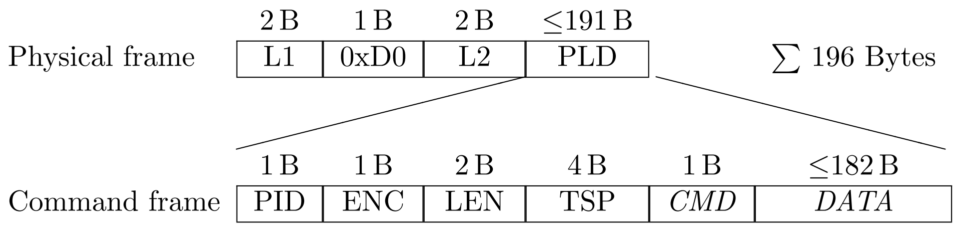


Figure 3.5: Physical and Command Frame of the Protocol. CMD and DATA Sizes for Unencrypted State.

	1 B	1 B	26 B	1B
Challenge	LEN(x_A)	LEN(x_B)	x_A	STATUS

	1 B	1 B	1 B	26 B	24 B	128 B	1 B
Response	LEN(x_B)	LEN(x_T)	LEN(s_T)	x_B	x_T	s_T	STATUS

Figure 3.6: DATA Frames for Authentication

Authentication DATA: The two different authentication DATA frames are depicted in Figure 3.6. Challenge and response communication is always unencrypted. Any transferred data is public data.

- Challenge (29 Bytes)
  - Length of challenge x_A (1 Byte): This is always 26 (0x1A).
  - Length of expected response (1 Byte): This is always 26 (0x1A).
  - Challenge x_A (26 Bytes): The challenge itself is 24 bytes long, two bytes are used to detect possible overflow or carry-bit errors.
  - Status byte (1 Byte): 0x00 if successful.
- Response (182 Bytes)
  - Length of response x_B (1 Byte): This is always 26 (0x1A).
  - Length of public key x_T (1 Byte): This is always 24 (0x18).
  - Length of signature s_T (1 Byte): This is always 128 (0x80).
  - Response x_B (26 Bytes): The response itself is 24 bytes long, two bytes are used to detect possible overflow or carry-bit errors.
  - Public key x_T (24 Bytes): The tags public key. Derived from the private key with Montgomery multiplication.
  - Signature s_T (128 Bytes): The signed public key. Signed with the $PrivSKey$.
  - Status byte (1 Byte): 0x00 if successful.

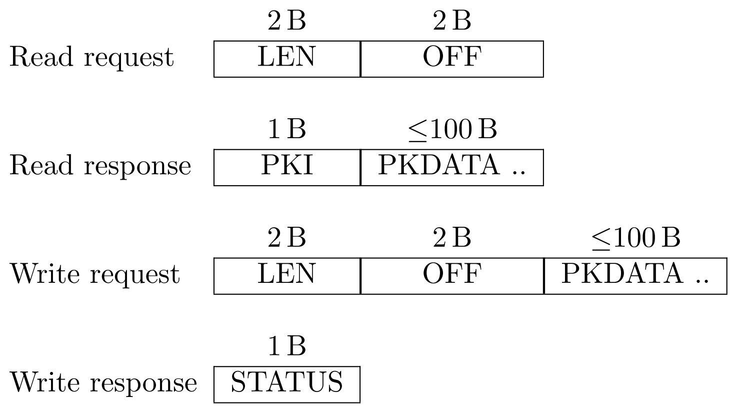


Figure 3.7: DATA Frames for Read and Write Operations

Pedigree DATA: What is visible to the smartphone users as *pedigree-read* operation actually consists of read and write commands. Those commands are divided into tag requests and corresponding responses. As the pedigree data can hold up to 4 kB, the message is split into 100-Byte packages. Figure 3.7 shows the four different DATA frames.

- Read request (4 Bytes)
  - LEN (2 Bytes): Length of data to be read. This is always 100 (0x64). The second byte allows bigger packages.
  - OFF (2 Bytes): The memory offset for the data. This enables reading the full pedigree in multiple parts.
- Read response ($\leq$ 101 Bytes)
  - PKI (1 Byte): Package Info. Additional information about the package. 0xFF if the pedigree consists of more data. If the package is the final part of the pedigree, then it is the length of the package ($\leq$ 0x64).
  - PKDATA ($\leq$100 Bytes): Package data. The read data.
- Write request ($\leq$ 104 Bytes)
  - LEN (2 Bytes): Length of data to be written. This is always 100 (0x64). The second byte allows bigger packages.
  - OFF (2 Bytes): The memory offset for the data. This enables writing the full pedigree in multiple parts.
  - PKDATA ($\leq$100 Bytes): Package data. The data to be written.
- Write response (1 Byte)
  - STATUS (1 Byte): Status byte. Should be 0x00.

3.3.2 Tag Data

The used RFID-tag prototype stores user data in an encrypted way. Here an XML-formatted product description is used as user data.

The *Secure Channel System* can store a pedigree with a size of up to 4 kB on a tag. The content of the pedigree can be split into

- *Basic Information*, which describes the product and the item itself, which never changes,
- *Additional Data*, which describes reader related information changing on every pedigree-read, and a
- *Pedigree Signature*, which signs the two parts mentioned above.

The basic information, describing product and item, is derived from the *detego SURVEYOR* database structure (see Section 3.2.1). The basic information never changes, as all data is directly linked to the tagged product. Examples for basic information include:

- Product data, such as
  - Name,
  - Description,
  - Article Number,
  - Global Trade Item Number (GTIN),
  - Colour,
  - Season,
  - Company,
  - Origin, and
  - Recommended Retail Price (RRP).
- Item data, such as
  - Electronic Product Code (EPC),
  - Creation Timestamp, and
  - Producer (Location).

The additional data contains information concerning the reading device and the product life cycle. A comprehensive history for life cycle management can then be extracted from the database. The additional data is intended to change on every pedigree-read. Examples for additional data include:

- Life cycle management information, such as
  - Sold status,
  - Complained status, and
  - Error status.

- Reader data, such as
  - Timestamp,
  - Location,
  - Global Positioning System (GPS) coordinates,
  - Reader Information (IMEI),
  - Read count, and
  - User information.

The signature appended can be referred to as additional data, as it is created on every pedigree-read. The basic information and additional data is signed and verified by the backend with ECDSA.

3.4 Android Application Design

Within the Secure Channel System, the smartphone is the bridge between backend and RFID-tag prototype. This is realized with an intuitive Android application, with a clear interface but intentionally limited background information. Only the tag authenticity and the product data are important and thus displayed to the user. As most of the communicated messages are encrypted, the Android application itself is not able to see any information sent.

3.4.1 Android User Interface

The Graphical User Interface (GUI) for the Secure Channel Android application is designed to be simple and useful. Only product data which is not sensible in any security concerning way is presented to the user.

Start-Up Screen: The application is started with the application icon or by scanning an NFC tag. If started by icon, a start-up screen (Figure 3.8) shows the user how to place the smartphone on the tag, which ensures a stable connection over the NFC interface. After connection between tag and reader is established, an *Android NFC-event* is triggered. Directly scanning an NFC tag without starting over the icon, triggers the same NFC event on the application.

Main Screen: The NFC event brings the application to its main screen (Figure 3.9). On the bottom of the screen there are three check boxes displaying system relevant information, such as web-service connection status, location data or whether a tag is in the reader's field or not.

Authentication: Upon first start, the main screen shows the "Authenticate" button[1] alongside a grey *Enso Detego* icon. The colour of the icon indicates the current authentication state. A grey icon means nothing has happened so far. The user can now tap the

[1]The buttons for authentication and pedigree-read operation are named "Authenticate Tag (GET)" and "Read from ePedigree (POST)".

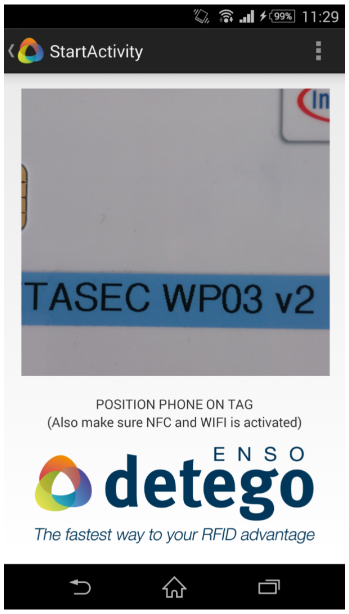

11:29
StartActivity
TASEC WP03 v2
POSITION PHONE ON TAG
(Also make sure NFC and WIFI is activated)
ENSO detego
The fastest way to your RFID advantage

Figure 3.8: Start-Up Screen of the Android Application

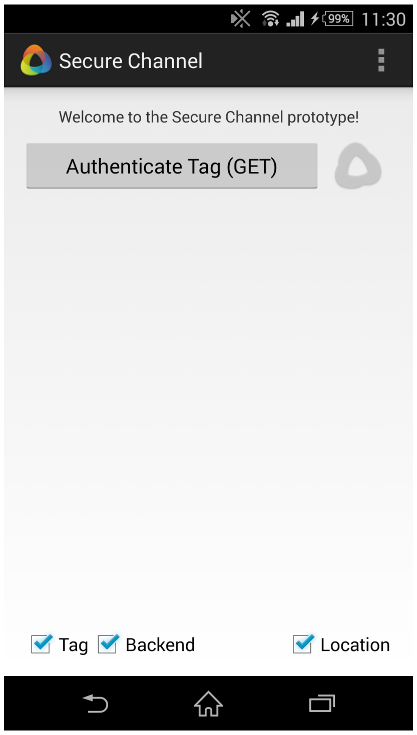


Figure 3.9: The Main Screen of the Secure Channel Android Application: Only authentication is possible. A grey indicator icon is showing the authentication state as “not yet run”.

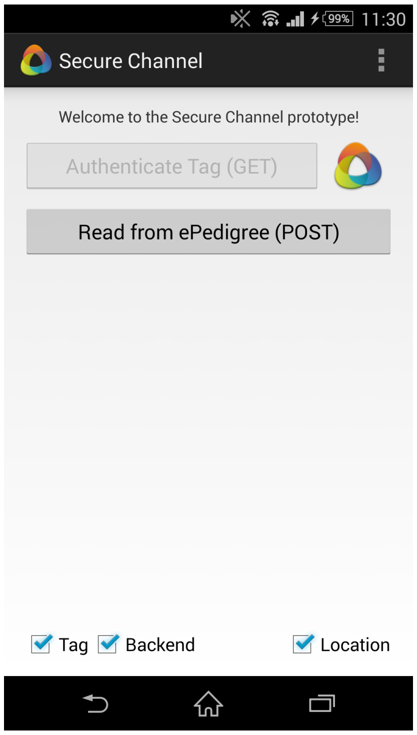
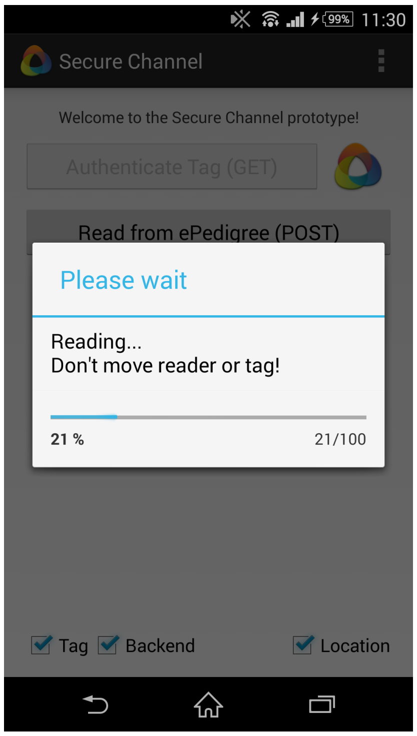
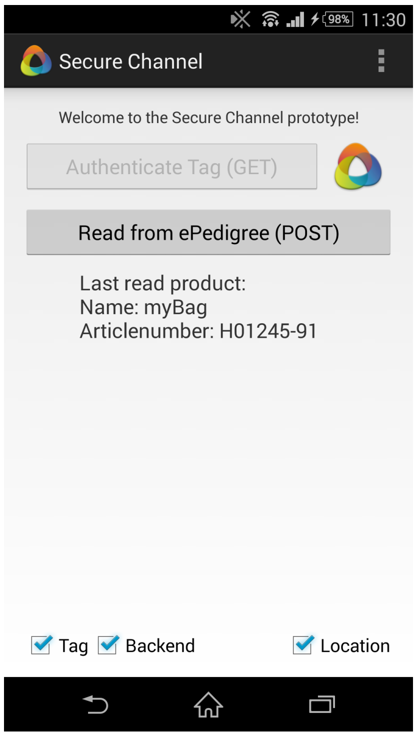

11:30
Secure Channel
Welcome to the Secure Channel prototype!
Authenticate Tag (GET)
Read from ePedigree (POST)
Tag
Backend
Location

Figure 3.10: The Main Screen After a Successful Authentication: Reading the pedigree is now possible. A coloured indicator icon is showing a successful authentication.



Reading...
Don't move reader or tag!

Figure 3.11: During the pedigree-read operation, a loading dialog is presented to the user.


11:30
Secure Channel
Welcome to the Secure Channel prototype!
Authenticate Tag (GET)
Read from ePedigree (POST)
Tag
Backend
Location

Last read product:
Name: myBag
Articlenumber: H01245-91

Figure 3.12: After successful pedigree-read, the product data corresponding to the read tag is shown on the main screen.

button and initiate the authentication process. The different methods, tasks and requests following a button click are described in Section 4.3.3. If the authentication fails, the icon becomes red. A successful authentication results in a colored icon. The "Authenticate" button is deactivated and the "Pedigree-Read" button is visible (see Figure 3.10).

Pedigree-Read: If the user taps the "Pedigree-Read" button, a background task is started, which shows a loading dialog as depicted in Figure 3.11. The progress indicates pending operations. Details on the implemented procedure are discussed in Section 4.3.3. Possible errors, like connection loss or signature validation failure are displayed in a dialog on screen. However, if the pedigree-read operation is successful, the product data is shown on the main screen (Figure 3.12).

3.5 Workflow

This section includes detailed descriptions of the communication between the three parties (tag, application and backend). Additionally sequence diagrams are used to describe communication.

Authentication: Figure 3.13 shows the authentication process. During this authentication the tag gets authenticated to the backend. The preconditions are a connection between the smartphone and the web service, as well as the tag being in the smartphone's read range. If the user initiates an *authentication*, the Android application sends the first *HTTP GET* command to the web service. This *ScGetChallenge()* command requests a valid challenge from the backend to initiate the implemented challenge response protocol (for additional information of the authentication protocol see Section 3.1.3). The backend calculates the challenge x_A and includes it into the command frame (see Section 3.3.1). The command frame is sent to the application, where it gets an additional NDEF frame. The NDEF message is then sent to the tag. The tag parses the NDEF frame as well as the command frame and calculates the response x_B. Together with the certificate (x_T, s_T) the response is returned to application as command frame inside an NDEF frame. The command frame is then sent to the backend using the HTTP POST command *ScVerifyResponse(cmd(*x_B*,*x_T*,*s_T*))*. On the backend the signature is verified. Assuming a valid signature, x_C is calculated and compared to x_B. A successful comparison means a successful authentication. In a final step the backend responds to the application with the authentication result.

Pedigree-Read: Figure 3.14 shows the entire *pedigree-read* operation, consisting of multiple read and write operations. Figure 3.15 and Figure 3.16 show an even more detailed view of those two operations. To start the *pedigree-read*, the tag has to be authenticated to the backend first. The user can then initiate the pedigree-read operation. The application then sends the HTTP GET command *ScGetReadCommand()*, whereas the backend builds the first read command for the tag (see Section 3.3.1 for details concerning the protocol). This command is encrypted with AES-OCB and returned to the application. After including the response in an NDEF frame, the application sends the message to the tag where it is parsed and decrypted. If the message can be dercypted successfully, the first 100 bytes are read from the pedigree memory and the response-command frame is built. After encrypting and including into an NDEF frame, the tag response is sent back to the application. The application forwards the encrypted frame to the backend using the HTTP POST command *ScRead(enc(cmd(pedigree part)))*. The message is then decrypted on the backend side and the command frame is parsed. The read pedigree part is stored in the backend. Included in the command frame, the package info byte informs about further read requests necessary to get the entire pedigree. If more parts are to be read, the backend responds to the HTTP POST with the next read command. The additional pedigree parts are read the same way as the first part. If the pedigree has been read entirely, the backend parses the complete pedigree at once. The included signature, which has to be the same as the backend's signature, is verified. The additional timestamp is saved in the database. For history data recording, after verifying the signature, the product details are checked: The product has to exist in the database beforehand. If any product data has changed

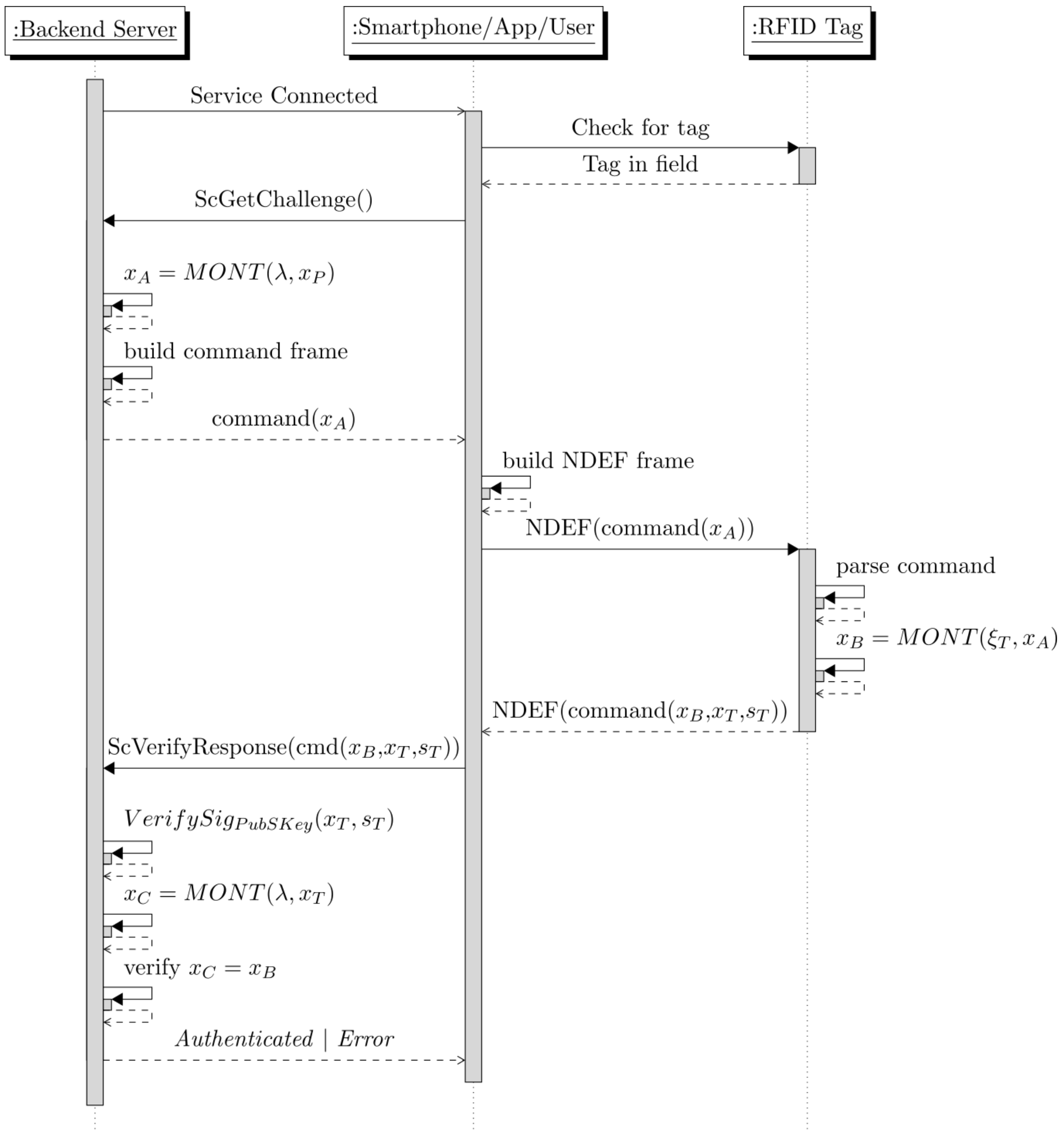


Figure 3.13: Sequence Diagram of the Authentication Process

at the backend, the pedigree is updated. A new timestamp is added and the package pedigree-timestamp is signed by the backend. The new pedigree consists of the product data, the timestamp, and the signature. This data is now sent to be written to the tag in one or more transmissions. If the backend is ready to write, the status *WRITEREADY* is sent as response to the preceding *ScRead* command. Now, the application initiates the write-part with the HTTP GET command *ScWrite()*. The backend builds the first pedigree package and builds the encrypted command frame, which is sent as response to the GET command. The application adds the NDEF frame and forwards it to the tag. After decryption and command parsing, the message part is written to the designated memory section. An encrypted *OK*-message is sent as a response from the tag to the application. The next HTTP POST command for the backend is *ScWrite(enc(cmd(OK)))*. If additional pedigree parts have to be written, the procedure is repeated. After the final *OK* from the tag, the backend responds to the application with a *DONE* message. The last HTTP GET command *ScFinalize()* is initiated. If the backend is in the correct state, the corresponding product data is sent back to the application where it is visualized for the user.

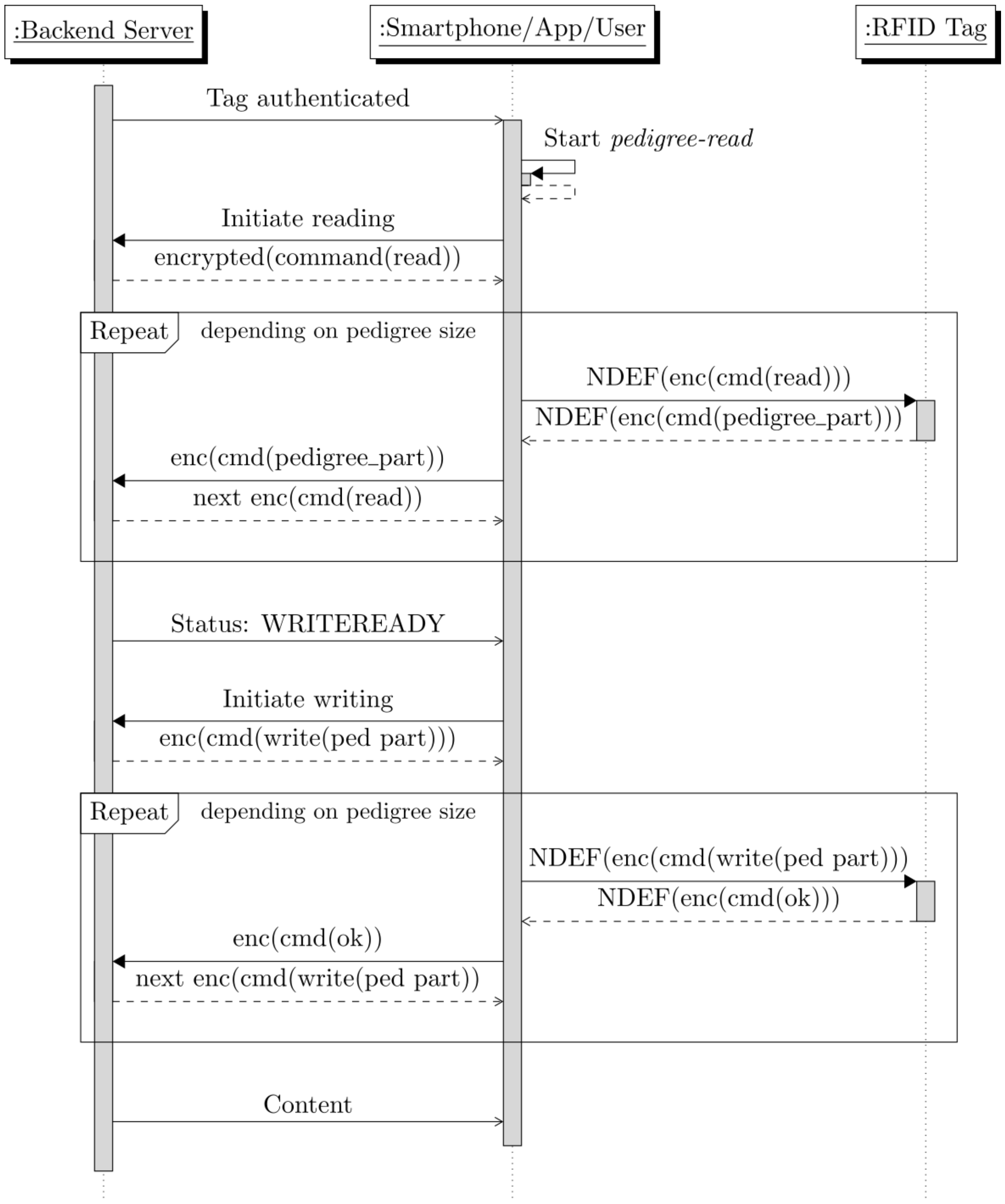


Figure 3.14: Sequence Diagram of the *Pedigree-Read*, Consisting of Read and Write Operations (Details see Figure 3.15 and Figure 3.16).

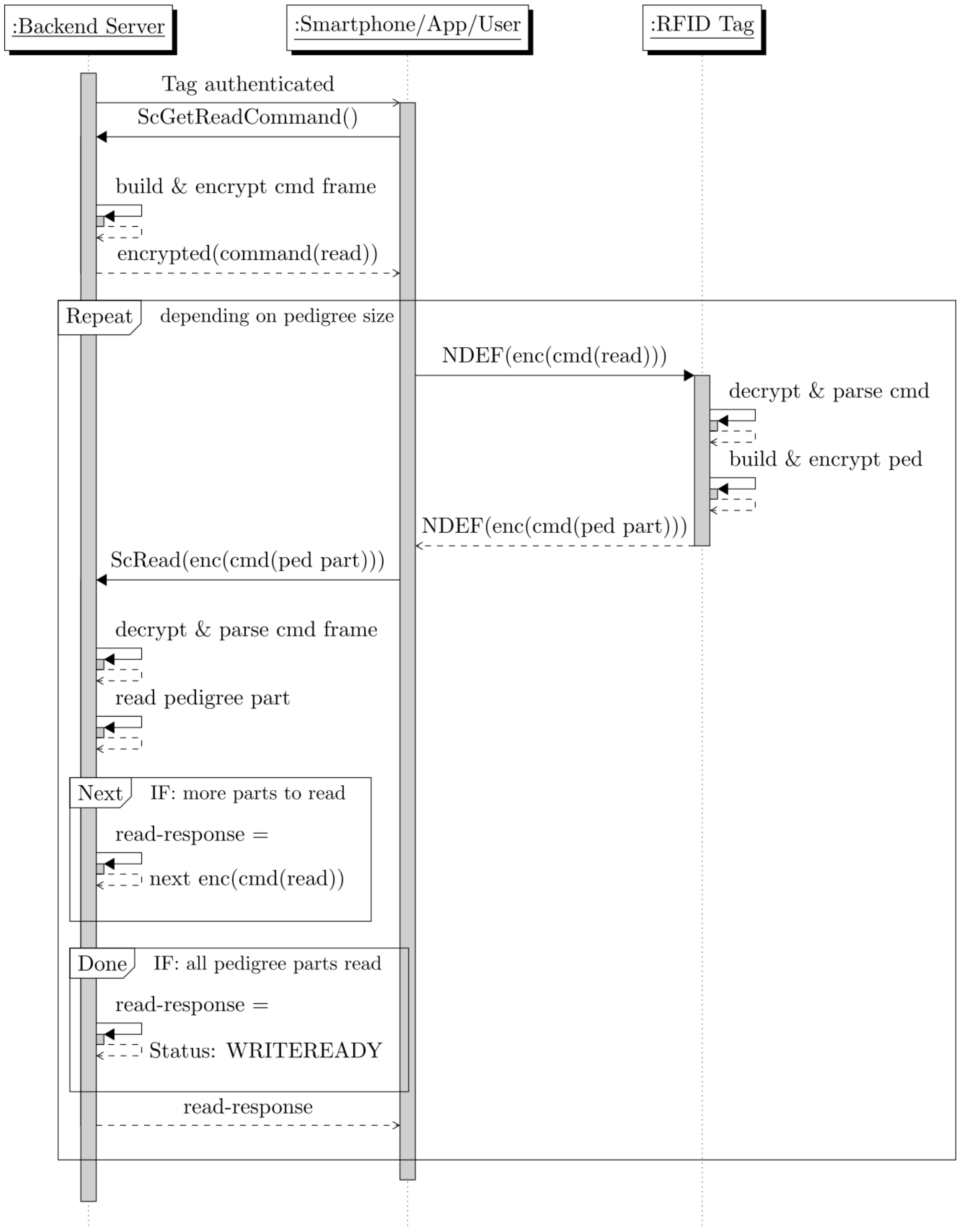


Figure 3.15: Detailed Sequence Diagram of the Read Operation

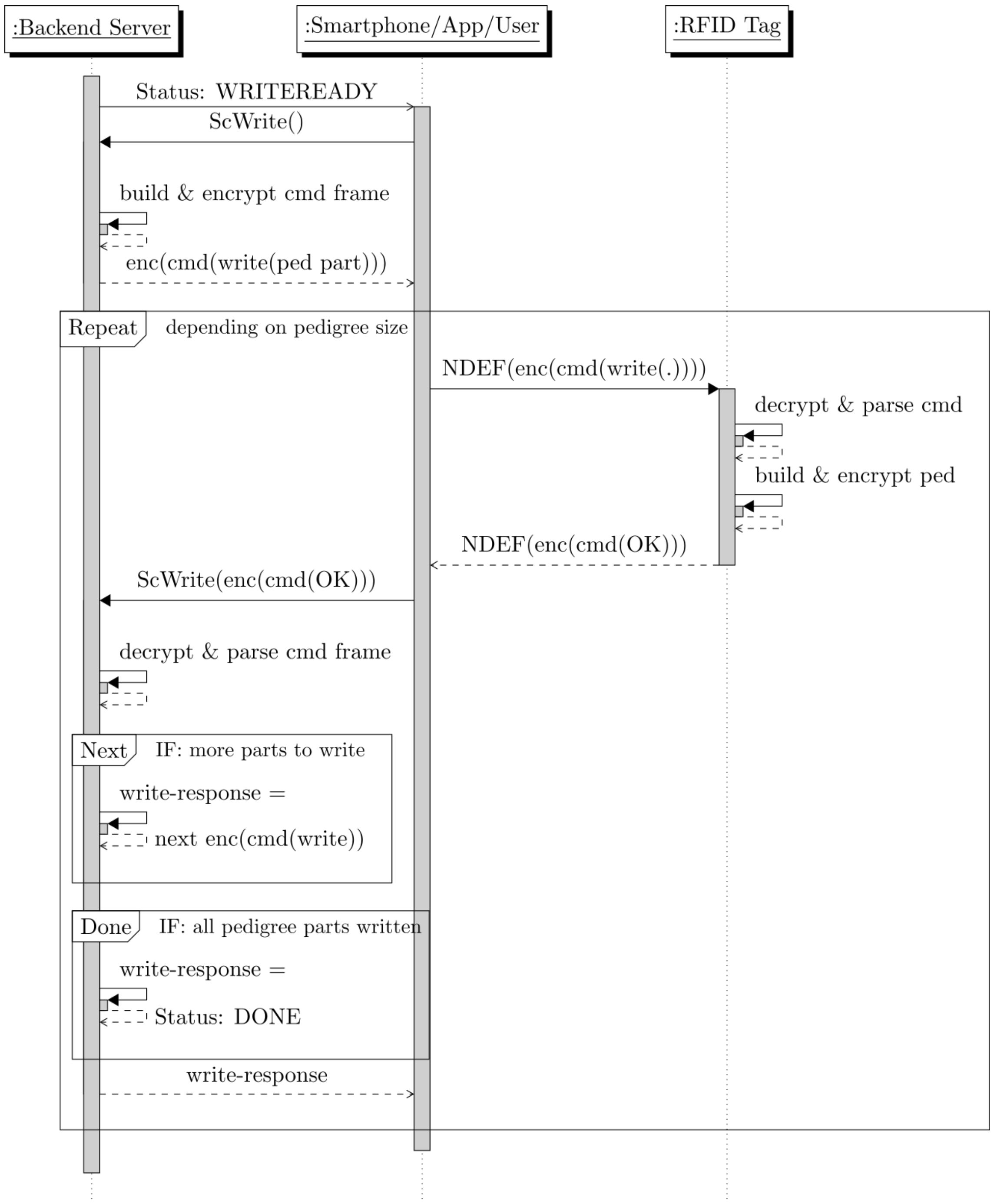


Figure 3.16: Detailed Sequence Diagram of the Write Operation

Chapter 4

Implementation

This chapter describes the detailed implementation of the secure NFC / backend system described in this thesis. It is divided into three main sections: Section 4.1 gives insight into the software simulation of the whole system. The implementation on the backend side is described in Section 4.2, whereas the Android application is described in Section 4.3. Implementation related to the RFID-tag prototype was done by our research-project partner and is not to be disclosed in this thesis.

4.1 Software Simulation

As a first step towards the creation of the designed system, a software simulation is performed. At the very beginning the cryptographic functionalities are tested using unit tests. After that, message communication over the secure channel is simulated within a C# environment. The environment is extended to use encryption and therefore a new protocol stack. The designed classes for this simulation are the same classes used for the backend implementation later on and details can be found in Section 4.2. The main reason for simulating the communication is to give the tag-implementing party a reference behaviour. Starting with implementing a basic unencrypted communication with the designed protocol stack, the tag authentication is simulated. After that, the encrypted communication is tested with an AES encryption, followed by AES-OCB encryption. Using this AES-OCB encryption, the secure channel functionalities are simulated.

Bouncy Castle: The Bouncy Castle Crypto APIs are used for most of the cryptographic functions [otBCI13]. This lightweight APIs enables the secure usage of elliptic curve cryptography and many other cryptographic primitives.

4.1.1 Key Generation

The first important part for the software simulation is the generation of proper keys. The following listing describes the individual key pairs and their respective properties:

- Elliptic Curve Key Pair (192 bit)
  Used for authentication of tag to backend

Curve: *secp192r1*
See Listing 4.1

- RSA Key Pair (1024 bit)
  Used during authentication for the signature
  Signature: SHA1 with RSA encryption
  See Listing 4.2

These key generation methods always generate new key pairs. After the initial generation the key pairs were stored into encrypted files (Listing B.8). Listing 4.3 shows how the stored keypairs are loaded from the encrypted files.

Listing 4.1 Keygen.cs: GenerateEcKeyPair

```
private static string ecCurvename = "P-192";

public ECPoint GetBasepoint()
{
 X9ECParameters x9ecp = NistNamedCurves.GetByName(ecCurvename);
 return x9ecp.G;
}
public AsymmetricCipherKeyPair GenerateEcKeyPair()
{
 X9ECParameters x9ecp = NistNamedCurves.GetByName(ecCurvename);
 ECDomainParameters ecParameters = new ECDomainParameters(x9ecp.Curve,
     x9ecp.G, x9ecp.N, x9ecp.H);
 ECKeyPairGenerator ecGenerator = new ECKeyPairGenerator();
 SecureRandom secureRandom = new SecureRandom();
 ECKeyGenerationParameters ecKeyGenParam = new ECKeyGenerationParameters(
     ecParameters, secureRandom);
 ecGenerator.Init(ecKeyGenParam);
 return ecGenerator.GenerateKeyPair();
}
```

Listing 4.2 Keygen.cs: GenerateRsaKeyPair

```
public AsymmetricCipherKeyPair GenerateRsaKeyPair()
{
 RsaKeyPairGenerator rsaGenerator = new RsaKeyPairGenerator();
 SecureRandom secureRandom = new SecureRandom();
 RsaKeyGenerationParameters rsaParameters =
  new RsaKeyGenerationParameters(
                       new BigInteger("65537", 16)
                       , secureRandom
                       , 1024
                       , 80);
 rsaGenerator.Init(rsaParameters);
 return rsaGenerator.GenerateKeyPair();
}
```

Listing 4.3 Keygen.cs: GetKeysFromFiles

```
public AsymmetricCipherKeyPair GetKeysFromFiles(FileStream publicfile,
    FileStream privatefile)
{
 Asn1Object ao = Asn1Object.FromStream(publicfile);
 SubjectPublicKeyInfo pubinfo =
  SubjectPublicKeyInfo.GetInstance(ao);
 var publicParam = PublicKeyFactory.CreateKey(pubinfo);

 ao = Asn1Object.FromStream(privatefile);
 EncryptedPrivateKeyInfo enpri = EncryptedPrivateKeyInfo.GetInstance(ao);
 PrivateKeyInfo priECinfo = PrivateKeyInfoFactory.CreatePrivateKeyInfo("
    Meta[:SEC:]".ToCharArray(), enpri);
 var privateParam = PrivateKeyFactory.CreateKey(priECinfo);

 return new AsymmetricCipherKeyPair(publicParam, privateParam);
}
```

4.1.2 Authentication (Simulation)

Authentication uses the above mentioned key pairs to simulate the tag authentication process. This simulation uses newly generated key pairs for every authentication run. Additional details about the authentication process can be found in Section 3.1.3. The private key ξ_T is the private key of the EC key pair. The public key x_T has to be calculated with a Montgomery Multiplication

$$x_T = MONT(\xi_T, x_P). \tag{4.1}$$

The signature s_T is computed using the private RSA key (Listing 4.4). The random nonce λ is created by generating an additional EC key pair and using the public part as nonce. The challenge x_A is then generated with the Montgomery Multiplication

$$x_A = MONT(\lambda, x_P). \tag{4.2}$$

The response is

$$x_B = MONT(\xi_T, x_A). \tag{4.3}$$

The signature is then verified using the public part of the RSA key pair. If the signature is approved, the response gets verified with another Montgomery Multiplication

$$x_C = MONT(\lambda, x_T) \tag{4.4}$$

followed by the comparison of $x_C = x_B$. If both values are equal, the authentication succeeds. Listing 4.5 shows the simulated authentication. The used classes Authentication.cs and Montgomery.cs can be found in the appendix (Listing B.2 and Listing B.3).

Listing 4.4 Authentication.cs: ComputeSignatureOfPublicEcKey

```
public Byte[] ComputeSignatureOfPublicEcKey(BigInteger publickey)
{
 Byte[] ecPublicKeyBytes = publickey.ToByteArray();
 ISigner signer = SignerUtilities.GetSigner("SHA1WITHRSAENCRYPTION");
 signer.Init(true, (RsaKeyParameters)(rsaKeyPair.Private));
 signer.BlockUpdate(ecPublicKeyBytes, 0, ecPublicKeyBytes.Length);
 Byte[] sig = signer.GenerateSignature();
 return sig;
}
```

Listing 4.5 Authentication Simulation (Unit Test)

```
[TestMethod]

public void AuthenticationComplete()
{
 Keygen keygen = new Keygen();
 AsymmetricCipherKeyPair authEcKeyPair = keygen.GenerateEcKeyPair();
 AsymmetricCipherKeyPair authRsaKeyPair = keygen.GenerateRsaKeyPair();

 var privatekey = ((ECPrivateKeyParameters)(authEcKeyPair.Private)).D;
 var xP = keygen.GetBasepoint().X.ToBigInteger();
 var publickey = privatekey.Multiply(xP).Mod(Montgomery.p);
                                                // = xT eg. (4.1)
 BigInteger xT = publickey;

 Authentication auth = new Authentication();
 auth.ecKeyPair = authEcKeyPair;
 auth.rsaKeyPair = authRsaKeyPair;

 BigInteger sT = new BigInteger(auth.ComputeSignatureOfPublicEcKey(
     publickey));                               // Listing 4.4

 BigInteger lambda = ((ECPublicKeyParameters)(keygen.GenerateEcKeyPair().
     Public)).Q.X.ToBigInteger();               // nonce
 BigInteger xA = auth.GetNewChallenge(lambda, xP);// challenge eg. (4.2)
 BigInteger xB = auth.GetResponse(xA);          // response  eg. (4.3)
 //real response consists of xb, xT and signature

 bool verified = auth.VerifySignature(xT, sT);
 if (verified)
 {
     BigInteger xC = Montgomery.Multiplication(lambda, xT); // eg. (4.4)
     Assert.AreEqual(xC, xB);
 }
 else Assert.IsTrue(verified);
}
```

4.1.3 OCB Simulation

The used AES mode OCB is implemented, in the class OCB.cs, using the C-reference given by Rogaway [Phi14]. Details explaining the OCB mode can be found in Section 2.1.4. To verify the functionality of the implemented OCB mode, all available test cases from the OCB reference are used.

Tag-Related Simulation: The tag-related simulation environment was developed prior to this thesis and can not be disclosed in detail. All functionalities are tested on byte-level. The protocol is implemented with a command builder class (described in Section 4.2.1). Different key sizes and hash tag lengths were tested. Insight gathered during the testing phase directly impacted decisions for the final implementation.

4.2 Backend Server

In this section, the actual implementation on the backend server is described. The backend server is running detego SURVEYOR software suite (see Section 3.2). The implemented components required for the secure tag authentication are integrated into the existing detego SURVEYOR web service. The existing detego SURVEYOR source code can not be disclosed in this thesis either. The existing classes IDataService.cs and the DataService.svc.cs were extended with new methods and new classes were added to the web service. The following list names all added classes:

- AES.cs
- Authentication.cs
- CommandBuilder.cs
- EPedigreeHandler.cs
- Keygen.cs
- Montgomery.cs
- OCB.cs

The AES.cs class includes the AES encryption and decryption using the standard System.Security.Cryptography namespace. The classes Authentication.cs, Keygen.cs, Montgomery.cs and OCB.cs have been discussed in the previous section. In this section the classes CommandBuilder.cs and EPedigreeHandler.cs are described in detail, followed by the extensions to the existing classes IDataService.cs and DataService.svc.cs.

TLS: The connection between smartphone and backend server is secured additionally using Transport Layer Security (TLS). The implementation of this security feature has been done in previous work [Ple14]. To enable TLS 1.2 on the backend server, three steps are required: First, a certificate for connections is created. Second, the web server is configured to use the certificate as well as the web service. Moreover, the web service is enforced to only accept HTTPS connections. Third, the pool of possible cipher suites is set to only allow specific TLS 1.2 cipher suites. Section 4.3 briefly describes the implementation of TLS for the smartphone application. The following steps are required to enable TLS 1.2 on a Windows based backend server:

1. Enable the use of TLS 1.2 in Windows Internet Options.
2. Enable the use of TLS 1.2 in Windows Registry.
3. Configure WCF Service for secure transport.
4. Configure Web Application for secure transport.
5. Configure https bindings for the websites in Internet Information Services.
6. Configure Windows Group Policy to only allow specific TLS cipher suites.

4.2.1 Command Builder

The command builder class CommandBuilder.cs handles the construction of the byte-array corresponding to the used protocol (see Section 3.3.1). The main method is called `Build`. The parameters include the command type as byte, the data bytes and an encryption flag. At first, the encryption of the command byte and the data is performed. The bytes are concatenated and encrypted using AES-OCB. The used AES parameters have a block size of 128 bit with a 256-bit key. The OCB parameters are the resulting hash tag length of 128 bit, the current timestamp and a 96-bit nonce that is derived from the timestamp. The timestamp is designed to be in Unix time [Wik14]. However, for the prototype implementation a counter is used as timestamp. The nonce can be derived from the timestamp, as it can be public and should not be used repeatedly. Deriving the nonce from the timestamp also saves the bytes needed to append the nonce for decryption. If the `Build` command is called with the parameter for an unencrypted command, the AES-OCB encryption is still computed, but not used. This ensures the same timing as if the message is encrypted. The encrypted command byte and data is then concatenated with the 4-byte timestamp, the 2-byte data length field, and the protocol id byte. The complete command byte array is finally returned. The whole CommandBuilder.cs can be found in the appendix in Listing B.4, the `Build` command is shown in Listing 4.6.

Listing 4.6 Command Builder.cs: Build

```
//command = [PID|ENC|LEN|LEN|TSP|TSP|TSP|TSP|CMD|DATA..TAG]
//          { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 .. +16]

public static byte[] build(CommandType command, byte[] data,
                bool encrypted)
{
 byte[] commandAsByteArray;

 if (data == null) data = new byte[0];

 int bytesToEncrypt = 1 + data.Length;

 byte[] requestToEncrypt = new byte[bytesToEncrypt];
 requestToEncrypt[0] = (byte)command;
 System.Buffer.BlockCopy(data, 0, requestToEncrypt, 1, data.Length);

 byte[] TSP = new byte[4]; TSP = getUnixTimestamp();
 byte[] N = new byte[15];

 System.Buffer.BlockCopy(TSP, 0, N, 11, 4);

 byte[] encryptedRequest = OCB.encrypt(AES.key, AES.key.Length * 8,
     requestToEncrypt, N, TSP, 128);

 if (!encrypted) encryptedRequest = requestToEncrypt;

 commandAsByteArray = new byte[encryptedRequest.Length + (int)
     PositionsPID02.POS_CMD];

 commandAsByteArray[(int)PositionsPID02.POS_PROTOCOL_ID] = (byte)PID.
     PROTOCOL_ID_02;

 commandAsByteArray[(int)PositionsPID02.POS_ENCRYPTION] = (byte)Encryptions
     .ENCRYPTED_AESOCB;
 if (!encrypted) commandAsByteArray[(int)PositionsPID02.POS_ENCRYPTION] = (
     byte)Encryptions.NONE;

 int noncebytes = (encrypted ? 16 : 0);
 commandAsByteArray[(int)PositionsPID02.POS_LENGTH_H] = (byte)((data.Length
      + PositionsPID02.POS_DATA - PositionsPID02.POS_TIMESTAMP + noncebytes)
      >> 8);
 commandAsByteArray[(int)PositionsPID02.POS_LENGTH_L] = (byte)((data.Length
      + PositionsPID02.POS_DATA - PositionsPID02.POS_TIMESTAMP + noncebytes)
      & 0xFF);

 System.Buffer.BlockCopy(encryptedRequest, 0, commandAsByteArray, (int)
     PositionsPID02.POS_CMD, encryptedRequest.Length);
 System.Buffer.BlockCopy(TSP, 0, commandAsByteArray, (int)PositionsPID02.
     POS_TIMESTAMP, 4);


 return commandAsByteArray;
}
```

4.2.2 Pedigree Handler

The data stored on the tag is called *pedigree*, as it is derived from the ePedigree standard (see Section 2.5). Details on the pedigree are described in Section 3.3.2.

The class EPedigreeHandler.cs implements the pedigree as well as the methods for creating the test pedigree as XML and parsing the pedigree from XML and writing it to XML, respectively. The implemented pedigree consists of the following properties:

- Product Name
- Product Category
- Article Number

- Origin
- Recommended Retail Price

The method `createDummyPedigree` is used to create the test pedigree for the prototype. This method is only used once, in conjunction with `convertXmlToByteArray`, to initially set the tag data. When the pedigree is read from the tag, the received byte array is first converted to XML with the method `convertByteArrayToXml` and then parsed as pedigree struct with `readPedigreeToStruct`. The read pedigree information is then compared with already stored products in the database. Changes to the pedigree data, like refreshing the timestamp and creating a new signature, are done on the byte level and do not require any conversion. The full EPedigreeHandler.cs can be found in the appendix in Listing B.5.

4.2.3 Secure Channel Commands

The important commands of the *Secure Channel System* are either `HTTP POST` or `HTTP GET` commands. In the detego SURVEYOR web service these commands are declared in the IDataService.cs class and implemented in the DataService.svc.cs class, respectively. Methods that are called without parameters are `HTTP GET` commands, whereas methods which hand over data as parameters are always executed as `HTTP POST` commands. A list of commands, followed by a detailed description, include:

- ScGetChallenge(),
- ScVerifyResponse(response_string),
- ScGetReadCommand(),
- ScRead(read_response),
- ScWrite(),
- ScCheckWrite(write_response),
- ScFinalize(), and
- ScServiceStatus().

Additionally, Listing 4.7 shows the declaration of the commands in the IDataService.cs class. Except for `ScFinalize`, all commands return objects of type *PropertyValue*. This type is an Entity Framework [Mic15] representation of entries in the database table PropertyValue. As the web service works with the Entity Framework model, it is best practice to always use generated types. Similarly, the method parameters are also implemented as PropertyValue objects. The command `ScFinalize` returns a Product object, as its functionality is to return the product details.

Listing 4.7 IDataService.cs: Declaration of the Secure Channel Commands

```
#region SecureChannel

[WebGet]
[OperationContract]
PropertyValue ScGetChallenge();

[WebInvoke(Method = "POST")]
[OperationContract]
PropertyValue ScVerifyResponse(PropertyValue response_string);

[WebGet]
[OperationContract]
PropertyValue ScGetReadCommand();

[WebInvoke(Method = "POST")]
[OperationContract]
PropertyValue ScRead(PropertyValue read_response);

[WebGet]
[OperationContract]
PropertyValue ScWrite();

[WebInvoke(Method = "POST")]
[OperationContract]
PropertyValue ScCheckWrite(PropertyValue write_response);

[WebGet]
[OperationContract]
Product ScFinalize();

[WebGet]
[OperationContract]
PropertyValue ScServiceStatus();

#endregion
```

In the DataService.svc.cs class, the commands, as well as additional helper methods are implemented. The following detailed descriptions start with those helper methods. The workflows described in Section 3.5 are important for understanding the behaviour of the main methods.

void ScReset()

This method is used to reset control variables and to set important variables to a predefined value. Possible connection losses and failed transmissions are the main reason to reset those values and variables at certain points during execution (see Listing 4.8).

Listing 4.8 DataService.scv.cs: ScReset

```
private void ScReset()
{
    lambda = new BigInteger(new byte[1]);
    read_pedigree_buffer = new byte[0];
    write_pedigree_buffer = new byte[0];
    MAXTRANSMISSION = 100;
    offset = 0;
    copyOffset = 0;
    current_productid = Guid.NewGuid();
}
```

PropertyValue ScCommandtoPropVal(command,name,state)

As mentioned above, the return values are typically PropertyValue objects. The helper method `ScCommandToPropVal` converts the `command` byte-array to a PropertyValue. On each call, the command is stored in the database as an entry in the PropertyValue table. The string parameter `name` indicates what kind of operation is performed and also defines the PropertyBase type for the database entry. An additional status can be set with the string parameter `status`. If the response should only contain the `status`, the `command` parameter is *null*. Listing 4.9 shows the method's implementation.

Listing 4.9 DataService.scv.cs: ScCommandToPropVal

```
public PropertyValue ScCommandToPropVal(byte[] command, string name, string
     state)
{
    var propBase = entities.PropertyBases.SingleOrDefault(pb => pb.Name ==
        name);

    if (propBase == null)
    {
        var propType = entities.PropertyTypes.SingleOrDefault(pt => pt.Type
             == "Binary");
        propBase = new PropertyBase()
        {
            Name = name,
            Description = "SecureChannel Command",
            TypeId = propType.PropertyTypeId,
            Readability = "RW",
            Visibility = "VISIBLE"
        };
        entities.PropertyBases.Insert(entities, propBase);
        entities.SaveChanges();
    }

    var propValue = new PropertyValue()
    {
        PropertyBaseId = propBase.PropertyBaseId,
        BinaryValue = command,
        StringValue = state
    };

    entities.PropertyValues.Insert(entities, propValue);
    entities.SaveChanges();

    return propValue;
}
```

PropertyValue ScServiceStatus()

Although this is a `HTTP GET` command, it can be seen as a helper method. Its only function is to check whether the service is connected or not (see Listing 4.10), which is used by the Android application to indicate the backend connection state (see Figure 5.2).

Listing 4.10 DataService.scv.cs: ScServiceStatus

```
public PropertyValue ScServiceStatus()
{
    return ScCommandToPropVal(null, "ScVerify", "TRUE");
}
```

PropertyValue ScGetChallenge()

This method is used during the authentication. The smartphone initiates authentication by calling `ScGetChallenge`.

An initial `ScReset` resets all imortant values to a predefined state. The Keygen class is used to get the Elliptic-Curve key-pairs from the encrypted key-pair files. Then, x_P is set with the curve's basepoint. The Authentication class is initialized using the key pair, and the nonce λ is generated from a new EC key pair. Using the method `GetNewChallenge`, the challenge x_A is created by a Montgomery Multiplication. The method returns the already back-transformed result, so it is necessary to transform the result into the Montgomery domain again. Converted to a byte array, the challenge is build into an unencrypted message using the CommandBuilders `Build` method. Finally, the `ScCommandToPropVal` method returns the encrypted message as PropertyValue. Listing 4.11 shows the implementation of `ScGetChallenge`.

Listing 4.11 DataService.scv.cs: ScGetChallenge

```
public PropertyValue ScGetChallenge()
{
    ScReset();
    Keygen keygen = new Keygen();
    FileStream EcPublicFile = new FileStream(@"D:/MetaSec_/Robert/trunk/
        detego.Surveyor/Service2/detego.Service.WCF/SecureChannel/
        EcPublicKey", FileMode.Open, FileAccess.Read);
    FileStream EcPrivateFile = new FileStream(@"D:/MetaSec_/Robert/trunk/
        detego.Surveyor/Service2/detego.Service.WCF/SecureChannel/
        EcPrivateKey", FileMode.Open, FileAccess.Read);
    AsymmetricCipherKeyPair authEcKeyPair = keygen.GetKeysFromFiles(
        EcPublicFile, EcPrivateFile);

    var xP = keygen.GetBasepoint().X.ToBigInteger();
    Authentication auth = new Authentication();
    auth.ecKeyPair = authEcKeyPair;

    lambda = ((ECPublicKeyParameters)(keygen.GenerateEcKeyPair().Public)).Q
        .X.ToBigInteger(); //random lambda
    BigInteger xA = auth.GetNewChallenge(lambda, xP);

    byte[] xA_ = Montgomery.Transform(xA).ToByteArray();
    byte[] lambda_ = lambda.ToByteArray();
    byte[] xP_ = xP.ToByteArray();

    byte[] challenge = new byte[26];
    System.Buffer.BlockCopy(xA_, 0, challenge, 25-xA_.Length, xA_.Length);

    byte[] challengedata = new byte[challenge.Length + 3];
    challengedata[0] = (byte)challenge.Length;
    challengedata[1] = (byte)challenge.Length;
    System.Buffer.BlockCopy(challenge, 0, challengedata, 3, challenge.
        Length);

    byte[] command = CommandBuilder.Build(CommandBuilder.CommandType.
        CHALLENGE, challengedata, false);

    return ScCommandToPropVal(command, "ScGetChallenge", null);
}
```

PropertyValue ScVerifyResponse(response_string)

This is the second method used during authentication, which is called with the tags `response_string` as parameter.

Similar to the `ScGetChallenge` method, the `ScVerifyResponse` loads the RSA key-pair from encrypted files and initializes the Authentication class. The response message is extracted from the PropertyValue object and split into response x_B, public key x_T and signature s_T. The signature s_T is verified to guarantee that x_T is a legitimate public key created by a legitimate tag. After a successful validation, the check value x_C is calculated with a Montgomery Multiplication of λ with x_T. Finally, if the check value x_C is equal to the response x_B, the authentication was successful and a PropertyValue with the state *TRUE* is returned.

The method can return an error in form of a PropertyValue with status *FALSE*. This error is returned if the `response_string` or the extracted response is *null*, if the response structure does not fit the protocol, if the signature could not be verified or if the comparison of $(x_C = x_B)$ shows inequality.

Listing 4.12 shows the methods implementation[1].

[1]In the late stages of testing it was clear that the tag did not return the correct response. It differed from the correct one in 2 out of 26 bytes and those two were computationally connected. The error could not be found in time, therefore a workaround that calculated the correct response, was implemented on the backend-side.

Listing 4.12 DataService.scv.cs: ScVerifyResponse

```
public PropertyValue ScVerifyResponse(PropertyValue response_string)
{
    if (response_string.StringValue != null) ScCommandToPropVal(null, "
        ClientLocation", response_string.StringValue);

    Keygen keygen = new Keygen();
    FileStream RsaPublicFile = new FileStream(@"D:/MetaSec_/Robert/trunk/
        detego.Surveyor/Service2/detego.Service.WCF/SecureChannel/
        RsaPublicKey", FileMode.Open, FileAccess.Read);
    FileStream RsaPrivateFile = new FileStream(@"D:/MetaSec_/Robert/trunk/
        detego.Surveyor/Service2/detego.Service.WCF/SecureChannel/
        RsaPrivateKey", FileMode.Open, FileAccess.Read);
    AsymmetricCipherKeyPair authRsaKeyPair = keygen.GetKeysFromFiles(
        RsaPublicFile, RsaPrivateFile);

    var xP = keygen.GetBasepoint().X.ToBigInteger();

    Authentication auth = new Authentication();
    auth.rsaKeyPair = authRsaKeyPair;

    if (response_string == null) return ScCommandToPropVal(null, "ScVerify"
        , "FALSE");
    byte[] response = response_string.BinaryValue;
    if (response == null) return ScCommandToPropVal(null, "ScVerify", "
        FALSE");

    if (response[0] != 0x02 || response[1] != 0x00 || response[3] != 0xBB
        || response[8] != 0x00) return ScCommandToPropVal(null, "ScVerify",
        "FALSE");
    byte[] xB_ = new byte[1 + response[9] - 1];
    byte[] xT_ = new byte[1 + response[10]];
    byte[] sT_ = new byte[response[11]];
    System.Buffer.BlockCopy(response, 13, xB_, 1, xB_.Length - 1);
    System.Buffer.BlockCopy(response, 14 + xB_.Length - 2, xT_, 1, xT_.
        Length - 1);
    System.Buffer.BlockCopy(response, 14 + xB_.Length - 2 + xT_.Length - 1,
         sT_, 0, sT_.Length);

    //Fix for the calculation problem
    int fix = (int)xB_[xB_.Length - 1];
    xB_[1] = (byte)(xB_[1] + fix);
    int tofix = xB_[16] * 256 + xB_[17];
    int fixd = tofix - fix;
    xB_[16] = (byte)(fixd / 256);
    xB_[17] = (byte)(fixd);
    byte[] xB_fix = new byte[xB_.Length - 1];
    System.Buffer.BlockCopy(xB_, 0, xB_fix, 0, xB_fix.Length);
    xB_ = xB_fix;

    BigInteger sT = new BigInteger(sT_);
    BigInteger xB = new BigInteger(xB_);
    BigInteger xT = new BigInteger(xT_);

    bool verified = auth.VerifySignature(xT, sT);
    if (verified)
    {
        BigInteger xC = Montgomery.Multiplication(lambda, xT);
        byte[] xC_ = xC.ToByteArray();
        if (xC.Equals(xB)) return ScCommandToPropVal(null, "ScVerify", "
            TRUE");
        return ScCommandToPropVal(null, "ScVerify", "FALSE");
    }
    return ScCommandToPropVal(null, "ScVerify", "FALSE");

}
```

PropertyValue ScGetReadCommand()

As start of the *pedigree-read* operation, the smartphone calls this method to get the first read request for the tag. After an initial reset, the command for reading the first 100 bytes is encrypted and embedded into a PropertyValue with the additional state *START* (see Listing 4.13).

Listing 4.13 DataService.scv.cs: ScGetReadCommand

```
public PropertyValue ScGetReadCommand()
{
    ScReset();
    byte[] command = CommandBuilder.Build(CommandBuilder.CommandType.READ,
        new byte[] { 0x00, (byte)MAXTRANSMISSION, (byte)(offset >> 8), (byte
        )(offset & 0xFF) }, true);
    return ScCommandToPropVal(command, "ScGetReadCommand", "START");
}
```

byte[] ScGetNextReadCommand()

Similar to `ScGetReadCommand`, this method returns the read command for the next 100 bytes of the pedigree. The offset gets incremented and the command is returned (see Listing 4.14). This method is *private* and only called from inside the `ScRead` method.

Listing 4.14 DataService.scv.cs: ScGetNextReadCommand

```
private byte[] ScGetNextReadCommand()
{
    offset += MAXTRANSMISSION;
    return CommandBuilder.Build(CommandBuilder.CommandType.READ, new byte[]
         { 0x00, (byte)MAXTRANSMISSION, (byte)(offset >> 8), (byte)(offset &
         0xFF) }, true);
}
```

PropertyValue ScRead(read_response)

Whenever a part of the pedigree is read from the tag, the smartphone transmits this `read_response` as parameter of the method `ScRead`. Each call of `ScRead` stores an additional part of the pedigree, until it is completed. Hence, the method is called multiple times during one *pedigree-read*. On successful execution, the returned PropertyValue includes a positive `state`. If there are further pedigree packages the state is *CONT*, if the reading is completed the state is *WRITEREADY*.

The initial reset has already been executed in the previous `ScGetReadCommand` method. First, the `read_response` is checked. If the included response is not `null` and its structure matches the protocol, it is parsed for its length. The encrypted part, the timestamp and the nonce are extracted from the response. The encrypted part gets decrypted and the included command byte, the package info as well as the pedigree part are checked. The package-info byte is used to indicate if there are still parts of the pedigree to be read from the tag. If the current part is the last part, the package-info byte includes the length of the data, otherwise it is `0xFF`. The pedigree part is appended to the already read parts in the `read_pedigree_buffer` variable. If there are further pedigree parts, the method returns the next read command (`ScGetNextReadCommand`) with the state *CONT*.

If the last pedigree part is read, the pedigree is fully stored in the buffer variable. Then, the appended timestamp and signature are extracted and the signature is verified. An additional Elliptic-Curve key-pair together with the Elliptic Curve Digital Signature Algorithm (ECDSA) is used for the pedigree signature. Therefore, the key pair is loaded from stored encrypted files. The class Authentication.cs provides the methods to verify, and to later recreate, the signature. After a successful verification, the pedigree is parsed using methods from the EPedigreeHandler class. The database is searched for products matching the pedigree information. A new timestamp is appended to the pedigree data and the concatenation is signed. The full pedigree is now stored in the buffer variable `write_pedigree_buffer` and the `ScRead` method returns with the state `WRITEREADY`. The smartphone application continues the *pedigree-read* process by calling `ScWrite`. The complete source code for the `ScRead` method can be found in the appendix in Listing B.6.

Possible errors and error messages include:

- `read_response` is *null*
  state = *NULLINPUT*
- Response included in `read_response` is *null*
  state = *NULLINPUTBYTES*
- Response structure does not fit protocol structure
  state = *MESSAGEERROR*
- Decrypted bytes are *null*
  state = *READERROR*
- Pedigree signature could not be verified
  state = *SIGERROR*

- New signature could not be verified
  state = *SIG_GEN_ERROR*

PropertyValue ScWrite()

After the pedigree has been read and verified, a new pedigree with a new timestamp and a new signature is stored on the tag. The method `ScWrite` builds the commands to write 100-byte parts of the pedigree. After multiple method calls, the new pedigree should have been written. Similar to the `ScRead` method, the returned state is either *CONT* if there are further packages, or *LAST* if the package was the last package. The method is called a last time to return the state *ALL_WRITTEN*. Listing 4.15 shows the method's implementation.

Listing 4.15 DataService.scv.cs: ScWrite

```
public PropertyValue ScWrite()
{
    string returnstate = "ERROR";

    if (write_pedigree_buffer.Length < offset) return ScCommandToPropVal(
        null, "ScWrite", "ALL_WRITTEN");

    byte[] package;
    if (write_pedigree_buffer.Length - offset >= MAXTRANSMISSION)
    {
        package = new byte[MAXTRANSMISSION];
        package[0] = 0xFF;
        returnstate = "CONT";
    }
    else
    {
        package = new byte[write_pedigree_buffer.Length - copyOffset + 1];
        package[0] = (byte)(write_pedigree_buffer.Length - copyOffset);
        returnstate = "LAST";
    }

    System.Buffer.BlockCopy(write_pedigree_buffer, copyOffset, package, 1,
        package.Length - 1);

    byte[] command = CommandBuilder.Build(CommandBuilder.CommandType.WRITE,
         (new byte[] { 0x00, (byte)package.Length, (byte)(offset >> 8), (
        byte)(offset & 0xFF) }).Concat(package).ToArray(), true);

    offset += MAXTRANSMISSION;
    copyOffset += MAXTRANSMISSION - 1;

    return ScCommandToPropVal(command, "ScWrite", returnstate);
}
```

Product ScFinalize()

The last method for the *pedigree-read* process it the `ScFinalize` method. Only after successful reading and writing of the pedigree from and to the tag, this method returns the Product object according to the pedigree information read (see Listing 4.16).

The returned object is not encrypted and can therefore be read and displayed by the smartphone application.

Listing 4.16 DataService.scv.cs: ScFinalize

```
public Product ScFinalize()
{
    //command = [PID|ENC|LEN|LEN|TSP|TSP|TSP|TSP|CMD|DATA..TAG]
    //          { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 .. +16]

    if (write_pedigree_buffer.Length == 0 || read_pedigree_buffer.Length ==
         0 || current_productid == null) return null;

    Product product = entities.Products.SingleOrDefault(
       p =>
       p.ProductId == current_productid);

    if (product == null) return null;

    ScReset();

    return product;
}
```

4.3 Android Application

This section describes all implementation details concerning the Android smartphone application. The application is based on an existing framework which facilitates basic communication between the smartphone and the RFID-tag prototype. The communication to the backend is additionally secured using TLS. In the Secure Channel Application the authentication process and the pedigree-read process are combined to offer a fluent user experience. Section 3.4 describes the design of the Android application. The class diagram is depicted in Figure 4.1 whereas the calling structure for the service methods is shown in Figure A.3.

Tag Communication: The communication between the smartphone and the tag is established over NFC and based on the NFC Data Exchange Format (NDEF). Tag-related commands are stored in a queue and processed one after another. A periodically executed `IsDeviceAliveCommand` ensures that the tag is in the reader's field and reachable. The rest of this section will focus on the backend connection.

TLS: Transport Layer Security (TLS) is used to secure the backend communication. Section 2.3.1 describes the functionality of TLS in detail. The implementation of this security feature has already been performed during a previous master's project [Ple14].

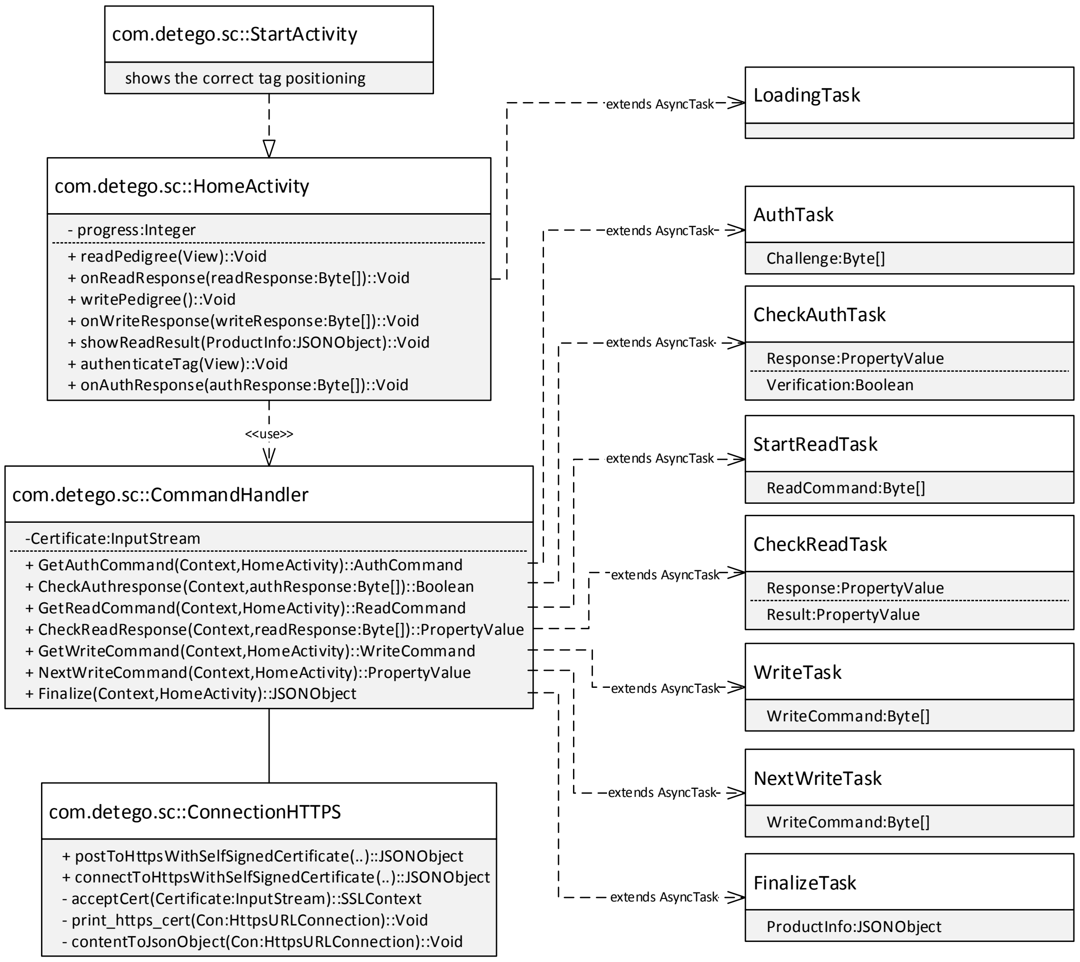

com.detego.sc::StartActivity
shows the correct tag positioning
com.detego.sc::HomeActivity
- progress:Integer
+ readPedigree(View)::Void
+ onReadResponse(readResponse:Byte[])::Void
+ writePedigree()::Void
+ onWriteResponse(writeResponse:Byte[])::Void
+ showReadResult(ProductInfo:JSONObject)::Void
+ authenticateTag(View)::Void
+ onAuthResponse(authResponse:Byte[])::Void
<<use>>
com.detego.sc::CommandHandler
-Certificate:InputStream
+ GetAuthCommand(Context,HomeActivity)::AuthCommand
+ CheckAuthresponse(Context,authResponse:Byte[])::Boolean
+ GetReadCommand(Context,HomeActivity)::ReadCommand
+ CheckReadResponse(Context,readResponse:Byte[])::PropertyValue
+ GetWriteCommand(Context,HomeActivity)::WriteCommand
+ NextWriteCommand(Context,HomeActivity)::PropertyValue
+ Finalize(Context,HomeActivity)::JSONObject
com.detego.sc::ConnectionHTTPS
+ postToHttpsWithSelfSignedCertificate(..)::JSONObject
+ connectToHttpsWithSelfSignedCertificate(..)::JSONObject
- acceptCert(Certificate:InputStream)::SSLContext
- print_https_cert(Con:HttpsURLConnection)::Void
- contentToJsonObject(Con:HttpsURLConnection)::Void
extends AsyncTask
LoadingTask
AuthTask
Challenge:Byte[]
CheckAuthTask
Response:PropertyValue
Verification:Boolean
StartReadTask
ReadCommand:Byte[]
CheckReadTask
Response:PropertyValue
Result:PropertyValue
WriteTask
WriteCommand:Byte[]
NextWriteTask
WriteCommand:Byte[]
FinalizeTask
ProductInfo:JSONObject

Figure 4.1: Class Diagram of the Secure Channel Android Application. Only thesis related details shown.

4.3.1 Class ConnectionHTTPS

The connection to the backend is established over Wi-Fi using the secure protocol HTTPS. `HTTP` methods `GET` and `POST` are used to contact the backend. All data is sent in the JavaScript Object Notation (JSON) format (see Section 3.2.2 for details).

The class `ConnectionHTTPS` implements the methods to connect to the backend over HTTPS (Listing 4.17). The used self-signed certificate is part of the application package and has to be put into the key store of the smartphone.

Listing 4.17 Overview of the class ConnectionHTTPS

```java
public class ConnectionHTTPS {
  private SSLContext AcceptCert(InputStream fin)
  private JSONObject ContentToJsonObject(HttpsURLConnection con)
  public JSONObject ConnectToHttpsWithSelfSignedCertificate(URL url,
      InputStream fin, String sessionId)
  public JSONObject PostToHttpsWithSelfSignedCertificate(URL url,
      InputStream fin, String sessionId, PropertyValue payload)
  private void Print_https_cert(HttpsURLConnection con) // used for
      debugging
  }
```

SSLContext AcceptCert(certificate)

This helper method returns an `SSLContext` object which is needed for the secured TLS connection. As mentioned before, the certificate has to be deployed into the phone's key store. A new key store is created with the used certificate as the only entry. This key store is only accessible within the application. Finally, the `SSLContext` is created and returned to the calling method. Listing 4.18 shows the relevant parts of the method.

Listing 4.18 AcceptCert

```java
private SSLContext AcceptCert(InputStream fin) {
  CertificateFactory cf;
  try {
      cf = CertificateFactory.getInstance("X.509");
      Certificate ca = cf.generateCertificate(fin);
      String keyStoreType = KeyStore.getDefaultType();
      KeyStore keyStore = KeyStore.getInstance(keyStoreType);
      keyStore.load(null, null);
      keyStore.setCertificateEntry("ca", ca);
      String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
      TrustManagerFactory tmf = TrustManagerFactory
              .getInstance(tmfAlgorithm);
      tmf.init(keyStore);
      SSLContext context = SSLContext.getInstance("TLS");
      context.init(null, tmf.getTrustManagers(), null);
      return context;
  } catch .. (additional error handling)
}
```

JSONObject ContentToJsonObject(HttpsUrlConnection)

The two following methods use this helper method to convert the HTTP responses to a `JSONObject`. The responses already are in JSON format, but need to be converted to the correct object format.

JSONObject ConnectToHttpsWithSelfSignedCertificate(url, certificate, sessionId)

The actual connection to the backend via `HTTP GET` commands is implemented in this method. An object of the type `HttpsUrlConnection` is created and the properties of the connection are then set. An option to include a session ID is also included. However, the session id has no significance in this prototype application yet. The object method `HttpsUrlConnection.getInputStream()` returns the actual response of the request. Therefore, it initiates the communication and waits for any response. Finally, the response is converted to a `JSONObject` and returned.

JSONObject PostToHttpsWithSelfSignedCertificate(url, certificate, sessionId, payload)

Similar to the `ConnectToHttpsWithSelfSignedCertificate` method, this method realizes the actual communication using HTTP POST commands. The functionality is equivalent to the GET request, except for the additional payload. Listing 4.19 shows the code of the `PostToHttpsWithSelfSignedCertificate` method.

Listing 4.19 PostToHttpsWithSelfSignedCertificate

```java
public JSONObject PostToHttpsWithSelfSignedCertificate(URL url, InputStream
     fin, String sessionId, PropertyValue payload) {
  try {
    SSLContext SSLctx = acceptCert(fin);
    HttpsURLConnection myConn = (HttpsURLConnection) url
            .openConnection();
    myConn.setSSLSocketFactory(SSLctx.getSocketFactory());
    myConn.setRequestProperty("SessionId", sessionId);

    while (!myConn.getDoOutput()) {
        myConn.setDoOutput(true);
    }
    myConn.setRequestMethod("POST");
    myConn.setRequestProperty("User-Agent", "Mozilla/5.0 ( compatible ) ");
    myConn.setRequestProperty("Accept", "*/*");
    myConn.setRequestProperty("Content-Type", "application/json");

    DataOutputStream dataOut = new DataOutputStream(
            myConn.getOutputStream());
    dataOut.writeBytes(new Gson().toJson(payload));
    dataOut.close();

    return ContentToJsonObject(myConn);

  } catch .. (additional error handling)
}
```

4.3.2 Class CommandHandler

The `CommandHandler` class is used to build the correct requests for the backend. Each request is built in a corresponding method which initiates a task for each connection. These tasks are derived from the `AsyncTask` object and use the `ConnectionHTTPS` methods to connect to the backend. As the methods and tasks are similar to each other, an overview is given in Table 4.1. Figure A.3 shows the calling context between the tasks of the `CommandHandler` class and the service methods implemented at the backend. Listing B.9 shows the entire code in the appendix.

Operation	Method	Task	Service Method
Authentication Get Challenge	GetAuthCommand	AuthTask	ScGetChallenge
Authentication Check Response	CheckAuthResponse	CheckAuthTask	ScVerifyResponse
Pedigree-Read Begin Reading	GetReadCommand	StartReadTask	ScGetReadCommand
Pedigree-Read Continue Reading	CheckReadResponse	CheckReadTask	ScRead
Pedigree-Read Begin Writing	GetWriteCommand	WriteTask	ScWrite
Pedigree-Read Continue Writing	NextWriteCommand	NextWriteTask	ScCheckWrite
Pedigree-Read Final Request	Finalize	FinalizeTask	ScFinalize

Table 4.1: Overview of the Methods and Tasks of the CommandHandler Class, the Underlying Operations and the Called Service Methods.

4.3.3 Class HomeActivity

This is the main class in the Secure Channel Android application. As there are only two buttons in the user interface, the implemented functionality for each is described separately.

Authentication: Related methods in `HomeActivity`:

- `AuthenticateTag`
- `OnAuthResponse`

After the application is started, only the "Authenticate" button is visible (see Figure 5.2 (center)). After the button is clicked, the method `AuthenticateTag` is called. The method itself calls the `CommandHandler.GetAuthCommand` method, which returns the encrypted command from the backend. The command is then passed to the

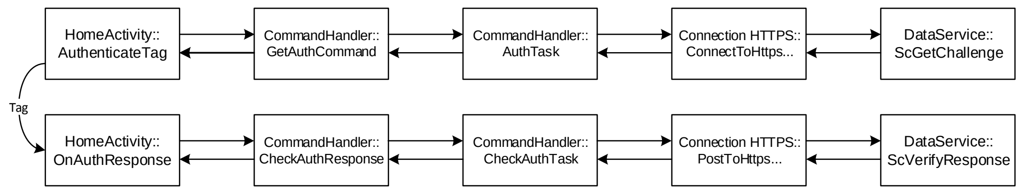

HomeActivity:: AuthenticateTag
CommandHandler:: GetAuthCommand
CommandHandler:: AuthTask
Connection HTTPS:: ConnectToHttps...
DataService:: ScGetChallenge
Tag
HomeActivity:: OnAuthResponse
CommandHandler:: CheckAuthResponse
CommandHandler:: CheckAuthTask
Connection HTTPS:: PostToHttps...
DataService:: ScVerifyResponse

Figure 4.2: Methods and Tasks Executed During Authentication

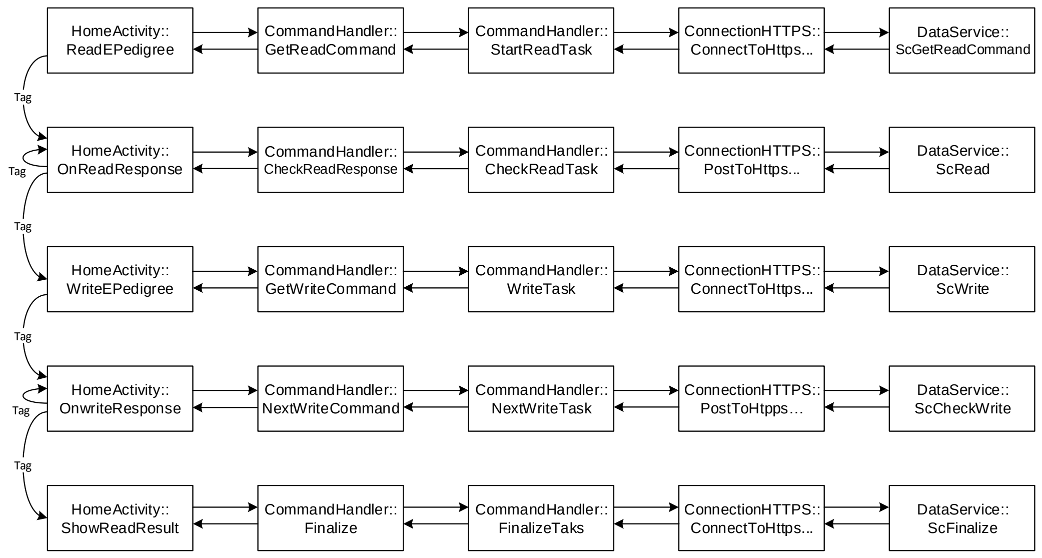

HomeActivity:: ReadEPedigree
CommandHandler:: GetReadCommand
CommandHandler:: StartReadTask
ConnectionHTTPS:: ConnectToHttps...
DataService:: ScGetReadCommand
Tag
HomeActivity:: OnReadResponse
CommandHandler:: CheckReadResponse
CommandHandler:: CheckReadTask
ConnectionHTTPS:: PostToHttps...
DataService:: ScRead
Tag
Tag
HomeActivity:: WriteEPedigree
CommandHandler:: GetWriteCommand
CommandHandler:: WriteTask
ConnectionHTTPS:: ConnectToHttps...
DataService:: ScWrite
Tag
HomeActivity:: OnwriteResponse
CommandHandler:: NextWriteCommand
CommandHandler:: NextWriteTask
ConnectionHTTPS:: PostToHtpps...
DataService:: ScCheckWrite
Tag
Tag
HomeActivity:: ShowReadResult
CommandHandler:: Finalize
CommandHandler:: FinalizeTaks
ConnectionHTTPS:: ConnectToHttps...
DataService:: ScFinalize

Figure 4.3: Methods and Tasks Executed During Pedigree-Read

tag communicating part of the application (`OnUserInitiatedCommand`), where it gets packaged into an NDEF message and is then sent to the tag. The tag response is parsed out of the NDEF message and passed to the method `OnAuthResponse(authResponse)`. The response is sent to the backend and the backend-response is returned as boolean. If the authentication was successful, the “Authenticate” button gets disabled, an indicator icon shows the success to the user and the “Read-Pedigree” button is shown (Figure 5.3 (left)). If the authentication fails, the indicator icon becomes red and the user can try again (Figure 5.2 (right)).

Figure 4.2 shows the methods and tasks that are called one after another and returning responses back to the individual caller.

Pedigree-Read: Related methods and background tasks in `HomeActivity`:

- `ReadEPedigree`
- `OnReadResponse`
- `WriteEPedigree`
- `OnWriteResponse`
- `ShowReadResult`
- `LoadingTask`

Clicking the "Pedigree-Read" button initiates the reading and writing of the pedigree (Figure 5.3 (left)). Since the entire operation takes several seconds, the user is presented with a loading screen (Figure 5.3 (center)). The included progress bar is updated periodically, as soon as the execution has reached certain stages of the operation.

The smartphone application starts the *pedigree-read* by calling the `readEPedigree` method. As shown in Table 4.1, the tasks for getting the first read command from the backend are executed one after another. The backend responds with the encrypted command. This encrypted command gets packaged into an NDEF message and is then sent to the tag (`OnUserInitiatedCommand(readCommand)`). The tag responds with the first part of the pedigree, encrypted and included in an NDEF message. The application extracts the encrypted tag-response and forwards it to the backend (*OnReadResponse*), by calling the `CommandHandler.CheckReadResponse` method. The backend itself returns the encrypted command for the next pedigree part to read, and so forth. Further parts are read until the pedigree is fully stored at the backend. The now outdated pedigree is verified and the timestamp and signature are renewed. The backend responds with a status message of *WRITEREADY*.

The application then calls the `WriteEPedigree` method. The `CommandHandler` starts the related task and sends the `HTTP GET` request for the encrypted write command to the backend (`GetWriteCommand`). The encrypted command, including the first pedigree part, is returned by the backend and then passed to the `OnUserInitiatedCommand`. The command is sent to the tag and the tag responds with a write-successful message. This response message is then forwarded to the backend and the next package is fetched. This continues until the pedigree is written completely. The backend responds to the next `ScGetWrite` call with the status message *ALL_WRITTEN*. Now, the `ShowReadResult` method of the `HomeActivity` is called. This calls all "Finalize" methods and tasks to get the product data from the backend. The final response is unencrypted and can be read by the Android application.

The application ends the loading task and shows the product information on the screen (Figure 5.3 (right)). Figure 4.3 shows the methods and tasks called during the *pedigree-read* operation.

Chapter 5

Results

The proposed system is designed for a specific use case, which is giving a potential customer the possibility to distinguish original products from counterfeits. The system consists of three main parts: an RFID-tag prototype, a backend system and an NFC-enabled smartphone. The customer's interface is the smartphone with an installed Android application. In the background, the smartphone is used as bridge technology for communication between backend and tag. To significantly reduce the risks imposed by a publicly available Android application, private communication between server and tag is always encrypted. Figure 5.1 shows an overview of the Secure Channel System.

The following sections give details on the implemented Android application and show a timing analysis of the entire system.

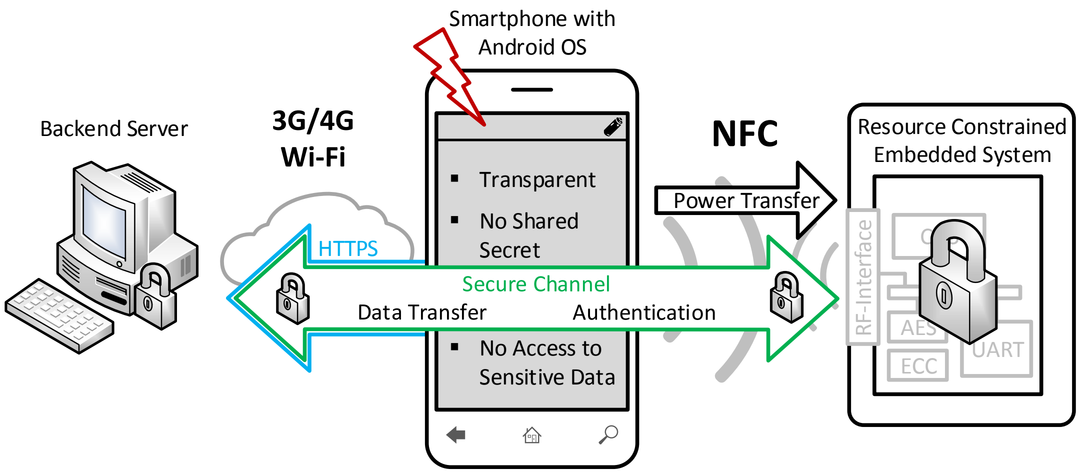


Figure 5.1: Overview of the Secure Channel System

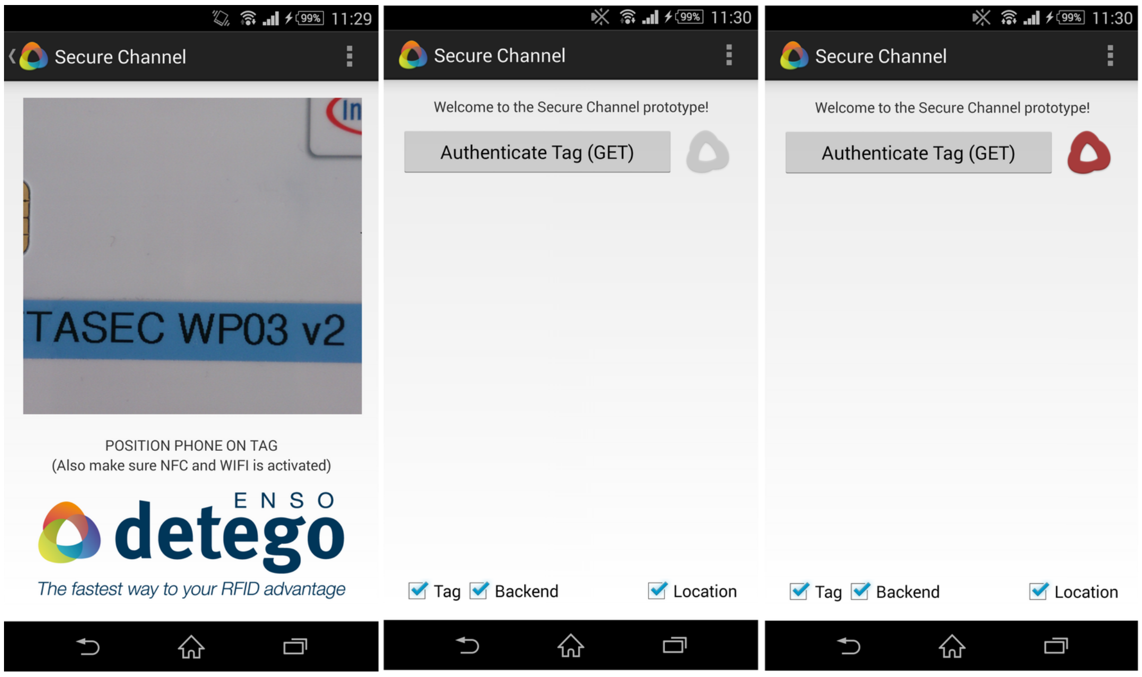

Secure Channel
TASEC WP03 v2
POSITION PHONE ON TAG
(Also make sure NFC and WIFI is activated)
ENSO detego
The fastest way to your RFID advantage
Welcome to the Secure Channel prototype!
Authenticate Tag (GET)
Tag
Backend
Location

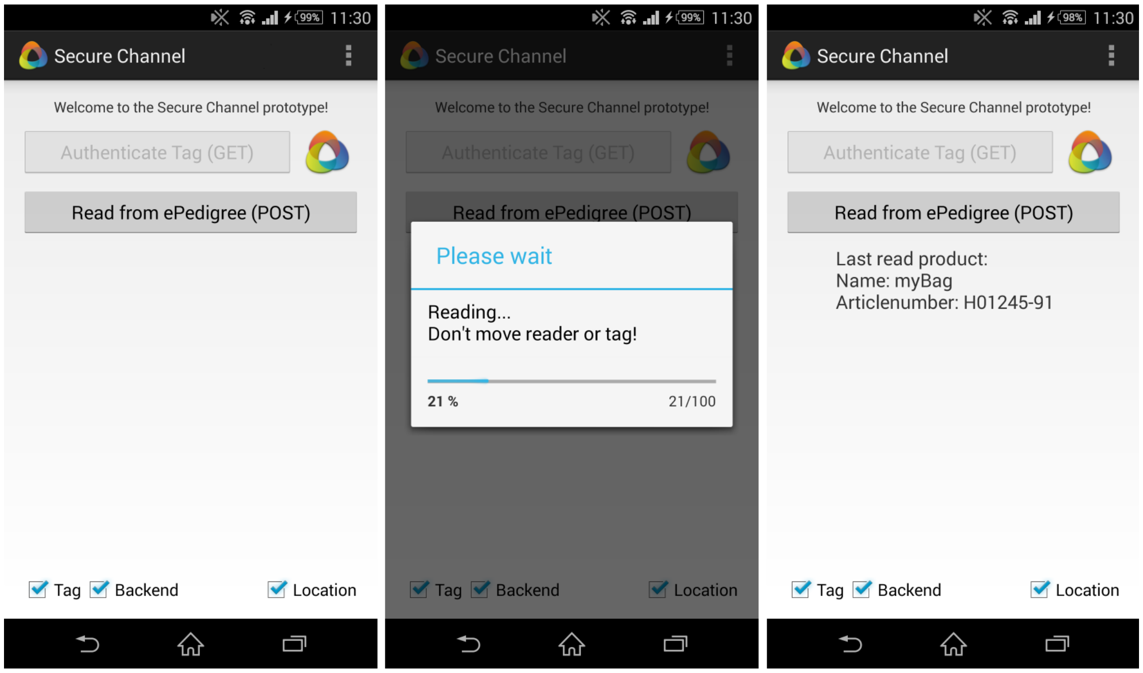
Figure 5.2: Screenshots of the Secure Channel Application: Start screen (left). After a supported tag is found, the user can start the authentication (center). If the authentication fails, the signal icon becomes red (right).

5.1 Android Application

The smartphone with the installed Android application is used as bridge technology, which enables the backend system to communicate with the RFID-tag prototype. This section shows the results of the application design and implementation.

To start the application, the user either clicks the designated icon on his smartphone or scans a tag directly. If started using the application icon, the start screen is shown, which directs the user to activate NFC and Wi-Fi on his smartphone. Moreover, it depicts how to position the tag prototype (see Figure 5.2). After a tag is scanned, the main screen is shown, which is designed to be intuitive and simplistic, thus giving the customer only information he really needs. At the bottom of the screen, check boxes indicate the state of tag and backend communication and if the smartphone is giving away location data to the application. The main screen supports all actions the user can execute within the application. The user can now start the authentication with a button click.

Next to the authentication button, a logo is indicating the authentication state, which is initially grey, red upon failure and coloured after a successful authentication (see Figure 5.3). A successful authentication also shows the button for the pedigree-read operation on the screen. Upon clicking this button, a dialog box shows the current state of the operation. After a successful pedigree-read execution, read product information is displayed on the main screen.



Welcome to the Secure Channel prototype!

Please wait

Reading...
Don't move reader or tag!

Last read product:
Name: myBag
Articlenumber: H01245-91

Figure 5.3: Screenshots of the Secure Channel Application: After a successful authentication the tag-data can be read (left). The process of the read-operation is shown in a process diaglog (center). After a successful read-operation product information is shown (right).

Execution Time	Iterations	Minimum	Maximum	Average	Standard Deviation
Authentication	240	0.68 s	2.15 s	1.21 s	0.23 s
Pedigree-Read	2 000	5.31 s	8.76 s	6.18 s	0.48 s
Read (4 cycles)	2 000	2.31 s	4.86 s	2.89 s	0.30 s
Write (4 cycles)	2 000	2.74 s	5.56 s	3.17 s	0.27 s

Table 5.1: Timing Analysis of Authentication and Pedigree-Read: The timings are measured during runtime within the Android application. Measurement is started upon executing the methods behind by the corresponding button click and stopped upon receiving the final message of the operation.

5.2 Timing Analysis

This section shows details on the timing behaviour of the Secure Channel System. The timings are measured during runtime within the Android application. Therefore, measurement affected the actual execution time.

There are two main operations for the application user to execute: the authentication and the pedigree-read. The measurement starts upon button click and ends after the final message had been received from the backend.

Authentication Figure 5.4 shows the time needed for 240 successful authentication operations. The average execution time is 1.21 s, with a standard deviation of 0.23 s (see Table 5.1). As the communicated messages only consist of insensitive data, this data is not encrypted. Therefore, authentication can be performed without delay of additional encryption overhead, in about one second.

Pedigree-Read Figure 5.5 depicts the timings of 2 000 successful pedigree-read operations. The execution time of the entire operation is marked with the black trace, whereas the blue and the red traces resemble the sum of read and write cycles respectively. One pedigree-read consists of four read and four write cycles. The entire operation executes in an average time of 6.18 seconds, with a standard deviation of 0.48 seconds (see Table 5.1). The execution time depends on the smartphone's CPU clock-frequency. This is shown in Figure 5.6. As the smartphone is constantly working during the measurements, the OS throttles the clock frequency to prevent overheating. This directly affects the execution time, since the execution time increases when the CPU frequency is reduced by the operating system.

Additionally, Figure 5.7 illustrates a timeline of one pedigree-read operation, where every timestamp marks points in time where a message is being sent to the tag. The first block marks the initial read-command request on backend, followed by four cycles of read operations. Then the first write-command is requested, followed by four cycles of writing data to the tag. Finally, the *Finalize* operation takes another 0.1 seconds to show the product information on the screen. Figure 5.8 shows a more detailed view of the pedigree-read timeline.

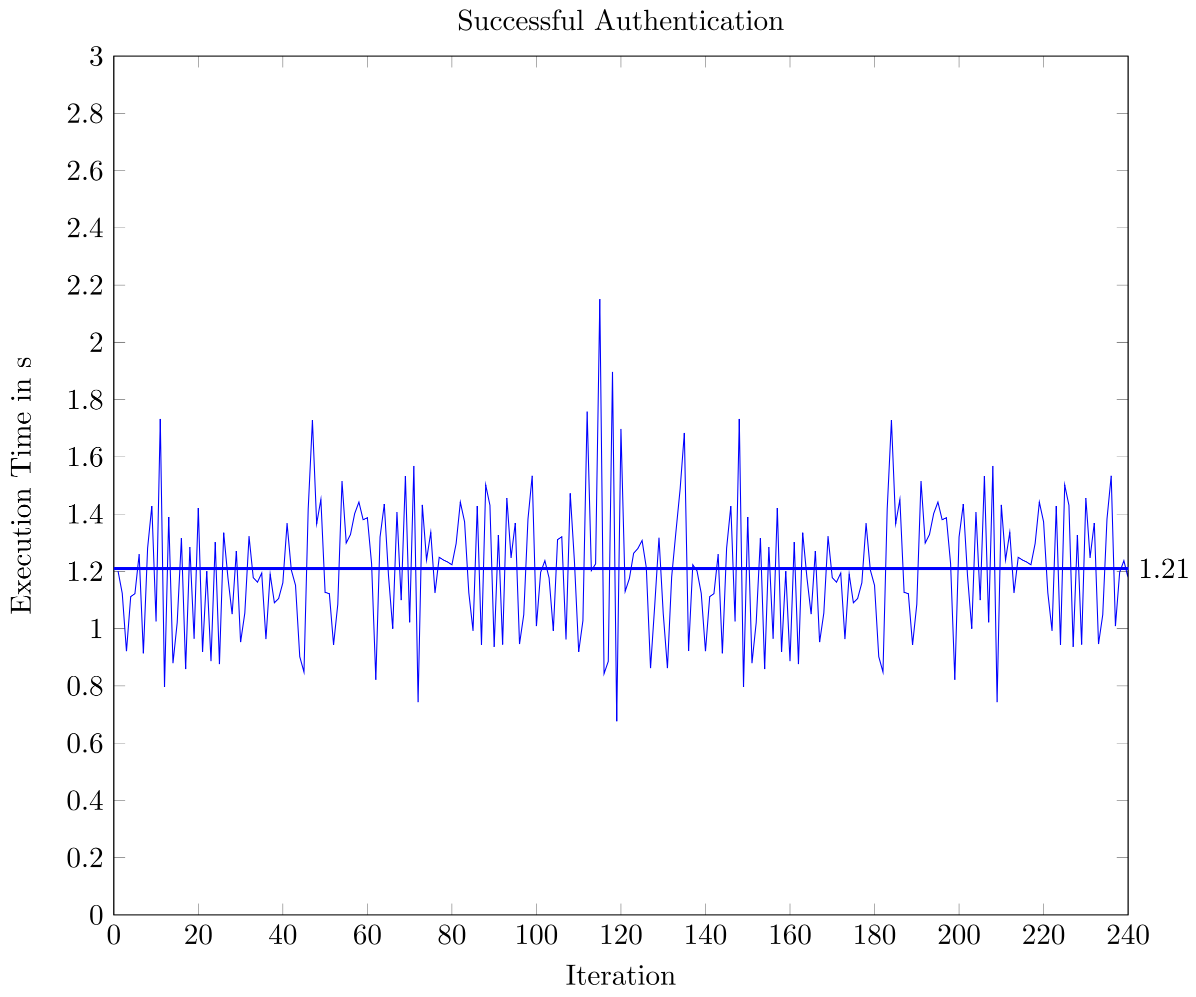


Figure 5.4: 240 Iterations of Successful Authentication Operations

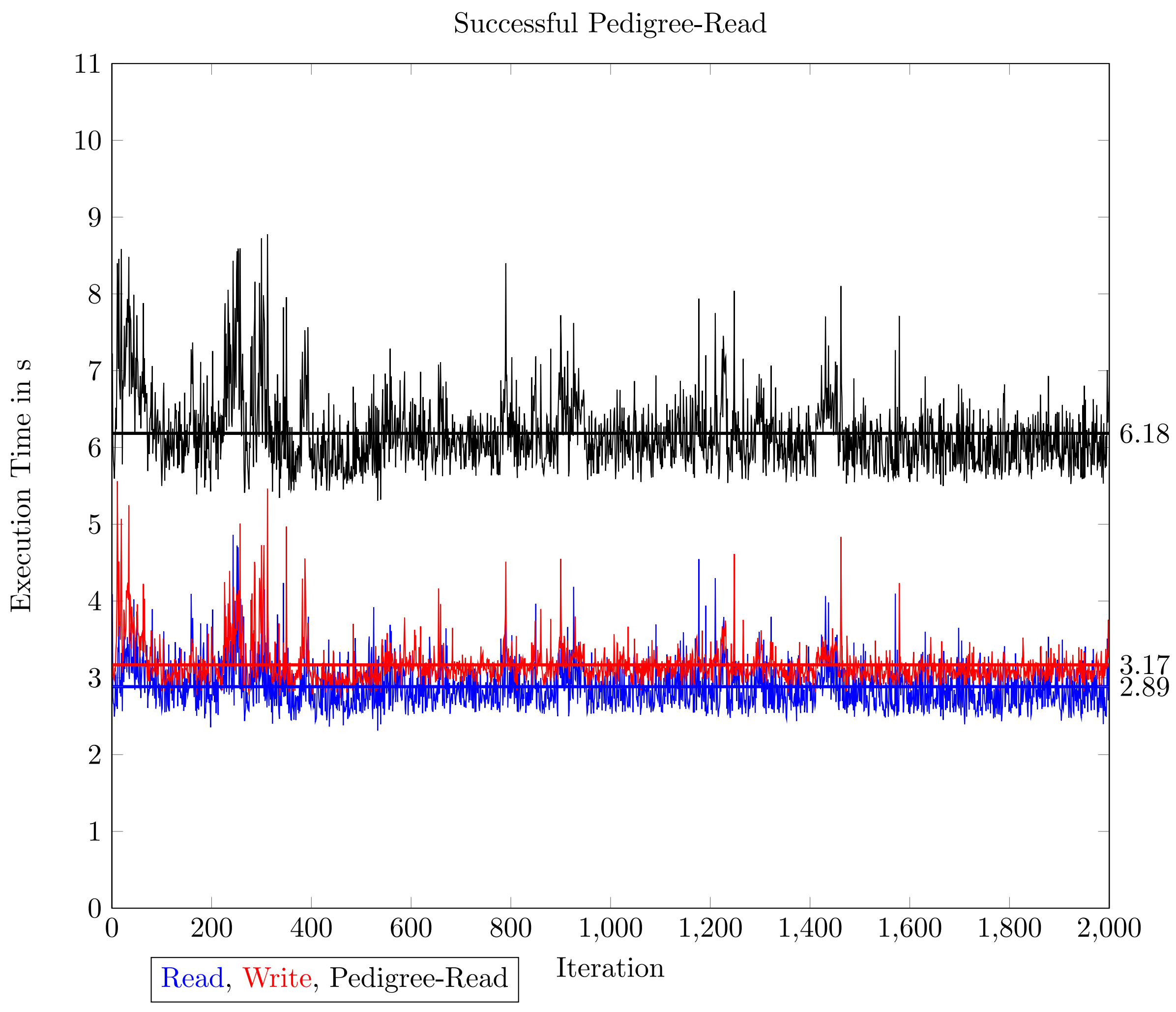


Figure 5.5: 2 000 Iterations of Successful Pedigree-Read Operations. Read and Write Operations Highlighted. One pedigree-read consists of four read and four write cycles.

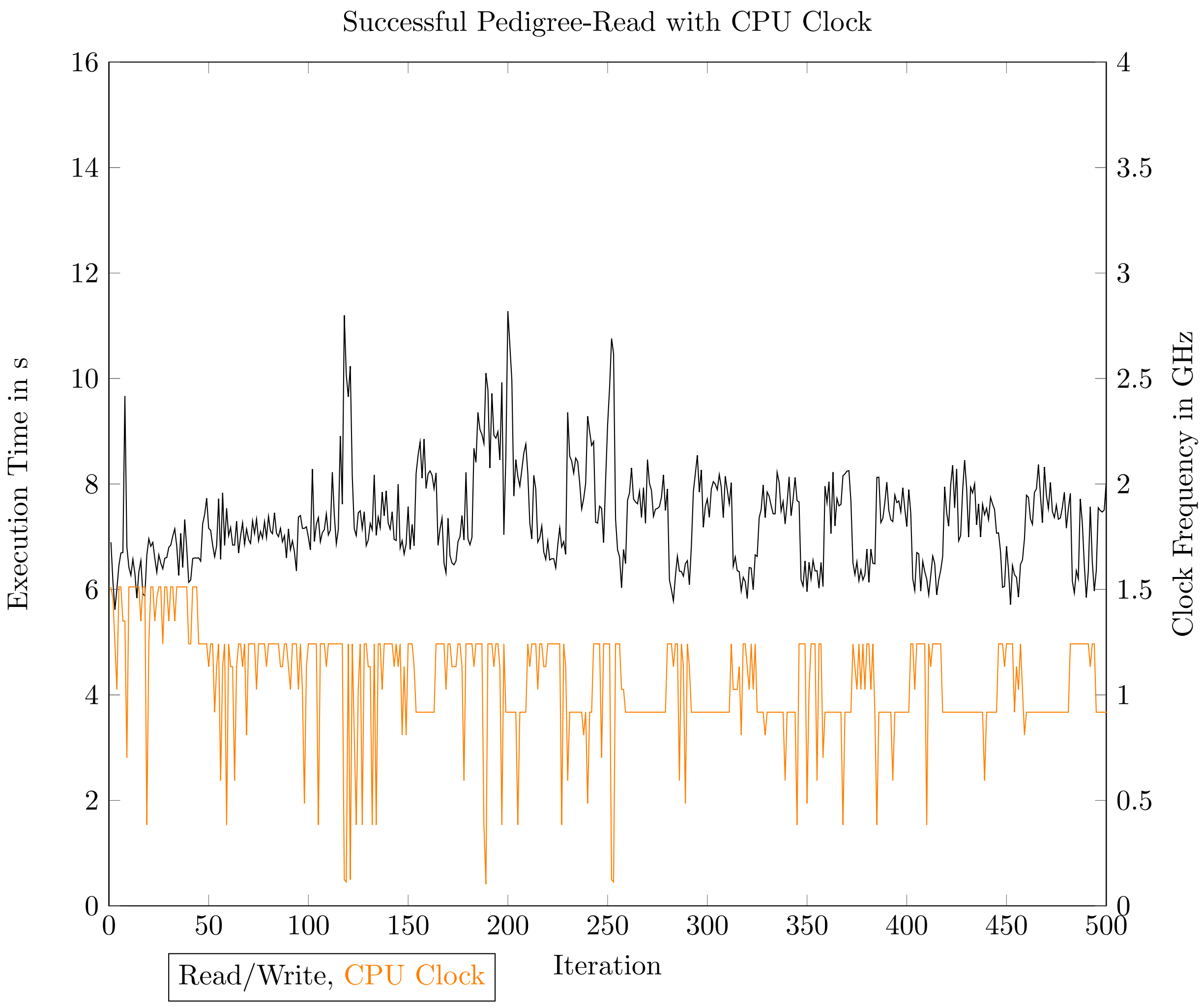


Figure 5.6: Timings and Smartphone CPU Clock-Frequencies of Successful Pedigree-Read Operations

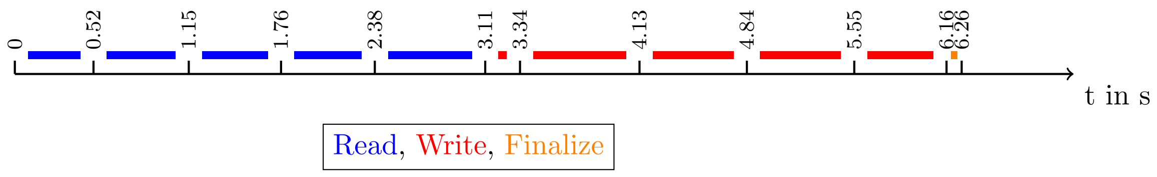


Figure 5.7: Execution of Pedigree-Read: Starting with the button click, timestamps mark points in time where a message is sent from smartphone to tag. One read-command request is followed by four read cycles, one write-command request and four write cycles.

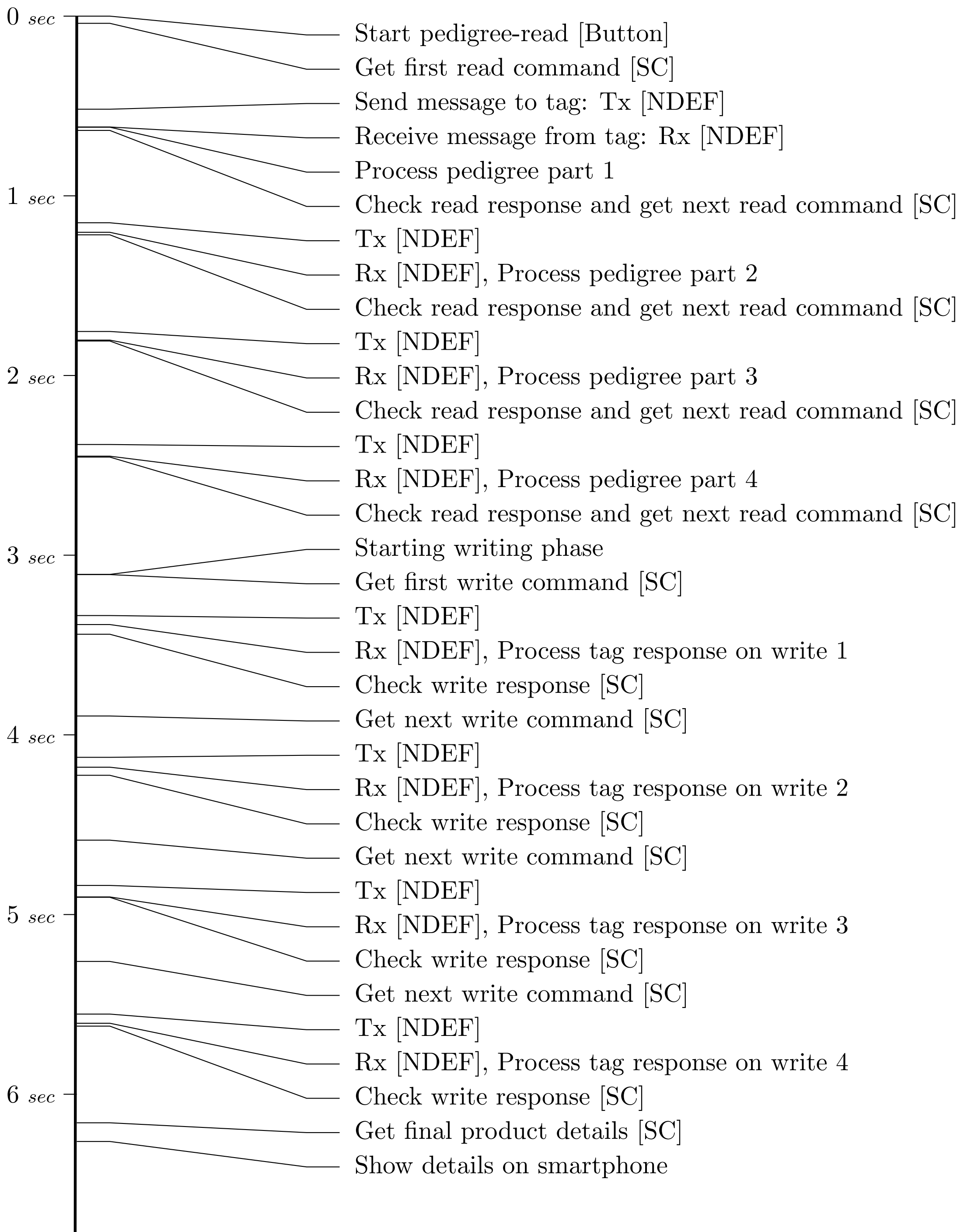

0 sec
1 sec
2 sec
3 sec
4 sec
5 sec
6 sec
Start pedigree-read [Button]
Get first read command [SC]
Send message to tag: Tx [NDEF]
Receive message from tag: Rx [NDEF]
Process pedigree part 1
Check read response and get next read command [SC]
Tx [NDEF]
Rx [NDEF], Process pedigree part 2
Check read response and get next read command [SC]
Tx [NDEF]
Rx [NDEF], Process pedigree part 3
Check read response and get next read command [SC]
Tx [NDEF]
Rx [NDEF], Process pedigree part 4
Check read response and get next read command [SC]
Starting writing phase
Get first write command [SC]
Tx [NDEF]
Rx [NDEF], Process tag response on write 1
Check write response [SC]
Get next write command [SC]
Tx [NDEF]
Rx [NDEF], Process tag response on write 2
Check write response [SC]
Get next write command [SC]
Tx [NDEF]
Rx [NDEF], Process tag response on write 3
Check write response [SC]
Get next write command [SC]
Tx [NDEF]
Rx [NDEF], Process tag response on write 4
Check write response [SC]
Get final product details [SC]
Show details on smartphone

Figure 5.8: Detailed Execution of Pedigree-Read: Tx and Rx remarks messages sent to and received from the tag.

Chapter 6

Conclusion

The system proposed in this thesis offers authentication and secure communication between a backend server and an RFID tag. An NFC-capable smartphone with Android OS is used as bridge technology between the two communicating parties. The phone communicates with the RFID tag over NFC and is connected to the backend over Wi-Fi. The backend connection is additionally secured using TLS. The proposed system supports two stand-alone features: a one-way authentication of the tag, and reading and writing tag data over an encrypted communication channel.

The one-way authentication authenticates the RFID tag to the backend. The used protocol is based on elliptic curve arithmetic, while communicating only with insensitive, public data. Montgomery Multiplication is used on both sides to execute the main calculations.

Tag-data operations send sensitive data over the communication channel created by and routed through the smartphone. To ensure that the smartphone application cannot leak any sensitive information, the communication is not decrypted on the device, but rather on the secured backend system. For this reason, the smartphone merely acts as a relay which establishes a secured link between the RFID tag and the backend system. AES in Offset Codebook Mode (AES-OCB) is used to encrypt the messages, which additionally creates a hash tag to authenticate the message. Thus, manipulated and erroneous messages can be detected upon decryption.

Data stored on the RFID tag is XML-structured, similar to the ePedigree standard for pharmaceutical drugs. Every access to the tag data requires an additional ECDSA signature from the processing entity (backend), which is then appended to the original data. Therefore tag data is updated on every read.

Whereas the sensitive information is transmitted in an encrypted way such that the smartphone application cannot potentially leak information, the final response, consisting of public product data is not encrypted and can therefore be displayed in the application.

A timing analysis showed that a secure authentication can be performed in 1.21 seconds on average, which is very fast, considering the communication overhead needed for NDEF encapsulation. This is due to the fact that only a single computation is needed on the tag. Therefore, the authentication facilitates many real life use-case scenarios, where fast authentication of tagged goods is of interest. Reading, and subsequently writing tag data takes more time, as all messages need to be encrypted and decrypted. For a tag-data length that needs four cycles of read and write operations (max. 100 B of data in each cycle), the average execution time is 6.18 seconds. For most real life use cases this might be too long, however, future optimizations could improve the execution time as outlined below:

- *Payload Size*
  The chosen maximum payload size is 100 bytes, therefore requiring multiple cycles for reading and writing tag data. Increasing the payload size would decrease the necessary cycle count, and thus execution time.

- *Suitable Key Length*
  Depending on the focused use case, the key lengths could be decreased. The used AES key has a length of 256 bit, which is not ideal for fast applications. Decreasing the key lengths could speed up encryption and decryption operations, however it may reduce security.

In addition, there are several options to improve the security features of the proposed system:

- *Authenticated Encryption with Associated Data (AEAD)*
  The used AEAD scheme is known as state-of-the-art for authenticated encryption. However, an ongoing cryptographic competition [D. 14] tries to determine standards for authenticated ciphers. The winners of this competition could offer features that might enhance the proposed system.

- *Replay Attack Mitigation*
  To detect replayed messages, which have already been sent over the secure channel, the protocol is designed to include a Unix timestamp [Wik14]. However, the timestamp resolution is limited to seconds, whereas the system often sends multiple messages within one second. Therefore, the timestamp has been replaced by a counter. As the backend cannot know the counter value stored on the tag, both sides reset their counters upon communication start. Defining a value that is increasing over time, such as the Unix timestamp, however more granularly to feature *e.g.* milliseconds, would greatly increase the mitigation of replay attacks.

- *Web-Service Session-ID*
  The proposed system already offers the possibility for a session ID for the web-service communication. However, session handling is deactivated. Configuring the web service to use the session handling, and forcing the smartphone application to login and use the given session key during communication, would add an additional layer of security.

Product counterfeit presents a problem for producers as well as their potential customers. Forgeries of high priced goods are often indistinguishable from their original counterparts. With the proposed system, customers can securely identify genuine products using their own smartphone in just over one second. An additional secure data transfer between the product tag and a backend server can be performed in about six seconds. All information sent and received by the smartphone application is either insensitive or encrypted.

Bibliography

[AFKM05] C. Adams, S. Farrell, T. Kause, and T. Mononen. Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP). RFC 4210 (Proposed Standard), September 2005. Updated by RFC 6712. (Cited on pages 27 and 37.)

[ARH08] S.I. Ahamed, F. Rahman, and E. Hoque. ERAP: ECC Based RFID Authentication Protocol. In *Future Trends of Distributed Computing Systems, 2008. FTDCS '08. 12th IEEE International Workshop on*, pages 219–225, 2008. (Cited on page 33.)

[BBD$^+$08] Holger Bock, Michael Braun, Markus Dichtl, Erwin Hess, Johann Heyszl, Walter Kargl, Helmut Koroschetz, Bernd Meyer, and Hermann Seuschek. A Milestone Towards RFID Products Offering Asymmetric Authentication Based on Elliptic Curve Cryptography. *Invited talk at RFIDsec*, 2008. (Cited on page 33.)

[BGK$^+$06] Lejla Batina, Jorge Guajardo, Tim Kerins, Nele Mentens, Pim Tuyls, and Ingrid Verbauwhede. An Elliptic Curve Processor Suitable For RFID-Tags. *IACR Cryptology ePrint Archive*, 2006:227, 2006. (Cited on page 33.)

[BGK$^+$07] L. Batina, J. Guajardo, T. Kerins, N. Mentens, P. Tuyls, and I. Verbauwhede. Public-Key Cryptography for RFID-Tags. In *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops '07. Fifth Annual IEEE International Conference on*, pages 217–222, 2007. (Cited on page 33.)

[BHM08] Michael Braun, Erwin Hess, and Bernd Meyer. Using Elliptic Curves on RFID Tags. *International Journal of Computer Science and Network Security*, 2:1–9, 2008. (Cited on pages 33, 43, and 44.)

[BSW01] Alex Biryukov, Adi Shamir, and David Wagner. Real Time Cryptanalysis of A5/1 on a PC. In *Fast Software Encryption*, pages 1–18. Springer, 2001. (Cited on page 26.)

[CER02] CERT Advisory. CA-2001-13 Buffer Overflow In IIS Indexing Service DLL. https://www.cert.org/historical/advisories/CA-2001-13.cfm, 2002. [Online; accessed 23-September-2014]. (Cited on page 34.)

[Cha14] Mike Chapple. Structured Query Language (SQL) — About.com - Databases. http://databases.about.com/od/sql/a/sqlbasics.htm, 2014. [Online; accessed 23-September-2014]. (Cited on page 47.)

[Chi07] Hung-Yu Chien. SASI: A New Ultralightweight RFID Authentication Protocol Providing Strong Authentication and Strong Integrity. *Dependable and Secure Computing, IEEE Transactions on*, 4(4):337–340, 2007. (Cited on page 33.)

[D. 14] D. J. Bernstein. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. http://competitions.cr.yp.to/caesar.html, 2014. [Online; accessed 23-September-2014]. (Cited on pages 28 and 96.)

[DA99] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176. (Cited on page 35.)

[DH76] W. Diffie and M.E. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, Nov 1976. (Cited on page 26.)

[Dob07] Daniel M. Dobkin. *The RF in RFID*. Elsevier, 2007. (Cited on page 32.)

[DR98] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. 1998. (Cited on page 26.)

[DR06] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), April 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176. (Cited on page 35.)

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176. (Cited on pages 35 and 36.)

[ECM13] ECMA-404. The JSON Data Interchange Standard. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, 2013. [Online; accessed 23-September-2014]. (Cited on page 49.)

[EJ03] Patrik Ekdahl and Thomas Johansson. A new version of the stream cipher SNOW. In *Selected Areas in Cryptography*, pages 47–61. Springer, 2003. (Cited on page 26.)

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology*, pages 10–18. Springer, 1985. (Cited on page 27.)

[EPC] EPCglobal. Pedigree Ratified Standard, Version 1.0., EPCglobal (2007). (Cited on pages 37, 38, and 39.)

[FHM$^+$12] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: an analysis of android SSL (in)security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 50–61, New York, NY, USA, 10 2012. ACM. (Cited on page 37.)

[FKK11] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic), August 2011. (Cited on page 35.)

[fSidIG14] Bundesamt für Sicherheit in der Informationstechnik Germany. BSI — Technische Richtlinie, Kryptografische Verfahren: Empfehlungen und Schlüssellängen (BSI TR-02102-1), 02 2014. (Cited on page 27.)

[FW07] F. Fürbass and J. Wolkerstorfer. ECC Processor with Low Die Size for RFID Applications. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 1835–1838, 2007. (Cited on page 33.)

[Goo13a] Google. Near Field Communication | Android Developers. https://developer.android.com/guide/topics/connectivity/nfc/index.html, 2013. [Online; accessed 23-September-2014]. (Cited on page 37.)

[Goo13b] Google. Security with HTTPS and SSL | Android Developers. https://developer.android.com/training/articles/security-ssl.html, 2013. [Online; accessed 23-September-2014]. (Cited on page 37.)

[Goo15] Google Inc. Google Play. https://play.google.com/, 2015. [Online; accessed 07-February-2015]. (Cited on page 42.)

[HE95] Kipp Hickman and Taher Elgamal. The SSL protocol. *Netscape Communications Corp*, 501, 1995. (Cited on page 35.)

[Hei11] Daniel Hein. Security Aspects in Software Development, XSS & Security Aspects of Data I/O, Lecture Slides, 2011. Graz University of Technology, Austria, Institute for Applied Information Processing and Communications. (Cited on page 35.)

[HFP10] Michael Hutter, Martin Feldhofer, and Thomas Plos. An ECDSA processor for RFID authentication. In *Radio Frequency Identification: Security and Privacy Issues*, pages 189–202. Springer, 2010. (Cited on page 33.)

[Hic95] Kipp Hickman. The SSL Protocol (Version 2). *Netscape Communication Cooperation*, 2, 1995. (Cited on page 35.)

[HVM04] Darrel Hankerson, Scott Vanstone, and Alfred J Menezes. *Guide to Elliptic Curve Cryptography.* Springer Science & Business Media, 2004. (Cited on page 31.)

[Inf14] Infineon Technologies Austria AG. Infineon Technologies. http://www.infineon.com/cms/austria/de/, 2014. [Online; accessed 23-September-2014]. (Cited on pages 33, 41, and 50.)

[Int] Internet Assigned Numbers Authority (IANA). Transport Layer Security (TLS) Parameters. https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml. [Online; accessed 23-September-2014]. (Cited on page 35.)

[JJ11] Li Jun and Wen Jun. Security guarantee in backend rfid system. In *Business Computing and Global Informatization (BCGIN), 2011 International Conference on*, pages 489–491, 2011. (Cited on page 35.)

[JQ13] Ben Smeets Jie Qian. IPsec and OpenVPN worked-out examples. http://ipseclab.eit.lth.se/tiki-index.php, 2013. [Online; accessed 23-September-2014]. (Cited on page 36.)

[KAF$^+$10] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K Lenstra, Emmanuel Thomé, Joppe W Bos, Pierrick Gaudry, Alexander Kruppa, Peter L Montgomery, Dag Arne Osvik, et al. Factorization of a 768-bit RSA modulus. In *Advances in Cryptology–CRYPTO 2010*, pages 333–350. Springer, 2010. (Cited on page 27.)

[Ker83] A Kerckhoffs. La Cryptographie Militaire. *L. Baudoin & Cie, Paris*, 1883. (Cited on page 25.)

[KF10] T. Kern and M. Feldhofer. Low-resource ecdsa implementation for passive rfid tags. In *Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on*, pages 1236–1239, 2010. (Cited on page 33.)

[KP06] Sandeep Kumar and Christof Paar. Are standards compliant elliptic curve cryptosystems feasible on RFID. In *Workshop on RFID Security*, pages 12–14, 2006. (Cited on page 33.)

[KP12] P. Kumar and R.K. Pateriya. A survey on SQL injection attacks, detection and prevention techniques. In *Computing Communication Networking Technologies (ICCCNT), 2012 Third International Conference on*, pages 1–5, 2012. (Cited on page 35.)

[KR14] T. Krovetz and P. Rogaway. The OCB Authenticated-Encryption Algorithm. RFC 7253 (Informational), May 2014. (Cited on page 28.)

[KT99] Kalle Kaukonen and Rodney Thayer. A stream cipher encryption algorithm "arcfour". *The Internet Society*, 1999. (Cited on pages 26 and 35.)

[Kul14] Adnan Kuleta. Design and Implementation of an optimized Authentication Protocol for Security Controller-based Applications. Master's Thesis, Graz University of Technology, Austria, Institute for Technical Informatics, 03 2014. (Cited on page 43.)

[Lam10] Mario Lamberger. IT Security, L2 - Public Key Distribution, Lecture Slides, 2010. Graz University of Technology, Austria, Institute for Applied Information Processing and Communications. (Cited on page 27.)

[LW07] Ticyan Li and Guilin Wang. Security Analysis of Two Ultra-Lightweight RFID Authentication Protocols. In *New Approaches for Security, Privacy and Trust in Complex Environments*, volume 232, pages 109–120. Springer US, 2007. (Cited on page 33.)

[Mic12a] Microsoft. SQL Server. http://www.microsoft.com/en-us/server-cloud/products/sql-server/, 2012. Latest Version: SQL Server 2012. (Cited on page 34.)

[Mic12b] Microsoft. Windows Communication Foundation. http://msdn.microsoft.com/en-us/library/dd456779.aspx, 2012. [Online; accessed 23-September-2014]. (Cited on page 49.)

[Mic13] Microsoft. Internet Information Services (IIS). http://www.iis.net/, 2013. Latest Version: 8.5. (Cited on page 34.)

[Mic15] Microsoft. Entity Framework. https://msdn.microsoft.com/en-gb/data/ef.aspx, 2015. [Online; accessed 14-February-2015]. (Cited on page 70.)

[Mon85] Peter L Montgomery. Modular Multiplication without Trial Division. *Mathematics of computation*, 44(170):519–521, 1985. (Cited on page 31.)

[Mor06] David Morgan. Web application security - SQL injection attacks. *Network Security*, 2006(4):4 – 5, 2006. (Cited on page 35.)

[MVOV10] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of Applied Cryptography, Chapter 10.* CRC Press, 2010. (Cited on page 28.)

[Nat99] National Institute of Standards and Technology. Recommended elliptic curves for federal government use. http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf, 1999. [Online; accessed 07-February-2015]. (Cited on page 44.)

[Neta] Netcraft. September 2013 Web Server Survey. http://news.netcraft.com/archives/2013/09/05/september-2013-web-server-survey.html. [Online; accessed 23-September-2014]. (Cited on page 34.)

[Netb] Netcraft. SSL: Intercepted today, decrypted tomorrow. http://news.netcraft.com/archives/2013/06/25/ssl-intercepted-today-decrypted-tomorrow.html. [Online; accessed 23-September-2014]. (Cited on page 34.)

[Nok13] Nokia Developers. Understanding NFC Data Exchange Format (NDEF) messages. `http://developer.nokia.com/community/wiki/Understanding_NFC_Data_Exchange_Format_(NDEF)_messages`, 2013. [Online; accessed 23-September-2014]. (Cited on pages 37 and 50.)

[OL10] Elisabeth Oswald and Mario Lamberger. IT Security, Lecture Notes, 03 2010. Graz University of Technology, Austria, Institute for Applied Information Processing and Communications. (Cited on page 27.)

[otBCI13] Legion of the Bouncy Castle Inc. Bouncy Castle Crypto APIs. `https://www.bouncycastle.org/`, 2013. [Online; accessed 14-February-2015]. (Cited on page 63.)

[Phi14] Phillip Rogaway. OCB Mode. http://www.cs.ucdavis.edu/~rogaway/ocb/, 2014. [Online; accessed 23-September-2014]. (Cited on pages 28, 29, 30, and 67.)

[Ple14] Robert Gerd Pleyer. Backend-Communication Security for NFC Systems with Android. IT Project, Graz University of Technology, Austria, Institute for Technical Informatics, 03 2014. (Cited on pages 32, 34, 67, and 79.)

[Rij11] Vincent Rijmen. Applied Cryptography, Lecture Slides, 2011. Graz University of Technology, Austria, Institute for Applied Information Processing and Communications. (Cited on pages 26 and 27.)

[rJ01] D. Eastlake 3rd and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Informational), September 2001. Updated by RFCs 4634, 6234. (Cited on page 35.)

[RSA78] Ronald L Rivest, Adi Shamir, and Len Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. (Cited on page 35.)

[SG10] Harald Schellnast and Susanne Gollatz. Security Aspects in Software Development, Lecture Notes, 10 2010. Graz University of Technology, Austria, Institute for Applied Information Processing and Communications. (Cited on page 35.)

[SM08] Boyeon Song and Chris J. Mitchell. RFID authentication protocol for low-cost tags. In *Proceedings of the first ACM conference on Wireless network security*, WiSec '08, pages 140–147, New York, NY, USA, 2008. ACM. (Cited on page 33.)

[SPW$^+$02] Stuart Staniford, Vern Paxson, Nicholas Weaver, et al. How to Own the Internet in Your Spare Time. In *USENIX Security Symposium*, pages 149–167, 2002. (Cited on page 34.)

[Sta96] IOF Standardization. ISO/IEC 7498-1: 1994 information technology-open systems interconnection-basic reference model: The basic model. *International Standard ISOIEC*, 74981:59, 1996. (Cited on page 35.)

[Sta98] William Stallings. SSL: Foundation for Web Security. *The Internet Protocol Journal*, 1(1):20–29, 1998. (Cited on page 105.)

[Sta99] Data Encryption Standard. Data encryption standard. *Federal Information Processing Standards Publication*, 1999. (Cited on page 26.)

[Sta11] IOF Standardization. ISO/IEC 9075-1:2011 information technology – database languages – sql – part 1: Framework (sql/framework). *International Standard ISOIEC*, page 68, 2011. (Cited on page 47.)

[Teca] Microsoft TechNet. Security Best Practices for IIS 8. http://technet.microsoft.com/en-us/library/jj635855.aspx. [Online; accessed 23-September-2014]. (Cited on page 34.)

[Tecb] Microsoft TechNet. Transact-SQL Reference. http://technet.microsoft.com/en-us/library/bb510741.aspx. [Online; accessed 23-September-2014]. (Cited on page 34.)

[Tecc] Microsoft TechNet. Web Server (IIS) Overview. http://technet.microsoft.com/en-us/library/hh831725.aspx. [Online; accessed 23-September-2014]. (Cited on page 34.)

[TP11] S. Turner and T. Polk. Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176 (Proposed Standard), March 2011. (Cited on page 35.)

[Tsu06] G. Tsudik. YA-TRAP: yet another trivial RFID authentication protocol. In *Pervasive Computing and Communications Workshops, 2006. PerCom Workshops 2006. Fourth Annual IEEE International Conference on*, pages 4 pp.–643, 2006. (Cited on page 33.)

[UP13] P. Urien and S. Piramuthu. Framework and authentication protocols for smartphone, NFC, and RFID in retail transactions. In *Intelligent Sensors, Sensor Networks and Information Processing, 2013 IEEE Eighth International Conference on*, pages 77–82, 2013. (Cited on page 33.)

[W3C04] W3C. W3C Working Group Note 11 February 2004. http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice, 2004. [Online; accessed 23-September-2014]. (Cited on page 49.)

[Wei11] SA Weis. RFID (Radio Frequency Identification): Principles and Applications. *Retrived from http://www.eecs.harvard.edu/cs199r/readings/rfidarticle.pdf*, 1, 2011. (Cited on page 32.)

[WEW12] Erich Wenger, Maria Eichelseder, and Mario Werner. IT Security, Exercise Notes, Implementing and Attacking Elliptic Curve Cryptography and RSA, 03 2012. Graz University of Technology, Austria, Institute for Applied Information Processing and Communications. (Cited on page 31.)

[WFF11] Erich Wenger, Martin Feldhofer, and Norbert Felber. Low-resource hardware design of an elliptic curve processor for contactless devices. In *Information Security Applications*, pages 92–106. Springer, 2011. (Cited on page 33.)

[Wik13] Wikipedia. 2013 global surveillance disclosures — Wikipedia, The Free Encyclopedia, 2013. [Online; accessed 4-November-2013]. (Cited on page 34.)

[Wik14] Wikipedia. Unix time — Wikipedia, The Free Encyclopedia, 2014. [Online; accessed 6-January-2015]. (Cited on pages 50, 68, and 96.)

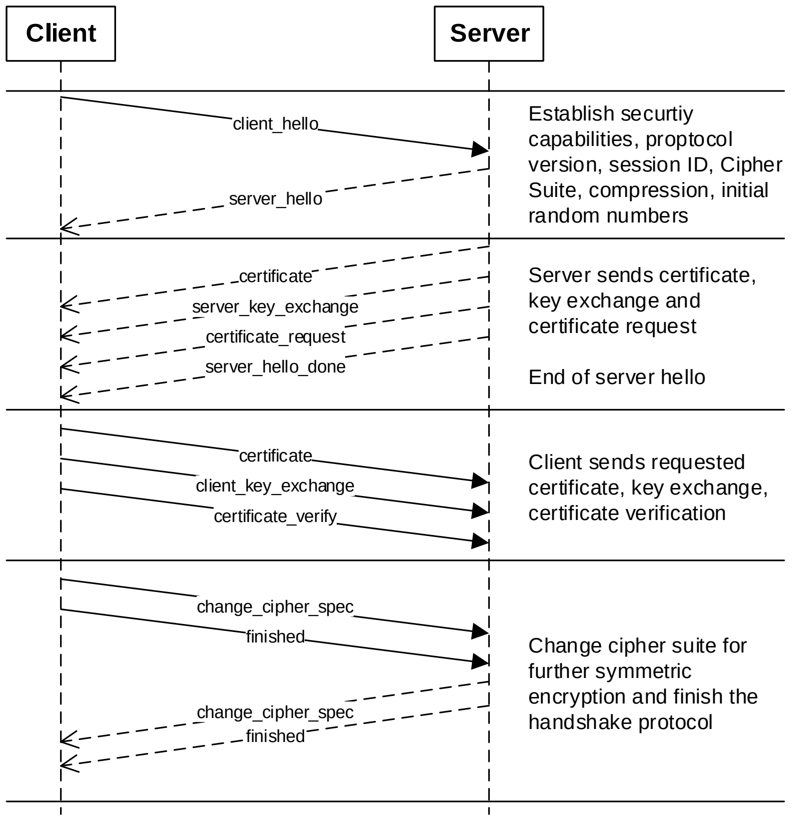
[WIS14] WISeKey. WISeKey. https://www.wisekey.com/, 2014. [Online; accessed 23-September-2014]. (Cited on page 38.)

[Wol04] Johannes Wolkerstorfer. *Hardware aspects of elliptic curve cryptography*. PhD thesis, Graz University of Technology, Austria, Institute for Applied Information Processing and Communications, 2004. (Cited on page 33.)

[XZZT13] Wei Xie, Chen Zhang, Quan Zhang, and Chaojing Tang. RFID Authentication Against an Unsecure Backend Server. *CoRR*, abs/1304.1318, 2013. (Cited on page 38.)

Appendix A

Figures



Client
Server
client_hello
server_hello
Establish securtiy capabilities, proptocol version, session ID, Cipher Suite, compression, initial random numbers
certificate
server_key_exchange
certificate_request
server_hello_done
Server sends certificate, key exchange and certificate request
End of server hello
certificate
client_key_exchange
certificate_verify
Client sends requested certificate, key exchange, certificate verification
change_cipher_spec
finished
change_cipher_spec
finished
Change cipher suite for further symmetric encryption and finish the handshake protocol

Figure A.1: Detailed View of the TLS Handshake Protocol
(adapted from [Sta98])

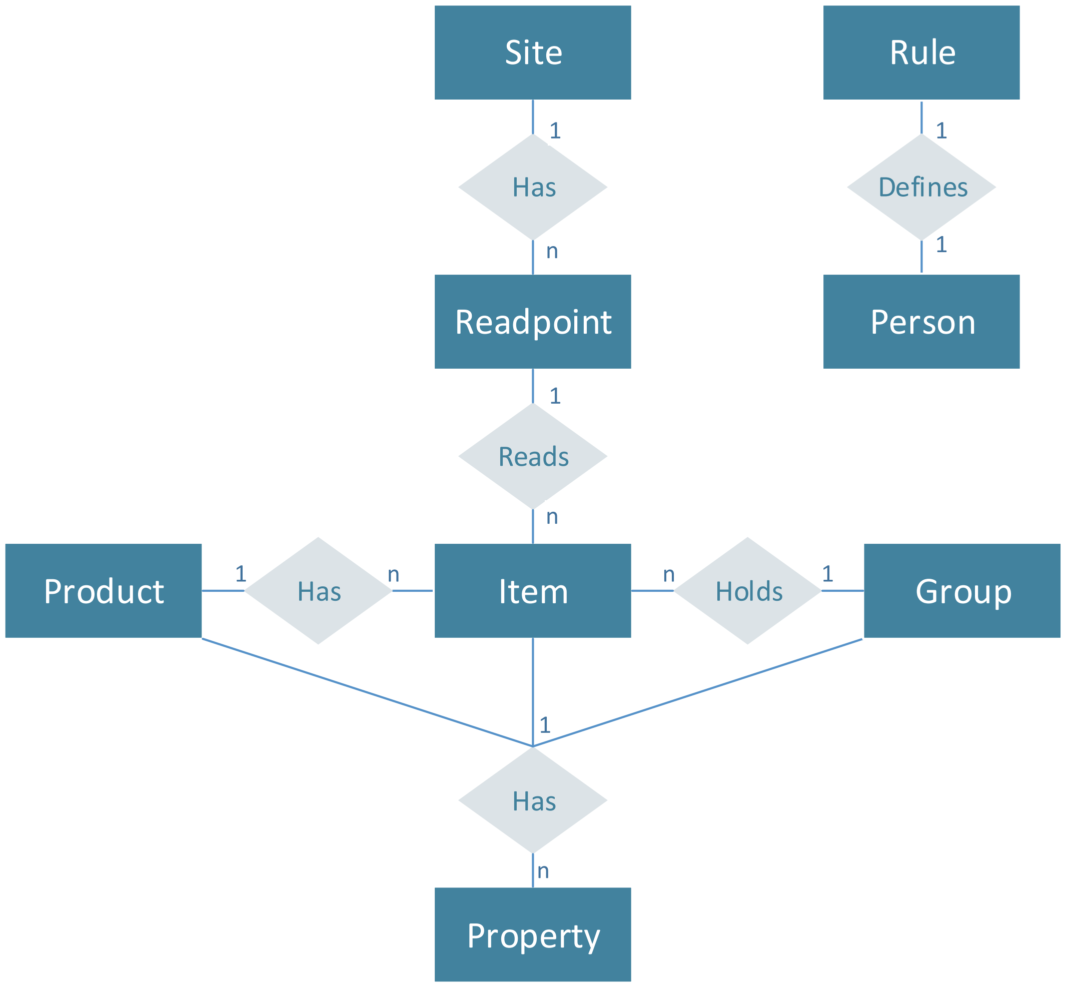


Figure A.2: Database Relation Diagram for Relevant Entities

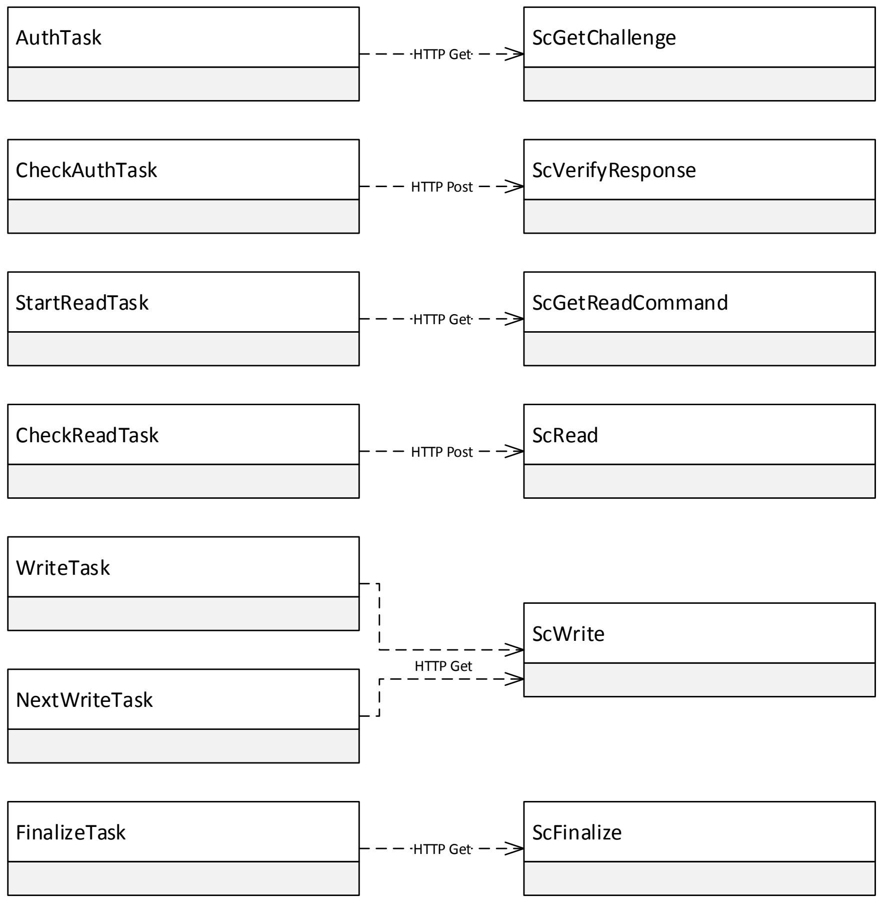


Figure A.3: Calling Structure of Android Application Tasks and Service Methods

Appendix B

Listings

Listing B.1 Keygen.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

using Org.BouncyCastle.Crypto;
using Org.BouncyCastle.Asn1.X9;
using Org.BouncyCastle.Asn1.Nist;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Crypto.Generators;
using Org.BouncyCastle.Security;
using Org.BouncyCastle.Math;
using Org.BouncyCastle.Math.EC;
using Org.BouncyCastle.Asn1;
using Org.BouncyCastle.Asn1.X509;
using Org.BouncyCastle.Asn1.Pkcs;
using Org.BouncyCastle.Pkcs;
using System.IO;

namespace detego.Service.WCF.SecureChannel
{
    public class Keygen
    {
        private static string ecCurvename = "P-192";

        public ECPoint GetBasepoint()
        {
            X9ECParameters x9ecp = NistNamedCurves.GetByName(ecCurvename);
            return x9ecp.G;
        }

        public AsymmetricCipherKeyPair GenerateEcKeyPair()
        {
            X9ECParameters x9ecp = NistNamedCurves.GetByName(ecCurvename);
            ECDomainParameters ecParameters = new ECDomainParameters(x9ecp.
                Curve, x9ecp.G, x9ecp.N, x9ecp.H);
            ECKeyPairGenerator ecGenerator = new ECKeyPairGenerator();
            SecureRandom secureRandom = new SecureRandom();
            ECKeyGenerationParameters ecKeyGenParam = new
                ECKeyGenerationParameters(ecParameters, secureRandom);
            ecGenerator.Init(ecKeyGenParam);
            return ecGenerator.GenerateKeyPair();
        }

        public AsymmetricCipherKeyPair GenerateRsaKeyPair()
        {
            RsaKeyPairGenerator rsaGenerator = new RsaKeyPairGenerator();
            SecureRandom secureRandom = new SecureRandom();
            RsaKeyGenerationParameters rsaParameters = new
                RsaKeyGenerationParameters(
                                                            new BigInteger(
                                                                "65537", 16)
                                                            , secureRandom
                                                            , 1024
                                                            , 80);
            rsaGenerator.Init(rsaParameters);
            return rsaGenerator.GenerateKeyPair();
        }

        public AsymmetricCipherKeyPair GetKeysFromFiles(FileStream
            publicfile, FileStream privatefile)
        {
            Asn1Object ao = Asn1Object.FromStream(publicfile);
            SubjectPublicKeyInfo pubinfo = SubjectPublicKeyInfo.GetInstance
                (ao);
            var publicParam = PublicKeyFactory.CreateKey(pubinfo);

            ao = Asn1Object.FromStream(privatefile);
            EncryptedPrivateKeyInfo enpri = EncryptedPrivateKeyInfo.
                GetInstance(ao);
            PrivateKeyInfo priECinfo = PrivateKeyInfoFactory.
                CreatePrivateKeyInfo("Meta[:SEC:]".ToCharArray(), enpri);
            var privateParam = PrivateKeyFactory.CreateKey(priECinfo);

            return new AsymmetricCipherKeyPair(publicParam, privateParam);
        }

    }
}
```

Listing B.2 Authentication.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Threading.Tasks;
using Org.BouncyCastle.Crypto;
using Org.BouncyCastle.Security;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Math.EC;
using Org.BouncyCastle.Asn1.X9;
using Org.BouncyCastle.Asn1.Nist;
using Org.BouncyCastle.Math;
using Org.BouncyCastle.Crypto.Prng;

namespace detego.Service.WCF.SecureChannel
{
    public class Authentication
    {
        public AsymmetricCipherKeyPair ecKeyPair { get; set; }
        public AsymmetricCipherKeyPair rsaKeyPair { get; set; }

        public Byte[] ComputeSignatureOfPublicEcKey(BigInteger publickey)
        {
            Byte[] ecPublicKeyBytes = publickey.ToByteArray();
            ISigner signer = SignerUtilities.GetSigner("
                SHA1WITHRSAENCRYPTION");
            signer.Init(true, (RsaKeyParameters)(rsaKeyPair.Private));
            signer.BlockUpdate(ecPublicKeyBytes, 0, ecPublicKeyBytes.Length
                );
            Byte[] sig = signer.GenerateSignature();
            return sig;
        }

        public BigInteger GetNewChallenge(BigInteger lambda, BigInteger xP)
        {
            return Montgomery.Multiplication(lambda, xP);
        }

        public BigInteger GetResponse(BigInteger xA)
        {
            return Montgomery.Multiplication(
                ((ECPrivateKeyParameters)(ecKeyPair.Private)).D
                , xA
                );
        }

        public bool VerifySignature(BigInteger xT, BigInteger sT)
        {
            ISigner verifier = SignerUtilities.GetSigner("
                SHA1WITHRSAENCRYPTION");
            verifier.Init(false, (RsaKeyParameters)(rsaKeyPair.Public));
            verifier.BlockUpdate(xT.ToByteArray()
                                 , 0
                                 , xT.ToByteArray().Length);
            return verifier.VerifySignature(sT.ToByteArray());
        }

        public byte[] signEPedigree(AsymmetricCipherKeyPair authEcKeyPair,
            byte[] epedigree)
        {
            ISigner signer = SignerUtilities.GetSigner("ECDSAWITHSHA1");
            signer.Init(true, (ECPrivateKeyParameters)(authEcKeyPair.
                Private));
            signer.BlockUpdate(epedigree, 0, epedigree.Length);
            return signer.GenerateSignature();
        }

        public bool verifyEPedigree(AsymmetricCipherKeyPair authEcKeyPair, byte[] epedigree, byte[] sig)
        {
            ISigner verifier = SignerUtilities.GetSigner("ECDSAWITHSHA1");
            verifier.Init(false, (ECPublicKeyParameters)(authEcKeyPair.Public));
            verifier.BlockUpdate(epedigree, 0, epedigree.Length);
            return verifier.VerifySignature(sig);
        }


    }
}
```

Listing B.3 Montgomery.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

using Org.BouncyCastle.Math;
using Org.BouncyCastle.Crypto.Parameters;

namespace detego.Service.WCF.SecureChannel
{
    public class Montgomery
    {
        static Org.BouncyCastle.Math.BigInteger BigIntZero = new Org.BouncyCastle.Math.BigInteger("0", 16);
        static Org.BouncyCastle.Math.BigInteger BigIntOne = new Org.BouncyCastle.Math.BigInteger("1", 16);
        static Org.BouncyCastle.Math.BigInteger BigIntTwo = new Org.BouncyCastle.Math.BigInteger("2", 16);

        //initialize for secp192r1

        public static Org.BouncyCastle.Math.BigInteger p = new Org.BouncyCastle.Math.BigInteger("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF", 16);
        //p = 2^192 - 2^64 - 1

        public static Org.BouncyCastle.Math.BigInteger R = BigIntTwo.Pow(24 * 8);
        //R = 2^(24*8) =^= 2^(WORDS_PER_BIGINT*BITS_PER_WORD)

        public static Org.BouncyCastle.Math.BigInteger R_squared = R.ModPow(BigIntTwo, p);
        //R^2 = R^2 mod p

        public static Org.BouncyCastle.Math.BigInteger p_strich = p.Negate().ModInverse(R);
        //p' = -p^(-1) mod R


        public static Org.BouncyCastle.Math.BigInteger Multiplication(Org.
            BouncyCastle.Math.BigInteger a, Org.BouncyCastle.Math.BigInteger
             b)
        {
            /* Montgomery Multiplication
             *
             * calculating c = a*b mod p
             *
             * 1. transform a and b to a_ and b_ (Montgomery representation
                 )
             * 2. calculate c_ = a_(star)b_     (star) denotes Mont.Mult.
             *    OR c_=MONT(a_,b_)
             * 3. transform c_ back to c
             *
             * constant value R, R > p, gcd(R,p)=1
             * R = 2^b (b that p has at most b bits)
             * so division by 2^b is right shift by b bits
             *
             * a_ = a * R mod p
             * a_ = MONT(a,R^2)
             * b_ = MONT(b,R^2)
             * c_ = MONT(a_,b_)
             * c  = MONT(c_,1)
             */

            Org.BouncyCastle.Math.BigInteger a_ = Transform(a);
            Org.BouncyCastle.Math.BigInteger b_ = Transform(b);
            Org.BouncyCastle.Math.BigInteger c_ = MONT(a_, b_);
            Org.BouncyCastle.Math.BigInteger c = Backtransform(c_);

            return c;
        }

        public static Org.BouncyCastle.Math.BigInteger Backtransform(Org.
            BouncyCastle.Math.BigInteger y)
        {
            return MONT(y, BigIntOne);
        }

        public static Org.BouncyCastle.Math.BigInteger Transform(Org.
            BouncyCastle.Math.BigInteger x)
        {
            return MONT(x, R_squared);
        }

        public static Org.BouncyCastle.Math.BigInteger MONT(Org.
            BouncyCastle.Math.BigInteger x, Org.BouncyCastle.Math.BigInteger
             y)
        {
            //d = x*y
            Org.BouncyCastle.Math.BigInteger d = x.Multiply(y);
            return Reduction(d);
        }

        private static Org.BouncyCastle.Math.BigInteger Reduction(Org.
            BouncyCastle.Math.BigInteger d)
        {
            /* Montgomery Reduction
             * s = ( d + (d*p' mod R) * p ) / R
             * v = (d*p' mod R)
             * w = ( d + v * p )
             *
             * p' = -p^(-1) mod R
             *
             * result r = s-p   if s>=p
             *            s     else
             */

            //BigInteger v = (d.Multiply(p_strich)).Mod(R);
            Org.BouncyCastle.Math.BigInteger Mask = R.Subtract(BigIntOne);
            Org.BouncyCastle.Math.BigInteger v = (d.Multiply(p_strich)).And
                (Mask);
            Org.BouncyCastle.Math.BigInteger w = d.Add((v.Multiply(p)));
            Org.BouncyCastle.Math.BigInteger s = w.ShiftRight(24 * 8);
            //same as s = w / R, just with Montgomery Magic

            //final reduction with fake reduction against timing attacks
            Org.BouncyCastle.Math.BigInteger r = s;
            Org.BouncyCastle.Math.BigInteger r_reduced = s.Subtract(p);
            int compare = s.CompareTo(p);
            if (compare >= 0)
                return r_reduced;
            else
                return r;
        }

    }
}
```

Listing B.4 CommandBuilder.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace detego.Service.WCF.SecureChannel
{
    public class CommandBuilder
    {
        private static readonly DateTime Epoch = new DateTime(1970, 1, 1,
            0, 0, 0,
                                                    DateTimeKind.Utc);

        /// <summary>
        /// Available commands on transponder side
        /// </summary>
        public enum PID : byte
        {
            PROTOCOL_ID_02 = 0x02
        }

        /// <summary>
        /// Available commands on transponder side
        /// </summary>
        public enum Command : byte
        {
            CMD_DATALINK_REFLECTOR = 0xC0,
            CMD_SECURITY_REFLECTOR = 0x80,
            CMD_SECURITY_GetVersion = 0x81,
            CMD_TOOLBOX_Authentication = 0x82,
            PROTOCOL_RESPONSE_OK = 0x00,
            PROTOCOL_RESPONSE_ACCESSDENIED = 0x02,
            CMD_STOREEPedigree = 0x42,
            CMD_READEPedigree = 0x43
        }

        /// <summary>
        /// Positions in PID = 0x02
        /// </summary>
        public enum PositionsPID02 : int
        {
            POS_PROTOCOL_ID = 0x00,
            POS_ENCRYPTION = 0x01,
            POS_LENGTH_H = 0x02,
            POS_LENGTH_L = 0x03,
            POS_TIMESTAMP = 0x04, // This is the associated data and has 4
                bytes
            POS_CMD = 0x08, // This can already be encrypted
            POS_DATA = 0x09 // This can already be encrypted
        }

        /// <summary>
        /// Supported encryptions
        /// </summary>
        public enum Encryptions : int
        {
            NONE = 0x00,
            ENCRYPTED_AES256 = 0x01,
            ENCRYPTED_AESOCB = 0x02
        }

        public enum CommandType : byte
        {
            CHALLENGE = Command.CMD_TOOLBOX_Authentication,
            READ = Command.CMD_READEPedigree,
            WRITE = Command.CMD_STOREEPedigree
        }


        public static UInt32 formerUnixTimestamp = 0;

        /// <summary>
        /// Returns a fake UnixTimeStamp as byte[4]
        /// </summary>
        /// <returns></returns>
        public static byte[] getUnixTimestamp()
        {
            formerUnixTimestamp++;
            byte[] bytes = BitConverter.GetBytes(formerUnixTimestamp);
            if (BitConverter.IsLittleEndian) Array.Reverse(bytes);

            return bytes;
        }

        /// <summary>
        /// Returns the current UnixTimeStamp as byte[4]
        /// </summary>
        /// <returns></returns>
        public static byte[] getRealUnixTimestamp()
        {
            Int32 unixTimestamp = (Int32)(DateTime.UtcNow.Subtract(new
                DateTime(1970, 1, 1))).TotalSeconds;
            byte[] bytes = BitConverter.GetBytes(unixTimestamp);
            if (BitConverter.IsLittleEndian) Array.Reverse(bytes);
            return bytes;

        }

        public static DateTime UnixTimeToDateTime(string text)
        {
            double seconds = double.Parse(text);
            return Epoch.AddSeconds(seconds);
        }

        //command = [PID|ENC|LEN|LEN|TSP|TSP|TSP|TSP|CMD|DATA..TAG]
        //          { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 .. +16]

        public static byte[] Build(CommandType command, byte[] data, bool
            encrypted)
        {
            byte[] commandAsByteArray;

            if (data == null) data = new byte[0];

            // calculate the bytes to encrypt
            // and combine it with the command byte (command also has to be
                 encrypted)
            // 1 ... command byte
            int bytesToEncrypt = 1 + data.Length;

            byte[] requestToEncrypt = new byte[bytesToEncrypt];
            requestToEncrypt[0] = (byte)command;
            System.Buffer.BlockCopy(data, 0, requestToEncrypt, 1, data.
                Length);

            byte[] TSP = new byte[4]; TSP = getUnixTimestamp();// Timestamp
                 (associated data)
            byte[] N = new byte[15]; // 96/8 (Nonce)

            System.Buffer.BlockCopy(TSP, 0, N, 11, 4); //nonce = [00..00|
                TSP_NOW]

            byte[] encryptedRequest = OCB.encrypt(AES.key, AES.key.Length * 8, requestToEncrypt, N, TSP, 128);

            if (!encrypted) encryptedRequest = requestToEncrypt; //Same timing as if encr.

            // everything is encrypted except the protocol id, the encryption, the associated data and the length
            commandAsByteArray = new byte[encryptedRequest.Length + (int)PositionsPID02.POS_CMD];

            commandAsByteArray[(int)PositionsPID02.POS_PROTOCOL_ID] = (byte)PID.PROTOCOL_ID_02;

            commandAsByteArray[(int)PositionsPID02.POS_ENCRYPTION] = (byte)Encryptions.ENCRYPTED_AESOCB;
            if (!encrypted) commandAsByteArray[(int)PositionsPID02.POS_ENCRYPTION] = (byte)Encryptions.NONE;

            int noncebytes = (encrypted ? 16 : 0);
            // Set the data length of the followed command
            // this is never changed because the length is needed to get the real length of the encrypted message
            commandAsByteArray[(int)PositionsPID02.POS_LENGTH_H] = (byte)((data.Length + PositionsPID02.POS_DATA - PositionsPID02.POS_TIMESTAMP + noncebytes) >> 8);
            commandAsByteArray[(int)PositionsPID02.POS_LENGTH_L] = (byte)((data.Length + PositionsPID02.POS_DATA - PositionsPID02.POS_TIMESTAMP + noncebytes) & 0xFF);

            System.Buffer.BlockCopy(encryptedRequest, 0, commandAsByteArray, (int)PositionsPID02.POS_CMD, encryptedRequest.Length);
            System.Buffer.BlockCopy(TSP, 0, commandAsByteArray, (int)PositionsPID02.POS_TIMESTAMP, 4);


            return commandAsByteArray;
        }
    }
}
```

Listing B.5 EPedigreeHandler.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Xml;
using System.IO;
using detego.Service.Contract.Entities;

namespace detego.Service.WCF.SecureChannel
{
    public class Pedigree
    {
        string _productName;
        string _productCategory;
        string _articleNumber;
        string _origin;
        double _rrp; //recommended retail price

        public Pedigree(string productName, string productCategory, string
            articleNumber, string origin, double rrp)
        {
            this._productName = productName;
            this._productCategory = productCategory;
            this._articleNumber = articleNumber;
            this._origin = origin;
            this._rrp = rrp;
        }

        public string ProductName { get { return _productName; } set {
            _productName = (string)value; } }
        public string ProductCategory { get { return _productCategory; }
            set { _productCategory = (string)value; } }
        public string ArticleNumber { get { return _articleNumber; } set {
            _articleNumber = (string)value; } }
        public string Origin { get { return _origin; } set { _origin = (
            string)value; } }
        public double RRP { get { return _rrp; } set { _rrp = (double)value
            ; } }
    }

    public class EPedigreeHandler
    {
        public void createDummyPedigree(string filename)
        {
            Pedigree myPed = new Pedigree("myBag", "Handbag", "H01245-91",
                "Le Handbag Producer", 1199.99);

            XmlWriterSettings xmlWriterSettings = new XmlWriterSettings()
            {
                Indent = true,
                IndentChars = "\t",
                NewLineOnAttributes = true
            };

            using (XmlWriter xwriter = XmlWriter.Create(filename,
                xmlWriterSettings))
            {
                xwriter.WriteStartDocument();
                xwriter.WriteStartElement("Pedigree");
                xwriter.WriteElementString("ProductName", myPed.ProductName
                    );
                xwriter.WriteElementString("ProductCategory", myPed.
                    ProductCategory);
                xwriter.WriteElementString("ArticleNumber", myPed.
                    ArticleNumber);
                xwriter.WriteElementString("Origin", myPed.Origin);
                xwriter.WriteElementString("RRP", myPed.RRP.ToString());
                xwriter.WriteEndElement();
                xwriter.WriteEndDocument();
            }
        }

        public void readPedigreeToConsole(string filename)
        {
            using (XmlReader xreader = XmlReader.Create(filename))
            {
                while (xreader.Read())
                {
                    if (xreader.IsStartElement())
                    {
                        switch (xreader.Name)
                        {
                            case "Pedigree":
                                if (xreader.Read())
                                    Console.WriteLine("Start <Pedigree>
                                        element.");
                                break;
                            case "ProductName":
                                if (xreader.Read())
                                    Console.WriteLine("  ProductName: " +
                                        xreader.Value);
                                break;
                            case "ProductCategory":
                                if (xreader.Read())
                                    Console.WriteLine("  ProductCategory: "
                                         + xreader.Value);
                                break;
                            case "ArticleNumber":
                                if (xreader.Read())
                                    Console.WriteLine("  ArticleNumber: " +
                                         xreader.Value);
                                break;
                            case "Origin":
                                if (xreader.Read())
                                    Console.WriteLine("  Origin: " +
                                        xreader.Value);
                                break;
                            case "RRP":
                                if (xreader.Read())
                                    Console.WriteLine("  RRP: " + xreader.
                                        Value);
                                break;

                        }
                    }
                }
            }
        }

        public Pedigree readPedigreeToStruct(string filename)
        {
            Pedigree ped = new Pedigree("dummy", "dummy", "dummy", "dummy",
                0.0);
            using (XmlReader xreader = XmlReader.Create(filename))
            {
                while (xreader.Read())
                {
                    if (xreader.IsStartElement())
                    {
                        switch (xreader.Name)
                        {
                            //case "Pedigree":
                            //    if (xreader.Read())
                            //        Console.WriteLine("Start <Pedigree>
                                  element.");
                            //    break;
                            case "ProductName":
                                if (xreader.Read())
                                    ped.ProductName = xreader.Value;
                                break;
                            case "ProductCategory":
                                if (xreader.Read())
                                    ped.ProductCategory = xreader.Value;
                                break;
                            case "ArticleNumber":
                                if (xreader.Read())
                                    ped.ArticleNumber = xreader.Value;
                                break;
                            case "Origin":
                                if (xreader.Read())
                                    ped.Origin = xreader.Value;
                                break;
                            case "RRP":
                                if (xreader.Read())
                                    ped.RRP = Convert.ToDouble(xreader.
                                        Value);
                                break;
                        }
                    }
                }
            }

            return ped;
        }

        public byte[] convertXmlToByteArray(string filename)
        {
            XmlDocument ped = new XmlDocument();
            ped.Load(filename);

            MemoryStream ms = new MemoryStream();
            ped.Save(ms);
            byte[] bytes = ms.ToArray();

            //Console.WriteLine(Encoding.UTF8.GetString(bytes));

            //foreach (byte byteValue in bytes)
            //    Console.Write("{0:X2} ", byteValue);

            return bytes;
        }

        public void convertByteArrayToXml(byte[] bytes, string filename)
        {
            XmlDocument ped = new XmlDocument();
            MemoryStream ms = new MemoryStream(bytes);
            ped.Load(ms);
            ped.Save(filename);

            //Console.WriteLine(Encoding.UTF8.GetString(bytes));

            //foreach (byte byteValue in bytes)
            //    Console.Write("{0:X2} ", byteValue);
        }
    }
}
```

Listing B.6 DataService.scv.cs: ScRead

```
public PropertyValue ScRead(PropertyValue read_response)
{
    //command = [PID|ENC|LEN|LEN|TSP|TSP|TSP|TSP|CMD|DATA..TAG]
    //          { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 .. +16]

    string returnstate = "CONT";

    if (read_response == null) return ScCommandToPropVal(null, "ScRead", "
        NULLINPUT");
    byte[] response = read_response.BinaryValue;
    if (response == null) return ScCommandToPropVal(null, "ScRead", "
        NULLINPUTBYTES");
    if (response[0] != 0x02 || response[1] != 0x02) return
        ScCommandToPropVal(null, "ScRead", "MESSAGEERROR");

    int length = (((int)response[2] << 8) + (int)response[3]);
    byte[] encrypted = new byte[response.Length - 8];
    System.Buffer.BlockCopy(response, 8, encrypted, 0, encrypted.Length);

    byte[] tsp = new byte[4];
    System.Buffer.BlockCopy(response, 4, tsp, 0, 4);

    byte[] nonce = new byte[15];
    System.Buffer.BlockCopy(tsp, 0, nonce, 11, 4);

    byte[] decrypted = OCB.decrypt(AES.key, AES.key.Length * 8, encrypted,
        nonce, tsp, 128);

    if (decrypted == null) return ScCommandToPropVal(null, "ScRead", "
        READERROR");

    byte command = decrypted[0];
    byte packageInfo = decrypted[1];
    int data_length = decrypted.Length - 3;

    if (packageInfo != (byte)0xff)
    {
        data_length = (int)packageInfo;
        returnstate = "DONE";
    }

    byte[] data = new byte[data_length];
    System.Buffer.BlockCopy(decrypted, 2, data, 0, data.Length);

    byte[] pedigree_memory = read_pedigree_buffer;
    read_pedigree_buffer = new byte[pedigree_memory.Length + data_length];
    System.Buffer.BlockCopy(pedigree_memory, 0, read_pedigree_buffer, 0,
        pedigree_memory.Length);
    System.Buffer.BlockCopy(data, 0, read_pedigree_buffer, pedigree_memory.
        Length, data.Length);

    if (returnstate == "CONT")
    {
        return ScCommandToPropVal(ScGetNextReadCommand(), "ScRead",
            returnstate);
    }

    byte[] pedigree = read_pedigree_buffer;
    int sig_length = 56;

    byte[] signature = new byte[sig_length];
    System.Buffer.BlockCopy(pedigree, pedigree.Length - sig_length,
        signature, 0, signature.Length);

    byte[] ped_w_tsp = new byte[pedigree.Length - sig_length];
    System.Buffer.BlockCopy(pedigree, 0, ped_w_tsp, 0, ped_w_tsp.Length);

    Keygen keygen = new Keygen();
    FileStream EcPublicFile = new FileStream(@"D:/MetaSec_/Robert/trunk/
        detego.Surveyor/Service2/detego.Service.WCF/SecureChannel/
        EcPublicKey_SIGNING", FileMode.Open, FileAccess.Read);
    FileStream EcPrivateFile = new FileStream(@"D:/MetaSec_/Robert/trunk/
        detego.Surveyor/Service2/detego.Service.WCF/SecureChannel/
        EcPrivateKey_SIGNING", FileMode.Open, FileAccess.Read);
    AsymmetricCipherKeyPair authEcKeyPair = keygen.GetKeysFromFiles(
        EcPublicFile, EcPrivateFile);

    Authentication auth = new Authentication();

    bool verified = auth.verifyEPedigree(authEcKeyPair, ped_w_tsp,
        signature);

    if (!verified) return ScCommandToPropVal(null, "ScRead", "SIGERROR");

    byte[] epedigree = new byte[ped_w_tsp.Length - 4];
    byte[] timestamp = new byte[4];
    System.Buffer.BlockCopy(ped_w_tsp, 0, epedigree, 0, epedigree.Length);
    System.Buffer.BlockCopy(ped_w_tsp, ped_w_tsp.Length - 4, timestamp, 0,
        4);
    if (BitConverter.IsLittleEndian) Array.Reverse(timestamp);
    int timestamp_int = BitConverter.ToInt32(timestamp, 0);

    EPedigreeHandler pedHandler = new EPedigreeHandler();
    pedHandler.convertByteArrayToXml(epedigree, @"D:/MetaSec_/Robert/trunk/
        detego.Surveyor/Service2/detego.Service.WCF/SecureChannel/pedigree.
        xml");
    Pedigree ped = pedHandler.readPedigreeToStruct(@"D:/MetaSec_/Robert/
        trunk/detego.Surveyor/Service2/detego.Service.WCF/SecureChannel/
        pedigree.xml");
    Product product = entities.Products.SingleOrDefault(
                p =>
                p.ArticleNumber == ped.ArticleNumber &&
                p.Name == ped.ProductName);

    if (product == null)
    {
        product = new Product()
                {
                    Name = ped.ProductName,
                    DisplayName = CommandBuilder.UnixTimeToDateTime(
                        timestamp_int.ToString()).ToString(),
                    ArticleNumber = ped.ArticleNumber
                };
        entities.Products.Add(product);
        entities.SaveChanges();
        return ScCommandToPropVal(null, "ScRead", "ADDED_NEW_PRODUCT");
    }

    current_productid = product.ProductId;
    product.TimeStampEvent = DateTime.UtcNow;
    product.DisplayName = CommandBuilder.UnixTimeToDateTime(timestamp_int.
        ToString()).ToString();

    byte[] new_timestamp = CommandBuilder.getRealUnixTimestamp();
    byte[] to_sign = new byte[epedigree.Length + 4];
    System.Buffer.BlockCopy(epedigree, 0, to_sign, 0, epedigree.Length);
    if (BitConverter.IsLittleEndian) Array.Reverse(timestamp);
    System.Buffer.BlockCopy(new_timestamp, 0, to_sign, epedigree.Length, 4)
        ;
    byte[] new_sig = auth.signEPedigree(authEcKeyPair, to_sign);
    while (new_sig.Length != 56) { new_sig = auth.signEPedigree(
        authEcKeyPair, to_sign); }
    byte[] new_pedigree_full = new byte[to_sign.Length + new_sig.Length];
    System.Buffer.BlockCopy(to_sign, 0, new_pedigree_full, 0, to_sign.
        Length);
    System.Buffer.BlockCopy(new_sig, 0, new_pedigree_full, to_sign.Length,
        new_sig.Length);

    if (auth.verifyEPedigree(authEcKeyPair, to_sign, new_sig) != true)
        return ScCommandToPropVal(null, "ScRead", "SIG_GEN_ERROR");

    returnstate = "WRITEREADY";
    offset = 0;
    copyOffset = 0;
    write_pedigree_buffer = new_pedigree_full;

    return ScCommandToPropVal(null, "ScRead", returnstate);
}
```

Listing B.7 DataService.scv.cs: ScCheckWrite

```
public PropertyValue ScCheckWrite(PropertyValue write_response)
{
    if (write_response.StringValue != null) ScCommandToPropVal(null, "ClientLocation", write_response.StringValue);

    if (write_response == null) return ScCommandToPropVal(null, "ScCheckWrite", "NULLINPUT");
    byte[] response = write_response.BinaryValue;
    if (response == null) return ScCommandToPropVal(null, "ScCheckWrite", "NULLINPUTBYTES");
    if (response[0] != 0x02 || response[1] != 0x02) return ScCommandToPropVal(null, "ScCheckWrite", "MESSAGEERROR");

    int length = (((int)response[2] << 8) + (int)response[3]);
    byte[] encrypted = new byte[response.Length - 8];
    System.Buffer.BlockCopy(response, 8, encrypted, 0, encrypted.Length);

    byte[] tsp = new byte[4];
    System.Buffer.BlockCopy(response, 4, tsp, 0, 4);

    byte[] nonce = new byte[15];
    System.Buffer.BlockCopy(tsp, 0, nonce, 11, 4);

    byte[] decrypted = OCB.decrypt(AES.key, AES.key.Length * 8, encrypted, nonce, tsp, 128);

    if (decrypted == null) return ScCommandToPropVal(null, "ScCheckWrite", "READERROR");
    if (decrypted[0] == 0x00 && decrypted[1] == 0x00) return ScCommandToPropVal(null, "ScCheckWrite", "OK");

    return ScCommandToPropVal(null, "ScCheckWrite", "ERROR");
}
```

Listing B.8 Store Keys to File

```
SubjectPublicKeyInfo RSAsubInfo = SubjectPublicKeyInfoFactory.CreateSubjectPublicKeyInfo(myRSAKeyPair.Public);
Asn1Object RSAaobject = RSAsubInfo.ToAsn1Object();
byte[] pubInfoByte = RSAaobject.GetEncoded();
FileStream RSAfs = new FileStream(@"D:/MetaSec/trunk/MASTER/CryptoTest1/Test1/Test1/bin/Debug/RsaPublicKey", FileMode.Create, FileAccess.Write);
RSAfs.Write(pubInfoByte, 0, pubInfoByte.Length);
RSAfs.Close();

//PrivateKeyInfo RSAprivateKeyInfo = PrivateKeyInfoFactory.CreatePrivateKeyInfo(myRSAKeyPair.Private);
//RSAaobject = RSAprivateKeyInfo.ToAsn1Object();
```

```

EncryptedPrivateKeyInfo enRSAprivate = EncryptedPrivateKeyInfoFactory.
    CreateEncryptedPrivateKeyInfo(
     "1.2.840.113549.1.12.1.3", //alog
     "Meta[:SEC:]".ToCharArray(),//pw
     new byte[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 },//salt
     100,//iterations
     myRSAKeyPair.Private);
byte[] priInfoByte = enRSAprivate.GetEncoded();
RSAfs = new FileStream(@"D:/MetaSec/trunk/MASTER/CryptoTest1/Test1/Test1/
    bin/Debug/RsaPrivateKey", FileMode.Create, FileAccess.Write);
RSAfs.Write(priInfoByte, 0, priInfoByte.Length);
RSAfs.Close();

SubjectPublicKeyInfo ECsubInfo = SubjectPublicKeyInfoFactory.
    CreateSubjectPublicKeyInfo(myECKeyPair.Public);
Asn1Object ECaobject = ECsubInfo.ToAsn1Object();
byte[] ECpubInfoByte = ECaobject.GetEncoded();
FileStream ECfs = new FileStream(@"D:/MetaSec/trunk/MASTER/CryptoTest1/
    Test1/Test1/bin/Debug/EcPublicKey_SIGNING", FileMode.Create, FileAccess.
    Write);
ECfs.Write(ECpubInfoByte, 0, ECpubInfoByte.Length);
ECfs.Close();

//PrivateKeyInfo ECprivateKeyInfo = PrivateKeyInfoFactory.
    CreatePrivateKeyInfo(myECKeyPair.Private);
//ECaobject = ECprivateKeyInfo.ToAsn1Object();

EncryptedPrivateKeyInfo enECprivate = EncryptedPrivateKeyInfoFactory.
    CreateEncryptedPrivateKeyInfo(
     "1.2.840.113549.1.12.1.3", //alog
     "Meta[:SEC:]".ToCharArray(),//pw
     new byte[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 },//salt
     100,//iterations
     myECKeyPair.Private);
byte[] ECpriInfoByte = enECprivate.GetEncoded();
ECfs = new FileStream(@"D:/MetaSec/trunk/MASTER/CryptoTest1/Test1/Test1/bin
    /Debug/EcPrivateKey_SIGNING", FileMode.Create, FileAccess.Write);
ECfs.Write(ECpriInfoByte, 0, ECpriInfoByte.Length);
ECfs.Close();
```

Listing B.9 CommandHandler.java

```java
package com.detego.sc;

import android.content.Context;
import android.os.AsyncTask;
import android.util.Base64;
import android.util.Log;

import com.detego.javaEntities.Product;
import com.detego.javaEntities.PropertyValue;
import com.infineon.dcgr.ccs.demo.avl.protocol.AuthCommand;
import com.infineon.dcgr.ccs.demo.avl.protocol.ReadCommand;
import com.infineon.dcgr.ccs.demo.avl.protocol.WriteCommand;

import org.json.JSONException;
import org.json.JSONObject;

import java.io.InputStream;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.concurrent.ExecutionException;

/**
 * Created by Robert.Pleyer on 01.07.2014.
 */
public class CommandHandler {

    private static ConnectionHTTPS connectionhttps;

    static {
        connectionhttps = new ConnectionHTTPS();
    }

    private static InputStream certificateStream;

    public static AuthCommand GetAuthCommand(Context context, HomeActivity
        homeActivity) {
        try {
            certificateStream = context.getResources().openRawResource(R.
                raw.cert);
            AuthTask authTask = new AuthTask();
            authTask.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
            byte[] authBytes = authTask.get();
            return new AuthCommand(authBytes, homeActivity);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        return null;
    }

    public static boolean CheckAuthResponse(Context context, byte[]
        authResponse) {
        try {
            certificateStream = context.getResources().openRawResource(R.
                raw.cert);

            PropertyValue propval = new PropertyValue();
            propval.binaryvalue = Base64.encodeToString(authResponse,
                Base64.NO_WRAP);

            CheckAuthTask checkAuthTask = new CheckAuthTask();
            checkAuthTask.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR,
                 propval);
            return checkAuthTask.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        return false;
    }

    public static PropertyValue CheckReadResponse(Context context, byte[]
        readResponse) {
        try {
            certificateStream = context.getResources().openRawResource(R.
                raw.cert);

            PropertyValue propval = new PropertyValue();
            propval.binaryvalue = Base64.encodeToString(readResponse,
                Base64.NO_WRAP);

            CheckReadTask checkReadTask = new CheckReadTask();
            checkReadTask.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR,
                 propval);
            return checkReadTask.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        return null;
    }

    public static JSONObject Finalize(Context context, HomeActivity
        homeActivity) {
        try {
            certificateStream = context.getResources().openRawResource(R.
                raw.cert);

            FinalizeTask finalizeTask = new FinalizeTask();
            finalizeTask.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
            return finalizeTask.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        return null;
    }


    public static class AuthTask extends AsyncTask<Void, Void, byte[]> {

        @Override
        protected void onPreExecute() {
        }

        @Override
        protected byte[] doInBackground(Void... params) {
            try {
                InputStream fin = certificateStream;
                URL url = new URL("https://grzdev0104.at.local/detego.
                    Service.WCF/DataService.svc/Json/ScGetChallenge");
                JSONObject commandObj = connectionhttps.
                    connectToHttpsWithSelfSignedCertificate(url, fin, null);
                return Base64.decode(commandObj.getString("BinaryValue"),
                    Base64.NO_WRAP);
            } catch (MalformedURLException e) {
                e.printStackTrace();
            } catch (JSONException e) {
                e.printStackTrace();
            }
            return null;
        }

    }

    public static class CheckAuthTask extends AsyncTask<PropertyValue, Void
        , Boolean> {

        @Override
        protected void onPreExecute() {
        }

        @Override
        protected Boolean doInBackground(PropertyValue... propertyValues) {
            try {
                InputStream fin = certificateStream;
                URL url = new URL("https://grzdev0104.at.local/detego.
                    Service.WCF/DataService.svc/Json/ScVerifyResponse");
                JSONObject commandObj = connectionhttps.
                    postToHttpsWithSelfSignedCertificate(url, fin, null,
                    propertyValues[0]);
                String response = commandObj.getString("StringValue");
                if (response.equals("TRUE")) return true;
            } catch (MalformedURLException e) {
                e.printStackTrace();
            } catch (JSONException e) {
                e.printStackTrace();
            }
            return false;
        }

    }

    public static class CheckReadTask extends AsyncTask<PropertyValue, Void
        , PropertyValue> {

        @Override
        protected void onPreExecute() {
        }

        @Override
        protected PropertyValue doInBackground(PropertyValue...
            propertyValues) {
            try {
                InputStream fin = certificateStream;
                URL url = new URL("https://grzdev0104.at.local/detego.
                    Service.WCF/DataService.svc/Json/ScRead");
                JSONObject commandObj = connectionhttps.
                    postToHttpsWithSelfSignedCertificate(url, fin, null,
                    propertyValues[0]);
                PropertyValue pv_response = new PropertyValue();
                pv_response.binaryvalue = commandObj.getString("BinaryValue
                    ");
                pv_response.stringvalue = commandObj.getString("StringValue
                    ");
                return pv_response;
            } catch (MalformedURLException e) {
                e.printStackTrace();
            } catch (JSONException e) {
                e.printStackTrace();
            }
            return null;
        }

    }

    public static ReadCommand GetReadCommand(Context context, HomeActivity
        homeActivity) {
        try {
            certificateStream = context.getResources().openRawResource(R.
                raw.cert);
            StartReadTask startReadTask = new StartReadTask();
            startReadTask.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR)
                ;
            byte[] readBytes = startReadTask.get();
            return new ReadCommand(readBytes, homeActivity);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        return null;
    }


    public static class StartReadTask extends AsyncTask<Void, Void, byte[]>
         {

        @Override
        protected void onPreExecute() {
        }

        @Override
        protected byte[] doInBackground(Void... params) {
            try {
                InputStream fin = certificateStream;
                URL url = new URL("https://grzdev0104.at.local/detego.
                    Service.WCF/DataService.svc/Json/ScGetReadCommand");
                JSONObject commandObj = connectionhttps.
                    connectToHttpsWithSelfSignedCertificate(url, fin, null);
                return Base64.decode(commandObj.getString("BinaryValue"),
                    Base64.NO_WRAP);
            } catch (MalformedURLException e) {
                e.printStackTrace();
            } catch (JSONException e) {
                e.printStackTrace();
            }
            return null;
        }

    }

    public static WriteCommand GetWriteCommand(Context context,
        HomeActivity homeActivity) {
        try {
            certificateStream = context.getResources().openRawResource(R.
                raw.cert);
            WriteTask writeTask = new WriteTask();
            writeTask.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
            byte[] writeBytes = writeTask.get();
            return new WriteCommand(writeBytes, homeActivity);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        return null;
    }

    public static class WriteTask extends AsyncTask<Void, Void, byte[]> {

        @Override
        protected void onPreExecute() {
        }

        @Override
        protected byte[] doInBackground(Void... params) {
            try {
                InputStream fin = certificateStream;
                URL url = new URL("https://grzdev0104.at.local/detego.
                    Service.WCF/DataService.svc/Json/ScWrite");
                JSONObject commandObj = connectionhttps.
                    connectToHttpsWithSelfSignedCertificate(url, fin, null);
                return Base64.decode(commandObj.getString("BinaryValue"),
                    Base64.NO_WRAP);
            } catch (MalformedURLException e) {
                e.printStackTrace();
            } catch (JSONException e) {
                e.printStackTrace();
            }
            return null;
        }

    }

    public static PropertyValue NextWriteCommand(Context context,
        HomeActivity homeActivity) {
        try {
            certificateStream = context.getResources().openRawResource(R.
                raw.cert);
            NextWriteTask nextWriteTask = new NextWriteTask();
            nextWriteTask.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR)
                ;
            return nextWriteTask.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        return null;
    }

    public static class NextWriteTask extends AsyncTask<Void, Void,
        PropertyValue> {

        @Override
        protected void onPreExecute() {
        }

        @Override
        protected PropertyValue doInBackground(Void... params) {
            try {
                InputStream fin = certificateStream;
                URL url = new URL("https://grzdev0104.at.local/detego.
                    Service.WCF/DataService.svc/Json/ScWrite");
                JSONObject commandObj = connectionhttps.
                    connectToHttpsWithSelfSignedCertificate(url, fin, null);
                PropertyValue pv_response = new PropertyValue();
                pv_response.binaryvalue = commandObj.getString("BinaryValue
                    ");
                pv_response.stringvalue = commandObj.getString("StringValue
                    ");
                return pv_response;
            } catch (MalformedURLException e) {
                e.printStackTrace();
            } catch (JSONException e) {
                e.printStackTrace();
            }
            return null;
        }

    }

    private static class FinalizeTask extends AsyncTask<Void, Void,
        JSONObject> {

        @Override
        protected void onPreExecute() {
        }

        @Override
        protected JSONObject doInBackground(Void... params) {
            try {
                Log.d("FinalizeTask", "FinalizeTask called");
                InputStream fin = certificateStream;
                URL url = new URL("https://grzdev0104.at.local/detego.
                    Service.WCF/DataService.svc/Json/ScFinalize");
                JSONObject commandObj = connectionhttps.
                    connectToHttpsWithSelfSignedCertificate(url, fin, null);
                return commandObj;
            } catch (MalformedURLException e) {
                e.printStackTrace();
            }
            Log.d("FinalizeTask", "@return null");
            return null;
        }
    }
}
```