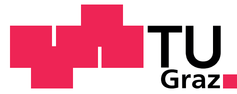


Martin Zöhrer, BSc.

Design and Implementation of an Efficient and Secured Transport Layer Protocol for NFC

Master's Thesis
to achieve the university degree of
Diplom-Ingenieur
Master's degree programme: Computer Science

submitted to

Graz University of Technology

Advisors
Ass.-Prof. Dipl-Ing. Dr.techn. Christian Steger
Dipl-Ing. Dipl-Ing. Thomas Ulz, Bsc

Institute for Technical Informatics
Head: Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Uwe Roemer

Graz, May 2019

Abstract

Due to the rapidly growing usage of Near Field Communication (NFC) throughout many different fields of application, the security of data being sent via NFC becomes more and more crucial. This includes payment transactions, social and health related data or information related to access controls, hence there is a strong need for the usage of efficient data protection mechanisms to ensure confidentiality, authenticity and integrity in an NFC-session. However, many mechanisms presented in existing research do not satisfy these requirements. Thus, this master's thesis presents a versatile cryptographic protocol, which provides the required level of security while still being efficient. Authentication is ensured by the usage of standard SSL certificates and authenticated encryption algorithms are used to protect the confidentiality and integrity of information. Even though this protocol was designed for NFC, it is also applicable for other wireless communication standards.

Within the scope of this thesis, a threat and performance analysis of an existing cryptographic protocol is conducted. Based on that analysis, modifications are presented to enhance the security level and the performance. Furthermore, application use cases are identified and the implementation processes of the proposed protocol and corresponding demo applications are presented within this work.

Kurzfassung

Aufgrund der rasanten Verbreitung von NFC in verschiedensten Anwendungsbereichen spielt die Sicherheit der übertragenen Daten eine immer größere Rolle. Unter anderem wird NFC für die Übertragung von Finanztransaktionen sowie für die Übermittlung von Gesundheitsdaten und für Zugriffskontrollmechanismen verwendet. Daher besteht die Notwendigkeit für einen effizienten Mechanismus zur Sicherung der übertragenen Daten, um Vertraulichkeit, Authentizität und Integrität jener Daten gewährleisten zu können. Jedoch unterstützen viele aktuelle forschungsrelevante Lösungen nur teilweise diese Kriterien, daher wird in dieser Arbeit ein sicheres und effizientes Protokoll zur Übertragung von vertraulichen Daten mit Hilfe von NFC vorgestellt. Dieses Protokoll erfüllt die bereits genannten Kriterien und ist dabei performanter als andere Lösungen. Auch wenn dieses Protokoll ursprünglich für NFC entwickelt wurde, so ist es doch auch für andere drahtlose Übertragungsmedien geeignet. Die Authentifizierung basiert auf standardmäßigen SSL-Zertifikaten, und die Verwendung von authentifizierten Verschlüsselungsalgorithmen garantiert die Vertraulichkeit und Integrität der übermittelten Daten. Im Zuge dieser Masterarbeit wird eine Sicherheits- und Gefahrenanalyse von existierenden kryptografischen Protokollen für NFC durchgeführt. Zusätzlich werden diese Protokolle auf Effizienz überprüft. Basierend auf den Resultaten dieser Analyse wird ein bestehendes Protokoll erweitert, um die Sicherheit und Effizienz zu verbessern. Außerdem werden in dieser Arbeit konkrete Anwendungsfälle definiert, und dementsprechend das beschriebene Protokoll und die dazugehörigen Anwendungen implementiert.

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.

Date

Signature

Acknowledgements

This master's thesis has been conducted within the context of the IoSense-project[1] at Graz University of Technology at the Institute of Technical Informatics. Therefore, I would like to thank Ass.-Prof. Dipl-Ing. Dr.techn. Christian Steger for his guidance, patience and valuable input during this time. Especially, I want to thank Dipl-Ing. Dipl-Ing. Thomas Ulz, Bsc for time he spent on supporting me during countless meetings and delightful discussions from the day I decided to work on this thesis to the day I write these lines of text.

Studying and pursuing my goals wouldn't have been possible without the help of my beloved family and friends. Therefore, I would like to thank my parents Anita and Reinhard and my grandparents Renate, Ernestine, Franz and Vinzenz for their endless love, infinite patience and for their never-ending encouragement, which eventually helped me getting to the point where I am right now. I am so grateful for having such a lovely and supportive family and I will never take this for granted. During the time of my studies I also got to know my girlfriend Manuela, who supported me in every single aspect of my life ever since, for which I am infinitely grateful. Finally, I would also like to thank my study colleagues and friends Bernhard, Markus, Martin and Philipp for raising my interest in research and for their continuous support and understanding during this challenging but also rewarding time.

Graz, May 2019 Martin Zöhrer

[1] http://www.iosense.eu/

Contents

1 Introduction **10**
1.1 Motivation 10
1.2 Introduction 11
1.2.1 Objectives 12
1.3 Structure of this thesis 13

2 Prerequisites & Related Work **14**
2.1 Prerequisites 14
2.1.1 General terms 14
2.1.2 NFC 14
2.1.3 Cryptographic primitives 16
2.1.4 Threat analysis of protocols 21
2.1.5 Operating systems 23
2.2 Related work 24
2.2.1 Threats on NFC 24
2.2.2 Securing NFC 26
2.3 Conclusion 32

3 Design **34**
3.1 Application use cases 34
3.1.1 NFC-based configuration 34
3.1.2 Other use cases 35
3.2 Security and threat analysis of QSNFC 35
3.2.1 MITM-Attack 36
3.2.2 Replay-Attack 41
3.3 Performance analysis towards a modified QSNFC 43
3.3.1 Certificate chain and compression 43
3.3.2 Available encryption algorithms 43
3.3.3 Abort handshakes 43
3.4 Modified QSNFC 44
3.4.1 Supported encryption algorithms and cryptographic primitives . . . 44
3.4.2 Prerequisites 46
3.4.3 Initial handshake 47
3.4.4 Subsequent handshake 50
3.4.5 SD-messages 54

3.4.6 Connection tear down . . . 54
3.4.7 Message types . . . 56

4 Implementation **60**
4.1 Requirements & Consequences . . . 60
4.1.1 Requirements . . . 60
4.1.2 Consequences . . . 61
4.2 Implementation environment and external libraries . . . 62
4.2.1 IDE . . . 62
4.2.2 Unit Test-framework . . . 62
4.2.3 External libraries and tools . . . 63
4.3 libQSNFC . . . 65
4.3.1 Interface . . . 66
4.3.2 OpenSSL-API usage . . . 69
4.4 Two NFC-enabled IoT-devices . . . 71
4.4.1 Setup . . . 72
4.4.2 Development environment . . . 72
4.4.3 Implementation . . . 74
4.5 An NFC-enabled IoT-device and an NFC-enabled smart phone . . . 75
4.5.1 Simple APDU Exchange Protocol . . . 75
4.5.2 APDU-commands . . . 76
4.5.3 Development environment . . . 76
4.5.4 Implementation-Android . . . 77
4.5.5 Implementation-RaspberryPi . . . 81

5 Evaluation **82**
5.1 Protocol evaluation . . . 82
5.2 Demo evaluation . . . 83
5.2.1 Test scenarios . . . 84
5.2.2 Overhead and timings . . . 84

6 Conclusion and Future Work **87**
6.1 Conclusion . . . 87
6.2 Known limitations . . . 88

List of Figures

1.1 Four-layer model comprising QSNFC 11
1.2 Handshake types 12

2.1 Man-in-the-Middle 23
2.2 Android architecture simplified 24
2.3 NFC-Relay attack 26
2.4 Connection establishment (initial handshake) 30
2.5 Subsequent handshake 32

3.1 Major use cases 36
3.2 MitM attack 1 38
3.3 MitM attack 2 40
3.4 Replay attack 42
3.5 Simplified flow diagram 45
3.6 Initial handshake of the modified QSNFC 49
3.7 Subsequent handshake of the modified QSNFC 51
3.8 Abort initial handshake 53
3.9 SD-messages of the modified QSNFC 55

4.1 Library implementation overview 61
4.2 Data flow diagram 68
4.3 Two NFC-enabled IoT-devices 72
4.4 Secure communication from a server to an IoT-device 79
4.5 Android app 80

5.1 Message overhead for libQSNFC 85
5.2 Execution time for libQSNFC 86

List of Tables

2.1 NFC-Tag types defined by NFC-Forum 15
2.2 Key lengths of RSA and ECC compared to the security level 20
2.3 Protocol feature comparison 33

3.1 Supported authenticated encryption algorithms 46
3.2 Supported elliptic curves 46
3.3 Abort reasons 49
3.4 Message types 56
3.5 Inchoate CH-message 57
3.6 RJ-message 57
3.7 Uncompressed certificate chain 57
3.8 Compressed certificate chain 58
3.9 Complete CH-message 58
3.10 SH-message 58
3.11 SD-message 58
3.12 AB-message 59

4.1 Status byte for SAEP 76

5.1 Performance evaluation for different encryption modes 86

List of Listings

4.1 Key-pair generation . . . 63
4.2 Issuing an entity-certificate . . . 64
4.3 Issuing an intermediate CA . . . 64
4.4 Verification of a certificate chain . . . 65
4.5 Revocation of a certificate . . . 65
4.6 Usage of compress and decompress functions . . . 65
4.7 QSNFC-Interface . . . 67
4.8 QSNFC-Callbacks . . . 69
4.9 OpenSSL-Generation of an EC-key . . . 70
4.10 OpenSSL-Shared secret computation . . . 71
4.11 OpenSSL-Certificate chain verification . . . 71
4.12 Starting GDB-server on RaspberryPi . . . 73
4.13 Starting remote debug session . . . 73
4.14 Interaction with NXP's Reader library . . . 74
4.15 Build OpenSSL for Android's ABI armeabi-v7a . . . 78
4.16 Usage of Android's JNI . . . 78
4.17 Accessing Java from JavaScript . . . 79

Chapter 1

Introduction

This first introductory chapter consists of several sections. The first section gives a short motivation to emphasize the importance of the proposed solution, followed by a brief introduction to the related work, which acts as the basis for the work presented in this thesis. The subsequent section comprises a concise overview of the main objectives and a structural overview of this thesis concludes this chapter.

1.1 Motivation

Today's world is full of secure data being transmitted over the web. Financial transactions, company confidential data and even personal social and health related data is just the tip of the iceberg. The necessity of securing confidential data is obvious and hence, Secure Socket Layer (SSL) / Transport Layer Security (TLS) evolved as the major security standard for modern web communication over the last decades, placed in the transport layer of the OSI-Model, which makes it flexible and applicable for many applications. Hence, applications located on top of the transport layer can rely on SSL/TLS and do not need to take care of communication security.

However, in the last couple of years a new technology called Near Field Communication (NFC) arose, which provides mobile phones and smart devices the tools to communicate without the need of the web or the necessity of a pairing routine, as required by Bluetooth. In a world full of connected devices, aka the Internet of Things (IoT), NFC plays an important role due to its simplicity and connectivity while still being powerful and energy efficient. Fields of application for NFC-enabled devices and things vary from simple URL stickers over information retrieval in the industrial environment to the transmission of confidential data such as bank transactions using mobile payment, initial configuration of electronic devices such as routers, mobile ticketing or access controls.

In contrary to modern web communication standards though, NFC does not support similar cryptographic protocols compared to SSL/TLS. Therefore, application developers are condemned to implement their own security protocols to secure confidential data, which is being transmitted via NFC. This results from the fact that eavesdropping and data manipulation are major security concerns for NFC. Therefore, Hameed et. al. [HJS16]

noted, that the lack of standardized security protocols limits the spread of NFC-usage for applications where confidential data must be transmitted.

Given the wide variety of sensitive data, which is being sent over NFC, makes it inherently vulnerable to attackers. Thus, current research tries to counteract those attack vectors. However, existing solutions provide either inefficient protocols or partial security only, thus an efficient protocol providing confidentiality, authenticity and integrity suited for NFC is desirable. Furthermore, having the data transfer rate as the limiting factor, reducing communication overhead needs to be considered with special care when designing a security protocol for NFC.

Hence, the work within this thesis focuses on the design, implementation and evaluation of an efficient and secure cryptographic protocol suited for NFC-enabled devices. In fact, the proposed solution, which resides in the transport layer of the communication standard, is not only applicable for NFC, but also for other wireless communication protocols, which provide a unidirectional or bidirectional way of communication. Figure 1.1 illustrates a four-layer model with respect to NFC, comprising the proposed solution in the transport layer.

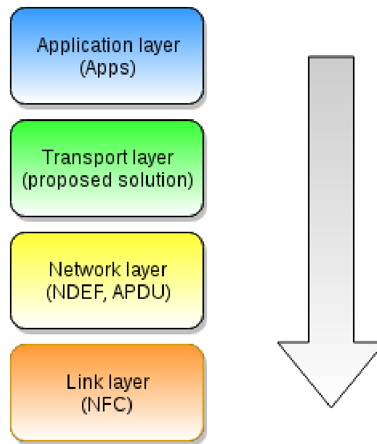
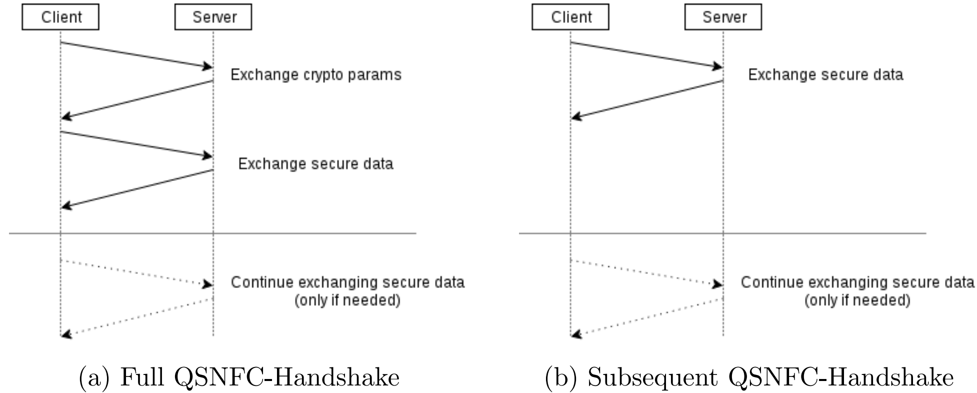


Figure 1.1: Four-layer model comprising QSNFC

1.2 Introduction

The protocol, which will be presented in this thesis is based on the work of Ulz et al. [UPS+18]. Therefore, the authors introduced *Quick and Secure Near Field Communication* (QSNFC), a protocol developed to secure NFC-sessions while keeping the data overhead as low as possible. To achieve overall security, QSNFC aims to ensure confidentiality and integrity by using authenticated encryption and authenticity by using SSL-certificates in a typical client-server model. In terms of NFC, the client is the active part, who initiates the NFC connection by approaching the passive NFC component, also referred to as the server. Even though the naming suggests multiple concurrent sessions, only one client communicates with one server at a time due to the nature of NFC.



(a) Full QSNFC-Handshake (b) Subsequent QSNFC-Handshake

Figure 1.2: Handshake types

QSNFC also applies a recent property of modern web browsers, the so called *Zero Round Trip Time (0-RTT)* property, which is for example supported by TLS 1.3 [KW16] for TCP connections and by QUIC [CLL+17] for UDP connections. The 0-RTT property entails the requirement to send confidential data in a secure manner, but without having to negotiate cryptographic primitives in an immediately prior handshake, which causes a significant speed up and less data overhead.

QSNFC fulfills the 0-RTT property in such a way, that, if client and server have never had a QSNFC-session before, they first need to establish trust and agree on a shared secret by applying an initial handshake. Within this initial handshake cryptographic primitives and certificates are exchanged, as shown in Figure 1.2a. After a successful initial handshake, the client is now in possession of a shared secret and therefore he can use this shared secret to secure the data for the remaining communication session. However, if client and server have already performed a QSNFC session successfully in the past, the client can start immediately sending secure data by using the stored shared secret from a previous session and thus, fulfilling the 0-RTT property, as depicted in Figure 1.2b.

Eventually, cache replacement strategies need to be considered for NFC-enabled devices, especially for those with memory constraints. This is because client and server must store cryptographic primitives. Therefore, the inventors of QSNFC identified several applicable ways to replace cached data. However, the appropriate cache replacement strategy depends on the actual field of application and therefore needs to be examined individually.

1.2.1 Objectives

The overall objectives of this thesis are to design, implement and evaluate a secure, efficient and easy applicable cryptographic protocol for a wireless communication. In particular, these main objectives are described in a more precise way in the following:

Design

In the design process, an efficient and secure wireless communication protocol needs to be defined, which provides confidentiality, integrity and authenticity. Hence, a security

analysis needs to be performed including common protocol attack scenarios. Moreover, as NFC is constrained by the data transfer rate, protocol overhead must be taken into account with special care.

Implementation

This objective comprises the implementation requirements. Therefore, a versatile and easy applicable library should be implemented, which contains the protocol related logic. This includes the protocol handshakes, cryptographic related operations and a comprehensive interface for applications, which are going to use this library.

Evaluation

The evaluating part should comprise the implementation of demo applications using the previously mentioned protocol library to cover the following use cases:

- An NFC-enabled mobile device and an IoT-Device communicate via NFC
- Two IoT devices communicate with each other via NFC

In addition to that, an evaluation of timings and protocol data overhead is needed.

1.3 Structure of this thesis

The subsequent content of this thesis is organized as follows: In *Chapter 2*, all necessary prerequisites are explained and selected related work is presented. This includes a brief introduction to NFC and corresponding modes of operation, cryptographic primitives and common threats on security protocols. In addition to that, selected topics from current research are going to be discussed, which cover security-related work on NFC. *Chapter 3* comprises the design process of the proposed protocol. Therefore, an existing protocol is going be analyzed and subsequently, a detailed protocol description based on that analysis is presented. This includes flow charts, message types and computational details. The subsequent *Chapter 4* illustrates the implementation processes of the protocol library and the demo applications. Therefore, the utilized development platforms and used programming languages are stated. The following *Chapter 5* recaps the security properties of the proposed protocol. Additionally, timings and overhead of the demo applications are evaluated. Finally, in *Chapter 6* the thesis is concluded by discussing limitations and outlining possible future work.

Chapter 2

Prerequisites & Related Work

First, this chapter discusses necessary prerequisites, starting with general terms such as NFC, but also cryptographic primitives such as encryption modes, key exchanges and freshness. In addition to that, different threat scenarios are presented which conclude the first section of this chapter. In the second half of this chapter, related work will be discussed. Among others, existing protocol solutions will be compared against each other with respect to the provided security features. Finally, a concise summary of the presented related work will complete this chapter.

2.1 Prerequisites

This section consists of high-level descriptions of selected prerequisites, which will be used throughout this thesis and are mandatory to understand.

2.1.1 General terms

Internet of Things

Huening [Hün19] describes the term *IoT* as the *Internet of Things*, where not only robots and expensive machines but also small sensors and other things are interconnected and equipped with computational power. This results from the constant miniaturization of electronic components.

Data compression

Salomon [SM10] defines data compression as follows:

> *Data is compressed by removing redundancies in its original representation, and these redundancies depend on the type of data. Text, images, video, and audio all have different types of redundancies and are best compressed by different algorithms which in turn are based on different approaches.*

2.1.2 NFC

NFC is a family of wireless communication standards based on RFID [TT10], which operate at 13.56 MHz at a maximum data rate of 424 Kbit per second [ISO07]. The active

Tag-type	Standards	Maximum memory size
Type 1	ISO/IEC 14443-3 Type A	2KB
Type 2	ISO/IEC 14443-3 Type A	2KB
Type 3	JIS-X-6319-4[3]	1 MB
Type 4	ISO/IEC 14443-3 Type A and ISO/IEC 7816-4	64KB

Table 2.1: NFC-Tag types defined by NFC-Forum

part generates an RF-field and powers the passive part, whereby this field is used for communication by modulation. NFC can operate in three different modes, as noted in [NFC], [Rol15], [JIC14]:

Reader-Writer Mode

The active device reads from a or writes to a passive device, while the passive device itself behaves as a tag. NFC-Tags can store small amounts of data in a simple memory structure. Table 2.1 illustrates the four distinct NFC-Tag types defined by NFC-Forum[1], the standards they are based on and their maximum memory size, as mentioned in [Rol15]. However, there exists also a fifth proprietary (to NXP Semiconductors[2]) NFC-Tag type. The Reader-Writer mode is compliant with the ISO-14443 standard [ISO08].

Peer-to-Peer Mode

In Peer-to-Peer mode, often abbreviated as P2P-mode, both entities can send data, which is compliant with the communication protocol presented in the ISO/IEC-18092 standard [ISO07]. Therefore, the Logical Link Control Protocol (LLCP) [LR10], which is located on top of the P2P-layer, provides a bidirectional communication between two NFC-enabled devices.

Card Emulation Mode

In Card emulation mode, a device emulates a passive component, for example a tag or a smart card. This way, an NFC-enabled device can communicate with another device which operates in NFC's Reader-Writer mode.

Application Protocol Data Unit (APDU):

Each NFC-chip provides a standardized set of commands (APDUs) for communication. These APDU-commands vary based on the implemented standards. Hence, an APDU command represents the lowest level of interaction with an NFC-device within this thesis.

[1] https://nfc-forum.org/
[2] www.nxp.com
[3] https://standards.globalspec.com/std/10070623/jsa-jis-x-6319-4

NFC Data Exchange Format

NFC-Forum defines within the NFC Data Exchange Format (NDEF) a standardized way how data, which is being sent over NFC, should be formatted. Jepson et. al. [JIC14] consider this as a major enhancement compared to the RFID-Technology. Furthermore it is noted, that NFC-sessions compliant with NDEF include so called NDEF-Messages, whereby an NDEF-Message can consist of one or more NDEF-Records. Each of these records come with a specific record type including a semantic meaning. This defines how an NFC-enabled device is supposed to handle the data contained in that record. Several defined record types do exist for NDEF-Records, as listed in the following:

- **Simple Text Records:** This type of record can contain arbitrarily text data, which does not trigger any particular behaviour at the receiving device.
- **URI Records:** These records represent URLs, which cause an NFC-enabled mobile device to open the browser, for example.
- **Smart Poster Records:** Smart posters may trigger the NFC-enabled device to send messages or add calendar entries, which however depends on the context.
- **Signature Records:** These records are supposed to contain a signature providing authenticity, however this record type is prone to attacks, as discussed later within this chapter.

Simple NDEF Exchange Protocol (SNEP)

Jepson et. al. [JIC14] consider SNEP [NFC11a] in their work as a possible implementation, how two NFC-enabled devices can communicate in a P2P manner.

In particular, SNEP defines a protocol which obeys a request-response scheme, similar to HTTP-requests. Thus, an initiator issues a request and the other communication party answers with a response. Hereby utilizes SNEP the underlying LLCP. Additionally, SNEP provides a set of defined request types and among others, also the *Get-Request* type, which is used within this thesis. While the initiating communication party requests data in the *Get-Request Message*, it can also send an NDEF message along with this request. The receiving party responds then with a *Success-Response code*, which includes an NDEF message as well. However, as stated in [NFC11a], the default SNEP-Server shall not accept *Get-Requests* and thus it must be implemented separately. In contrary to that, a *Put-Request Message* does not return an NDEF message.

2.1.3 Cryptographic primitives

This subsection illustrates selected cryptographic primitives. The first half of this subsection covers security fundamentals such as symmetric and asymmetric encryption, followed by encryption modes, message authentication codes and key exchange protocols. The other half discusses common attack scenarios on cryptographic protocols such as Man-in-the-Middle, Replay-or Relay attacks. To illustrate complex matters, the paragraphs within this subsection obey the following common definition: Two communication parties, historically referred to as *Alice* and *Bob*, want to exchange messages via an insecure channel without any further assumptions.

General terms

The following properties are desired within a secure communication, which involves multiple entities:

- **Confidentiality:** A message is confidential if only sender and receiver can read the content of the message, but any other outside party cannot.
- **Integrity:** Integrity of a message is given if the receiver can be assured that the message has not been altered.
- **Authenticity:** An authentic message guarantees, that a receiver can be assured that both the sender has sent the message and the message itself has not been altered.
- **Freshness:** A message is considered to be fresh, if it does not stem from a prior communication but is new (fresh).

Symmetric/Asymmetric Encryption

In general, confidential messages between two parties *Alice* and *Bob*, in the following abbreviated with A and B, can be encrypted in two different ways. In a symmetric encryption scheme, two parties encrypt and decrypt messages by using the same key, often referred to as common shared secret. A popular symmetric encryption scheme is the Advanced Encryption Standard (AES) [AES01]. In contrary, an asymmetric encryption scheme requires both communication parties to possess a private/public key pair. A message can be encrypted by party A using B's public key and subsequently only B can decrypt it using his own private key and vice versa. Asymmetric encryption evolved from a fundamental concept of cryptography, namely *Public Key Cryptography*, which will be described in a later paragraph. Among others, RSA [RSA78] includes a representative of a popular asymmetric encryption schema.

Message Authentication Codes

Message Authentication Codes (MAC) provide a way to verify the integrity of a message. A Message Authentication Code is typically computed by a one-way function h, often referred to as hash function, which must fulfill the following three requirements, as Rogaway and Shrimpton [RS04] noted:

- **Preimage-resistance**: It must be infeasible for any given hash value $h(y)$ to find an input x such that $h(x) = h(y)$, whereby y is unknown.
- **2nd-preimage resistance**: Given a message x and its hash value $h(x)$, it must be infeasible to find a second message x', where $x \neq x'$ such that $h(x) = h(x')$
- **Collision resistance**: It must be infeasible to find messages x and x', where $x \neq x'$, such that $h(x) = h(x')$

Authenticated Encryption

Bellare and Namprempre [BN00] introduced the term Authenticated Encryption, which carries out the idea of combining encryption with authentication. The authors identified three distinct modes of operation, as described briefly in the following:

- **Encrypt-and-MAC:** The plaintext is encrypted, and the MAC is computed separately and both outputs get stringed together.
- **MAC-than-Encrypt:** The MAC is computed first and then it gets added to the plaintext. Afterwards the bundle is going to be encrypted together.
- **Encrypt-then-MAC:** The plaintext gets encrypted first to a cipher text, which will then be used to compute a MAC. Both Ciphertext and MAC are transmitted.

The authors additionally recommend the use of Encrypt-then-MAC, as it provides the best level of security among those. Furthermore, Black [Bla11] mentions an extension to authenticated encryption, a so-called *Authenticated Encryption with Associated Data* (AEAD). Therefore, a message gets encrypted in an authenticated manner, but additional content may also be added to the MAC, which need not be encrypted but authentic. Hence, to ensure both confidentiality and integrity, the output of an authenticated encryption algorithm does not only consist of a cipher, but also contains an authentication tag.

Block Ciphers

Block Ciphers mark a crucial building block in the design of modern encryption algorithms, therefore Knudsen [Knu11] defines Block Ciphers as follows:

> *A block cipher with n-bit blocks and a k-bit key is a selection of 2k permutations (bijective mappings) of n bits. For any given key k, the block cipher specifies an encryption algorithm for computing the n-bit ciphertext for a given n-bit plaintext, together with a decryption algorithm for computing the n-bit plaintext corresponding to a given n-bit ciphertext*
> ...
> *Most block ciphers are so-called iterated ciphers where the output is computed by applying in an iterative fashion a fixed key-dependent function r times to the input. We say that such a cipher is an r-round iterated (block) cipher. A key-schedule algorithm takes as input the user-selected k-bit key and produces a set of subkeys*

AES

In 1997, the National Institute for Standards and Technology (NIST) conducted a competition seeking for a new Advanced Encryption Standard. The winner of this competition is Rijndael [DR02], which is generally referred to as AES. AES represents a symmetric encryption algorithm for block ciphers of an arbitrary length, but under the condition that it's a multiple of 32 bits. However, NIST restricts the input block length sizes of AES to 128, 196 and 256 bits, whereas the key is fixed to 128 bits. Blazhevski et. al. [BBSP13] noted that AES can be operated in different modes. Among others, there exist

also modes of operation for AES which do provide AEAD, as stated in [ZMH07], [Wät18] and [WHF03]:

- **Galois-Counter Mode (GCM):** Operates on a Finite Field (2^{128}). A Finite Field is a mathematical body which contains a finite number of elements, for which standard mathematical operations such as addition and multiplication are defined.
- **Counter-with CBC-MAC (CCM):** Combines AES-Counter Mode [LRW01] with CBC-MAC [Pre11].

Public Key Cryptography

The term PKC was mentioned first in Paragraph 2.1.3 on Symmetric/Asymmetric Encryption within this chapter. Hence, it will be discussed in greater detail in the following. As already noted previously, an entity must possess a private/public key pair to work with PKC. In particular, PKC is the basis of several fields of application:

- **Digital Signatures:** A party A can prove the authenticity of a message by generating a signature with its private key. Any other outside party can then verify the authenticity of the message by applying A's public key on the received message and comparing it to the received signature. Kaliski [Kal11] presents a Digital Signature Scheme.
- **Key Exchanges:** In [BPPT17] the authors describe a *Key Exchange Protocol* as a cryptographic primitive for two parties A and B to agree on a shared secret over an insecure channel. In 1976, Diffie and Hellman [DH76] proposed the *Diffie Hellman Key Exchange Protocol*.
- **Asymmetric Encryption:** Asymmetric encryption has already been described in a previous paragraph and is added to this list only for completeness.

Certificates:

The term *Certificate* is related to PKC. Carlisle Adams [Ada11a] defines it as follows:

> *A certificate is a data structure signed by an entity that is considered (by some other collection of entities) to be authoritative for its contents. The signature on the data structure binds the contained information together in such a way that this information cannot be altered without detection. Entities that retrieve and use certificates (often called relying parties) can choose to rely upon the contained information because they can determine whether the signing authority is a source they trust and because they can ensure that the information has not been modified since it was certified by that authority.*

Furthermore, the author notes that certificates primarily contain information about the public part of a private/public key pair along with other metadata, but also relevant validity information, usage of cryptographic protocols and any other constraints. Certificates are involved in the process of authentication among (partial) untrusted parties.

Security level (bits)	RSA-Key length (bits)	ECC-Key length (bits)
80	1024	160-223
112	2048	224-255
128	3072	256-283
192	7680	384-511
256	15360	512-571

Table 2.2: Key lengths of RSA and ECC compared to the security level

Certificate Authority:

A *Certificate Authority (CA)* is a privileged entity with given trust of a group of unprivileged entities. CAs are in charge of creating and distributing certificates and provide ways to verify the content of a certificate. Moreover, CAs may grant other entities a certificate or the privilege to create and distribute certificates themselves. Consequently, this creates a so-called *chain of trust*, which starts from a Root-CA over possible Intermediate-CAs to a certificate of an untrusted entity. In an authentication process, an untrusted entity can be verified by starting from its certificate until a trusted CA is found, which comprises the traversal of a certificate chain. If no trusted CA is found, the entity remains untrusted.

X.509

According to De Cock [De11], X.509 certificates are based on the ASN.1 syntax. These certificates contain different types of records, whereby some of them are mandatory and some are optional. All certificates mentioned within this thesis are based on X.509 certificates.

Elliptic Curve Cryptography

Koblitz [Kob87] proposes a cryptographic system based on elliptic curves, which can be used for PKC. Hence, this system is called Elliptic Curve Cryptography (ECC), whereby ECC can be applied for encryption, signatures and key exchange algorithms. Thus, Johnson et. al. [JMV01] present a Signature-protocol in their work called Elliptic Curve Digital Signature Algorithm, in short ECDSA, which is based on ECC. Compared to RSA-DSA, ECDSA requires significantly fewer key bits to achieve the same level of security, as noted in [MHS10]. Therefore, the authors compare the security level in bits and key sizes in bits for RSA and ECDSA, which is depicted in Table 2.2. Additionally, the authors in [RMF+15] present Elliptic Curve Diffie Hellman (ECDH), a way how the *Diffie Hellman Key Exchange Protocol* can be implemented using ECC.

Key Derivation Function

A *Key Derivation Function* (KDF) derives one or more keys from a shared secret, typically obtained from a prior key exchange protocol, as described previously. Additionally, human defined passwords can also be used as input for key derivation functions. In either way, it is generally recommended to use a separate key derivation function to obtain keys which can then be used for encryption. Krawczyk [Kra10] describes an HMAC-based key derivation

function (HKDF). Furthermore, *Password-Based Key Derivation Function* (PBKDF) are described in [MKR17].

Challenge-Response protocols

According to Just [Jus11], a challenge-response protocol can be used in an identification process, which typically involves two entities, a *challenger* and a *verifier*. Therefore the challenging entity generates a challenge and transmits it to the proving entity, which solves the challenge and sends it back to the challenger. Subsequently, the response can be verified by the challenging entity. Another application of challenge-response protocols is to ensure freshness, which is done by sending a new challenge each time the protocol is going to be applied.

Nonces

The above described challenge-response protocols use challenges, which are only used once to ensure freshness. Within this context, a so called *Nonce* is a number which should only be used once. In challenge-response protocols, the challenging entity typically transmits a nonce, which is guarded by a security mechanism, to its communication partner. On receipt of that nonce, the communication partner is going to apply a function on that nonce and responds the result. After verification, the challenger can be assured that the received message is fresh and does not stem from a prior session.

Pseudo Random Number Generator

According to Koeune [Koe11], many cryptographic primitives require random numbers. These random numbers are going to be used for encryption algorithms, challenge-response schemes, unique identifiers or key derivation algorithms. However, it is difficult to generate a real random number on a PC, therefore the concept of *Pseudo Random Number Generators (PRNG)* was introduced. A PRNG is seeded with a secret random value and produces a randomly-looking sequence of values, which is deterministic though. OpenSSL[4] states that a so called *strong pseudo-random number* can be used for cryptographic primitives whereas a *weak pseudo-random number* should only be used for non-cryptographic purposes.

Initialization Vectors

An initialization vector (IV) is used to seed an encryption algorithm. Some encryption modes require that an IV in conjunction with a key must not be used twice to avoid leakage of information. Furthermore, IVs are used in key generation and derivation functions.

2.1.4 Threat analysis of protocols

This subsection discusses common threat scenarios for cryptographic protocols. First, the concept of Man-in-the-Middle (MITM) is described in detail and furthermore, several

[4]https://www.openssl.org

attacks based on MITM are presented. Finally, a brief introduction to Denial-of-Service attacks concludes this subsection.

Man-in-the-Middle

Schneier [Sch95] mentioned the concept of *Man-in-the-Middle* in his work on Applied Cryptography, in which two honest and trustworthy parties *Alice* and *Bob* want to communicate by sending messages over an insecure channel. An insecure channel implies that none of the two parties is neither in control of what happens along the communication line, nor able to detect any passive or active malicious adversaries. That is, *Alice* and *Bob* may send and receive messages, however these messages are unprotected during the transmission. Hence, both do not have any knowledge of whether the message has been eavesdropped or altered, neither each of the parties can be assured to whom he or she is talking to. Thus, Schneier introduced *Eve*, an evil third party who desires to eavesdrop as much information as possible, as depicted in Figure 2.1a. Even worse, a malicious *Mallory* is not only able to read messages, but also may forge any sent message without the knowledge of *Alice* and *Bob*. Figure 2.1b illustrates *Mallory*, who is intercepting messages between *Alice* and *Bob*. Based on the concept of MITM, several attacks scenarios can be formulated, as given in the following list:

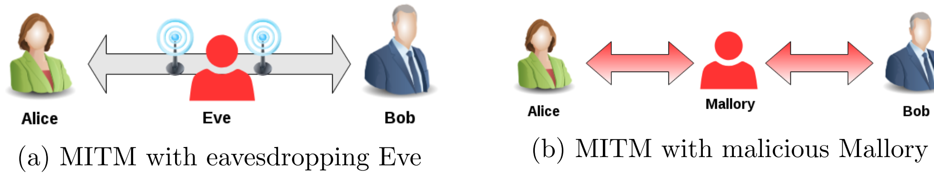
- **Relay-Attack:** In a Relay-Attack, an evil adversary delays the transmission of a message. As a practical example, *Alice* uses a wireless key to receive a permission entrance to her car *Charlie*. Suppose *Alice* is physically far away from *Charlie*. Then *Eve*, who is in the proximity of *Alice*, receives the signal from the wireless key and transmits it to *Evelyn*, who is standing right in front of *Charlie* and subsequently gets access by re-transmitting the received signal. Hence, the evil parties *Eve* and *Evelyn* successfully mounted a Relay-Attack. Another example of a Relay-Attack is given in [Des11].

- **Replay-Attack:** Adams [Ada11b] defines Replay-Attacks as a two-step procedure. First, an evil party *Eve* is going to eavesdrop and record (parts of) a communication by *Alice* and *Bob*. In the next step, *Eve* replays (parts of) the previously recorded message to *Alice* or *Bob* or both.

- **Interleaving-Attack:** In this attack, an evil adversary uses interleaving sessions with *Alice* (and *Bob*) to inject malicious data.

- **Reflection-Attack:** In a Reflection-Attack an evil adversary tries to fool a communication partner by sending messages back to the originator, for example in a challenge-response scenario.

- **Downgrade-Attack:** Downgrade-Attacks try to weaken a communication by 'downgrading' the negotiated level of security. Therefore, an evil *Eve* would try to inject/alter a message in a communication between *Alice* and *Bob*, where these two honest parties agree on the level of security.



(a) MITM with eavesdropping Eve

(b) MITM with malicious Mallory

Figure 2.1: Man-in-the-Middle

Denial of Service

Eric Cronin [Cro11] defines the term *Denial of Service*, abbreviated as DoS, as the unavailability of a service to a legitimate entity. However, in practice this term is only used if the unavailability is the result of an intended attack, which forces the system to behave differently as it is supposed to. DoS can be achieved, among others, by applying MITM-based attacks.

2.1.5 Operating systems

This subsection lists all used operating systems and describes each of them briefly. Furthermore, an outline of the usage of each operating system within the context of this thesis is given.

Raspbian

Raspbian[5] is a free operating system based on Debian, which includes pre-built libraries for the RaspberryPi-Platform[6]. A RaspberryPi is a small hand-sized computer providing several interfaces such as USB, Ethernet, WLAN and HDMI. In addition to that it contains numerous General-Purpose-Input-Output (GPIO) pins to extend the functionality with an NFC-Shield, for instance. Within this thesis, RaspberryPis running Raspbian with an attached NFC-Shield are used as NFC-enabled IoT-devices.

Android

Android[7] is a mobile operation system developed by Google. It is based on a Linux kernel and runs each application in its own sandbox. Peripherals such as NFC, Camera and Storage can be accessed by using defined interfaces in code. A detailed reference on programming Android apps can be found on the Android-Developer website[8], moreover Figure 2.2 illustrates Android's architecture. An NFC-enabled, Android-based mobile phone is used within this thesis.

[5] https://www.raspbian.org/
[6] https://www.raspberrypi.org/
[7] https://www.android.com/
[8] https://developer.android.com/

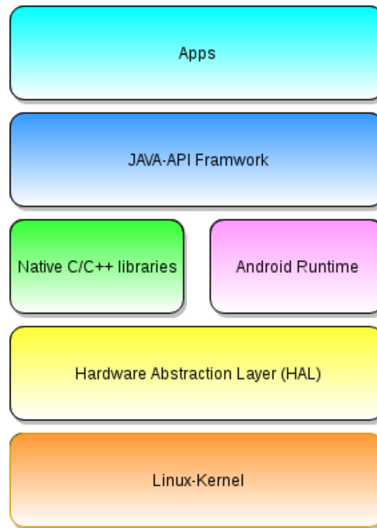


Figure 2.2: Android architecture simplified

2.2 Related work

This section discusses related research publications, whereby relevant papers for both the design process and for the implementation will be presented and analyzed. The first part of this section primarily focuses on theoretical attack vectors for NFC, while the second part presents practical attacks and the third part is dedicated to work about securing NFC connections. Finally, the subsection *Conclusion* summarizes the crucial points of the related work and furthermore, the implications of the individual results on this thesis will be highlighted.

2.2.1 Threats on NFC

This subsection discusses threats on NFC. The first part comprises attacks based on simple RF-signal transmission, while the second part describes several other types of attacks.

Attacks based on RF-transmission

Haselsteiner and Breitfuss [HB06] identify the following threat scenarios in their work on *Security in NFC*:

- **Eavesdropping:** Due to the wireless way of communication entailed with NFC, it is implicitly vulnerable to eavesdropping. That is, in an NFC-session, two parties communicate via RF waves, thus an attacker could simply receive those waves via an antenna. However, such an NFC-session is usually conducted within a range of centimeters, therefore the attack scenario highly depends on how close the attacker is to the transmitting part. Moreover, the used equipment of both sender and receiver plays an important role in such a scenario, which includes quality of antennas, RF-signal decoders and the overall location (buildings, walls or other impacting obstacles). Hence, it is not possible to give an exact distance without further assumptions on the environment.

Another important factor, which must be considered when applying such an attack is whether the sender's device operates in active or in passive mode. According to the authors, it is much harder to eavesdrop a communication if the sending device does not generate its own RF-field for communication (passive-mode) in contrary to if the sender generates its own RF-field (active-mode). For active mode, the authors estimate the possible eavesdropping distance at about 10 meters, whereas the passive mode is only applicable within a range of 1 meter.

- **Data corruption:** In contrary to passive eavesdropping, attackers could also corrupt data within a communication by emitting a noise signal at the right time. Therefore, the attacker must be aware of the available frequency spectrum for the communication. This would imply that the originally intended receiver would just receive invalid data.

- **Data modification:** While the goal from the previous attack is to just interrupt a communication with invalid data, this attack aims to modify transmitted data. The authors state that this attack highly depends on how the signal is modulated and which encoding is used. However, if the right modulation and the proper encoding is chosen, an attack as feasible.

- **Data insertion:** Instead of just modifying existing data, the authors also mention a data insertion attack scenario. In this scenario, the attacker tries to insert data in an existing communication. However, this is only feasible under certain conditions according to the authors:

  > *This means that the attacker inserts messages into the data exchange between two devices. But this is only possible, in case the answering device needs a very long time to answer. The attacker could then send his data earlier than the valid receiver. The insertion will be successful, only, if the inserted data can be transmitted, before the original device starts with the answer. If both data streams overlap, the data will be corrupted.*

- **Man-In-The-Middle:** The basic concept of a MITM-attack has already been described in the previous section, however this paragraph examines the applicability of a MITM-attack based on the context used above, that is two communication parties want to communicate via NFC and an attacker equipped with RF-transmission and receiving devices. Therefore, the above-mentioned authors investigated this scenario, however they concluded that it is practically impossible to mount such a MITM-attack, which is based on RF-Signal transmission and interception, due to nature of NFC.

Attacks based on relaying communication

In the work of Maass et al. [MMS+15], the authors present a Relay-Attack on top of NFC. Therefore, two Android based mobile devices were used to delay a communication between an NFC-Tag and an NFC-Reader, as depicted in Figure 2.3. In the presented attack, data from an NFC-Tag was captured by an NFC-enabled mobile device. Subsequently this data was transmitted via a web protocol to another, physically far located NFC-enabled

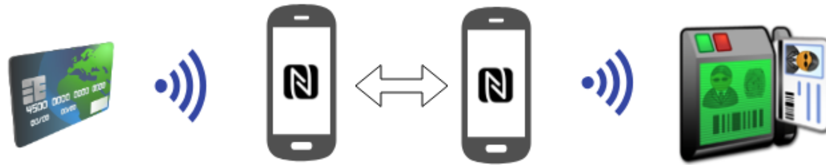


Figure 2.3: NFC-Relay attack

mobile device, which in turn transmits the received data to an NFC-Reader. Hence, the authors were able to use this attack scenario in conjunction with a popular contactless card payment system and an electronic passport document. Roland et. al. [RLS13] used a similar attack setup, where they conducted a Relay-Attack on Google-Wallet.

Using the presented attack scenarios, an attacker is not only able to successfully perform a passive MITM-attack and eavesdrop the communication, but also actively modify, insert or delete data, which opens the doors for all sorts of attacks related to MITM, as described in the previous preliminary section.

Conclusion

Within this subsection, multiple threat scenarios for NFC were presented. First, attacks based on RF-Signal transmission were discussed, yielding multiple attack scenarios where an attacker with the proper equipment can easily eavesdrop data, which depends on the actual setting though. Moreover, the attacker may flip certain bits or interrupt an NFC session by sending noise on a proper frequency. In the second part, a MITM-attack based on NFC was presented, which enables an attacker to arbitrarily forge messages.

2.2.2 Securing NFC

This subsection covers related work, which aims to secure an NFC connection. Within this subsection several approaches are discussed. First, an NDEF record type definition providing authenticity and integrity is given, followed by a scheme which aims to provide confidentiality. Subsequently, several other schemes providing confidentiality, authenticity and integrity are presented and analyzed. Finally, a concise comparison of all the presented schemes concludes this chapter.

NDEF-Signature record type

As already stated in the preliminary section of this thesis, NDEF contains a definition for a Signature Record Type[9]. A Signature Record Type Definition (Signature RTD) aims the ensure integrity and authenticity of an NDEF message, whereby multiple successive NDEF-records can be included in one Signature-record. However, Roland et. al. [RLS11] pointed out several security vulnerabilities in their work on the NDEF Signature RTD from 2011. First, the authors introduced the term *Record Composition Attack*, which is a two step attack:

[9] https://nfc-forum.org/purpose-nfc-ndef-signature-records/

1. At first, (possible unrelated) NDEF-records, which were signed by the same entity, are selected.
2. Subsequently, all records from different contexts are combined into a single context, which yields the content of a new message, for which the previously generated signature is valid.

According to the authors, such an attack could lead to a DoS or fraud, which in turn destroys the trust relationship between a consumer and a service provider.

In that same paper, the authors describe another attack scenario based on the NDEF Signature RTD. Thereby, the feature of remote signatures and certificates by URIs is identified as potential weakness, because it could lead to security vulnerabilities and could cause privacy issues. Remote signatures and certificates within an NDEF-Signature RTD are supposed to provide authenticity and integrity by referencing to a remote URI, which provides the actual signature/certificate of the included data. Hereby, the crucial point is that the URI itself is not protected by any kind of security measure. Therefore attackers could exploit this issue by triggering maloperation, for example:

- Access services by using cookies in URIs, which are normally only available to a certain user.
- Usage of services, which are only available on the receiving device.
- Execution of malware via buffer overflows.

However, according to Saeed and Walter [SW11], the presented record composition attack described above needs further changes to be carried out, such that a valid signature can be obtained. The authors conclude that the easiest way to counteract such attack scenarios is to sign all header fields within an NDEF message, which is practically not possible though. Thus, a signature scheme is proposed, which may contain multiple record chunks and hence, making it more difficult to exploit vulnerabilities in the NDEF-Signature RTD. In addition to that, NFC-Forum released in 2015 a new version of the NDEF-Signature RTD (version 2.0), addressing the above-mentioned issues in [NFC14]. Furthermore, Thomas Korak and Lukas Wilfinger [KW12] discuss the implementation related details on using the NFC-Signature RTD with respect to the record creation and validation.

Protected NFC data exchange

Hammed et. al. [HJS16] propose to extend the existing range of NDEF record type definitions by a so-called *Encryption Record Type Definition* (ERTD). The authors carry out the idea to design such an ERTD like the above described Signature RTD. Additionally, several encryption algorithms are supported within this definition. Furthermore, the authors anticipate the usage of both ERTD and Signature RTD in conjunction in future to provide authenticity, integrity and confidentiality.

ECMA-Standards

ECMA[10] is a standardization organization in the field of information and communication systems, which distributes, among others, the NFC-SEC standard [Ecm15a] for secure channels and shared secret services for NFC. This standard describes a protocol comprising several cryptographic mechanisms such as key agreement, key confirmation and Protocol Data Unit (PDU)-security. [Ecm15b] and [Ecm15c] describe how NFC-SEC can be used together with ECDH and AES/AES-GCM to ensure confidentiality while [Ecm17a] and [Ecm17b] describe how entity authentication could be performed, either symmetric or asymmetric. These standards state though, that the presented security protocol does not address security mechanisms for particular applications including NFC tags, as they usually do not provide any computational power. Another limiting factor which comes with the use of NFC-SEC is that according to [Uri14] only a few NFC-chips have implemented these features and moreover modern smart phones are not supported. Furthermore, as the last two named references involve patent rights, this thesis does not cover them in greater detail nor use them in any of the following chapters, but those are solely listed here for completeness.

NFC in health care

Dunnebeil et. al. [DKK+11] propose the use of NFC tags for emergency cases to store medical data. As this information might be confidential the authors introduce the so called *Encrypted NFC emergency tags*, where personal medical data can be stored on NFC tags in an encrypted way. In addition to that, Sethia et. al. [SGM+14]. present an NFC-based health care system. This proposal involves the transmission of confidential data by using the three different NFC-modes.

Transport Layer Security

According to Heinrich [Hei11], the TLS-standard contains a set of protocols for securing modern web communication, which is used in mobile banking, E-Commerce and in confidential transactions. All modern browsers do support TLS, which is the successor of SSL. TLS defines a set of protocols and cryptographic primitives to secure session-based communications in a client-server fashion. In the OSI-Model, it is based on top of the transport layer.

TLS over QSNFC

Pascal Urien [Uri14] proposes the idea to port TLS to NFC and include it in the LLCP layer, which acts as the basis for the Peer-2-Peer communication mode. However, TLS, which was originally designed for modern web communication, entails a significant overhead for NFC connections. Thus, the author reports that a full TLS session opens in 2 seconds in a test scenario with two NFC-devices.

[10] https://www.ecma-international.org/

Quick and Secure NFC

As already stated in the introductory part of this thesis, QSNFC [UPS+18] acts as the fundamental basis of this thesis, hence this paragraph describes it in greater detail, as it is mandatory to understand the concepts for the threat analysis and the design part presented in the next chapter. First, the protocol is described in detail, which includes a concise textual description and an illustrative flow diagram. Subsequently, message types and included fields are given and an overall summary of the findings by the authors concludes this section.

QSNFC is a Transport Layer protocol suited for NFC-based communications, which is similar to TLS for modern web communications. QSNFC aims to ensure confidentiality, integrity and authenticity, therefore confidentiality and integrity are ensured by the usage of authenticated encryption algorithms and authenticity is guaranteed by using certificates. In the presented model, a client wants to establish a secure connection with a server via QSNFC. In terms of NFC, the client is the active part, who initiates the NFC session by approaching the passive NFC component, also referred to as the server.

In a typical web communication protocol, a client authenticates a server by using certificates and certificate authorities. Hence, also QSNFC relies on this kind of authentication mechanism. Therefore, the protocol contains corresponding fields for certificates, however due to the significant data overhead entailed with the usage of certificates, the next chapter of this thesis elaborates the data overhead in greater detail.

Additionally, QSNFC provides a 0-RTT property, which reduces the limiting factor of data overhead. In particular, the 0-RTT property requires, that for a handshake between two parties, which takes place again, no re-occurring key agreement should be needed to agree on a new shared secret. In QSNFC, client and server agree on cryptographic primitives in a so-called *initial handshake*. In all successive sessions, client and server only need to perform a *subsequent handshake*. Hence, they re-use these cryptographic primitives to establish a secure communication without the need of another initial handshake, thus fulfilling the 0-RTT. However, in the first connection attempt, when client and server do not share any common cryptographic primitives or secrets, the 0-RTT property is not fulfilled. The next two paragraphs describe the concept of initial and subsequent handshakes in detail:

Initial Handshake: In a typical communication scenario, where client and server have never met before, a handshake is needed to perform a key agreement procedure and to enable the 0-RTT property for successive round trips. Thus, QSNFC defines the *Initial Handshake*, which is used in the first connection attempt, that is, client and server do not share any common cryptographic primitives. A client can initiate a QSNFC session by sending an *inchoate Client-Hello (CH)* message containing the client's id to the server. Subsequently, a server can respond with a *Reject (RJ)* message, which contains the following fields:

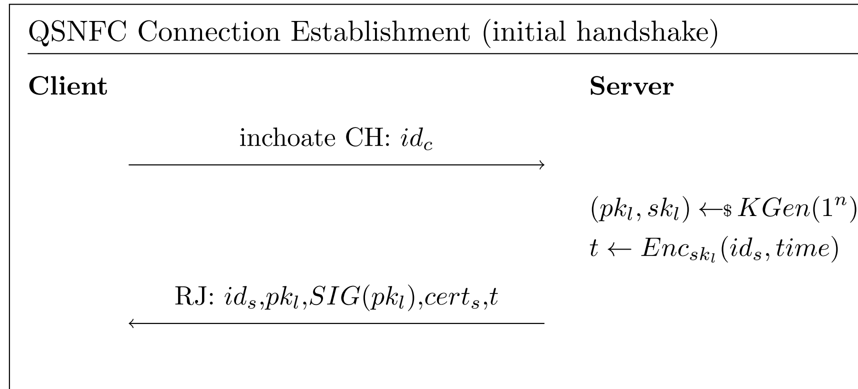


Figure 2.4: Connection establishment (initial handshake)

- The server's long-term public key, which is utilized in all subsequent handshakes to derive a new shared secret. This shared secret is then used for authenticated encryption.
- The server's certificate and the corresponding certificate chain, which must be verified by the client. By verifying this chain, the client can be assured that he or she is talking to the server.
- A signature of the server's long-term public key signed with the private part of the server's certificate key.
- A so-called source address token, that contains the server's id and a nonce chosen by the server. The client transmits this token in every subsequent handshake to prove knowledge of the server's identity.

Subsequently, when the client receives the RJ-message from the server, the authenticity of the message must be verified first. To do so, the client tries to build a certificate chain, starting with the server's certificate and the certificate chain and hence, yielding the result whether the server is trustworthy or not. If the client is able to build the certificate chain and trusts the server, the client must also verify, that the server is in possession of the private key corresponding to the public key within the received certificate. Therefore, the client verifies the integrity of the received long term public key by computing the signature of it and compares it with the received signature within the RJ-message. Figure 2.4 illustrates the initial handshake by using the fields described above.

Subsequent handshake: While the initial handshake was used by the client to authenticate the server by verifying the certificate chain, the subsequent handshake is designed to provide a secure way of communication. Within the initial handshake, the client received the server's long-term public key, which is subsequently used to derive shared secrets. To do so, the client generates his own ephemeral private/public key pair and derives in conjunction with the received long-term public key a temporary shared secret. This temporary shared secret is going to be used to secure data within that subsequent handshake. Therefore the client transmits the complete Client-Hello (CH)-message to the server, containing the following fields:

- The client's id to identify the client by the server.
- The client's ephemeral public key. This key is needed by the server to compute the shared secret.
- The source address token to prove the client's ownership of the server's identity.
- Confidential data secured by an authenticated encryption algorithm using the initial shared secret.

Upon receipt of the complete CH-message, the server computes the shared secret using the private part of his own long-term key and the received ephemeral public key from the client. Afterwards, the server decrypts the received message and verifies the integrity. Subsequently, the server generates his own ephemeral key pair, which is intended to be used for any further message within the current communication, but after the subsequent handshake. As a response to the client, the server composes the Server-Hello (SH)-message, which contains the following fields:

- The server's id used by the client to identify the server.
- The server's ephemeral public key used to create a shared secret for the successive messages within this communication session.
- Confidential data, which is secured with an authenticated encryption algorithm by using the initial shared secret.

Once the client has received the SH-message, both entities can compute the final shared secret by using the own private part and the opponent's public part of the ephemeral keys which were exchanged in the prior subsequent handshake. This final shared secret can then be used to secure data in successive Standard-Data (SD) messages within the communication session. These SD-messages are going to be sent in a request-response manner. Figure 2.5 illustrates both the subsequent handshake and the exchange of SD-messages in a request-response manner. Hereby, the messages above the dotted line represent the subsequent handshake. The messages underneath the dotted line represent SD messages, which can be sent in arbitrary order and direction. The function $\mathrm{Enc}_{\mathrm{k}}(\mathrm{data})$ denotes an authenticated encryption function using key k. The parameters for the subsequent handshake are: the client's and server's ephemeral public key ($\mathrm{pk}_{c\,|\,s}$, $\mathrm{sk}_{c\,|\,s}$) and additionally, the client has to transmit the source address token.

Security: The authors state, that the key agreement process is based on QUIC [CLL+17]. Thus, the security properties do not differ. However, as it will be shown in the next chapter of this thesis, QSNFC consists of several security vulnerabilities.

Protocol overhead: In a resource constrained scheme, protocol overhead needs to be considered with special care. Thus, the inventors of QSNFC consider the certificate chain as the major impacting factor for data overhead, stating that a standard certificate is approximately 1kB in size. However, as the certificate chain may include more than one certificates in the chain, this must be treated as an important aspect in the design phase. Thus, the 0-RTT was applied to reduce the number of necessary handshakes and hence, the number of certificates which have to be transmitted over NFC.

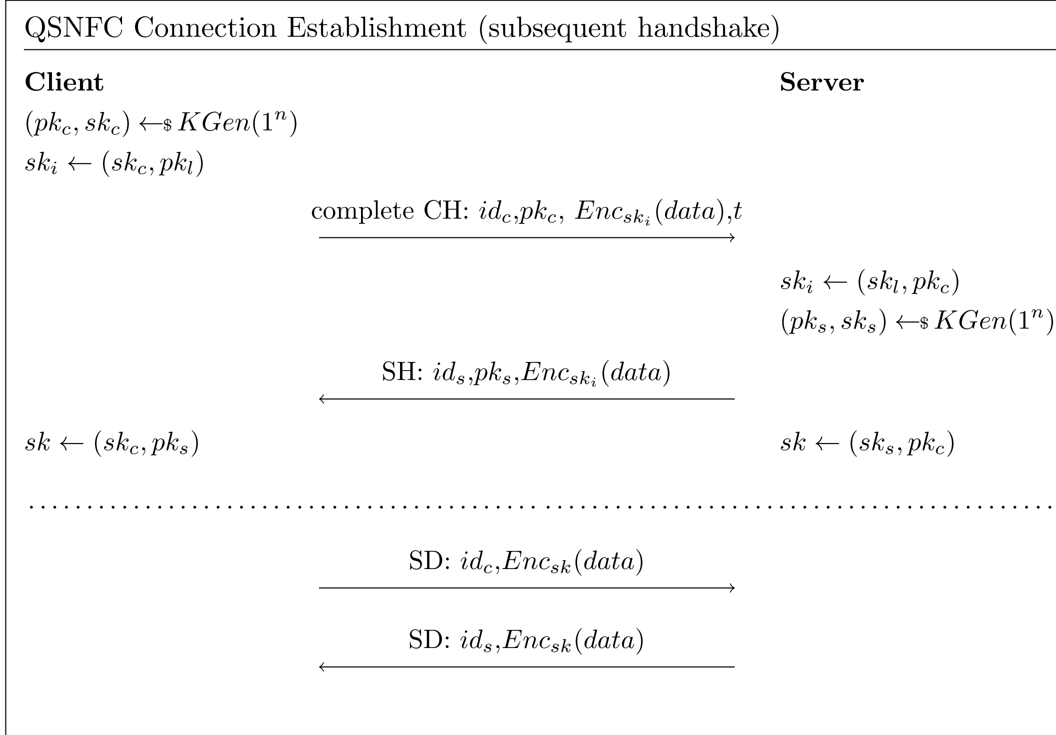


Figure 2.5: Subsequent handshake

2.3 Conclusion

This section summarizes the presented related work, compares the identified properties of each solution and gives an outline of desired features for the proposed protocol within this thesis. The Section 2.2.1 *Threats on NFC* depicts several vulnerabilities for NFC sessions. In particular, not only eavesdropping is a potential threat but also malicious attacks based on an active MitM. Hence, NFC sessions must be secured to ensure authenticity, integrity and confidentiality and moreover, the solution must be efficient while still being flexible and adaptable. Clearly, such a protocol requires computational power from both communication partners, hence this work focuses NFC-enabled devices, but not on simple NFC-tags. However, none of the presented security countermeasures does fulfill these requirements. NDEF itself is shipped with a Signature RTD as mentioned in Section 2.2.2, but at least an earlier version is prone to attacks and it does only provide authenticity. Subsequently, an *Encryption Record Type Definition* was introduced, which defines multiple encryption algorithms but does not consider authenticity nor integrity. ECMA defines several standards for NFC-SEC, but due to the involved patents and missing implementation on many modern devices, this is not an option. Furthermore, practical examples for confidential data transmission via NFC were illustrated within the context of health care, however the presented solutions are bound to certain fields of application and are not applicable for other use cases. Hence, the concept of *TLS over QSNFC* introduced the idea to use an existing standardized protocol, which has been used in modern web communication for decades. However, the significant overhead entailed with TLS is not

Solution	Authenticity	Integrity	Confidentiality	Efficient	Free
Signature-RTD	✓	-	-	-	✓
Encryption-RTD	-	-	✓	-	✓
TLS over NFC	✓	✓	✓	-	✓
ECMA	✓	✓	✓	✓	-
QSNFC	✓	✓	✓	✓	✓

Table 2.3: Protocol feature comparison

applicable for NFC in an efficient way. Finally, QSNFC was introduced, a versatile transport layer protocol for wireless communication, providing authenticity by using certificates in the same fashion as TLS does. Moreover, integrity and confidentiality are ensured by authenticated encryption, while still being lightweight enough to provide an efficient way of communication. Table 2.3 compares the presented solutions.

Even though QSNFC sounds promising, there are several security vulnerabilities, which are going to be shown in the next chapter. Thus, modifications are necessary to eliminate the identified security vulnerabilities, which are also going to be discussed in the next chapter.

Chapter 3

Design

Based on the results of the previously discussed related work, this chapter focuses on the design of a secure protocol, which provides authenticity, integrity and confidentiality. First of all, application use cases are given, for which the proposed protocol should be applied. After that, a comprehensive security analysis of an existing protocol is going to be conducted. The outcoming results will then be used to strengthen the security properties of this protocol and furthermore, several mechanisms to reduce the data overhead will be discussed and applied. This leads to a detailed protocol definition including flow diagrams, low level descriptions of computational operations and message types.

3.1 Application use cases

This thesis is conducted within the context of the IoSense-project (Flexible FE/BE Sensor Pilot Line for the Internet of Everything). Therefore, application scenarios based on smart sensor solutions are listed in the following. However, this proposed solution is by no means bound to an industrial field of application, hence other application scenarios are highlighted as well.

3.1.1 NFC-based configuration

In the context of the IoSense project, the *TrustWorSys - Towards Trustworthiness for IoT Sensors*[1] is mentioned, whereby smart sensors equipped with NFC-enabled devices are used for configuration in an industrial environment. Ulz et. al. [UPH^{+}17] mentioned configuration of smart sensor systems via NFC.

Smart sensor - mutual configuration

In a Machine-to-Machine scenario, two smart sensors with an NFC-enabled device can configure each other in an uncontrolled environment. Hence, to secure the configuration and to prevent any eavesdropping or malicious configuration modification, a secure protocol is needed.

[1] http://www.iosense.eu/index.php/project/demonstrator/

Human configures smart sensor

Not only machines can configure machines but also humans with a corresponding NFC-enabled mobile devices, for example due to a configuration update. Hence, also the communication between a smart phone and a machine must be secured.

3.1.2 Other use cases

The protocol presented within this thesis is not only applicable in the context of NFC-based configuration, but also in several other fields of application, where confidential messages are being transmitted. Especially, in a context with a mobile smart phone as initiating part and an IoT-device as passive part, the following use cases can be defined:

- **Mobile banking:** A human with an NFC-enabled smart phone pays at a banking terminal, which is also equipped with NFC.
- **Identification (Access control):** In order to get access to a certain entity such as a building or a car, a human with an NFC-enabled smart phone communicates with an NFC-equipped receiver terminal.

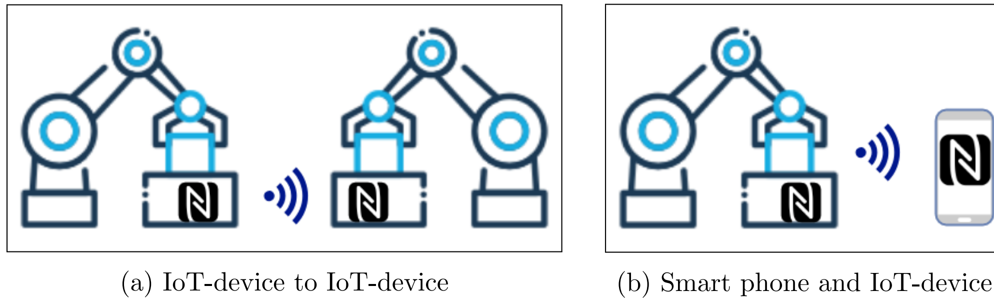
Clearly, more application scenarios can be formulated, however the above described use cases can be grouped into two major use case settings:

- **IoT-device to IoT-device:** In the first major use case setting, two IoT-devices equipped with NFC communicate in an unsecured environment. Hereby, one IoT-device acts as an active NFC-entity, also referred to as the client. This client initiates an NFC session with the second NFC-enabled IoT-device, which acts as the server. In a successive NFC session however, the roles may change. That is, the prior client acts now as the server and vice versa. Figure 3.1a illustrates such a major use case setting.
- **Smart phone and IoT-device:** The second major use case setting comprises an NFC-enabled smart phone and an NFC-enabled IoT-device. Here, the smart phone acts as the client and initiates a communication with the IoT-device, also referred to as the server. This second use case setting is depicted in Figure 3.1b.

There is also a third major use case, which involves a simple physical NFC-tag and another NFC-enabled device. However, due to the fact, that a simple NFC-Tag does not provide any computational power, which is needed by the proposed protocol, this use case is not considered and demands special treatment.

3.2 Security and threat analysis of QSNFC

This section investigates the existing protocol definition of QSNFC with respect to security. It will be shown that the existing QSNFC consists of several security vulnerabilities. First, a MITM-attack will be performed to show an attack vector, where an active adversary exploits unprotected public keys to inject its own keys and messages and subsequently is capable of gathering confidential information. The second attack presented in this section



(a) IoT-device to IoT-device (b) Smart phone and IoT-device

Figure 3.1: Major use cases

makes use of the fact, that QSNFC does not use any kind of challenge-response mechanisms to ensure freshness during multiple sessions. Hence, a Replay-Attack can be conducted to trigger unintended behaviour. In both presented attack scenarios, the participants of the QSNFC session, *Client* (C) and *Server*(S), are not able to detect the evil interference by an attacker on the protocol level.

3.2.1 MITM-Attack

Within this subsection, a Man-in-the-Middle attack is performed on the subsequent handshake. Therefore, an evil adversary *Eve*, subsequently denoted as E, tries to gather as much information or inject as much (malicious) content as possible without getting caught. That is, without the detection of the interference by the two original communication partners.

The subsequent handshake, which was already presented in the previous chapter, is used by the attacker to mount the MITM-Attack. In the following, the necessary prerequisites for a successful attack are listed. Afterwards, the attack itself, which in fact can be mounted in two different ways, will be explained in detail including a flow diagram. Finally, consequences and limitations of these attacks are highlighted:

Prerequisites

Within this attack scenario, the attacker must be able to eavesdrop and forge any message of the communication parties C and S, hence an active MitM. In the description of QSNFC, the source address token is used to demonstrate ownership of the server's identity, but since this token can be eavesdropped within the initial handshake, it is assumed that E is already in possession of such a valid source address token. Furthermore, for the first variant of the MITM-attack, it is assumed that messages can be sent without encrypted content. This assumption is legit, as QSNFC does not prohibit encrypted messages of length zero. However, even with such a restriction, there is still a severe vulnerability, as shown in the second variant. Other than that, no prerequisites or assumptions are needed to mount these attacks successfully.

Attack description - Variant 1

The first variant of this attack works as follows: At first, C and S perform the initial handshake without any interference by E. After that, the client is already in possession of the server's long-term public key pk_l contained in the received RJ-message and furthermore C has already created his own ephemeral key pair (pk_c and sk_c). Based on that, the client can already compute the shared secret sk_i for securing data within the subsequent handshake. Subsequently, the client initiates the subsequent handshake by sending the complete CH-message, including the public part of his ephemeral key pair and encrypted data, which is secured by an authenticated encryption algorithm. However, this message gets intercepted by E, who generates two distinct key pairs to mount the attack. That is, (pk_{e_s}, sk_{e_s}) is intended to be used for the communication with the server and (pk_{e_c}, sk_{e_c}) is going to be used for the communication with the client. After that, E truncates the received complete CH-message by the encrypted payload. That is, E sets the length of the encrypted payload to zero and replaces the pk_c with her own key pk_{e_s} and sends the modified message to S. The server S generates his own ephemeral key pair (pk_s, sk_s) and computes in good faith the shared secret sk_{e1} with his long-term private key sk_l and the received public key pk_{e_s}. Furthermore, to encrypt data within SD messages, S computes the final shared secret sk_{e3} by using the same public key pk_{e_s} and his own ephemeral private key sk_s. The shared secret sk_{e1} is used by S to encrypt data in the subsequent handshake, whereby this encrypted data is sent along with the public part of the server's ephemeral key pair within the SH-message. This message gets again intercepted by E, who is now able to decrypt the received encrypted payload. This can be done by computing the shared secret sk_{e1} by using sk_{e_s} and pk_l. Hence, E is now able to read the server's response.

Furthermore, if E wants to continue the attack, she computes the shared secret sk_{e3} by using sk_{e_s} and pk_s in order to encrypt and decrypt SD-messages to and from the server. Additionally, she has to truncate the received SH-message in the same manner as before. That is, E sets the length of the encrypted payload to zero and additionally, she replaces the server's ephemeral key with her own public key pk_{e_c}. E then sends the modified message to the client, who generates in good belief the shared secret sk_{e2} by using his private key sk_c and the received public key pk_{e_c}. In all following SD messages, E can impersonate both the client and the server. Moreover, she can read and modify messages from the client by using sk_{e2}, and using sk_{e1} for messages from the server respectively, without the knowledge of the two original communication partners. Even worse, E can send as many SD-messages as desired to both of the communication partners without the need of an existing SD message from C or S.

Figure 3.2 illustrates the above described attack, where C and S perform an initial handshake without any interception. However, after the initial handshake, E mounts the attack and is therefore able to read confidential data or inject malicious data without the knowledge of the communication partners. Within that figure, all malicious or manipulated entities (messages, shared secrets and keys) are marked in red.

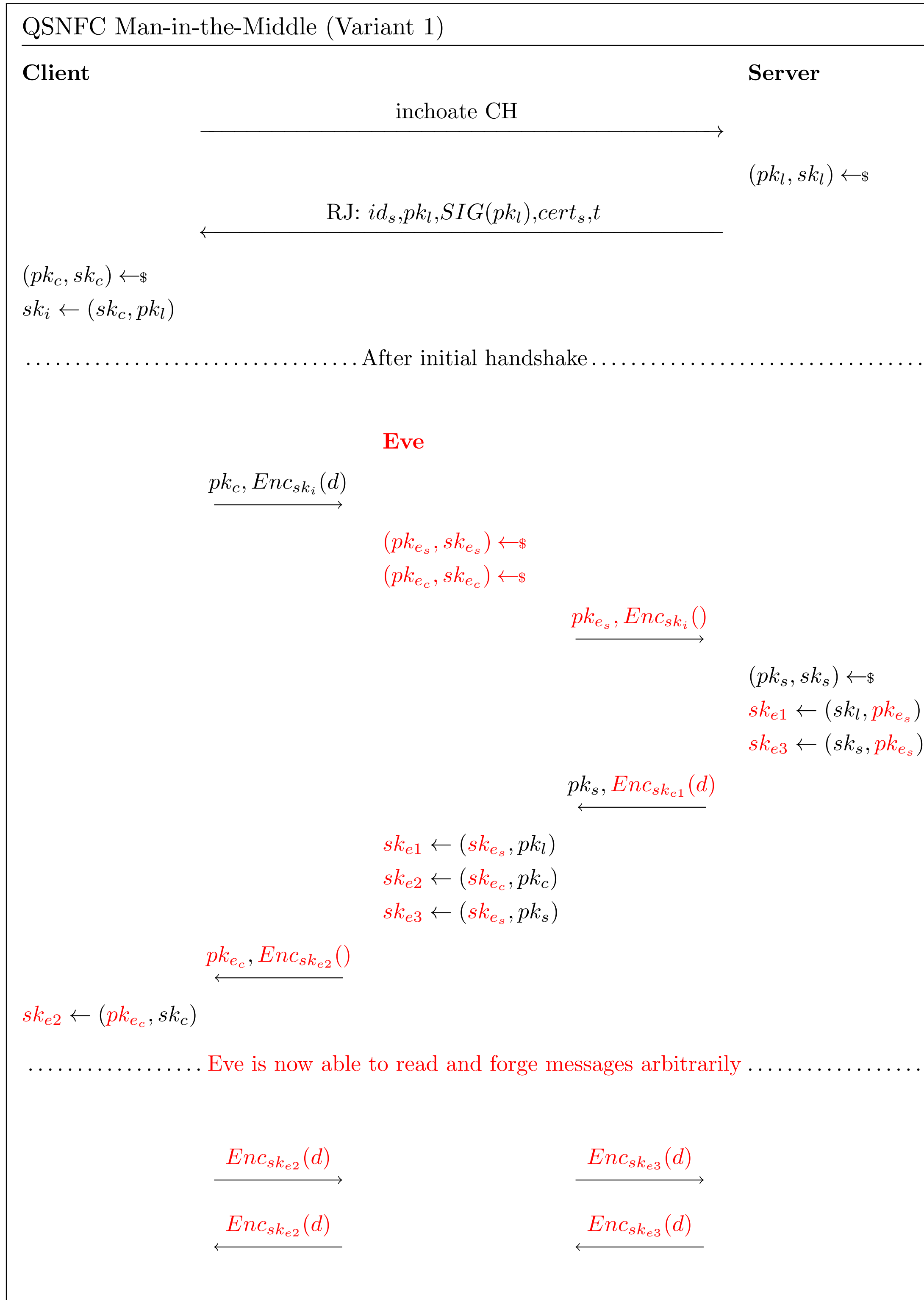


Figure 3.2: MitM attack 1

Attack description - Variant 2

If QSNFC would prohibit messages with no encrypted payload in the subsequent handshake though, there is still a severe vulnerability. That is, the unprotected public key in the SH-message (pk_s). To mount such an attack, E is not going to intercept but only eavesdrop the initial handshake and the first part of the subsequent handshake. Therefore, E stores the client's ephemeral public key pk_C, which is sent within the complete CH-message. In turn, the server S generates his ephemeral key pair (pk_s,sk_s) and computes the shared secret sk_i by using its long-term private key sk_l and the client's public key pk_c. Subsequently, the server responds with the SH message containing his ephemeral public key pk_s and data encrypted with the common shared secret sk_i. However, this message gets intercepted by E, who replaces the public key with her own public key pk_e and forwards the message to the client. In addition to that, E computes the shared secret sk' by using her own private key sk_e and the client's public key pk_c. In turn, the client computes in good belief the same shared secret sk' by using his private ephemeral key sk_c and Eve's pk_e.

From now on, E can impersonate the server and exchange arbitrary SD-messages with the client. Moreover, the server is also not aware of the fact, that the session has been intercepted. Figure 3.3 illustrates this second variant of the MitM-attack and similarly, the malicious forged content is depicted in red text color.

Consequences and limitations

The two previously shown variants of the MitM-attack on QSNFC imply severe security vulnerabilities, as in both cases an evil adversary is able to impersonate the server and send messages with arbitrary content to the client. Furthermore, in the first variant, it also possible of impersonating the client within a QSNFC session.

Within the first variant, the encrypted payload of the complete CH-message and the SH-message is set to zero, because an authenticated encryption algorithm would detect a forged message. Therefore, E needs to truncate the encrypted payload. Otherwise, the receiving part would detect the malicious forgery due to the fact, that the public keys are inherently correlated with the shared secret. Hence, decryption of the encrypted payload within the two above mentioned messages would fail. Therefore, the adversary is not able to decrypt the first message of each communication party. However, this attack is the more severe attack compared to the second variant, as it gives an adversary the opportunity to forge SD-messages arbitrarily to both client and server without their knowledge.

In contrary to the above described variant, the second variant does not require Eve to truncate the encrypted payload of the complete CH-message and the SH-message. Hence, such an adversary is not able to decrypt the encrypted payload of the complete CH-message and the SH-message, neither is he able to suppress the receiving entity from receiving the message. However, an adversary can subsequently impersonate the server and can continue the session with the client and send arbitrary SD-messages.

The fundamental issue, which causes this attack to work, is twofold. First, unprotected public keys open the door for Man-in-the-middle attacks. Second, the missing constraint

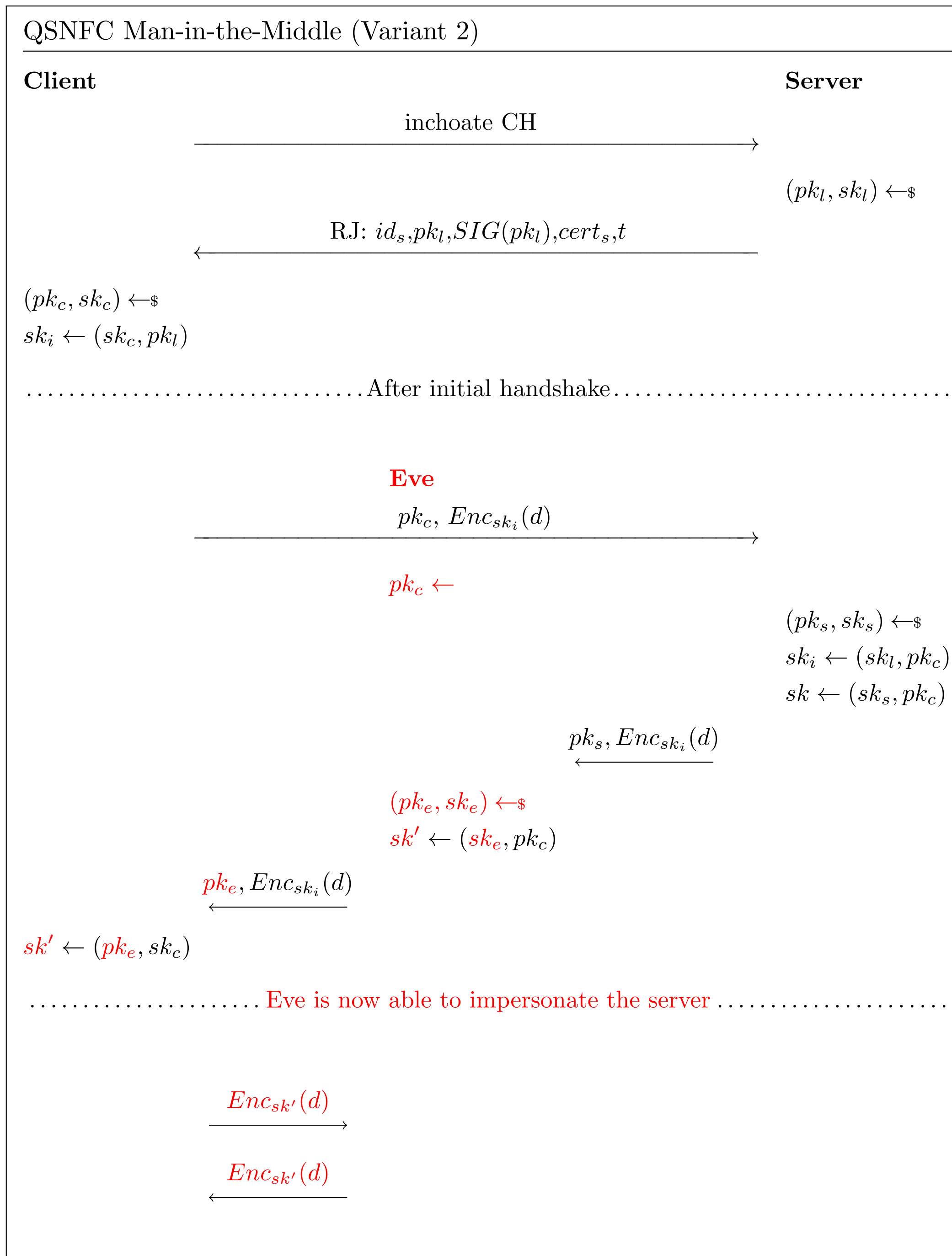


Figure 3.3: MitM attack 2

that the encrypted payload of a message must not be empty enables even more attack vectors. Fortunately, these issues can be fixed rather easily by adding the keys as additional authenticated content to the authenticated encryption algorithm and by ensuring that the encrypted payload length is never zero.

3.2.2 Replay-Attack

In the last subsection different variants of the MitM-attack were presented. These variants are applicable under the condition that an evil adversary is able to alter any given message within a QSNFC session between a client and a server. However, if such an adversary is not able to intercept one or more messages and alter the contents, there is still a severe vulnerability in the QSNFC-protocol. Therefore, an adversary is going to eavesdrop an existing QSNFC-session, and at a later point in time, he impersonates the client by re-sending the recorded message again to the server. This way, the adversary can force the server to perform unintended actions.

The following paragraphs describe such a Replay-Attack on QSNFC. At first, prerequisites and assumptions are stated, followed by a detailed attack description and including a flow diagram. Finally, a short paragraph on limitations and consequences concludes this attack.

Prerequisites

To conduct this attack, an evil adversary E must be able to eavesdrop a QSNFC-session between a client C and a server S, however it is not necessary to intercept and alter messages within the current session. Furthermore, E must be able to send messages to S. Other than that, no assumptions are needed to successfully apply this attack.

Attack description

At first, E eavesdrops a communication between C and S, whereby it is sufficient to record all the messages starting from the subsequent handshake. At a later point in time, E re-transmits the eavesdropped messages to S to trigger unintended behaviour. To make this attack applicable, it is important that E sends the exact same messages, including the encrypted content, the public key and source address token. Figure 3.4 illustrates this Replay-Attack.

Consequences and limitations

Within the above described Replay-Attack, an evil adversary is able to re-transmit eavesdropped messages from a prior QSNFC-session between two communication parties. Thus, by re-transmitting these messages to the server, an adversary may trigger unintended behaviour at the server. This could be exploited if the client uses QSNFC to get access to an entity. An adversary could eavesdrop the sent messages in order to get access to that entity and replay it at a later point in time, thus getting access to an originally prohibited area. However, an adversary cannot read encrypted data within the scope of this attack. As already discussed in 2.1.4, Replay-Attacks can be applied on protocols without any

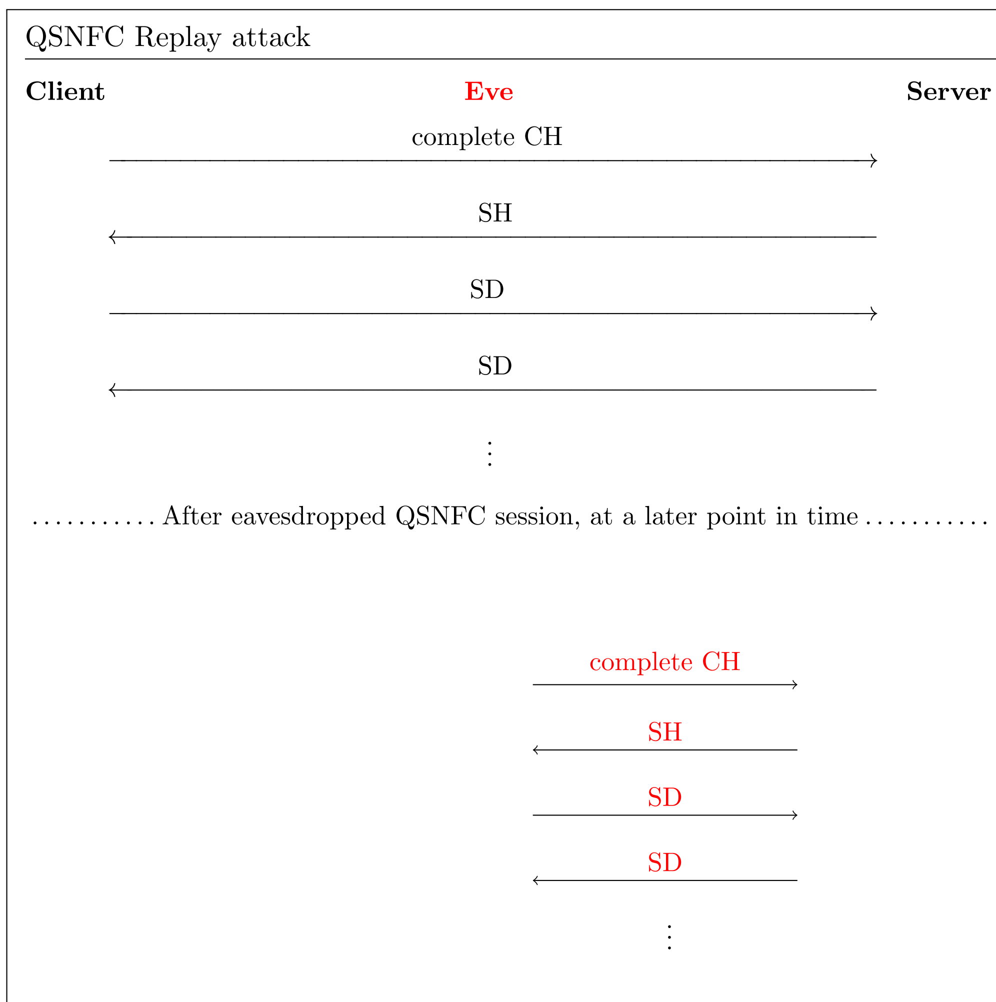


Figure 3.4: Replay attack

involving freshness properties, which is exactly the case for QSNFC. Hence, the communication partners are not able to verify the freshness of messages. However, this problem can be eliminated by adding nonces as additional authenticated data to the messages.

3.3 Performance analysis towards a modified QSNFC

This section analyses the existing QSNFC protocol with respect to performance and flexibility. First, the certificate chain handling is going to be evaluated and after that, some considerations on the use of different encryption algorithms are given and finally the concept of the Abort (AB)-message is introduced.

3.3.1 Certificate chain and compression

The authors in [UPS+18] state, that the certificate chain causes the major data overhead in a QSNFC-session. Therefore, the 0-RTT property is introduced as a countermeasure to reduce the number of bytes being sent via NFC. Nevertheless, the certificate chain must still be sent during the initial handshake. However, if the client is already in possession of a (partial) certificate chain prior to an initial QSNFC handshake, the server would still send the complete certificate chain. QUIC [CLL+17] solves this problem by introducing the so-called cached certificates. The idea behind cached certificates is, that the client proactively sends suitable information about trusted certificates to the server. In particular, QUIC defines this information to be a 64-bit *FNV-1a* hash. This way, the client sends hashes of suitable trusted certificates to the server in advance, then the server processes the received certificate hashes and matches them with the certificates in his certificate chain. Any certificate from the certificate chain, which was already received as a hash by the server is replaced with the corresponding hash. Taken into account that one certificate is roughly 1000 Byte in size, a significant drop of data overhead can be achieved by using cached certificates. Following up with the definition of QUIC, the certificate chain, which is sent by the server, is also going to be data compressed to reduce the data overhead even further. Consequently, due to the significant drop of data overhead, these features are also included in the modified QSNFC protocol, which is proposed within this thesis.

3.3.2 Available encryption algorithms

In contrary to other cryptographic protocols such as TLS, QSNFC does not support the usage of different encryption algorithms. Clearly, this could be useful for future versions of this protocol, especially if the client or the server is not capable of certain encryption algorithms, the protocol can still be continued with a different algorithm. In such a case, however, Downgrade-Attacks need to be considered as special threat. QUIC defines the use of two distinct encryption algorithms, and moreover Hameed et. al. [HJS16] proposed the use of other encryption algorithms in their work. Hence, this feature will also be included in the protocol version proposed within this work.

3.3.3 Abort handshakes

In the previous subsection, the usage of different encryption algorithms was explained. Therefore, if the server decides that a proposed encryption algorithm by the client does

not provide the desired security level, the server should return an AB-message, which contains information why the handshake was aborted. Thus, such an AB-message could also be used to contain information about the certificate chain, similar to the cached certificates described earlier. That is, a client may initiate a QSNFC session with a server by sending a complete CH-message using cryptographic primitives from a previous session. However, if the server is not in possession of the corresponding cryptographic primitives anymore due to cache replacement strategies, such an AB-message can be used to inform the client that a new initial handshake is needed. Hence, such an AB-message will be presented in the next section within this work.

3.4 Modified QSNFC

This section proposes a modified version of the QSNFC-protocol presented by Ulz et. al [UPS+18], whereby these modifications are based on the previously discussed findings. The names of the contained messages are based on the original version, however the content is different. Furthermore, several additional cryptographic computations are necessary to ensure the desired security properties, which entail confidentiality, integrity, authenticity and freshness. The modified version of the QSNFC protocol is presented in the following, starting with a simplified flow protocol of the modified version, which is depicted in Figure 3.5. Furthermore, supported encryption algorithms and primitives such as the definition of curves used for ECC are given. Subsequently, the initial and the subsequent handshakes will be explained in detail, including the AB-message, all involving cryptographic operations and the handling of certificates. After that, the handling of SD-messages is described in detail, whereby these SD-messages represent the successive messages in each subsequent handshake. Finally, the message types itself are listed, including a concise description of the message fields and lengths.

3.4.1 Supported encryption algorithms and cryptographic primitives

The following paragraph discusses the available encryption algorithms and furthermore, several cryptographic primitives such as the applied elliptic curves or hash functions are listed. As already stated in the original QSNFC version, an authenticated encryption algorithm is used to ensure confidentiality and authenticity. Hence, Table 3.1 depicts the available encryption algorithms. Within this work, AES is used in two different modes of operation and with three different input block sizes. Furthermore, the values for the certain encryption algorithms are selected in a way that allows the combination of several encryption algorithms by a simple XOR-operation, as it will be shown later in the protocol definition.

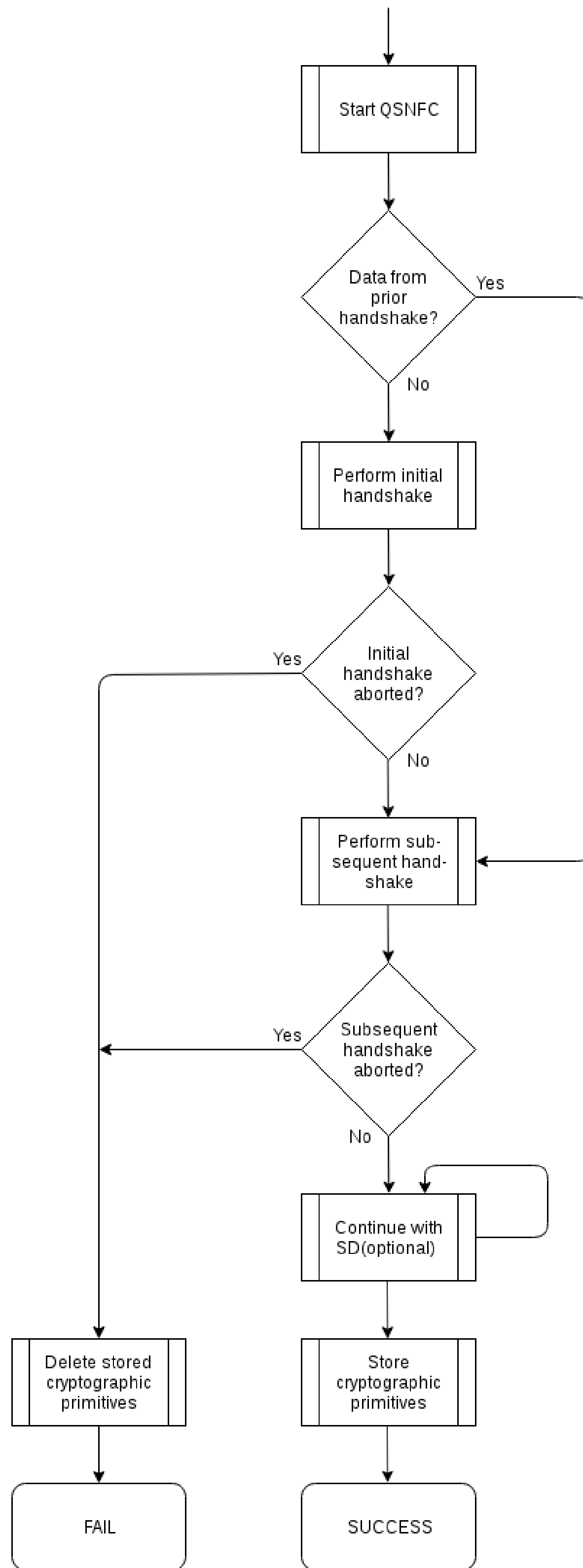


Figure 3.5: Simplified flow diagram

Hex-Value	Encryption algorithm
0x01	AES128-GCM
0x02	AES192-GCM
0x04	AES256-GCM
0x08	AES128-CCM
0x10	AES192-CCM
0x20	AES256-CCM
Others	Reserved for future use

Table 3.1: Supported authenticated encryption algorithms

Input block length (in bits)	Elliptic curve [SEC00]
128	Secp128r1
192	Secp192k1
256	Secp256k1

Table 3.2: Supported elliptic curves

Additionally, three distinct elliptic curves are used for PKC, depending on which authenticated encryption algorithm is chosen. In particular, the chosen authenticated encryption algorithm determines the elliptic curve needed for signatures and shared key generation, as illustrated in Table 3.2. Furthermore, the following enumeration describes all used cryptographic primitives briefly:

- **Signatures:** Within this work, an *Elliptic Curve Digital Signature Algorithm* is used to generate and verify signatures. The curve used for the computation is determined by the chosen encryption algorithm, is already noted above.
- **Key generation:** *Elliptic Curve Diffie Hellman* is used to agree on a shared secret, whereby the underlying curve results from the chosen encryption scheme, as already noted previously.
- **Key derivation:** The modified QSNFC uses a HKDF to generate keys from a previously negotiated shared secret.
- **Hash function:** SHA-256 is used for all hashing operations, as well as for the key derivation function.

The following subsections illustrate the modified QSNFC protocol, which is based on the original QSNFC. At first, prerequisites are given, then both the initial and the subsequent handshake is described and finally SD-messages are discussed. These SD-messages can be used to continue a QSNFC session after the subsequent handshake.

3.4.2 Prerequisites

It must be feasible for both the client and the server to perform cryptographic computations such as en- and decryption, signature creation and validation, key agreements, key

derivation and hashing values using the above described cryptographic primitives. Furthermore, both parties must be in possession of a unique id, subsequently referred to as id_C for the client id and id_S for the server id, respectively. Moreover, the server must be in possession of a valid certificate $cert_S$ including an associated key pair (sk_{cert}, pk_{cert}). Therefore, a certificate chain can be constructed starting from the server's certificate via optional intermediate certificate authorities to a root CA. Additionally, as already shown in the previous chapter, the truncation of encrypted payload can cause severe security vulnerabilities. Therefore, this protocol prohibits messages with an empty encrypted payload field. If such an empty payload is detected, the communication must be aborted.

3.4.3 Initial handshake

Within the initial handshake, a client C and a server S agree on the cryptographic primitives, which are going to be used in the following sessions. Furthermore, the client uses the information from the initial handshake to establish trust in the server by authenticating him using a certificate chain. Additionally, the client also authenticates the server's long-term public key, which is going to be used in every subsequent handshake until a new initial handshake takes place.

Inchoate CH-message

At first, the client sets the *prim* field, which corresponds to the supported encryption algorithms, as already mentioned previously in this section. The client may choose one or more algorithms by combining the corresponding hex value using a XOR-operation. Additionally, the client may compose a list of cached certificates in order to send information of already cached and trusted certificates to the server to reduce the data overhead implied with the certificate chain. Therefore, the client computes for each suitable and trusted certificate a hash value and after that he concatenates these hash values and stores the result in $cached_C$. Furthermore, he generates a nonce n_{iCH} and sends this data along with his id id_C within the inchoate CH-message to the server.

RJ-message

After receiving the inchoate CH-message from the client, the server validates the proposed cryptographic primitives in the *prim* field of the received message. If a suitable encryption algorithm can be found based on the proposed value, the server selects the corresponding encryption algorithm value and stores this value in the s_{prim} field. This encryption algorithm is going to be used throughout the following subsequent handshakes.

Additionally, the server creates a nonce n_{RJ}, which will be used to ensure freshness of the communication. Next, the server generates a long-term key pair (pk_l,sk_l), which will be used for all sessions with that client until a new initial handshake takes place. Moreover, the server computes the signature of the previously created long-term public key pk_l using the key associated within his certificate (sk_{cert}). This way, the client can later verify the authenticity of the long-term public key pk_l. The yielding signature of the long-term key is stored in the s_k field. Subsequently, the server concatenates the received nonce n_{iCH}, his own nonce n_{RJ} and the value of the selected encryption algorithm s_{prim} together. If the client has also sent cached certificates within the $cached_C$ field of the

inchoate CH-message, the content of this field is also added to the concatenated data bytes. Subsequently, to prove the ownership the private part of the long-term key, these concatenated data bytes get signed by the server with the corresponding private key sk_l. This yields a signature, which is stored in the s_p field to ensure freshness of the initial handshake.

Next, the server builds the certificate chain. Therefore, he first checks whether the client has included any cached certificates within the inchoate CH-message. Subsequently, the server concatenates the certificates from the chain starting with his certificate, followed by potential intermediate certificates up to but not including the certificate of the root CA. This concatenated certificate chain is then stored in the $chain_S$ field. However, if the client has included cached certificates within the $cached_C$ field of the inchoate CH-message, the server compares the hash-values with the ones in his certificate chain and if there is a match, he substitutes the certificate with the corresponding hash value within the $chain_S$ field.

Finally, the server sends all these fields along with his id id_S within the RJ-message back to the client.

Verification of the RJ-message

Immediately upon receipt of the RJ-message, the client is going to verify the content of this message. At first, the client tries to build and verify the certificate chain using the received certificate chain from the server and potential cached certificates. Moreover, the client has to verify the integrity of the server's long-term public key pk_l by computing the signature using the public key from the server's certificate. The yielding signature must then be compared with the received signature s_k. After that, the client concatenates his own nonce n_{iCH}, the server's nonce n_{RJ}, the received value of the selected encryption algorithm by the server, and potential cached certificates, which were transmitted by the client in the inchoate CH-message. Subsequently, the client computes the signature of the concatenated data by using the server's long-term public key pk_l and compares it with the received value s_p.

IV generation

After the successful verification of the negotiated parameters and the authentication process, both client and server compute the IVs for the shared secret generation. Each communication party computes the same two IVs, iv_{iC} and iv_{iS}. iv_{iC} can be computed by concatenating the client's initial nonce n_{iCH} and the server's initial nonce n_{RJ} and computing the hash value of it. iv_{iS} is computed in the same manner, however the order of the concatenated nonces is reversed.

Figure 3.6 illustrates the initial handshake in a more representative way, containing the inchoate CH and the RJ-message, the necessary verification steps and the IV-generation. $RAND$ denotes a function which generates a strong pseudo random number of length n and $KGEN$ generates a new private-public key pair. The function $Sign_k(data)$ computes the signature of $data$ using the key k and $VerifySig_k(sig)$ verifies a signature sig by using the key k. Finally, $VerifyCertChain(chain)$ checks, whether a certificate is trustworthy or not.

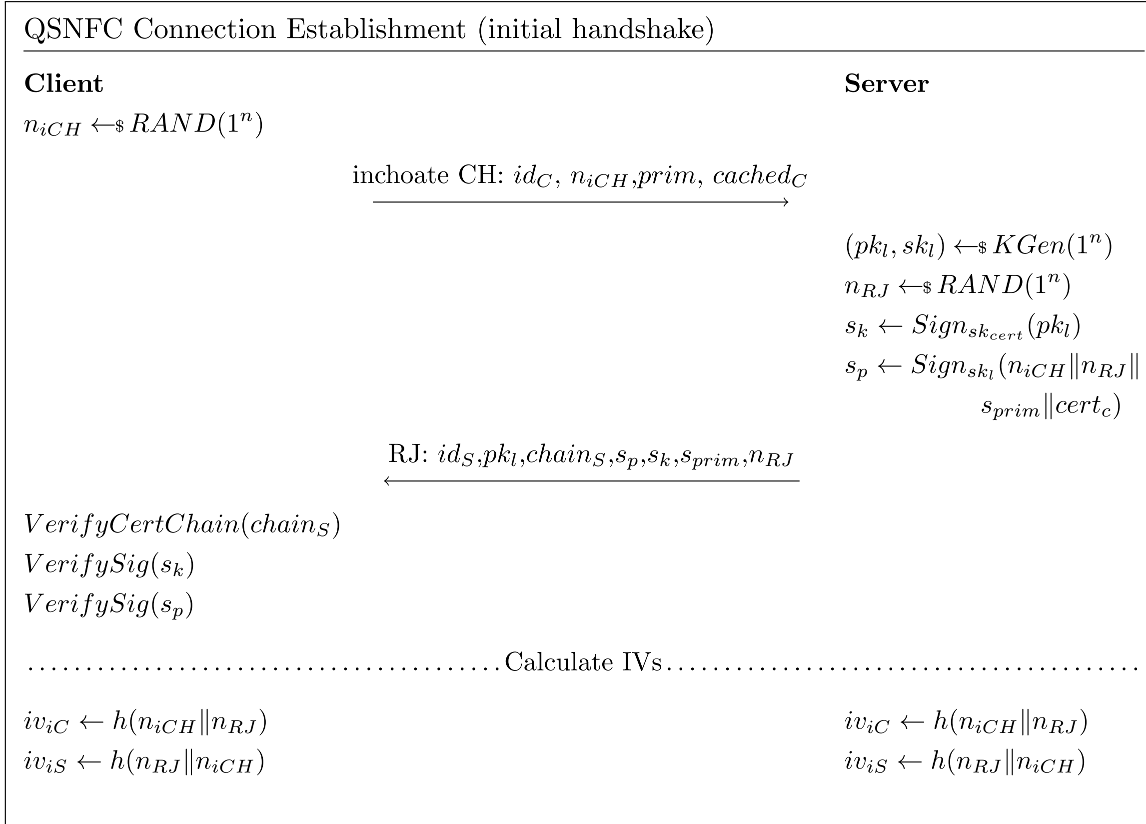


Figure 3.6: Initial handshake of the modified QSNFC

Hex-Value	**Abort reason**
0x01	Insufficient cryptographic primitives
0x02	Previous session keys missing
0x03	Forged message detected
Others	Reserved for future use

Table 3.3: Abort reasons

Aborting the initial handshake

In the initial handshake, the client proposes a set of supported cryptographic primitives to the server and if the server is satisfied with the proposal, he responds with the RJ-message. If, however, the proposed cryptographic primitives within the s_{prim} field of the inchoate CH-message are not sufficient for the server, he will respond with an AB-message. Anyhow, there are multiple situations in which such an AB-message can be send, as illustrated in Table 3.3, the following description does only apply for AB-messages within initial handshakes though. Instead of just letting the client know, that he proposed insufficient cryptographic primitives, the server can already use this AB-message to transfer the certificate chain. Hence, the server stores the abort reason in the *ab* field of the AB-message, and generates a long-term public key pair (pk_l,sk_l) in the same way as if he would respond with an RJ-message. Additionally, the server computes a signature s_k of pk_l by using the private part of the certificate key pair (sk_{cert}, pk_{cert}). In contrary to the RJ-message though, the server creates an abort nonce n_{AB}, concatenates it to the received client nonce n_{iCH}, the received *prim* and the abort reason *ab*. Subsequently, the server generates a signature s_p of those concatenated parameters using his long-term private key sk_l, which ensures freshness of the communication. All these parameters are sent along with the server's id id_S within the AB-message back to the client.

AB-message verification

On receival of the AB message, the client first verifies the certificate chain in the same fashion as described above in the RJ-verification paragraph. Next, the integrity of the received public key pk_l must be checked by verifying the received signature s_k and additionally, the signature s_p must also be verified. After the verification, the client is already in possession of the certificate chain. Thus, if he considers the chain as trustworthy, he can use this information within the next initial handshake by sending cached certificates to reduce the amount of data overhead.

3.4.4 Subsequent handshake

In the initial handshake, client and server negotiated and agreed on cryptographic primitives, which are going to be used for subsequent QSNFC-sessions. Moreover, the client has established trust in the server by authenticating him using a certificate chain. Within the subsequent handshake, the client can already exchange secured data with the server by using the complete CH-message and the SH-message, as depicted in Figure 3.7. Hereby, the function $AE_k^{iv}(data, aad)$ denotes an authenticated encryption function with the symmetric key k and initialized with the IV iv. Furthermore, the first parameter of this function *data* represents the plaintext, which is going to be encrypted in an authenticated manner. Moreover, the optional parameter *aad* symbolizes additional authenticated data, which gets also added to the authenticated encryption algorithm. This encryption algorithm yields the cipher and an authentication tag, which is needed to verify the integrity. Consequently, the function $AD_k^{iv}(cipher, aad, tag)$ acts as the corresponding authenticated decryption function with the symmetric key k and the IV iv, whereby the first parameter *cipher* is the data to decrypt and the parameter *aad* represents the additional authenticated data, which needs to be verified using the authentication tag *tag*.

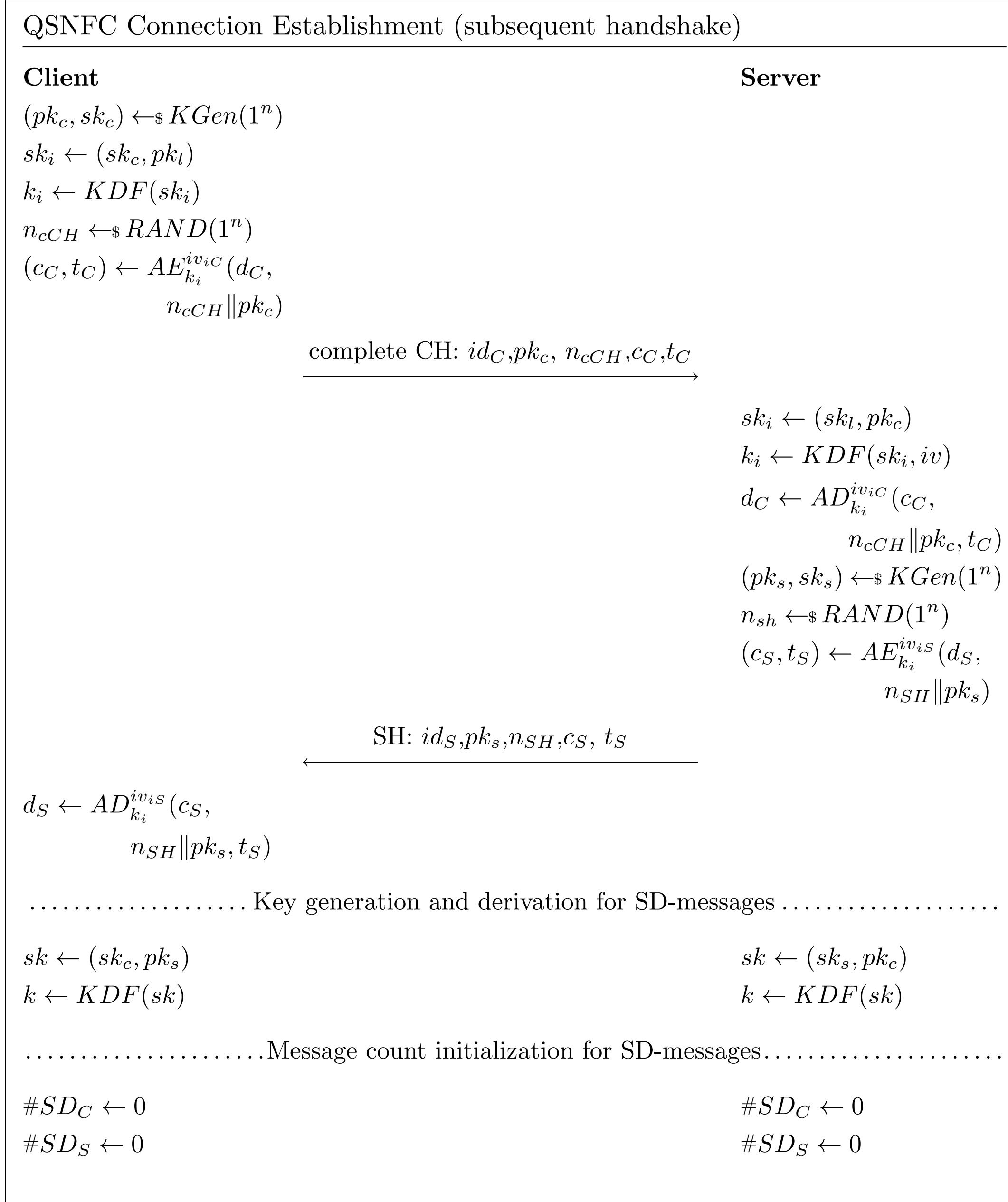


Figure 3.7: Subsequent handshake of the modified QSNFC

The following paragraph comprises a detailed description of the complete CH-message and the SH-message. Additionally, the abort of the subsequent handshake is highlighted and the key generation for SD-messages is explained.

Complete CH-message

To send the complete CH-message, the client has to generate an ephemeral key pair (pk_c,sk_c) at first, which will then be used to compute the intermediate shared secret sk_i by using the client's own private key sk_c and the server's long-term public key pk_l. Based on that shared secret, a symmetric encryption key k_i can be derived using a key derivation function. To ensure freshness of the subsequent handshake, the client furthermore has to generate a nonce n_{cCH}. Afterwards, the client is able to encrypt data d_C using an authenticated encryption algorithm with the key k_i and the IV iv_{iC}. This IV was either computed in the initial handshake or stored from a prior subsequent handshake, as shown in one of the next sections within this chapter. To guarantee the integrity of these parameters, which are going to be transmitted within the complete CH-message, the nonce n_{cCH} and the public key pk_c are concatenated and added to the authenticated encryption algorithm as additional authenticated data. This yields the cipher c_C and the corresponding authentication tag t_C. Subsequently, the client sends his id id_C, his ephemeral public key pk_c, the nonce n_{cCH}, the cipher c_C and the authentication tag t_C within the complete CH-message to the server. In turn, the server is now also able to generate the same shared secret sk_i, which is then used to derive the intermediate symmetric key k_i. This key is used along with the IV iv_{iC} to decrypt the received cipher c_C, which yields the plain text d_C from the client. Additionally, the received authentication tag t_C is used to verify the integrity of the received cipher text.

SH-message

In response to the previously received complete CH-message, the server has to respond with the SH-message. To do so, he first generates an ephemeral key pair (pk_s,sk_s) and the nonce n_{SH}, which ensures freshness of the following SH-message. Subsequently, the server encrypts the message d_S using an authenticated encryption algorithm with the previously derived intermediate key k_i and the IV iv_{iS}. This IV was computed either in the initial handshake or in a prior subsequent handshake. Additionally, the nonce n_{SH} gets concatenated with the public key pk_s. This concatenated data is going to be added as additional authenticated data to ensure integrity and authenticity. Hence, the encryption algorithm yields the corresponding cipher c_S and the authentication tag t_S. After that, the server sends the cipher and the authentication tag along with his id id_S, the fresh nonce n_{SH} and his ephemeral public key pk_s within the SH message to the client. In turn, the client decrypts the received cipher c_S and verifies its integrity by using the received authentication tag t_S. Additionally, the client has to verify the additional authenticated data provided within that SH-message. This additional authenticated data consists of the nonce n_{SH} and the ephemeral public key pk_s.

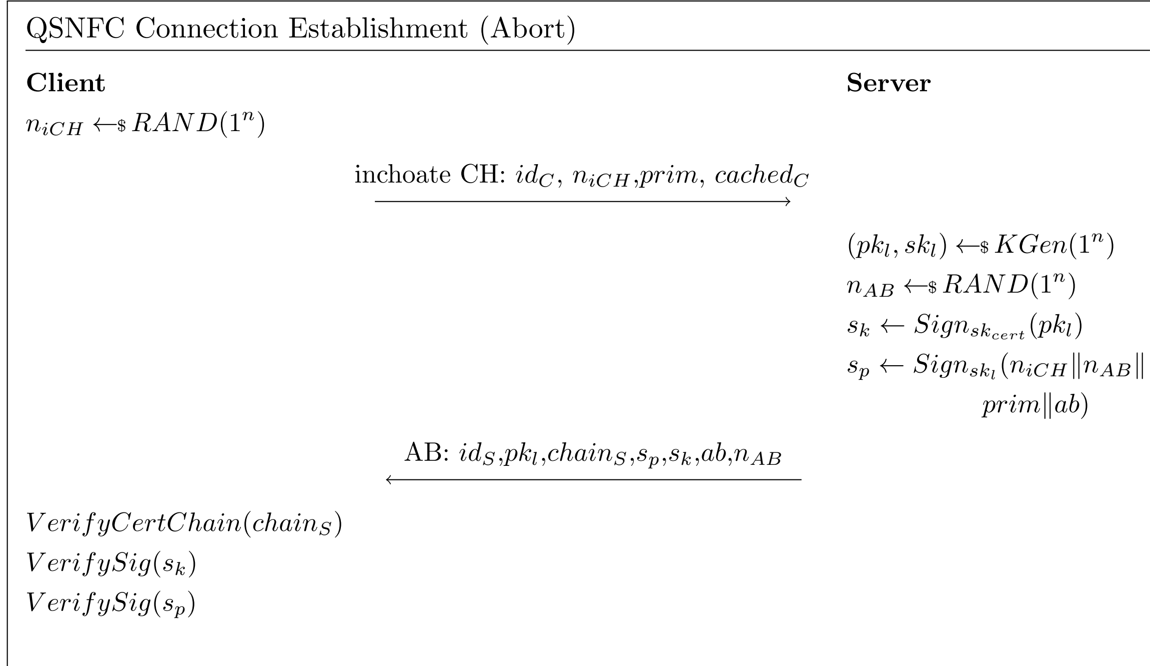


Figure 3.8: Abort initial handshake

Parameters for SD-messages

After a successful subsequent handshake, both client and server are now in possession of mutual ephemeral public keys. Therefore, both communication parties compute the final shared secret sk by using their own ephemeral private key $sk_{c|s}$ and the opponents ephemeral public key $pk_{c|s}$. By using this shared secret, client and server can now derive a symmetric key k, which is going to be used for the following SD-messages within the current QSNFC-session. Furthermore, the SD-messages make use of message counters to ensure freshness and proper message ordering within the remaining QSNFC session. Therefore, those message counters of both communication parties must be set to 0 in the subsequent handshake.

Abort of subsequent handshakes

In the initial handshake, the concept of an AB-message was introduced, where such an initial handshake is aborted due to insufficient cryptographic primitives. However, as already stated in Table 3.3, several reasons can lead to the cancellation of a QSNFC-session. Hence the server can also send the AB-message with the proper abort reason ab instead of the SH-message. Such an AB-message comprises the same content as described in the initial handshake and shown in Figure 3.8, but neglecting the certificate chain $chain_S$.

3.4.5 SD-messages

Within the initial and the subsequent handshake QSNFC follows a request-response model, where the client sends requests (inchoate/complete CH-message) and the server answers with the corresponding response (RJ/SH-message). However, after the subsequent handshake, client and server may continue the communication by using so called SD-messages. SD messages can be sent in arbitrary order. Therefore, it is up to the actual implementation of the protocol how to handle the message flow. To send an SD-message, the client concatenates the previously received nonces n_{cCH} and n_{SH} together with the message count $\#SD_C$, which represents all sent SD-messages by the client within this QSNFC session. The hash of this concatenated data yields the IV iv_{sC}, which is then used to encrypt a plaintext d_C using the key k. The resulting cipher c_C and the authentication tag t_C is sent along with the client's id id_C to the server. Upon receival, the server generates the IV in the same way as the client, which is subsequently used to decrypt the received cipher c_C using the key k and the tag t_C. Finally, both communication partners increase the $\#SD_C$ by one. The other direction, where the server sends an SD-message to the client, works similar, however this time the following IV iv_{sS} is used for en- and decryption. This IV can be computed by hashing the concatenated values of n_{SH}, n_{cCH} and $\#SD_S$. $\#SD_S$ denotes the number of previously sent SD-messages by the server. Figure 3.9 illustrates the exchange of SD-messages between client and server.

Abort of SD-messages

Within the subsections on the initial and subsequent handshake multiple ways to cancel a QSNFC handshake were discussed. This is due to the fact, that the client expects a response to a request, however as SD-messages can be sent in arbitrary order and direction, there is not necessarily a need for an AB-message. Hence, this work does not propose an abort mechanism for SD-messages.

3.4.6 Connection tear down

QSNFC does not support a connection teardown, however certain steps are necessary once a QSNFC-session between a client and a server is finished. To enable the 0-RTT property for subsequent sessions, both communication parties need to store session-specific information. In particular, client and server have the store the following properties for a future communication:

- **pk_l:** The server's long-term public key, which is used by the client for the next subsequent handshake to compute a shared secret and to derive a new symmetric encryption key.
- **sk_l:** The server's long-term private key, which is used by the server for the next subsequent handshake to compute a shared secret and to derive a new symmetric encryption key. This value is only known to the server and therefore only the server needs to store this private key.
- **s_{prim}:** The value of the negotiated encryption algorithms. This value is used to encrypt data with the corresponding authenticated encryption algorithm and to apply the correct key size.

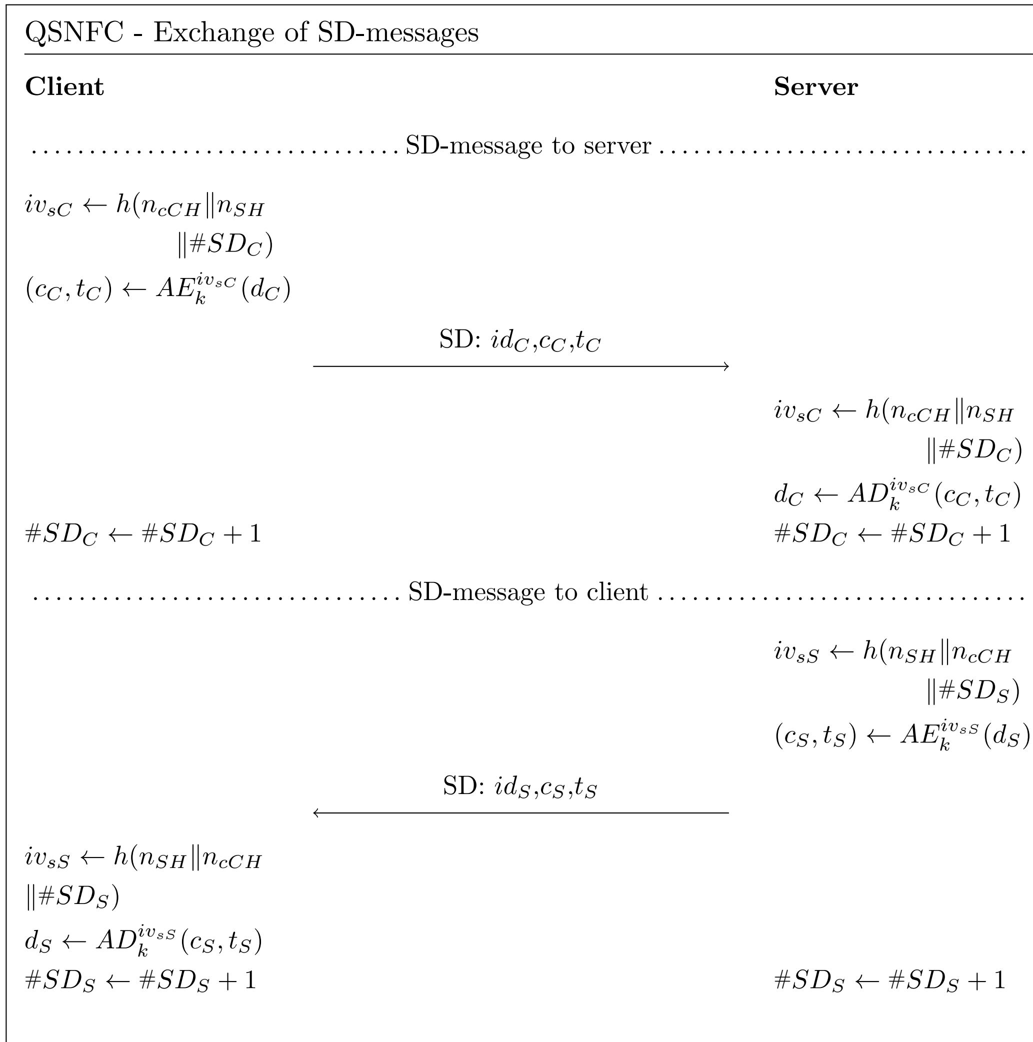


Figure 3.9: SD-messages of the modified QSNFC

Type	Value (in Hex)
Inchoate CH	0x0
RJ	0x1
Complete CH	0x2
SH	0x3
SD	0x4
AB	0xFF

Table 3.4: Message types

- iv_{iC}**:** The client's new initial IV for the next QSNFC session, which can be computed by hashing the current subsequent IV, that is $iv_{iC} = hash(iv_{sC})$.
- iv_{iS}**:** The server's new initial IV for the next QSNFC session, which can be computed by hashing the current subsequent IV, that is $iv_{iS} = hash(iv_{sS})$.
- $id_{C|S}$**:** The client's/server's id, respectively. This id is used to identify the communication partner in the next session.

3.4.7 Message types

Within the last subsections, the modified QSNFC protocol was presented in great detail, however, the actual message formats were neglected. Hence, the corresponding message formats including all relevant fields and possible variants are shown in this subsection. Additionally, the lengths of the certain fields are given in total bytes whenever possible and if not, for example when the length of a field depends on a cryptographic primitive, a relative length is stated instead. The basic structure of all QSNFC-messages is taken from the original QSNFC [UPS+18] proposal, however some fields are adapted to suit the modified QSNFC protocol.

General structure

All presented messages of the modified QSNFC follow the message structure of the original QSNFC. Thus, all messages start with a message type field, followed by a length-field of the non-encrypted part. After that, all corresponding non-encrypted fields are placed within the message. Subsequently, the length of the encrypted part is given and finally the encrypted content concludes the message. Table 3.4 illustrates all available message types.

Inchoate CH-message

The inchoate CH-message is sent by the client in the initial handshake to start a QSNFC-session. Table 3.5 depicts the message format for such an inchoate CH-message. The field $cached_C$ is optional, however if the client wants to send cached certificates, he concatenates the hashes of these cached certificates and puts the resulting concatenated data into the $cached_C$ field. Finally, the encrypted content remains empty, hence $LenE$ is set to zero.

Type	**LenP**	**Client ID** (id_C)	**Nonce** (n_{iCH})	**Crypto primitives** ($prim$)
1 Byte	2 Byte	8 Byte	8 Byte	1 Byte

Cached certificates($cached_C$)	**LenE**
LenP-Bytes - 20	2 Byte

Table 3.5: Inchoate CH-message

RJ-message

The RJ-message is sent as a response to the inchoate CH-message by the server, if the proposed cryptographic primitives are sufficient. Table 3.6 illustrates the composition of such an RJ-message. The length of the public key (KeyLen) and the length of the signatures (SigLen) vary depending on the chosen cryptographic primitives given in the s_{prim} field. Hence parsing the value of this field at first will yield the corresponding lengths. Moreover, the certificate chain length can be computed by using LenP, the keyLen and SigLen.

Type	**LenP**	**Server ID** (id_S)	**Selected primitives** (s_{prim})	**Long-term key** (pk_l)
1 Byte	2 Byte	8 Byte	1 Byte	KeyLen Byte

Nonce(n_{RJ})	**Signature Parameters** (s_p)	**Certificate chain** ($chain_s$)	**LenE**
8 Byte	SigLen Byte	CertChainLen Byte	2 Byte

Signature key(s_k)
SigLen Byte

Table 3.6: RJ-message

This certificate chain follows the structure described in Table 3.7, whereby the first byte is used to indicate an uncompressed certificate chain. After that, the length of a certificate (using 2 bytes) is given, followed by that serialized certificate. This pattern is repeated for all successive certificates.

0x0	**LenC1**	**Cert1**	**LenC2**	**Cert2**	**...**	**LenCN**	**CertN**
1 Byte	2 Byte	LenC1 Byte	2 Byte	LenC2 Byte	...	2 Byte	LenCN Byte

Table 3.7: Uncompressed certificate chain

However, as the certificate chain can be data-compressed as well, a special serialization format is used, as presented in Table 3.8 for a compressed certificate chain. The first byte indicates that data compression is used, followed by the length of the uncompressed certificate chain. These fields are used to decompress the compressed data. The compressed data itself consists of certificate tuples, where the first part indicates the length of the certificate (takes 2 bytes) and the second part represents the serialized certificate itself.

Complete CH-message

The complete CH-message is sent by the client to initiate the subsequent handshake. Therefore, the message format depicted in Table 3.9 is used, whereby the length of the

0x1	**DataLen**	**Compressed data**						
		LenC1	Cert1	LenC2	Cert2	...	LenCN	CertN
1 Byte	2 Byte	2 Byte	LenC1 Byte	2 Byte	LenC2 Byte	...	2 Byte	LenCN Byte

Table 3.8: Compressed certificate chain

serialized key KeyLen depends on the selected encryption algorithm.

Type	**LenP**	**Client ID** (id_C)	**Public key** (pk_c)	**Nonce** (n_{cCH})	**LenE**	**Tag**(t_C)	**Cipher** (c_C)
1 Byte	2 Byte	8 Byte	KeyLen Byte	8 Byte	2 Byte	16 Byte	LenE - 16 Byte

Table 3.9: Complete CH-message

SH-message

The SH-message is sent by the server as a response to a client's complete CH-message. Table 3.10 illustrates the composition of such an SH-message.

Type	**LenP**	**Server ID** (id_S)	**Public key** (pk_s)	**Nonce** (n_{SH})	**LenE**	**Tag**(t)	**Cipher** (c)
1 Byte	2 Byte	8 Byte	KeyLen Byte	8 Byte	2 Byte	16 Byte	LenE - 16 Byte

Table 3.10: SH-message

SD-message

After a successful subsequent handshake, both client and server exchange data using SD-messages, which are structured similarly to the complete CH and the SH message, but without a public key and a nonce, as depicted in Table 3.11.

Type	**LenP**	**ID** ($id_{C\|S}$)	**LenE**	**Tag**(t_S)	**Cipher** (c_S)
1 Byte	2 Byte	8 Byte	2 Byte	16 Byte	LenE - 16 Byte

Table 3.11: SD-message

AB-message

If the server wants to abort either the initial or a subsequent handshake, he sends the AB-message, which is structured as illustrated in Table 3.12. The $chain_s$ parameter however should only be sent when aborting the initial handshake, but not within subsequent handshakes.

Type	**LenP**	**Server ID (id_S)**	**Selected primitives (s_{prim})**	**Long-term key (pk_L)**
1 Byte	2 Byte	8 Byte	1 Byte	KeyLen Byte

Nonce(n_{AB})	**Abort reason (ab)**	**Signature Parameters (s_p)**	**Certificate chain ($chain_s$)**
8 Byte	1 Byte	SigLen Byte	CertChainLen Byte

LenE	**Signature key**(s_k)
2 Byte	SigLen Byte

Table 3.12: AB-message

Chapter 4

Implementation

Within this chapter, the implementation processes of both the protocol library and two demo applications are given. Therefore, the first part of this chapter elaborates the requirements for a software component which implements the previously described protocol definition. After that, the implementation environment is stated, the used external libraries are listed and finally, the actual implementation and important code snippets are shown. After that, the second part of this chapter comprises two demo applications, which are going to use the previously implemented protocol implementation. These demo applications cover the two required use cases from the *Design Chapter* by including the protocol implementation libQSNFC. Subsequently, the development processes of these two demo applications are going to be presented. Furthermore, the development environment and all necessary steps to build and run these applications in conjunction with libQSNFC are illustrated.

4.1 Requirements & Consequences

In the previous chapter, the modified QSNFC-protocol was presented in a formal way, this section however comprises the implementation relevant details. Thus, requirements are identified at first and subsequently consequences for the actual implementation are listed.

4.1.1 Requirements

The following list contains the requirements for the protocol library implementation.

- **Consistency:** The implementation should be compliant to the protocol definition given in the previous chapter, including all listed message types. Furthermore, the implementation should store negotiated primitives for subsequent sessions in the future.
- **Flexibility:** The implementation should be flexible, that is, not bound to NFC as communication layer.
- **Configurability:** Both communication parties should be able to configure the protocol implementation according to their needs. That is, defining available cryptographic primitives (encryption algorithms). Furthermore, the client should be able

to set the certificates in which he has already established trust, and which certificates he intends to send as cached certificates to reduce the data overhead.

- **Availability:** The resulting library should be versatile and applicable. Therefore, a programming language should be chosen which is supported by many different architectures.
- **Usability:** The protocol library implementation should be easy to use and easy to integrate in existing applications.

4.1.2 Consequences

Within this subsection, consequences are listed which result from the requirements described above.

Flexibility

One possible way to approach the implementation of the modified QSNFC is to tightly couple the communication layer, in this case NFC, with the implementation of the cryptographic protocol. This however implies the drawback that the protocol implementation is not applicable for other communication layers, such as Bluetooth, without modifying the actual protocol implementation. Therefore, this implementation should be designed in such a way that the protocol implementation and communication layer are not coupled at all to provide the flexibility for application to exchange each of them easily. Figure 4.1 illustrates the process between a client and a server, which want to communicate in secure fashion using QSNFC and a communication layer (COM). As can be seen in this figure, the communication layer and protocol implementation are not connected at all, thus, providing the flexibility to exchange the transportation layer. Another advantage of keeping the communication layer and the protocol implementation loosely coupled is the fact, that the implementation itself can be tested easily without the existence of a wireless communication protocol, as will be shown within this chapter.

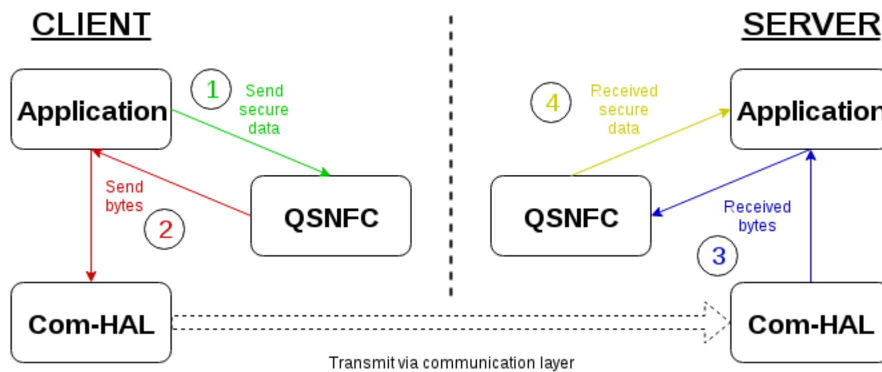


Figure 4.1: Library implementation overview

Availability

In the previous chapter, two major use cases for the modified QSNFC were identified. The first one comprises two NFC-enabled IoT-devices communicating with each other, while the second use case consists of a smart phone and an IoT-device which communicate via NFC. Hence, the programming language C will be chosen for the implementation, as many IoT-devices are capable of running programs written in C. Moreover, modern mobile phone operating systems such as Android or iOS provide integration for C-code as well.

4.2 Implementation environment and external libraries

This section describes the implementation environment, which comprises the used Integrated Development Environment (IDE) and test frameworks. In addition to that, all used external libraries are explained briefly and their usage in the protocol implementation is highlighted.

4.2.1 IDE

An Integrated Development Environment consists of multiple tools needed for the development of a piece of software. Depending on the used technology stack, these tools may vary. Additionally, many IDEs provide a way to extend the functionality by offering several installable additional plugins.
During the development of the modified QSNFC protocol implementation the free and open-source IDE Apache NetBeans was used. It comes with a build-in support for JavaEE and Web-programming (HTML5, PHP, JavaScript & CSS), however NetBeans does also support C/C++. The C-support includes also the possibility to build a piece of software for different platforms. That is, tools such as compiler, assembler and linker can be set independently for each platform configuration. This beneficial feature is used to build the library for all needed platforms such as AMD and ARM. Additionally, NetBeans provides a suitable graphical user interface to run unit tests and evaluate the results, as shown and described below.

4.2.2 Unit Test-framework

In modern professional software engineering, unit testing is a method to ensure quality of a piece of software during the entire development process. Hereby, the term quality implies correctness and completeness of the software functionality. In a unit test, only a single functionality (a unit) of a (large) piece of software is tested.
Therefore, within the protocol implementation, the Unit Test-framework CUnit was used to write and run unit tests. CUnit is a lightweight C-framework to debug and run unit-tests for programs written in C. It provides several useful assertion-functions and a built-in NetBeans plugin, which includes a rich graphical user interface to interact with the written unit-tests.
During the development of the protocol implementation, unit-tests were used to test certain functionality, whereby a simple buffer-based transport layer was used to simulate a communication between two entities.

4.2.3 External libraries and tools

A library extends the functionality of a piece of software, thus many operating systems ship with a default set of libraries, which can be included in a program. However, these internal libraries may not be sufficient for a special functionality. Hence, in need of a special functionality, tons of external libraries are available. This section lists the used external libraries and tools, explains them briefly and gives a short outline on their purposes within the proposed protocol implementation. Additionally, important code samples using those external libraries are shown.

OpenSSL

OpenSSL is the standard tool set for TLS and SSL protocols. Additionally, it provides a wide range of cryptographic primitives, starting from symmetric and asymmetric encryption algorithms, key derivation functions, hash functions and support for certificate operations. Moreover, it supports elliptic curve cryptography and authenticated encryption schemes, hence the OpenSSL-library is used within the scope of this thesis for all cryptographic operations. However, OpenSSL does not only offer a C-library which can be used in programs to access cryptographic functionality in a programmatic way, but it also provides a set of command line tools for Linux-based operating systems. The following enumeration lists and explains several useful tool commands and library functions, which were used during the development of the protocol implementation.

Create a Root-CA (Certificate Authority): Modern web communication protocols establish trust in a server by verifying its certificate chain and typically, each browser has its own set of trusted Root-CAs. While such a Root-CA is immediately trusted, a certificate must be checked whether there exists a certificate chain which ends in a trusted Root-CA. Therefore, a certificate can be issued by an intermediate or root CA. For the purpose of the development of the modified QSNFC protocol, a self-signed Root-CA was created. Therefore, a new private/public key pair *rootca.key* has to be generated, in this case an elliptic curve key based on the Secp521 random curve. After that, a self-signed certificate *rootca.cert* can be created by using the previously generated key *rootca.key*, as shown in Listing 4.1.

```
$> openssl ecparam -name secp521r1 -genkey -out rootca.key
$> openssl  req -new -x509 -days 365 -key rootca.key
    -out rootca.crt
```

Listing 4.1: Key-pair generation

Issue a Server-certificate: Typically, an entity, which wants to obtain a valid certificate, first generates its own private/public key pair. Using this key pair, the entity issues a certificate signing request to a certificate authority. Based on that signing request, the certificate authority signs this certificate with its own private key, adds the resulting signature to the entity's certificate and returns this certificate. Thus, the certificate chain

can be verified by any outside party. The following commands in Listing 4.2 illustrate the above described process using the previously created Root-CA.

```
$> openssl ecparam -name secp128r1 -genkey -out ../client.key
$> openssl req -new -key ../client.key -out ../client.csr
$> openssl ca -batch -config ../configs/rootca.conf -notext -in .
    ./client.csr -out ../client.crt
```

Listing 4.2: Issuing an entity-certificate

Hereby, the file *rootca.conf* contains parameters which define how to certificate is going to be created. Among others, the following parameters are set within this file:

- **Key:** The file path to the Root-CA's key pair, which is used to sign the certificate.
- **Validity:** Determines how long the yielding certificate is going to be valid.
- **Key usage:** This field states for which kind of applications the certificate can be used to prove trust.
- **CRL and OSCP distribution points:** These fields are used to determine whether the certificate is still trustworthy or has already been revoked or invalidated.

Intermediate certificate authorities: In practice however, the above described process, where the Root-CA directly signs a certificate for an entity, comes with a certain risk. That is, if the key from the Root-CA ever gets compromised during the signature creation, the Root-CA's certificate would be invalidated and therefore it would lose its invaluable status of unlimited trustworthiness. Hence, typically an Intermediate-CA is issued by a Root-CA, which then subsequently issues new certificates or new Intermediate CAs. The following Listing 4.3 shows how to create an Intermediate-CA using OpenSSL:

```
$> openssl ecparam -name secp521r1 -genkey
    -out intermediate_ca.key
$> openssl req -new -sha256 -key intermediate_ca.key
    -out intermediate_ca.csr
$> openssl ca -batch -config ../configs/rootca.conf -notext
    -in intermediate_ca.csr -out intermediate.crt
```

Listing 4.3: Issuing an intermediate CA

Certificate chain validation: To verify the validity of a certificate chain, an entity has to traverse that certificate chain and verify each certificate in that chain separately, until a trusted Root-CA is reached, or an invalid certificate is found. The following Listing 4.4 depicts the verification of an untrusted *entity.crt* by using a certificate chain *chain.pem*, whereby the Root-CA *root.crt* is trustworthy.

```
$> openssl verify -CAfile "root.crt" -untrusted chain.pem entity.crt
```

Listing 4.4: Verification of a certificate chain

Revoking certificates: In the previous paragraph, valid certificates were issued by either a Root-CA or an Intermediate-CA, however for the purpose of testing, it is also necessary to revoke such a certificate. This can be done using *Certificate Revocation Lists (CRL)* or using the *Online Certificate Status Protocol (OSCP)*. While OCSP is not within the scope of this thesis, CRLs can be created with the following commands, where a Root-CA first revokes the certificate of an entity (*entity.crt*). Subsequently, the Root-CA generates the CRL in PEM-format (Ascii) and finally the CRL is converted into the binary DER-format, as depicted in the following Listing 4.5:

```
$> openssl ca -config in.conf -revoke ../entity.crt
$> openssl ca -config ca.conf -gencrl -out ../root.crl.pem
$> openssl crl -inform PEM -in ../root.crl.pem -outform DER
    -out ../root.crl
```

Listing 4.5: Revocation of a certificate

zlib

As already mentioned in the *Design Chapter* of this work, data compression can be used to reduce the data overhead which comes along with the transmission of the certificate chain. Therefore, the implementation of the modified QSNFC uses the zlib[1]-library. This library was written in C and is a free, general purpose data compression library which uses gzip format for file operations but can also operate on data-stream formats. The following Listing 4.6 shows the usage of compress and decompress functions within the implementation of the modified QSNFC:

```
//compress
int retval = compress2(*zipped_bytes, &zipped_byte_len,
            bytes_to_zip, bytes_to_zip_len,
            Z_BEST_COMPRESSION);

//decompress
int retval = uncompress(unzipped_bytes, &unzipped_byte_len,
            bytes_to_unzip, zipped_bytes_len);
```

Listing 4.6: Usage of compress and decompress functions

4.3 libQSNFC

This section describes the implementation of the modified QSNFC-protocol. Therefore, for the remaining part of this thesis, this implementation is denoted as *libQSNFC*. As

[1] https://www.zlib.net

already mentioned in the first section of this chapter, the used programming language is C, therefore the output is a shared object file (libQSNFC.so), which can be linked during runtime. Within this section, the corresponding interface of this library is given, including important code snippets and concise descriptions.

4.3.1 Interface

The protocol implementation libQSNFC is intended to be used by a wide variety of applications, therefore, a comprehensive interface is defined within the work of this thesis. It acts as the main entry point for applications and provides the functionality to successfully run a QSNFC session. Hence, carrying forward the client-server pattern, which was introduced in the *Design Chapter*, both entities use this interface to interact with this library.

Setting QSNFC-related parameters (configuration)

At first, both client and server configure the QSNFC library according to their needs. Within the interface of the libQSNFC, several functions are provided to configure the library, as shown in the following Listing 4.7. This includes for the client, among others, the path to the trusted certificate store and the path to the cached certificates. Moreover, the server can set the path to its own certificate key, to the long-term key and the path to the certificate chain.

```
/*init client methods*/
  void QSNFC_init_client();
  void QSNFC_init_client_set_id(
          const QSNFC_byte id[]);
  void QSNFC_init_client_set_expected_server_id(
          const QSNFC_byte expected_server_id[]);
  int QSNFC_init_client_set_trusted_store(
          const unsigned char path_to_trusted_certs_folder[]);
  int QSNFC_init_client_add_cached_certificate(
          const unsigned char path_to_cached_cert[]);
  void QSNFC_init_client_Set_crp(
          const unsigned char crp);

   /*init server methods*/
  void QSNFC_init_server();
  void QSNFC_init_server_set_id(const QSNFC_byte id[]);
  int QSNFC_init_server_set_path_to_certificate(
          const unsigned char path_to_servers_certificate[]);
  int QSNFC_init_server_set_path_to_cert_chain(
          const unsigned char path_to_chain_file[]);
  int QSNFC_init_server_set_path_to_long_term_key(
          const unsigned char path_to_long_term_key[]);
  int QSNFC_init_server_set_supported_crs(
          const unsigned char crs);
```

```
    int QSNFC_init_server_set_compression(
            const unsigned char compression);
```

Listing 4.7: QSNFC-Interface

Data and message callbacks

In the *Requirements*-section of this chapter, the need of flexibility was explained, that is, the loose coupling between the communication layer and the protocol implementation. Hence, libQSNFC defines four callback functions to ensure the required flexibility, as shown in Listing 4.8. Additionally, these four callbacks are described in the following:

- **SendMessageCallback:** This callback is used by both entities to send a secure message. The libQSNFC subsequently handles the encryption-mechanism.
- **ReceiveMessageCallback:** Using this callback, the communicating entities can receive a secure message, which is already decrypted by the library.
- **SendBytesCallback** In order to send data via the communication channel, libQSNFC provides this callback, which is called by the library whenever data must be transmitted via that communication channel. That is, whenever one of the QSNFC-messages needs be sent.
- **ReceiveBytesCallback:** This callback is used, when data was received via the communication channel and is intended for the libQSNFC. Therefore, the library eventually parses the received data bytes (which consists of a QSNFC-message) and triggers the proper actions.

Additionally, Figure 4.2 contains a flow diagram where the usage of these callbacks is illustrated. At first, both client and server initialize the library (with the certificate paths, supported cryptographic primitives and so on) and after that, the callbacks are set. Now, both communication parties start the QSNFC-session.

Hence, if the client wants to start a secure communication, he sends a confidential message $msg1$ to the library using the $sendMessage()$-callback. Next, libQSNFC internally starts the initial and subsequent handshake using the provided callbacks $sendBytes$ and $receiveBytes$. Upon receival of the SH-message, which contains a confidential message $msg2$ from the server, libQSNFC decrypts and authenticates this confidential message. Finally, the corresponding callback $receiveMessage()$ gets called to provide the client the decrypted confidential message.

In turn, the server waits until he receives the inchoate CH-message using the provided callback $receiveBytes$. Upon receival, the library internally performs the initial handshake. After the complete-CH message is received, the callback $receiveMessage()$ is used to inform the server about the received confidential message $msg1$. Subsequently, the server responds with a confidential message $msg2$, which is forwarded to the library by using the callback $sendMessage()$. Finally, libQSNFC wraps the encrypted message $msg2$ within the SH-message and sends it to the client via the $sendBytes$ callback.

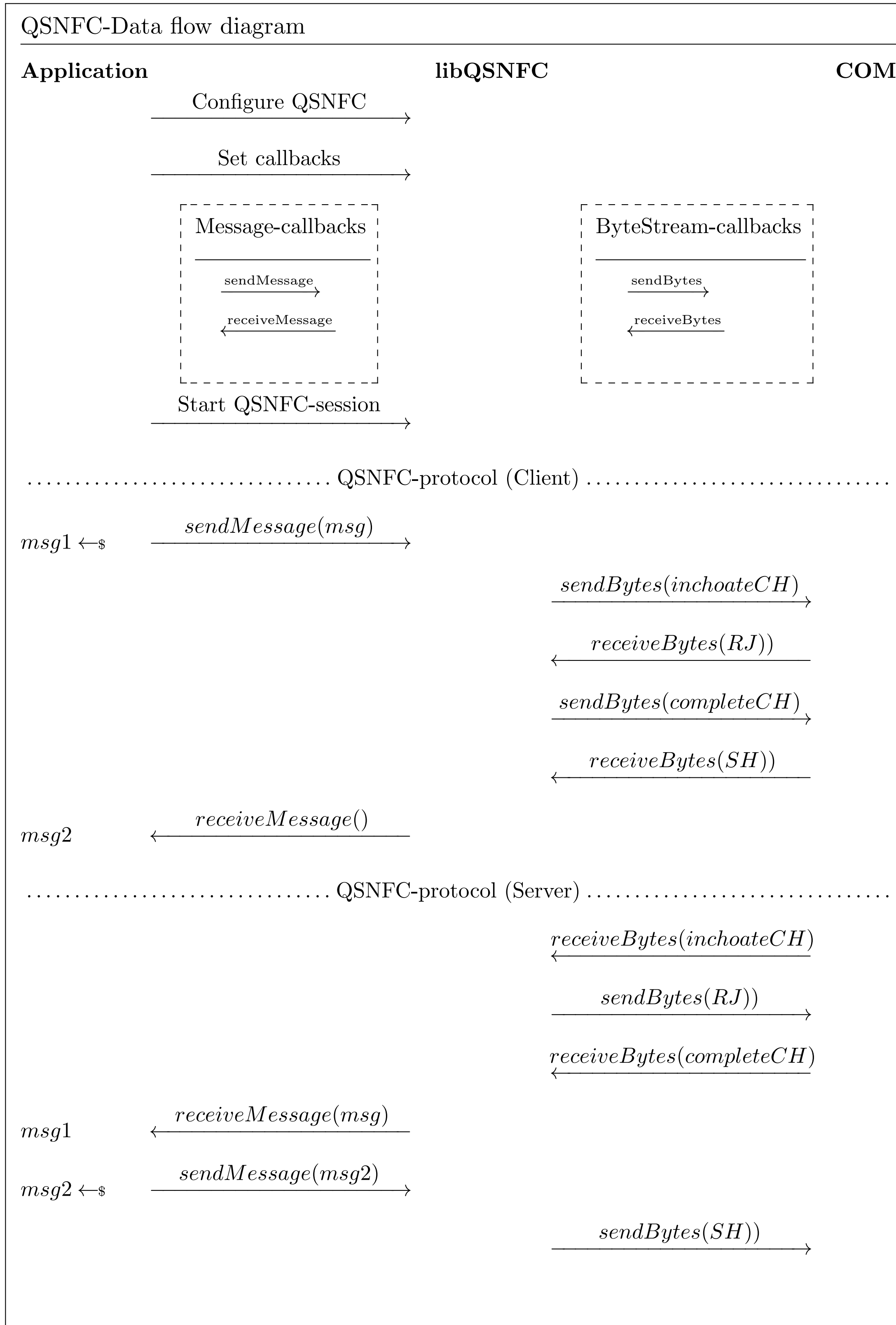


Figure 4.2: Data flow diagram

```
//define function pointer
  typedef int (*QSNFC_sendBytes)(const unsigned char*, size_t);
  typedef int (*QSNFC_readBytes)(unsigned char**, size_t*);

  /*callback methods*/
  int QSNFC_init_Set_SendMessage_Callback(
        QSNFC_readBytes callback);
  int QSNFC_init_Set_ReceiveMessage_Callback(
        QSNFC_sendBytes callback);
  int QSNFC_init_Set_SendBytes_Callback(
        QSNFC_sendBytes callback);
  int QSNFC_init_Set_ReceiveBytes_Callback(
        QSNFC_readBytes callback);
```

Listing 4.8: QSNFC-Callbacks

4.3.2 OpenSSL-API usage

As already mentioned previously, OpenSSL is the library used within the protocol implementation for all cryptographic-related operations. Therefore, the most important concepts and code snippets are listed and explained in the following:

EVP_PKEY

Whenever PKC is involved, OpenSSL uses the struct *EVP_PKEY*, which contains the key related parameters based on the specified key type. In the case of this thesis, an Elliptic-Curve key is used, however that same struct can also be used for RSA keys, for example. Such a key can then be used to compute a shared secret, to sign data or to verify a signature or a certificate chain. Listing 4.9 illustrates the generation of an elliptic curve key, whereby sanity checks are omitted for better readability:

```
int generate_new_key(EVP_PKEY** pkey, int curve) {

  EVP_PKEY_CTX *pctx, *kctx;
  EVP_PKEY *params = NULL;

  /* Create OpenSSL context for params */
  pctx = EVP_PKEY_CTX_new_id(EVP_PKEY_EC, NULL);

  /* Initialise params-context */
  EVP_PKEY_paramgen_init(pctx);

  EVP_PKEY_CTX_set_ec_paramgen_curve_nid(pctx, curve);

  /* Create parameters */
  EVP_PKEY_paramgen(pctx, &params);
```

```
    /* Kontext for key generation */
    kctx = EVP_PKEY_CTX_new(params, NULL);

    /* Generate  key */
    EVP_PKEY_keygen_init(kctx);
    EVP_PKEY_keygen(kctx, pkey);

    return 0;
}
```

Listing 4.9: OpenSSL-Generation of an EC-key

Compute a shared secret

Using the above generated key and another key received from the communication partner, a shared secret can be computed, as depicted in Listing 4.10, whereby sanity checks are omitted. The information, which key agreement protocol is used, is stored within the keys, and depending on this information the proper scheme is chosen by OpenSSL. The computed secret can then be used directly as an encryption key, or as input for a key derivation function.

```
QSNFC_byte calculate_secret(EVP_PKEY* private,
          EVP_PKEY* peer_public_key, size_t* secret_len) {

    EVP_PKEY_CTX *ctx = NULL;
    QSNFC_byte* secret = NULL;
    /*  context for shared secrets */
    ctx = EVP_PKEY_CTX_new(private, NULL)

    /* Initialization */
    EVP_PKEY_derive_init(ctx);

    /* Provide the oponent's public key */
    EVP_PKEY_derive_set_peer(ctx, peer_public_key);

    /* buffer size for shared secret */
    EVP_PKEY_derive(ctx, NULL, secret_len);

    /* Allocacte buffer */
    secret = OPENSSL_malloc(*secret_len);

    /* Compute the shared secret */
    EVP_PKEY_derive(ctx, secret, secret_len);

    EVP_PKEY_CTX_free(ctx);
```

```
    return secret;

}
```

Listing 4.10: OpenSSL-Shared secret computation

X509

The *X509* struct is used by OpenSSL for all certificate-related operations. This struct contains information about the corresponding key, issuer name, validity period and much more. OpenSSL has a built-in mechanism to verify the certificate chain, as shown in Listing 4.11 below:

```
  //init store and context
  X509_STORE* store = X509_STORE_new();
  X509_STORE_CTX* ctx = X509_STORE_CTX_new();

  //load trusted certificates
  X509_STORE_load_locations(store, NULL, "path_to_trusted_certs");

  //init context for verification with server's certificate
  //and corresponding certificate chain
  X509_STORE_CTX_init(ctx, store, server_cert, cert_chain);

  //verify certificate chain
   int retval = X509_verify_cert(ctx);

   if (ret != 1) {
      printf("Verification failed\n");
      return;
   }

   printf("Verified\n");
```

Listing 4.11: OpenSSL-Certificate chain verification

4.4 Two NFC-enabled IoT-devices

As previously mentioned, the demo applications are based on the use cases presented in the *Design-Chapter* of this thesis. Thus, the first use case, which is going to be implemented, comprises two NFC-enabled IoT-devices. In the following, the general setup is illustrated and subsequently, the development environment is going to be explained. Finally, important use-case related build steps are highlighted.

4.4.1 Setup

The setup for this demo application consists of two RaspberryPis Model 3B, both with an attached Explore-NFC shield. Both RaspberryPis use Raspbian as operating system whereby Figure 4.3 illustrates the corresponding setup.

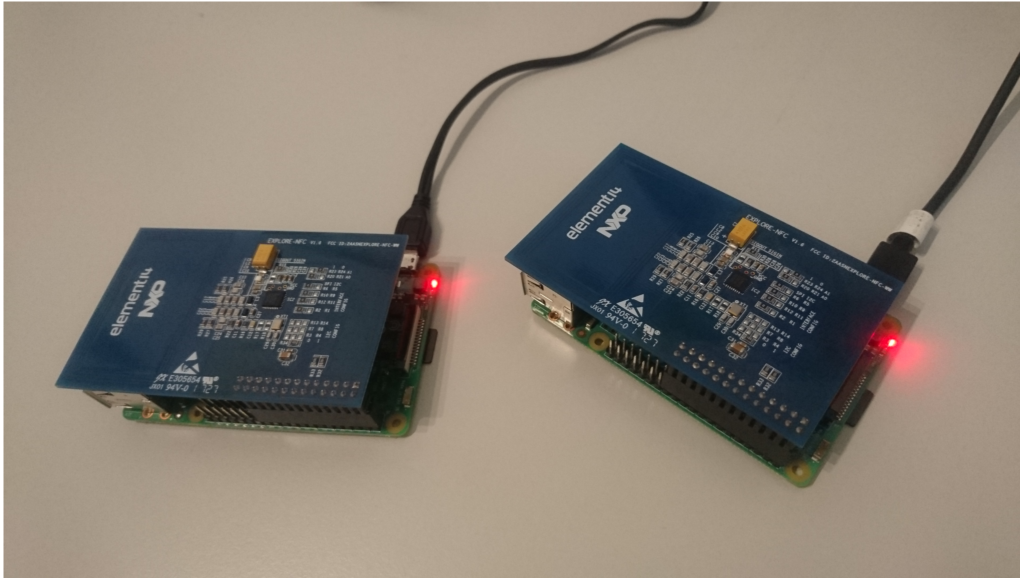


Figure 4.3: Two NFC-enabled IoT-devices

4.4.2 Development environment

This subsection describes the used development tools, moreover it gives short overview of the compilation process including the used toolchain and the used libraries. Moreover, the process of remote debugging is presented.

IDE

During the development of the RaspberryPi-to-RaspberryPi demo application, the free and open source cross-platform IDE KDevelop[2] was used, which provides support for C/C++, Python, JavaScript and PHP. KDevelop is based on the KDE platform and on the QT-Framework, and runs on Linux, macOS and on Windows. Moreover, it also supports the use of CMake-projects out of the box and comes bundled with GDB-integration.

CMake

CMake[3] is an open source tool to build and package software. It is used within this thesis to automate the build process of the demo application within KDevelop. Furthermore, cross compiler directives are set within CMake to build for another platform, in this case for the RaspberryPi.

[2] https://www.kdevelop.org/
[3] https://cmake.org/

GDB

The GNU Project Debugger[4] is a versatile debugging tool for applications written in C and C++. Many IDEs provide built-in support, however the GDB is also accessible via the command line. Additionally, it supports remote debugging, as presented later within this section.

NXP-Reader library

To communicate with the attached NFC-Shield, NXP's Reader-library is used. This C-library abstracts the hardware interaction with the NFC-controller, and it supports NFC's P2P and Card-Emulation mode.

Cross-compilation

The development of this demo application requires a cross compilation tool chain, because the programming tasks are performed on an PC using an AMD64 architecture, however the RaspberryPi is based on an ARMv8-architecture. Hence, the Linux ARM cross compiler toolchain[5] was used to build both the libQSNFC and the actual demo application. This toolchain can be found for many Linux-based operating systems in the corresponding package archives. Other used libraries such as OpenSSL or lzip need not be cross-compiled as there exist prebuilt-artifacts in the Raspbian package archives.

Remote-Debugging

As already mentioned previously, GDB supports remote-debugging. Therefore, the following listings illustrate the process how to setup such a remote debugging session, under the assumption that the RaspberryPi and the development machine are connected in the same network. Listing 4.12 depicts the startup command for the gdbserver:

```
$rpi> gdbserver localhost:2000 qsnfcDemo
```

Listing 4.12: Starting GDB-server on RaspberryPi

To debug an executable, which was built for a different platform, a special debugger (*arm-linux-gnueabihf-gdb*) for ARM-executables was used. This debugger is provided by Linaro Toolchain Binaries[6]. The following Listing 4.13 illustrates the process how to connect to the remote gdbserver, which was mentioned above.

```
$host> arm-linux-gnueabihf-gdb qsnfcDemo
$(gdb)> target remote <ip_of_raspberry_pi>:2000
```

Listing 4.13: Starting remote debug session

[4] https://www.gnu.org/software/gdb/
[5] https://packages.debian.org/sid/gcc-arm-linux-gnueabihf
[6] https://launchpad.net/linaro-toolchain-binaries/

4.4.3 Implementation

This subsection covers implementation-relevant topics. At first, a concise overview of the SNEP protocol is given, which is used within this demo application as transport layer protocol. After that, the interaction with NXP's Reader-library is highlighted and important code snippets are shown.

SNEP-Simple NDEF Exchange protocol

As already noted in the *Prerequisites Chapter*, SNEP is based on NFC's P2P mode. Therefore, SNEP will be used within this demo application as transport layer protocol. Both SNEP and libQSNFC follow a client/server-pattern, therefore the term client refers to both the client within SNEP as well as the client within libQSNFC, and consistently the same applies for the server. Within this demo application, SNEP-Get requests are used for communication. Hence, the client wraps the data he wants to send in a SNEP-Get request, whereby this request contains the serialized QSNFC-message (inchoate-CH, complete-CH or SD-message). Subsequently, this request is sent to the server. In turn, the server answers with an NDEF message, whereby this response contains a wrapped QSNFC-message (AB, RJ, SH or SD).

Interaction with NXP's Reader library

The following code snippet in Listing 4.14 illustrates the combined usage of the libQSNFC callbacks and NFC's Reader library, in particular the SNEP-Get request. The function *sendBytes*() gets called whenever libQSNFC needs to send data bytes to a communication partner, whereby *clientbuf* is going to contain the responded NDEF-message from the SNEP-Get request.

```
/* SNEP component holder */
phnpSnep_Sw_DataParams_t snpSnepClient;
/* buffer for NDEF-response */
uint8_t clientbuf[10000];
uint32_t clientlen;
int sendBytes(uint8_t* data, size_t len)
{
    phStatus_t ret_status = phnpSnep_Get(&snpSnepClient,
                    data, len,clientbuf, &clientlen, 10000 );
    if(ret_status != PH_ERR_SUCCESS)
    {
        printf("Error in send bytes \n");
        return -1;
    }
    return 0;
}
```

Listing 4.14: Interaction with NXP's Reader library

4.5 An NFC-enabled IoT-device and an NFC-enabled smart phone

This section describes the second developed demo application within the context of this thesis. The application covers the second use case, which was highlighted at the beginning of the *Design-Chapter*. Hence, this application comprises an NFC-enabled IoT-device and an NFC-enabled smart phone communicating in a secure manner via NFC. As the development of this demo application requires a new communication protocol, SAEP will be introduced. After that, the development environment and all used tools will be listed, including some remarks on the cross-compilation.

4.5.1 Simple APDU Exchange Protocol

As presented in the previous section of this chapter, the first demo application uses two RaspberryPis with an attached NFC-Shield. Hence, full access to the underlying communication layer is provided by the corresponding NFC-Hardware Abstraction Layer, in this case NXP's NFC-Reader library. However, Android restricts the access to its NFC-implementation. That is, even though an NFC-P2P communication is possible via Android Beam[7], it requires a user interaction for each message round trip. While this is sufficient for simple applications, it is not suitable for a general-purpose communication protocol with multiple round trips. Hence, the Simple APDU Exchange Protocol (SAEP) was designed within the scope of this thesis to counteract this issue. The following description highlights the main ideas of SAEP:

Objective

The objective of this protocol is to provide a request-response like protocol using a non-rooted Android phone with limited access to its NFC-interface, whereby the other communication partner is either an NFC-enabled IoT-device (which is the case within this demo application) or another Android phone with limited NFC-access.

Setup

Following up the client/server model, a client issues requests and a server answers with responses, therefore the client has to be set into NFC's Reader/Writer-Mode and the server has to emulate an NFC-tag by using NFC's Card Emulation mode. Hence, this demo application comprises a RaspberryPi with an attached NFC-Shield representing the server, and an NFC-enabled Android phone, which represents the client.

Protocol

To establish such a request-response protocol, the client writes into the data storage of the tag and sets a status byte. This status byte is located within the first byte of the NFC-memory container. The second and third byte represent the length of the following data, which starts at the fourth byte. Furthermore, the status gets polled by the server periodically and indicates whether the write-procedure is completed or not. If this is not

[7] https://developer.android.com/guide/topics/connectivity/nfc/nfc#p2p

the case, then the provided tag memory size is too small for the amount of data being sent via NFC. Therefore, the data transfer has to be chunked. Hence, the server must cache this information somewhere else as long as the status byte indicates that the write-procedure is completed. Subsequently, the server processes the cached information and writes the response into his own NFC-memory container. Meanwhile, the client polls the status byte in the same fashion as described before, and depending on whether the server's response is chunked or not, the client waits until the write-process is finished. Table 4.1 list all implemented status bytes, whereby *x1* is set when a new chunk has been written to the NFC-container. That is, one or more subsequent chunks are going to follow. A subsequent chunk is identified by the value x2 in the status byte. Once the write process is finished, the value *x3* is written in the status byte. Finally, the value *00* indicates, that the device is ready to be written.

Value (in Hex)	**Meaning**
0x00	Ready to be written
0x01	New chunk (Client)
0x02	Subsequent chunk write (Client)
0x03	Write finished (Client)
0x11	New chunk (Server)
0x12	Subsequent chunk (Server)
0x13	Write finished (Server)

Table 4.1: Status byte for SAEP

4.5.2 APDU-commands

Within this demo application, an NFC-Tag 4 type was used for the implementation. Therefore, this subsection lists the used APDU commands, which are needed to read and write data and for querying the capability container. A more detailed description of these commands can be found in [NFC11b].

- **Select (Instruction-byte 0x4A):** This command is used to select both the capability container and the NDEF container, which comprises the actual data.
- **Read (Instruction-byte 0xB0):** The Read-command is used to retrieve data from the capability container and to read data from the NDEF-container.
- **Write (Instruction-byte 0xD6):** This command is used to write data in the NDEF-container.

4.5.3 Development environment

Within this subsection, the development environment for the Android part of this demo application is going to be explained. This includes the used IDE, build systems and interfaces. The development for the RaspberryPi part of this application is similar to the one described within the previous section of this chapter.

Android-Studio

Android-Studio[8] is used within the work of this thesis to develop the Android-app for this demo application. This IDE is based on the IntelliJ-platform and is available free of charge. It runs on Linux, Windows and macOS, and it requires an installed Java-Development-Kit. Android-Studio is designed to support the development of Android-applications written both in Java and Kotlin, and additionally it provides a powerful GUI-Designer. The Android Software-Development-Kit comes bundled within Android-Studio, which is the responsible tool for compiling and packaging Android applications.

JNI

JNI is an abbreviation for the Java-Native-Interface. The purpose of JNI is to enable Java-Code to call native C functionality. The JNI interface is used within this demo application to call C-functions from the libQSNFC.

Android-NDK

The Android Native-Development-Kit (NDK) is a set of tools to compile native C Code for the Android platform. This toolset can be downloaded and installed within Android-Studio using the included SDK-Manager. Once installed, C code can be edited and built within Android-Studio, and subsequently functions from this C-code can be called within Java via JNI.

Gradle

Gradle is a build system to maintain library dependencies, manage the build order of Java/Kotlin Code, change run configurations and set version codes. Gradle comes included with Android-Studio, however it is not bound to Android/Java applications but can also be integrated in the development of C++ or Python -based applications.

Hardware

For the development of this application, a Sony Xperia Z5 Compact was used, running Android version 7.1.1 (non-rooted).

4.5.4 Implementation-Android

This subsection describes implementation-relevant details of the second demo application, in particular the Android part. Furthermore, important code snippets are shown and complex build steps are explained.

Building native C-code for Android

Instead of re-writing the libQSNFC in Java, this paragraph presents a way to utilize an existing C-library for Android. Therefore, Java-Code within an Android application calls native C functions provided by the protocol implementation libQSNFC.

[8] https://developer.android.com/studio

Building dependencies: As already presented in the *Implementation Chapter*, libQSNFC uses two external libraries, which comprises zlib for data compression and OpenSSL for cryptographic computations. Whereas zlib is included within the Android NDK, and thus, doesn't have to be compiled manually, OpenSSL is not. Hence, due to missing official distributions for Android, OpenSSL must be compiled for Android manually. Due to Android's support of different CPUs and thus, different *Application Binary Interfaces (ABI)*[9] with their own instruction set, it is required to build a native C-library for all supported ABIs to ensure functionality for all supported Android phones. Within this work, the instruction set *armeabi* from ABI *armeabi-v7a* was chosen because the Android phone, which was used during development, is based on that architecture. Subsequently, Android's NDK is used to build OpenSSL for this platform using the commands presented in Listing 4.15.

```
$> tar xvfz openssl-1.1.0f.tar.gz
$> cd openssl-1.1.0f/
$> ./Configure android-armeabi -L<ANDROID_SDK_PATH>/ndk-bundle/
      toolchains/arm-linux-androideabi-4.9/prebuilt/linux-x86_64/
      lib/gcc/arm-linux-androideabi/4.9.x/armv7-a
      -L<ANDROID_SDK_PATH>/ndk-bundle/toolchains/
      arm-linux-androideabi-4.9/prebuilt/linux-x86_64/
      arm-linux-androideabi/lib/armv7-a -latomic
```

Listing 4.15: Build OpenSSL for Android's ABI armeabi-v7a

Build libQSNFC: After the successful build of OpenSSL, libQSNFC can be compiled by adding this new OpenSSL build as a link-asset to the Android-Studio project. In particular, libQSNFC is added as a CMake project reference. Hence, by using JNI, the original libQSNFC can be accessed via the Android app. Listing 4.16 depicts the usage of JNI to access functionality from the protocol implementation.

```
public class JNI_Interface {

    //load libQSNFC
    static {
        System.loadLibrary("libqsnfc");
    }

    /* native QSNFC methods */
    public native void QSNFC_init_client();
    public native void QSNFC_init_server();
    ...
  }
```

Listing 4.16: Usage of Android's JNI

[9] https://developer.android.com/ndk/guides/abis.html

WebNFC

The WebNFC-API[10] is a currently unstable specification of an NFC-API, which enables the access of the NFC-interface on a smart phone using JavaScript. This API is supported only by a few experimental browser versions. However, within the context of this demo application, the idea of WebNFC-API is used to increase the practical applicability of libQSNFC. Due to the fact that QSNFC is neither an official protocol specification nor the corresponding implementation, libQSNFC, is anywhere available in modern cryptographic tool suites, WebNFC clearly does not support QSNFC. Thus, the Android application is designed in such a way that it emulates a browser with rudimentary access to libQSNFC via JavaScript. The confidential information can then be sent from JavaScript via libQSNFC to the communication partner. Consequently, a web server sends confidential data protected by TLS to the mobile phone, and then the data gets sent to the NFC communication partner via QSNFC. Hence, this provides a secure end-to-end encryption, as depicted in Figure 4.4. Therefore, presuming that libQSNFC is included in future versions of browsers, no additional app is needed to communicate via QSNFC on a smart phone. In addition to that, libQSNFC could use the browser's certificate store for its configuration with trusted certificates.

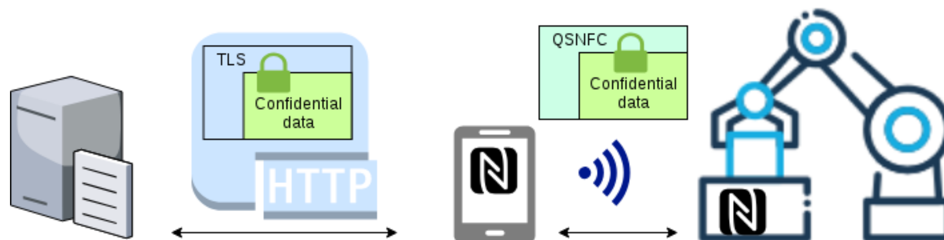


Figure 4.4: Secure communication from a server to an IoT-device

Android's JavaScript Interface

In order to expose Java-functionality to JavaScript, methods can be marked with the @*JavascriptInterface*-Annotation, as depicted in Listing 4.17.

```
//This function can be called from Javascript
//e.g. window.JSInterface.SetSecuredDate('Confidential',callback);
@JavascriptInterface
public void SetSecuredDate(String data, String callbackFunction)
{
    //process data
}
```

Listing 4.17: Accessing Java from JavaScript

[10] https://w3c.github.io/web-nfc/

Backend-Server

Within the scope of this demo application, a rudimentary servlet was developed to act as endpoint for the Android-application, which establishes an HTTPS connection with that server. Subsequently, an HTML-page with JavaScript-Code is received by the app. Using this JavaScript in the emulated Browser, the app-user can initiate a QSNFC-session. The already described IDE NetBeans was used to implement and host that code on an Embedded-Web-Server.

Android-App

Figure 4.5 illustrates the final Android-app, where the App acts as QSNFC-client and communicates with a RaspberryPi, which acts as a server.

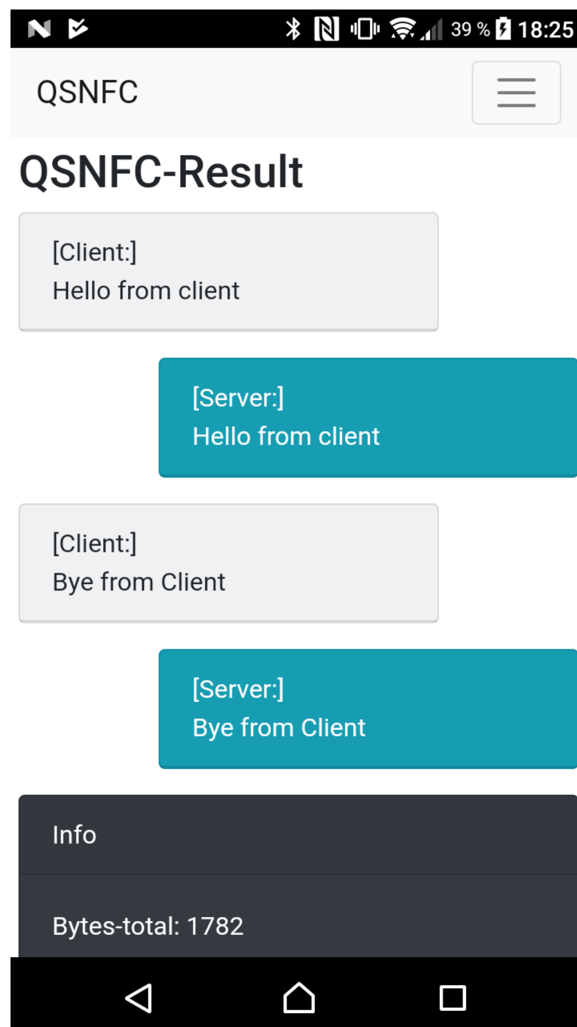


Figure 4.5: Android app

4.5.5 Implementation-RaspberryPi

The other part of this demo application is implemented on the RaspberryPi-platform by using the same IDE, tools and libraries from the first demo application, where two RaspberryPis were communicating via P2P. This time however, the RaspberryPi acts as the server within the QSNFC-session. Therefore, it has to operate in NFC's Host-Card emulation mode to run the previously defined SAEP, as P2P is not supported by the Android implementation. Within this demo application, NXP's Reader Library is used for the emulation of an NFC-Tag 4 type.

Chapter 5

Evaluation

This chapter covers a comprehensive evaluation of the modified QSNFC-protocol, both for the design and the implementation of libQSNFC. At first, the presented security threats of the existing QSNFC are considered again with respect to the chosen protocol design of the modified QSNFC protocol. After that, a message overhead and timing comparison for different modes of operation of the demo applications concludes this chapter.

5.1 Protocol evaluation

This section recaps the threats identified within the *Related Work Chapter* of this thesis and analyzes the corresponding security countermeasures. Furthermore, it evaluates the countermeasures of the security vulnerabilities of the original QSNFC, which were presented within this thesis.

Eavesdropping

Due to the usage of authenticated encryption algorithms, confidential messages are protected against eavesdropping, however eavesdropping itself cannot be avoided. Even though the modified QSNFC contains unprotected fields in the QSNFC-messages, an attacker cannot use this information to mount an attack.

Data corruption/modification/insertion

Any data corruption/modification/insertion of confidential or authenticated data is going to be noticed by the receiver due to the usage of authenticated encryption algorithms, however, the altering of data cannot be avoided.

Man-in-the-Middle

There is no way to prevent a MitM-attack, but due to the above described reasons, a MitM is neither able to gather confidential data nor is he able to alter messages without the notice of the receiving communication party. Furthermore, a MitM-attack is not able to impersonate the server due to the use of certificates.

Denial-of-Service

It is not feasible to protect a communication on protocol level against DoS-attacks, as a possible interference by an attacker can be detected but not avoided.

Downgrade-Attack

An attacker cannot downgrade the chosen encryption algorithm without the detection of the client, as the prior chosen value is compared with the one comprised in the received signature by the server.

MITM on the original QSNFC

In the *Design Chapter*, two variants of a MitM-attack on the original QSNFC were shown. The presented countermeasures involve the constraint, that confidential payloads must not be empty. Additionally, the public keys are now protected, as these keys are included as additional authenticated data to the authenticated encryption algorithm. Hence, such a MITM is no longer possible.

Freshness

As presented in the *Design Chapter*, the original QSNFC is vulnerable against Replay-Attacks due to missing nonces. Therefore, the modified QSNFC includes nonces, which are protected either by signatures (in the initial handshake), or added as additional authenticated data to the authenticated encryption algorithm in the subsequent handshakes and for SD messages. Thus, such a Replay-Attack is no longer possible.

5.2 Demo evaluation

This section covers the evaluation of the previously implemented demo applications. Therefore, these implementations are going to be tested using multiple test scenarios and thus, evaluating the performance with respect to message overhead and timing. At first, the proposed requirements within the protocol implementation are revised. After that, test scenarios for the software implementations are stated, and subsequently, the test results according to these test scenarios are presented for the two demo applications.

- **Flexibility:** The implementation is not bound to NFC at all, in fact there is not a single line within the code related to NFC. As already highlighted in the *Implementation Chapter*, during the development a simple buffer was used to simulate a communication between two entities.
- **Configurability:** The library provides an interface to set the desired options and flags for subsequent QSNFC-sessions.
- **Availability:** The libQSNFC was implemented in C, therefore it is applicable for many fields of application. The presented demo applications utilize the protocol implementation libQSNFC by using it from C and Java programs.

5.2.1 Test scenarios

This subsection lists test-scenarios for both demo applications, whereby these applications are going to be evaluated with respect to the message overhead and execution time. Hence, libQSNFC is going to be tested with different message sizes, while using all supported encryption algorithms and all performance-improving mechanisms such as cached-certificates, data compression and the usage of subsequent handshakes, as stated in the list below:

- **Demo application:** Within the evaluation, both demo applications should be tested. That is, the first demo application use case where two RaspberryPis with attached NFC-Shield communicate via NFC's P2P mode. Furthermore, the second demo application, where a mobile phone based on Android and a RaspberryPi communicate with the SAEP, is also going to be evaluated.
- **Encryption algorithms:** This evaluation of the libQSNFC should comprise all available authenticated encryption algorithms for QSNFC, that is AES-128, AES-192 and AES-256 in CCM and GCM mode.
- **Data compression:** The usage of data compression for the certificate chain was introduced within the *Design Chapter*, therefore the impact on the data overhead by using data compression should be evaluated.
- **Cached certificates:** The concept of cached certificates was presented in the *Design-Chapter* of this thesis. Hence, the proposed performance improvement should be shown within this evaluation.
- **Subsequent handshake:** Without the significant data overhead from the initial handshake, the benefit of the 0-RTT-property will be verified in this evaluation by using subsequent handshakes only.
- **Different message inputs:** The demo applications are going to be evaluated using different message inputs and multiple round trips. Therefore, not only subsequent handshakes but also the exchange of SD messages are involved.

5.2.2 Overhead and timings

This subsection presents selected results based on the previously defined test scenarios. At first, the message overhead will be shown. After that, the execution time is going to be evaluated, and finally, the time and message overheads with respect to the available encryption algorithms are presented.

Message data overhead

The first evaluation comprises the message data overhead for QSNFC-sessions. The data overhead is computed according to Equation 5.1:

$$overhead = \#totalBytes - \#payloadBytes \tag{5.1}$$

Therefore, Figure 5.1 compares the data overhead in bytes, for 100 payload bytes (confidential data) per message, using one and two round trips, respectively. Additionally, the corresponding overhead for a QSNFC-session using uncompressed and compressed certificate chains is shown. Furthermore, the significant decrease of data overhead is depicted by using cached certificates and subsequent handshake only. For this evaluation, the first demo application (2 NFC-enabled IoT-devices) was used together with AES128-CCM as encryption algorithm. As can be seen clearly within this figure, the usage of cached certificates and subsequent handshakes decreases the overall message overhead significantly. Furthermore, the introduced data compression almost halves the message overhead compared to a QSNFC-session using uncompressed certificate chains.

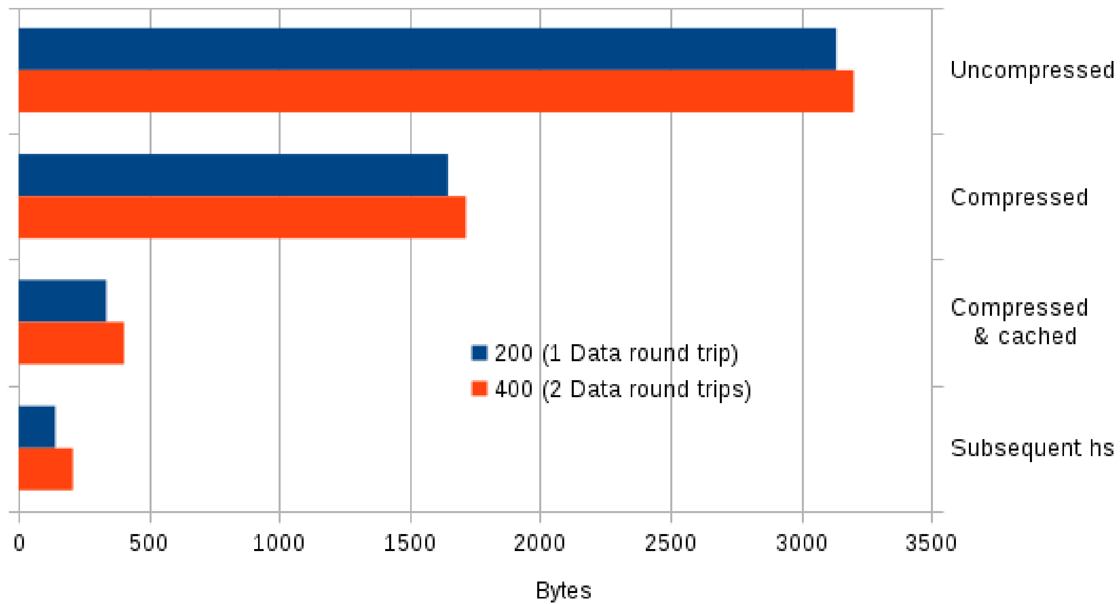


Figure 5.1: Message overhead for libQSNFC

Execution time

Figure 5.2 depicts the execution time of libQSNFC to successfully perform a QSNFC session between an Android phone and a RaspberryPi including one data round trip with 10 and 100 bytes, respectively. The categorization within that figure follows the categorization in the last figure. Similarly, the execution time evaluation depicted within the current figure also emphasizes the significant performance speed up by using data compression, cached certificates and subsequent handshakes (0-RTT).

Usage of different encryption algorithms

Table 5.1 illustrates the evaluation of QSNFC-sessions for all available block sizes and authenticated encryption modes. Therefore, within a subsequent-handshake only session, a confidential payload with 100 data bytes per message is used at first, yielding a total

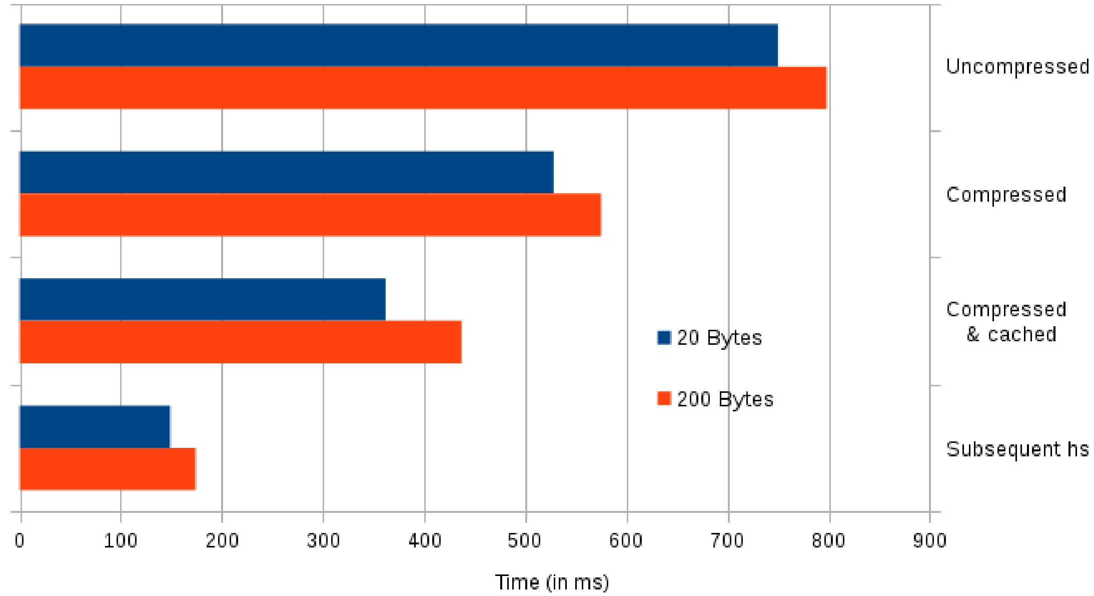


Figure 5.2: Execution time for libQSNFC

amount of 400 data message bytes for the subsequent handshake and one SD message round trip. Secondly, 10 data bytes per message are send, which results in a total of 20 data message bytes for the subsequent handshake. Within the above-mentioned Table, execution time and data message overhead are shown. Hence, it can be seen that the usage of different encryption modes using the same payload has only a slightly impact on the message overhead and on the execution time.

	400 data bytes		**20 data bytes**	
	Overhead (in Bytes)	Execution time (in ms)	Overhead (in bytes)	Execution time (in ms)
AES128-CCM	209	306	141	149
AES196-CCM	225	308	157	159
AES256-CCM	241	321	173	185
AES128-GCM	209	306	141	147
AES196-GGM	225	312	157	156
AES256-GCM	241	322	173	171

Table 5.1: Performance evaluation for different encryption modes

Chapter 6

Conclusion and Future Work

The following chapter concludes this thesis, therefore a compact summary of the presented work is given at first. Afterwards, limitations of the proposed protocol and its implementation are stated and based on that limitations, an outline for possible future work is given.

6.1 Conclusion

Within the scope of this master's thesis, a secure and efficient protocol for wireless communication was presented. In the Chapter *Prerequisites and Related work*, all necessary primitives were described and after that, the current state of the art was analyzed. Therefore, all threats for an NFC-session were examined. Subsequently, already existing security countermeasures were listed and described, and among others, the original version of QSNFC was presented.

In the *Design Chapter* of this thesis, application use cases for the proposed protocol were identified at first. After that, a security and threat analysis of the original version of QSNFC was conducted, which yielded several severe vulnerabilities in the protocol flow. Furthermore, this original version of QSNFC was analyzed with respect to performance. Based on the security and performance analysis, a modified QSNFC-protocol was presented, which is no longer vulnerable to the identified security threats and includes several performance improvements.

The subsequent chapter covers the implementation related topics of the modified QSNFC-protocol and the demo applications. Therefore, the requirements of such an implementation are identified and furthermore, the development process of the corresponding C-library libQSNFC is highlighted. Additionally, the development processes of two demo applications are shown.

Finally, an *Evaluation Chapter* concludes the work of this thesis. Therefore, the fulfillment of the given requirements is analyzed. Furthermore within this chapter, two demo implementations for the identified use cases are evaluated with respect to overhead and performance.

6.2 Known limitations

This section discusses several limitations, which are implied by both the design and the implementation of the proposed modified QSNFC protocol. The following list enumerates and describes each of the design-related limitations briefly and gives a short outline on how to overcome those issues within future work, whenever possible:

1. **Caching:** Both client and server need to cache cryptographic primitives for a subsequent session to fulfill the 0-RTT property. While this is not limiting factor in a small setting, it may require cache replacement strategies for large settings, as already discussed in the original QSNFC protocol description. However, the proper strategy depends on the field of application.

2. **Server's long-term key:** If the server's long-term key for a specific client ever gets compromised, an attacker can impersonate the server and thus, the server's certificate must be revoked. Hence, even if this long-term key gets compromised, the certificate should not need to be revoked.

3. **Computational power:** Clearly, the proposed design is not suitable for simple NFC tags, as QSNFC requires computational power for encryption and authentication. Therefore, another protocol scheme has to be developed or the existing QSNFC-protocol has to be extended, which is out of the scope of this thesis and thus, subject for future work.

While the list above highlights the limitations on protocol level, the following lists enumerates the implementation relevant limitations with respect to the library implementation of QSNFC, libQSNFC:

1. **Secure Storage:** Currently, all the cryptographic primitives which are used in subsequent handshakes are stored in memory. However, in a future development step it is advisable to store those credentials in a Secure Element.

2. **Caching:** The current software implementation does not provide a cache replacement strategy, so an application which uses the library needs to take care of the strategy implementation. Therefore, future version of the libQSNFC should provide an interface, where the application can decide how the credentials are stored and which cache replacement strategy is used.

3. **OCSP:** At the moment, the protocol implementation does not support OCSP for certificate chain validation. Thus, a future version of the library implementation should provide support for OCSP validation.

4. **Crypto-implementation:** Currently, OpenSSL is used for all cryptographic computations within the library. As the imposed code overhead by using this library is not an issue for mobile phones and IoT-devices such as the RaspberryPi, it is desirable to substitute OpenSSL with another cryptographic library, which provides only a minimal coding footprint, such as *mbed TLS*[1]. Therefore, an interface for the usage of different cryptographic libraries has to be defined in a future libQSNFC-version.

[1] https://tls.mbed.org/

5. **Android:** As already mentioned in the last chapter, Android supports all NFC-modes, however the applicability of P2P is limited though due to the required user interaction within each round trip. Hence, a simple protocol was presented to overcome this limitation on a stock Android phone. However, if Android weakens the restrictions on the use of its P2P-mode, a more concise implementation would be possible without having to deal with APDU-commands.

6. **iOS:** In contrary to Android, iOS does only provide support for reading proper encoded NFC-tags, but not for writing. This is caused by a software-based access restriction to the built-in NFC-chip on iOS devices. Hence, libQSNFC is not applicable for iOS devices.

7. **WebNFC:** The specification of WebNFC is still in an early and unstable state, therefore only a rudimentary interface was implemented within this thesis. Thus, once the specification becomes more precise, a more advanced approach could be used.

8. **Communication Layer:** The demo applications use NFC as communication layer, however due to the flexibility of libQSNFC, it also possible to apply QSNFC for other wireless communication protocols such as Bluetooth, which could be implemented and evaluated in another demo application.

Bibliography

[Ada11a] Carlisle Adams. *Certificate*, pages 188–189. Springer US, Boston, MA, 2011.

[Ada11b] Carlisle Adams. *Replay Attack*, pages 1042–1042. Springer US, Boston, MA, 2011.

[AES01] Advanced encryption standard (AES). Technical report, nov 2001.

[BBSP13] Dobre Blazhevski, Adrijan Bozhinovski, Biljana Stojcevska, and Veno Pachovski. MODES OF OPERATION OF THE AES ALGORITHM. 04 2013.

[Bla11] J. Black. *Authenticated Encryption*, pages 52–61. Springer US, Boston, MA, 2011.

[BN00] Mihir Bellare and Chanathip Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In Tatsuaki Okamoto, editor, *Advances in Cryptology — ASIACRYPT 2000*, pages 531–545, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[BPPT17] Mugurel Barcau, VicenŢiu Paşol, Cezar Pleşca, and Mihai Togan. On a Key Exchange Protocol. In Pooya Farshim and Emil Simion, editors, *Innovative Security Solutions for Information Technology and Communications*, pages 187–199, Cham, 2017. Springer International Publishing.

[CLL+17] Y. Cui, T. Li, C. Liu, X. Wang, and M. Khlewind. Innovating Transport with QUIC: Design Approaches and Research Challenges. *IEEE Internet Computing*, 21(2):72–76, Mar 2017.

[Cro11] E. Cronin. *Denial of service*, pages 143–144. Springer US, Boston, MA, 2011.

[Des11] Yvo Desmedt. *Relay Attack*, pages 1042–1042. Springer US, Boston, MA, 2011.

[De11] Danny DeCock. *X.509*, pages 1395–1395. Springer US, Boston, MA, 2011.

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

[DKK+11] S. Dunnebeil, F. Kobler, P. Koene, J. M. Leimeister, and H. Krcmar. Encrypted NFC Emergency Tags Based on the German Telematics Infrastructure. In *2011 Third International Workshop on Near Field Communication*, pages 50–55, Feb 2011.

[DR02] Joan Daemen and Vincent Rijmen. The Design of Rijndael. 01 2002.

[Ecm15a] Ecma international. *ECMA-385 NFC-SEC: NFCIP-1 Security Services and Protocol*, 6 2015. 4th Edition.

[Ecm15b] Ecma international. *ECMA-386 NFC-SEC-01: NFC-SEC Cryptography Standard using ECDH and AES*, 6 2015. 3rd Edition.

[Ecm15c] Ecma international. *ECMA-409 NFC-SEC-02: NFC-SEC Cryptography Standard using ECDH-256 and AES-GCM*, 6 2015. 2nd Edition.

[Ecm17a] Ecma international. *ECMA-410 NFC-SEC-03: NFC-SEC Entity Authentication and Key Agreement using Asymmetric Cryptography*, 6 2017. 3rd Edition.

[Ecm17b] Ecma international. *ECMA-412 NFC-SEC-03: NFC-SEC Entity Authentication and Key Agreement using Symmetric Cryptography*, 6 2017. 3rd Edition.

[HB06] Ernst Haselsteiner and Klemens Breitfuss. Security in Near Field Communication (NFC) Strengths and Weaknesses. 2006.

[Hei11] Clemens Heinrich. *Transport Layer Security (TLS)*, pages 1316–1317. Springer US, Boston, MA, 2011.

[HJS16] S. Hameed, U. M. Jamali, and A. Samad. Protecting NFC data exchange against eavesdropping with encryption record type definition. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pages 577–583, April 2016.

[Hün19] Felix Hüning. *Internet of Things und Industrie 4.0*, pages 1–10. Springer Berlin Heidelberg, Berlin, Heidelberg, 2019.

[ISO07] Near Field Communication - Interface and Protocol(NFCIP -1. Standard, Internatial Organization for Standardization, 2007.

[ISO08] Identification cards - Contactless integrated circuit cards. Standard, Internatial Organization for Standardization, 2008.

[JIC14] Brian Jepson, Tom Igoe, and Don Coleman. *Beginning NFC*. O'Reilly Media, 2014.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. The Elliptic Curve Digital Signature Algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, Aug 2001.

[Jus11] Mike Just. *Schnorr Identification Protocol*, pages 1083–1083. Springer US, Boston, MA, 2011.

[Kal11] Burt Kaliski. *RSA Digital Signature Scheme*, pages 1061–1064. Springer US, Boston, MA, 2011.

[Knu11] Lars R. Knudsen. *Block Ciphers*, pages 152–157. Springer US, Boston, MA, 2011.

[Kob87] Neal Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[Koe11] François Koeune. *Pseudorandom Number Generator*, pages 995–996. Springer US, Boston, MA, 2011.

[Kra10] Hugo Krawczyk. Cryptographic Extraction and Key Derivation: The HKDF Scheme. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, pages 631–648, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[KW12] Thomas Korak and Lukas Wilfinger. Handling the NDEF signature record type in a secure manner. pages 107–112, 11 2012.

[KW16] H. Krawczyk and H. Wee. The OPTLS Protocol and TLS 1.3. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 81–96, March 2016.

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

[LRW01] Helger Lipmaa, Phillip Rogaway, and David Wagner. CTR-Mode Encryption. 05 2001.

[MHS10] Vctor Martinez, Luis Hernandez, and Carmen Snchez. A Survey of the Elliptic Curve Integrated Encryption Scheme. *Journal of Computer Science and Engineering*, 2:7–13, 01 2010.

[MKR17] Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. PKCS #5: Password-Based Cryptography Specification Version 2.1. RFC 8018, January 2017.

[MMS+15] Max Jakob Maaß, Uwe Müller, Tom Schons, Daniel Wegemer, and Matthias Schulz. NFCGate - An NFC Relay Application for Android. In *Proceedings of the 8th ACM Conference on Security \& Privacy in Wireless and Mobile Networks*, WiSec, June 2015.

[NFC] NFC-Forum. What are the operating modes of NFC devices. `https://nfc-forum.org/resources/what-are-the-operating-modes-of-nfc-devices/`. [NFC-Forum. Accessed February 23, 2019].

[NFC11a] NFC-Forum. Simple NDEF Exchange Protocol. `https://nfc-forum.org/resources/what-are-the-operating-modes-of-nfc-devices/`, 2011.

[NFC11b] NFC-Forum. *Type 4 Tag Operation Specification*, 6 2011. NFCForum-TS-Type-4-Tag2.0.

[NFC14] NFC-Forum. What is the Purpose of NFC NDEF Signature Records? `https://nfc-forum.org/purpose-nfc-ndef-signature-records/`, 2014. [NFC-Forum. Accessed February 23, 2019].

[Pre11] Bart Preneel. *CBC-MAC and Variants*, pages 184–188. Springer US, Boston, MA, 2011.

[RLS11] M. Roland, J. Langer, and J. Scharinger. Security Vulnerabilities of the NDEF Signature Record Type. In *2011 Third International Workshop on Near Field Communication*, pages 65–70, Feb 2011.

[RLS13] M. Roland, J. Langer, and J. Scharinger. Applying relay attacks to Google Wallet. In *2013 5th International Workshop on Near Field Communication (NFC)*, pages 1–6, Feb 2013.

[RMF+15] O. Raso, P. Mlynek, R. Fujdiak, L. Pospichal, and P. Kubicek. Implementation of Elliptic Curve Diffie Hellman in ultra-low power microcontroller. In *2015 38th International Conference on Telecommunications and Signal Processing (TSP)*, pages 662–666, July 2015.

[Rol15] Michael Roland. *Security Issues in Mobile NFC Devices*. Springer International Publishing, 2015.

[RS04] Phillip Rogaway and Thomas Shrimpton. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In Bimal Roy and Willi Meier, editors, *Fast Software Encryption*, pages 371–388, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, feb 1978.

[Sch95] Bruce Schneier. *Applied Cryptography (2Nd Ed.): Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[SEC00] Recommended Elliptic Curve Domain Parameters. Standard, STANDARDS FOR EFFICIENT CRYPTOGRAPHY - Certicom Research, 2000.

[SGM+14] D. Sethia, D. Gupta, T. Mittal, U. Arora, and H. Saran. NFC based secure mobile healthcare system. In *2014 Sixth International Conference on Communication Systems and Networks (COMSNETS)*, pages 1–6, Jan 2014.

[SM10] David Salomon and Giovanni Motta. *Handbook of Data Compression*. Springer London, 2010.

[SW11] M. Q. Saeed and C. D. Walter. A Record Composition/Decomposition attack on the NDEF Signature Record Type Definition. In *2011 International Conference for Internet Technology and Secured Transactions*, pages 283–287, Dec 2011.

[TT10] Gerrit Tamm and Christoph Tribowski. *RFID-Technologie*, pages 9–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[UPH+17] T. Ulz, T. Pieber, A. Hller, S. Haas, and C. Steger. Secured and Easy-to-Use NFC-Based Device Configuration for the Internet of Things. *IEEE Journal of Radio Frequency Identification*, 1(1):75–84, March 2017.

[UPS+18] T. Ulz, T. Pieber, C. Steger, S. Haas, and R. Matischek. QSNFC: Quick and secured near field communication for the Internet of Things. In *2018 IEEE International Conference on RFID (RFID)*, pages 1–8, April 2018.

[Uri14] P. Urien. LLCPS: A new secure model for Internet of Things services based on the NFC P2P model. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–6, April 2014.

[Wät18] Dietmar Wätjen. *Der Galois-Counter-Modus (GCM)*, pages 255–262. Springer Fachmedien Wiesbaden, Wiesbaden, 2018.

[WHF03] D. Whiting, R. Housley, and N. Ferguson. Counter with CBC-MAC (CCM). Technical report, sep 2003.

[ZMH07] G. Zhou, H. Michalik, and L. Hinsenkamp. Efficient and High-Throughput Implementations of AES-GCM on FPGAs. In *2007 International Conference on Field-Programmable Technology*, pages 185–192, Dec 2007.