Sicherheit von NFC-basierten Anwendungen auf Smartphone Betriebssystemen

Bachelorarbeit

an der Hochschule Ulm
Fakultät Informatik
Studiengang Technische Informatik

Vorgelegt von
Stefan Cepcik

Oktober 2013

1. Gutachter: Prof. Dr. rer. nat. Markus Schäffter
2. Gutachter: Prof. Dr. rer. nat. Stefan Traub

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die im Literaturverzeichnis angegebenen Hilfsmittel verwendet habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht und mit genauer Quellenangabe dargelegt habe.

Ulm, den 4. Oktober 2013 Stefan Cepcik

Abstract

Modere Smartphones unterstützen Near Field Communication, NFC, d.h. die direkte und drahtlose Kommunikation mit anderen NFC Geräten und Tags. Dies bringt für den Anwender einen Gewinn an Bedienkomfort mit sich, da so eine Kommunikationsverbindung zu anderen Geräten alleine durch berühren der beiden Geräte hergestellt werden kann. Doch häufig bedeutet der Gewinn an Komfort gleichzeitig einen Verlust an Sicherheit. Somit werden durch NFC neue Angriffe möglich, etwa durch Social Engineering. Die vorliegende Arbeit beschreibt ein Konzept für einen sicheren Informationsaustausch unter Berücksichtigung der Gefährdung beim Einsatz von NFC als auch des Sicherheitsmodells des Smartphone Betriebssystems Windows Phone 8. Dieses Konzept sieht zunächst das Hardcoden des Public Key des Readers in der Smartphone App vor, damit dieser nicht über einen ungesicherten Kommunikationskanal übertragen werden muss. Als nächstes zeigt der Reader einen Code am Bildschirm an, den der Anwender in die App eingeben muss, damit der Public Key der App sicher an den Reader übertragen werden kann. Daraufhin werden die Diffie-Hellman-Teilgeheimnisse digital signiert und ausgetauscht. Anschließend erzeugen beide den geheimen Schlüssel und setzen die Kommunikation mit einer symmetrischen Verschlüsselung fort. Dieses Konzept wird in der Implementierung dieser Arbeit nahezu vollständig umgesetzt.

Danksagung

An dieser Stelle möchte ich mich zunächst bei Herrn Prof. Dr. Markus Schäffter bedanken. Ihr kontinuierliches Feedback, die häufigen Treffen und Diskussionen haben mir geholfen aus dieser Arbeit das zu machen, was daraus geworden ist. Wie Sie neulich selbst sagten, ging ich häufig mit mehr Ideen heim, als ich mit Fragen gekommen bin. Vielen Dank hierfür!

Ein Dankeschön geht ebenfalls an meinen Kommilitonen Michael Schreiber für die Unterstützung rund um das Betriebssystem Android.

Ein besonderer Dank geht auch an Erich Beer und Benjamin Schanzel für die investierte Zeit für das Korrekturlesen.

Danken möchte ich auch meinen Eltern an dieser Stelle, die mich während meiner Schulischen Laufbahn immer wieder darauf hingewiesen haben, wie wichtig Bildung ist. Wo ich heute ohne euren positiven Einfluss stehen würde, kann ich nicht sagen. Danke!

Inhaltsverzeichnis

1 Einleitung

Heutzutage wird in vielen modernen Smartphones mittlerweile standardmäßig *Near Field Communication* (NFC) als Hardwarekomponente verbaut. NFC als solches ist eine Technologie, die auf Basis von *Radio-Frequency Identification* (RFID) entwickelt wurde. Der Informationsaustausch bei NFC findet über kurze Distanzen statt. Typischerweise ist die Distanz zweier Geräte auf wenige Zentimeter beschränkt. Die vom NFC Standard unterstützen Datenübertragungsraten sind 106 kBit/s, 212 kBit/s und 424 kBit/s. Bedingt durch diese niedrigen Übertragungsraten eignet sich NFC nicht zum Übertragen vieler Informationen oder großer Dateien. Vielmehr kann NFC dazu genutzt werden, um z.B. eine URL zu einer Webseite zu übermitteln oder zum Austausch von komplexen Konfigurationsparametern um zwei Geräte miteinander zum Koppeln. Schon heute ist es möglich mit NFC-fähigen Smartphones beispielsweise den Austausch eines Bildes zu initiieren. NFC wird lediglich dazu genutzt, um die Bluetooth-Konfigurationsparameter auszutauschen und um den Datentransfer über Bluetooth anzustoßen. Dies ist für den Anwender ein Komfortgewinn, da dieser sich nicht mit der Bluetooth-Konfiguration seines Smartphones auseinandersetzten muss. Des Weiteren ist die Geste, zwei Geräte aneinander zu halten, sehr einfach für den Anwender erlernbar. Da NFC lediglich ein Interface ist, gibt es sehr viele mögliche Anwendungsfälle für NFC. Momentan geht der Trend in der Industrie jedoch dahin, z.B. Fahrscheine oder Kreditkarten in das Smartphone zu virtualisieren oder auch Geräte, wie z.B. den Bluetooth-Lautsprecher mit dem Smartphone, das Musik abspielt, zu koppeln.

Ziel dieser Arbeit ist es zunächst die Sicherheit von NFC auf unterster Ebene zu analysieren, um zu sehen welche Angriffe durchführbar sind. Als nächstes wird die Sicherheit von Smartphone Betriebssystemen als Plattform für NFC-basierte Anwendungen bewertet. Anschließend werden eine Desktop Anwendung und eine Smartphone App umgesetzt, um einen sicheren Kanal über NFC aufzubauen.

2 Grundlagen

In diesem Kapitel werden die relevanten NFC Standards und die grundlegende Funktionsweise von NFC erläutert. Um die Thematik nachvollziehen zu können, bedarf es eines bestimmten Grundwissens, die nachfolgende Referenzen adressieren: In [HB06] und [Abd11] werden die Communication Modes, die Operating Modes und die Sicherheit der unterschiedlichen Modulationsverfahren näher betrachtet. Zusätzlich wird in [Abd11] der Handshake bei NFC erläutert. In [Nok11] werden die zugehörigen Standards des NFC Protokollstacks sowie das NDEF Datenübertragungsformat erläutert.

2.1 Standards

In NFC gibt es unterschiedliche internationale Standards, die für Interoperabilität zwischen den verschiedenen Geräteherstellern sorgen sollen. Das NFC Forum, ein Konsortium das von den Unternehmen NXP Semiconductors, Sony und Nokia gegründet wurde, definiert NFC Standards, die zur Standardisierung an die *International Organization for Standardization* (ISO) freigegeben werden. Dadurch, dass die ISO mit der Normungsorganisation *International Electrotechnical Commission* (IEC) kooperiert, entstehen Standards, die unter anderem mit dem Kürzel *ISO/IEC* beginnen, statt nur mit *ISO*. Im Wesentlichen sind die Standards ISO/IEC 14443 [ISO14443] und ISO/IEC 18092 [ISO18092] relevant. ISO/IEC 18092 ist auch bekannt als NFCIP-1 und ECMA-340. Im weiteren Verlauf dieser Arbeit wird auf *NFCIP*-1 statt auf ISO/IEC 18092 verwiesen.

2.1.1 ISO/IEC 14443

ISO/IEC 14443 definiert eine Normenreihe, bestehend aus 4 Teilen, die den Aufbau und die Funktionswiese einer kontaktlosen Smartcard beschreiben. Eine kontaktlose Smartcard ist eine Chipkarte mit integriertem Schaltkreis. Die Teile 2 und 3 der Normenreihe definieren einen Protokollstack, welcher auf Niveau der Bitübertragungsschicht und der Sicherungsschicht des ISO/OSI Referenzmodells operiert. NFC nutzt diese Teile des Standards, um die grundlegende Kommunikation zwischen zwei NFC Geräten zu ermöglichen. Zusätzlich werden die Teile 2 und 3 von ISO/IEC 14443 in eine A und eine B Version unterteilt, die unterschiedliche Modulations- und Bitübertragungstechniken definieren. Die Bitübertragungsschicht (ISO/IEC 14443-2) beschreibt in der jeweiligen A oder B Version das eingesetzte Modulationsverfahren und die Bitenkodierung, die in Kapitel 2.2 näher erläutert werden. In der Sicherungsschicht (ISO/IEC 14443-3) wird das framing und low-level protocol für die Versionen A und B beschrieben. Auf der Transportschicht (ISO/IEC 14443-4) wird das *command protocol* definiert, welches *nicht* in eine A und B Version unterteilt wird. Der Aufbau des Protokollstacks wird in Abbildung 1 beschrieben.

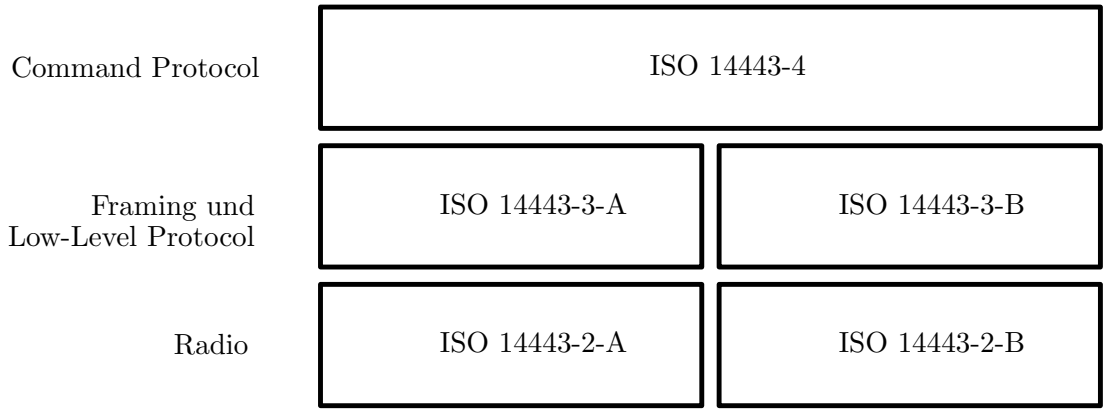

Abbildung 1: Aufbau des ISO/IEC 14443 Protokollstacks.

2.1.2 NFCIP-1

NFCIP-1 ist der wesentliche Standard, der es NFC Geräten ermöglicht, miteinander zu kommunizieren. Die Abkürzung NFCIP-1 steht für *Near Field Communication – Interface and Protocol*. Der NFCIP-1 Standard basiert auf den ISO/IEC 14443 Standard, welcher grundlegende Übertragungsfunktionalität bereitstellt. Beide Standards unterscheiden sich jedoch in dem Punkt, dass NFCIP-1 das *command protocol* (ISO/IEC 14443-4) ersetzt. Um dies zu veranschaulichen, ein Beispiel in Abbildung 2.

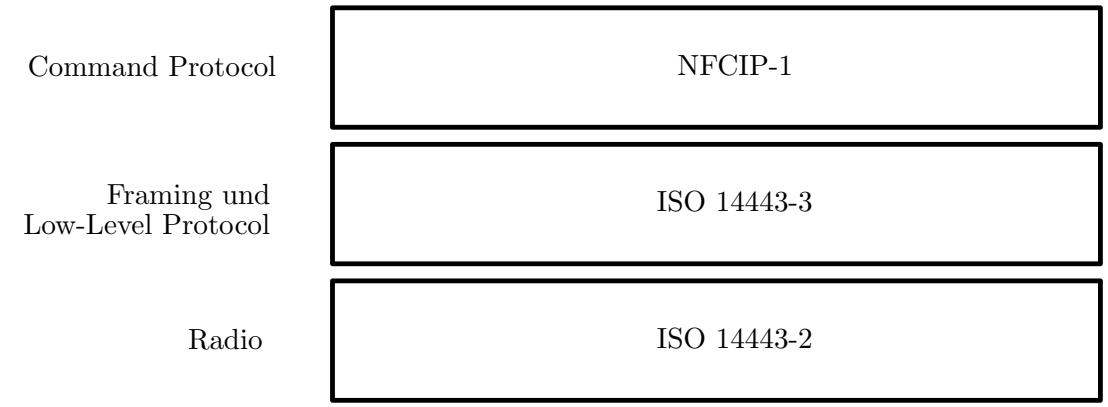

Abbildung 2: Aufbau des NFCIP-1 Protokollstacks.

Von NFCIP-1 werden zwei Communication Modes definiert, die in Kapitel 2.2 näher erläutert werden.

2.1.3 NDEF

Um Datenaustausch zwischen verschiedenen Herstellern und Geräten zu ermöglichen, bedarf es eines Datenformats, das herstellerunabhängig und interoperabel ist. Aus diesem Grund wurde das NDEF Datenformat vom NFC Forum spezifiziert. NDEF steht für *NFC*

Data Exchange Format. Die NDEF Spezifikation [NDEF] beschreibt ein Datenformat, welches es ermöglicht, beispielsweise URLs oder Klartextnachrichten an ein anderes NFC Gerät zu übermitteln. Der Aufbau einer sogenannten *NDEF message* ist wie folgt:

- Eine NDEF message besteht aus *n Records*.
- Ein Record besteht aus einem *Header* und einem *Payload*.
- Ein Header besteht aus *Identifier*, *Length* und *Type*.

In Abbildung 3, wird der Aufbau verdeutlicht.

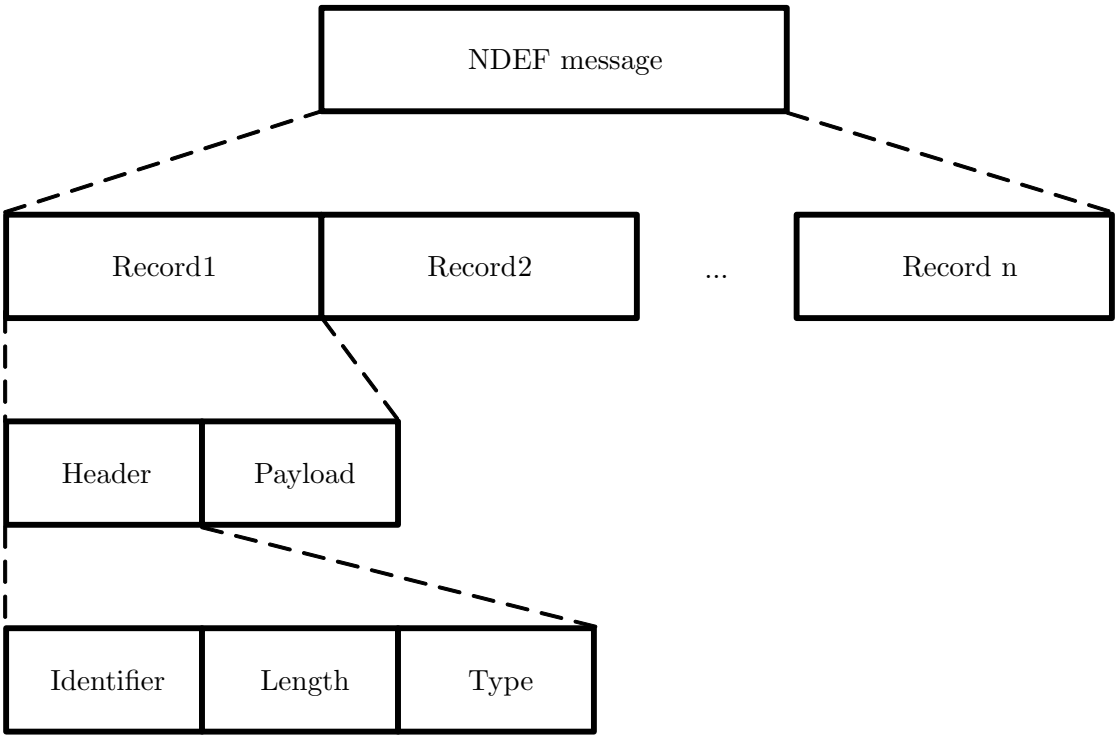

Abbildung 3: Aufbau einer NDEF message.

Die NDEF message repräsentiert die komplette Nachricht. In einem Record wird der verwendete Datentyp spezifiziert, auch bekannt als *Record Type Definition* (RTD). Ein Record wird in Header und Payload (Nutzdaten) unterteilt. Im Header des Records wiederum ist mit Type definiert, um welche RTD es sich handelt. Im Feld Length wird die Länge des Payloads in Byte angegeben. Das *optionale* Feld Identifier erlaubt es Anwendungen den Payload eindeutig zu identifizieren.

Die vom NFC Forum spezifizierten RTDs sind der Tabelle 1 zu entnehmen.

Record Type Definition	**Beschreibung**
Text	Eine Klartext Nachricht.
URI	Eine URI, z.B. zu einer Webseite.
Smart Poster	Typischerweise eine URI mit einem Titel als beschreibenden Text.
Signature	Enthält eine digitale Signatur zu einem oder mehreren Records in der NDEF message.

Tabelle 1: Übersicht der spezifizierten RTDs vom NFC Forum.

Beispielsweise ist es möglich, dass in einer NDEF message mehrere Records sequentiell aufeinander folgen. Das *Smart Poster* ist ein komplexerer Datentyp, der im Payload eine NDEF message enthält. Im Payload des Smart Posters muss sich der URI Datentyp befinden. Die anderen möglichen Datentypen (Title, Action, Icon, Size und Type) im Payload des Smart Poster sind optional. Typischerweise wird zu der URI im Payload der *Titel* Datentyp als beschreibender Text verwendet. Jedoch ist es durch fehlerhafte Software-Implementierungen, wie Collin Mulliner in [Mul08] gezeigt hat, möglich durch geschickte Wahl eines Titels die Modifikationen an der URI durch den Angreifer zu verschleiern. Bei modernen Smartphone-Betriebssystemen wie z.B. Windows Phone 8 oder Android 4.3 existiert diese Problematik jedoch nicht, da Smart Poster *nur* als URIs behandelt werden und die verbleibenden Records im Payload des Smart Posters ignoriert werden. Dennoch existiert unter Android 4.3 eine andere Problematik: Apps können durch ein URI-Schema ohne Nachfragen beim Anwender geöffnet werden, wie in Kapitel 5.2 beschrieben wird.

2.2 Communication Modes

Die Kommunikation in NFC erfolgt immer nur zwischen zwei Geräten. Das NFC Gerät das die Kommunikation startet, wird *initiator* genannt. Das Zielgerät, mit dem der initiator kommunizieren möchte, wird *target* genannt. Die möglichen communication modes, *active mode* und *passive mode*, beschreiben, ob ein NFC Gerät selber das RF Feld erzeugt, respektive das RF Feld nutzt, das von einem anderen NFC Gerät erzeugt wurde. Daher haben NFC Geräte im *active mode* typischerweise eine eigene Energieversorgung, während NFC Geräte im *passive mode* die Energie aus dem generierten RF Feld beziehen. NFC Geräte im *passive mode* werden auch als *Tag* bezeichnet, insofern diese nur aus einem Speicherbaustein und einer Antenne, als einziges Interface, bestehen. Die möglichen Übertragungsraten, die ein *initiator* wählen kann, beschränken sich auf 106 kBit/s, 212 kBit/s und 424 kBit/s. In Abbildung 4 wird ein gängiges Szenario dargestellt, bei dem ein Smartphone im *active mode* einen Tag mit Energie durch das generierte RF Feld versorgt.

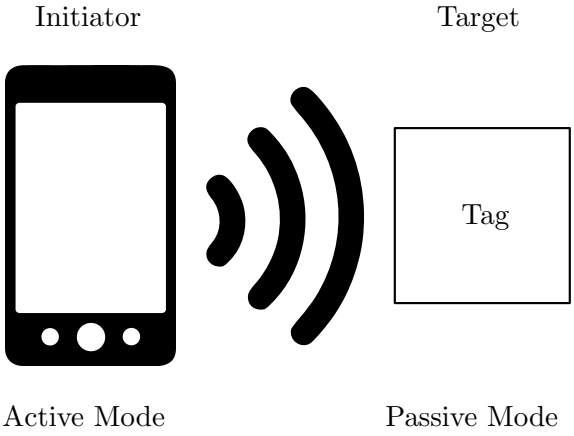

Abbildung 4: Ein Smartphone liest einen Tag aus.

Dadurch dass ein NFC Gerät sich im *active mode* oder *passive mode* befinden kann, entstehen drei mögliche Kombinationen, die in Tabelle 2 gezeigt werden.

Gerät A	**Gerät B**	**Beschreibung**
active mode	passive mode	NFC Gerät A generiert das RF Feld. NFC Gerät B bezieht die Energie aus dem generierten RF Feld.
passive mode	active mode	NFC Gerät B generiert das RF Feld. NFC Gerät A bezieht die Energie aus dem generierten RF Feld.
active mode	active mode	NFC Gerät A und B wechseln sich mit dem Generieren des RF Feldes ab. D.h. wenn NFC Gerät A im active mode ist, *muss* sich NFC Gerät B im passive mode befinden und umgekehrt.

Tabelle 2: Mögliche Kombinationen wie NFC Geräte kommunizieren.

Befindet sich in ein NFC Gerät dauerhaft im *active mode* und das andere NFC Gerät ebenso im *passive mode*, so handelt es sich hierbei um einen *passive communication mode*. Wechseln sich jedoch die NFC Geräte in ihren communication modes ab, so kommunizieren beide Geräte im *active communication mode*.

2.2.1 Active Mode

Im *active mode* wird zur Übertragung von Daten das *amplitude shift keying* (ASK) Verfahren verwendet. D.h. das 13,56 MHz RF Feld wird mit den Daten anhand eines Kodierungsschemas moduliert und die Daten werden übertragen. Je nach Übertragungsgeschwindigkeit, wird ein anderes Kodierungsschema verwendet.

Bei einer Übertragungsrate von 106 kBit/s wird das *modifizierte Miller* Kodierungsschema angewendet mit 100% Modulation. D.h., dass die Sendeleistung bei 100% ist, wenn Symbole übertragen werden, und die Sendeleistung bei 0% ist, wenn keine Sym-

bole zum Kodieren von Bits übertragen werden. Ein zu übertragendes Bit wird in zwei sog. *half bits* unterteilt. Abhängig davon, ob im *ersten* half bit kein Signal gesendet wird (Pause) und im *zweiten* half bit ein Signal gesendet wird und umgekehrt, wird entsprechend eine 0 oder 1 kodiert. Tabelle 3 zeigt, welche Symbole zum Kodieren einer 0 oder 1 gesendet werden müssen.

Bit	**Symbole**	
	1. half bit	**2. half bit**
0	Pause	Senden
1	Senden	Pause

Tabelle 3: Verwendetes Kodierungsschema.

Mögliche Bitfolgen, die auftreten können, wären 00, 01, 10 und 11. Mit der Bitfolge 10 in der Millerkodierung ist es daher möglich, dass *zwei* Pausen aufeinander folgen. Um dies zu vermeiden, wurde die *modifizierte Millerkodierung* eingeführt. Die modifizierte Millerkodierung vermeidet diesen Fall durch folgende Regel: Sobald nach einer 1 eine 0 folgt, wird das *erste* half bit des Bits 0 mit *Senden* statt einer Pause kodiert. Somit werden Verdopplungen bei Pausen vermieden. In Tabelle 4 werden die Unterschiede zwischen der *Millerkodierung* und der *modifizierten Millerkodierung* für die Bitfolge 10 gezeigt.

Bitfolge	**Millerkodierung**				**Modifizierte Millerkodierung**			
00	Pause	Senden	Pause	Senden	Pause	Senden	Pause	Senden
01	Pause	Senden	Pause	Senden	Pause	Senden	Senden	Pause
10	Senden	Pause	*Pause*	Senden	Senden	Pause	*Senden*	Senden
10	Senden	Pause	Senden	Pause	Senden	Pause	Senden	Pause

Tabelle 4: Gegenüberstellung der Millerkodierung und modifizierten Millerkodierung.

In Kapitel 3 wird die Sicherheit der *modifizierten Millerkodierung* bei einer Übertragungsgeschwindigkeit von 106 kBit/s bewertet.

Bei Übertragungsgeschwindigkeiten von 212 kBit/s und 424 kBit/s wird die *Manchesterkodierung* angewendet mit 10% Modulation. Die Funktionsweise der Manchesterkodierung in NFC ist recht einfach, wie es in [Poo] beschrieben wird. Der Übergang von *Pause* nach *Senden* ist im ersten, respektive zweiten half bit kodiert eine 0. Der umgekehrte Fall kodiert entsprechend eine 1. Daher lässt sich das in Tabelle 3 beschriebene Kodierungsschema auch auf die Manchesterkodierung anwenden. Dabei werden keine Verdopplungen der Pause vermieden, wie es in der *modifizierten Millerkodierung* der Fall ist. Ein Beispiel zur Manchesterkodierung in Abbildung 5.

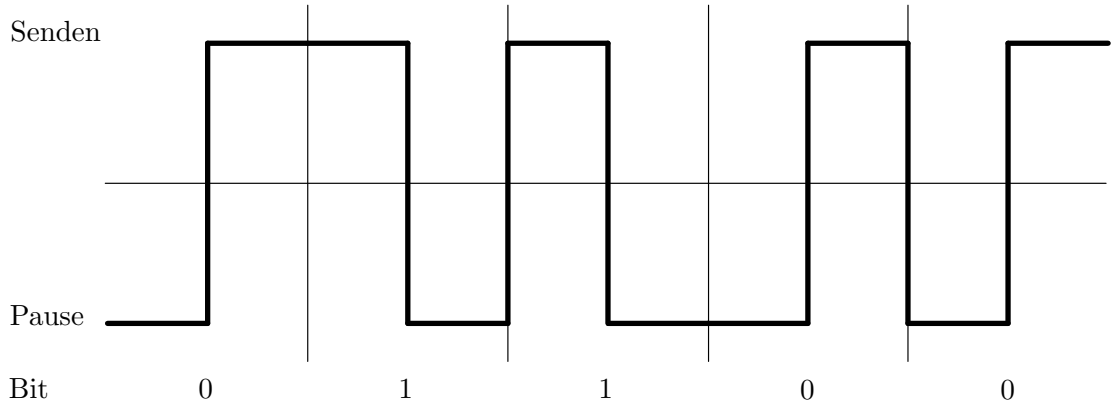

Abbildung 5: Die Bitfolge 01100 mit der Manchesterkodierung kodiert.

Bedingt dadurch, dass die *Manchesterkodierung* mit 10% Modulation angewendet wird, ist die Sendeleistung bei 100%, wenn Symbole übertragen werden und wenn keine Symbole übertragen werden, bei 82% Sendeleistung. Diese Eigenschaft, dass die Sendeleistung bei 82% ist, wenn keine Symbole übertragen werden, ist sicherheitstechnisch relevant. Dieser Aspekt wird näher betrachtet in Kapitel 3. Im Gegensatz dazu ist bei einer Übertragungsgeschwindigkeit von 106 kBit/s die Sendeleistung bei 0% und somit wird kein RF Feld generiert, wenn keine Symbole übertragen werden.

2.2.2 Passive Mode

Im Gegensatz zum *active mode* wird im *passive mode* für alle Übertragungsgeschwindigkeiten ASK mit 10% Modulation verwendet. Dabei wird als Kodierungsschema die *Manchesterkodierung* verwendet. Bei Übertragungsraten von 106 kBit/s wird die Unterträgerfrequenz genutzt zur Modulation, bei Übertragungsraten höher als 106 kBit/s wird das RF Feld bei 13,56 MHz moduliert.

2.3 Handshake

NFC basiert auf einem Message-Reply Konzept. Ein *initiator* sendet eine Nachricht an ein target, das sich im RF Feld vom initiator befindet. Das *target* moduliert das von initiator generierte RF Feld, um die Nachricht dem initiator zu übermitteln. Es ist nicht möglich, dass ein *target* einem beliebig *anderen* initiator eine Antwort zuschickt, ohne dass zuvor dieser initiator eine Verbindung zu dem *target* aufgebaut hat. Ein Verbindungsaufbau läuft wie folgt ab: Der *initiator* generiert ein RF Feld und sendet ein *Kommando* an das *target*. Sobald das *target* genügend Energie hat, antwortet es mit seiner ID als *acknowledge*. Anhand der ID des targets ist der *initiator* in der Lage zu erkennen, welcher Tag sich im RF Feld befindet. In Abbildung 6 wird der Ablauf verdeutlicht.

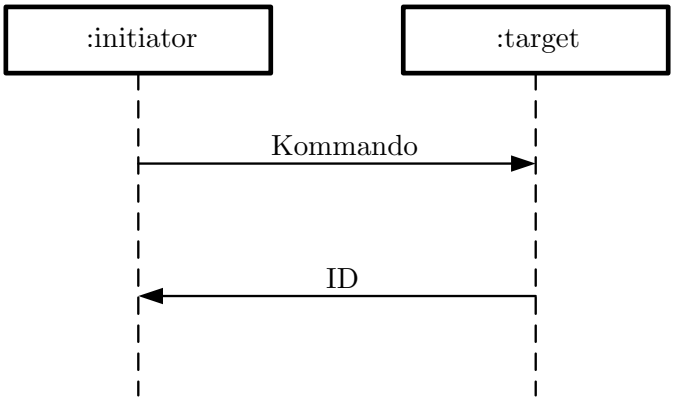

Abbildung 6: Handshake zwischen initiator und target.

Nach dem Aufbau der Kommunikation zwischen initiator und target wartet das target auf weitere Befehle vom initiator. Die möglichen Befehle beschränken sich auf das Lesen und Schreiben von Daten durch die Befehle *read* bzw. *write*, als auch *sleep* zum Beenden der Kommunikation. Um Daten auszulesen, wird ein *read* Befehl an das target gesendet, das den gewünschten Speicherblock an Daten als Antwort an den initiator sendet. Um Daten zu schreiben wird ein *write* Befehl mit den Daten und der Zielspeicheradresse an das target gesendet, dass die Daten in den gewünschten Speicherblock schreibt und die geschriebenen Daten als Antwort an den initiator zur Verifikation der Daten zurück sendet. In Abbildung 7 wird der Ablauf des Lese- und Schreibvorgangs zwischen initiator und target beschrieben.

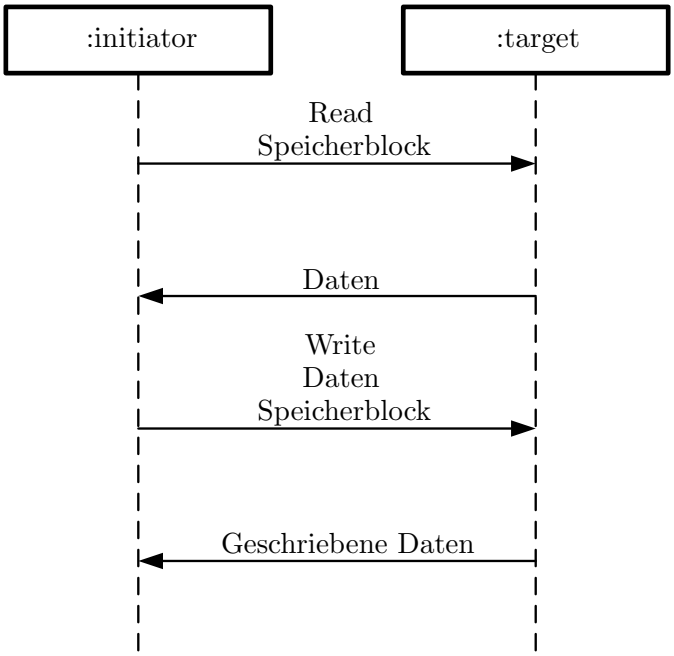

Abbildung 7: Initiator sendet ein read und ein write Kommando an das target.

Nach einem Schreibvorgang sollten die vom target empfangen Daten, die zur Verifikation gesendet wurden, überprüft werden, ob diese identisch mit den gesendeten Daten sind. Sind die gesendeten als auch die empfangenen Daten in jeder Bitposition identisch, kann davon ausgegangen werden, dass alles ordnungsgemäß abgelaufen ist.

2.4 Operating Modes

Mithilfe der operating modes in NFC wird es ermöglicht, unterschiedliche Anwendungsfälle zu realisieren. Daher wird zwischen drei operating modes, *Reader/Writer*, *Peer-To-Peer* und *Card Emulation*, unterschieden.

2.4.1 Reader/Writer

Ein NFC Gerät, das sich im *Reader/Writer Mode* befindet, kann einen Tag auslesen bzw. beschreiben. Beispielsweise ist ein Tag NDEF formatiert und speichert eine URL. Das NFC Gerät generiert als initiator ein RF Feld und erzeugt die benötigte Energie, die der Tag benötigt, damit die URL vom Tag übermittelt werden kann.

2.4.2 Peer-To-Peer

Zwei NFC Geräte, die Daten austauschen wollen, befinden sich im *Peer-To-Peer Mode*. Bedingt durch die niedrige Übertragungsgeschwindigkeit von NFC eignet sich NFC nicht zum Übertragen von großen Datenmengen. Vielmehr wird NFC dazu genutzt um komplexe Konfigurationsparameter auszutauschen. Um eine Peer-To-Peer Verbindung

zu ermöglichen, müssen sich NFC Geräte im *active communication mode* befinden. Dadurch wird bedingt, dass sich beide NFC Geräte mit der initiator-Rolle abwechseln. Beispielsweise werden beim Versenden eines Bildes an ein anderes NFC Gerät Bluetooth-Konfigurationsparameter ausgetauscht, um das Bild über eine Bluetooth Verbindung zu transferieren.

2.4.3 Card Emulation

Im *Card Emulation Mode* präsentiert sich das NFC Gerät als ein *Tag* oder eine *kontaktlose Smartcard* einem *anderen* NFC Gerät, das sich im Reader/Writer Mode befindet. Ein möglicher Anwendungsfall für den Card Emulation Mode ist das *Virtualisieren* kontaktloser Smartcards, wie z.B. eines Studentenausweises. Ein virtueller Studentenausweis auf dem Smartphone würde dem Lesegerät in der Mensa wie ein physikalischer Studentenauswies erscheinen. Der Einsatz vom Card Emulation Mode erfordert ein *Secure Element*, welches ein Hardwaremodul in einem Smartphone ist. Mit einem Secure Element können kryptografische Operationen und die sichere Speicherung von vertrauenswürdigen Informationen durchgeführt werden [Zef12]. Um auf ein Secure Element zugreifen zu können, bedarf es an spezieller Berechtigung für die entsprechende Betriebssystem API. Smartphone Betriebssystemhersteller wie Microsoft und Google verweigern unautorisierten Zugriff auf die Secure Element APIs. Auf das Secure Element wird im Weiteren nicht näher eingegangen.

3 Sicherheitsanalyse von NFC

In diesem Kapitel wird die Sicherheit von NFC auf unterster Ebene betrachtet. Die möglichen Angriffstypen werden beschrieben und Lösungsansätze werden präsentiert um die Sicherheitsanfälligkeit zu reduzieren. Die möglichen Angriffstypen, die diskutiert werden, sind *Eavesdropping* (a), *Data Corruption* (b), *Data Modification* (c), *Data Insertion* (d) und *Man-In-The-Middle* (e). Die einzelnen Angriffstypen lassen sich mittels Piktogrammen beschreiben, wie Abbildung 8 zeigt.

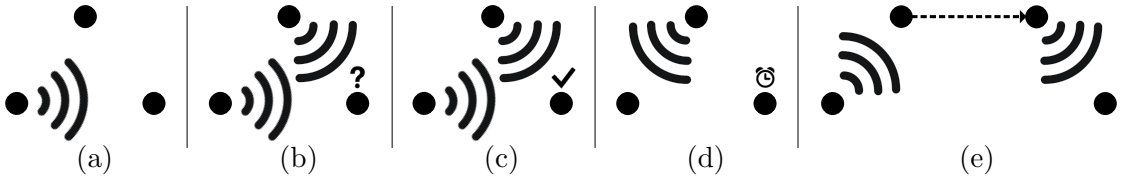

Abbildung 8: Piktogramme beschreiben die unterschiedlichen Angriffstypen.

3.1 Eavesdropping

Da NFC eine Technologie ist, die über drahtlose Kommunikation abläuft, ist es offensichtlich, dass die Kommunikation prinzipiell von einem Angreifer abgehört werden kann. NFC Geräte kommunizieren über kurze Distanzen, typischerweise weniger als 10 cm. Tests im Labor ergaben mit Smartphones wie dem Nokia Lumia 925 und einem Tag eine etwaige Distanz von 3,0cm. Bei Smartphones, wie dem LG Nexus 4, beträgt die Distanz zwischen dem Smartphone und dem Tag etwa 2,5cm. Anhand dieser Angaben ist ersichtlich, dass die mögliche Distanz zwischen zwei NFC Geräten variiert und demzufolge unter anderem auch hardwareabhängig ist. Die Frage, wie weit entfernt ein Angreifer sein kann, hängt von vielen Faktoren ab, wie sie in [HB06] beschrieben werden:

- RF Feld Charakteristiken vom Gerät, das sendet,
- Charakteristiken der Antenne des Angreifers,
- Qualität des Receivers des Angreifers,
- Qualität des Signal-Decoders des Angreifers,
- Ort, an dem der Angriff durchgeführt wird,
- Energie, die vom NFC Gerät ausgestrahlt wird.

Bedingt durch diese vielen Parameter kamen E. Haselsteiner und K. Breitfuß zu der Erkenntnis, dass keine verbindliche Aussage getroffen werden kann, wie nah der Angreifer sein muss, vor allem da auch der eingesetzte communication mode ein entscheidender Faktor ist. NFC Geräte, die im *active mode* operieren, sind wesentlich leichter abzuhören als NFC Geräte, die im *passive mode* betrieben werden. Grund hierfür ist,

dass je nach communication mode die Daten *unterschiedlich* übermittelt werden. Auch wenn es schwierig ist hier eine allgemein gültige Aussage zu treffen, wie nah der Angreifer sein muss, sind die Unterschiede etwa wie folgt: Ein Angreifer kann ein NFC Gerät, das im *active mode* sendet, bei einer Distanz von bis zu 10m abhören. Bei einem NFC Gerät, das im *passive mode* seine Antwort an den initiator übermittelt, kann immer noch bei einer Distanz von bis zu 1m abgehört werden. Da die Kommunikation zwischen NFC Geräten unverschlüsselt abläuft, muss manuell auf Applikations-Ebene ein sicherer Kanal aufgebaut werden um die Vertraulichkeit der Daten zu wahren. In Kapitel 7 wird gezeigt, wie ein sicherer Kanal aufgebaut werden kann, mithilfe von asymmetrischer Verschlüsselungsverfahren durch Private Keys und Public Keys, des Diffie-Hellman Schlüsselaustauschverfahren und der symmetrischen Verschlüsselung AES.

3.2 Data Corruption

In dieser Form eines Angriffs beschreiben E. Haselsteiner und K. Breitfuß in [HB06], wie die Kommunikation zweier NFC Geräte derart gestört wird, dass der *Receiver* eines NFC Geräts nicht versteht, was das andere NFC Gerät gesendet hat. Diese Art von Datenkorruption gleicht einem Denail of Service (DoS) Angriff. Im Wesentlichen muss der Angreifer zulässige Frequenzen vom Datenspektrum zum richtigen Zeitpunkt übermitteln, damit die empfangenen Daten für den Receiver nicht verständlich sind. Solch ein Angriff ist erkennbar, wenn das antwortende NFC Gerät das RF Feld während der Übertragung auch überprüft. Das kann dadurch begründet werden, dass die benötigte Energie vom Angreifer signifikant höher ist und somit auch erkennbar ist. Bei solch einem Angriff kann als Konsequenz die Verbindung vom Sender beendet werden.

3.3 Data Modification

In diesem Angriffsszenario versucht ein Angreifer die übertragenen Daten so zu manipulieren, dass diese für den Empfänger nach wie vor valide und verständlich sind. In [HB06] wird beschrieben, dass die Durchführbarkeit dieses Angriffs stark davon abhängt, ob ASK mit 100% Modulation oder ASK mit 10% Modulation zum Einsatz kommt. Der Angriff besteht lediglich darin, dass der Decoder statt einer 1 eine 0 dekodiert und um gekehrt.

Bei ASK mit 100% Modulation kommt die *modifizierte Millerkodierung* zum Einsatz. Wenn Daten gesendet werden, ist die Sendeleistung bei 100%. Wenn keine Daten gesendet werden (Pause), wird kein RF Feld generiert und somit ist die Sendeleistung bei 0%. Um ein Bit von 0 auf 1 zu ändern und umgekehrt, muss der Angreifer in der Lage sein das RF Feld bei *beiden* half bits zu manipulieren. Beispielsweise möchte ein Angreifer ein Bit von 0 auf 1 ändern. Das Bit 0 wird sowohl in der *modifizierten Millerkodierung* als auch in der *Millerkodierung* mit den beiden half bits *Pause — Senden* kodiert, wie bereits in Tabelle 3 gezeigt wurde. Um das Bit von 0 auf 1 zu kippen, müsste der Angreifer die *Pause* im *ersten* half bit mit einem RF Feld der Sendeleistung 100% auffüllen. Dies ist soweit durchführbar. Als nächstes muss aber das *zweite* half bit eine *Pause* sein, damit der Decoder das Bit als eine 1 interpretiert. D.h. der Angreifer müsste das RF

Feld perfekt überlappen, damit der Decoder dies als eine Pause interpretiert. E. Haselsteiner und K. Breitfuß kommen zu dem Entschluss, dass dies so gut wie unmöglich ist. *Dennoch* ist es möglich, ein Bit von 1 auf 0 zu kippen. Bedingt durch die *modifizierte Millerkodierung* gibt es den Sonderfall, wenn eine 1 von einer 0 gefolgt wird, dass das Bit 0 mit den beiden half bits *Senden — Senden* kodiert wird. Daher ergibt sich die Möglichkeit für Angreifer, bei einer Bitfolge 11 die *Pause* des zweiten Bits mit einem RF Feld aufzufüllen. Dadurch, dass der Decoder im zweiten Bit *keine* Pause erkennt, wird dieses Bit als 0 kodiert, da davor 1 kodiert wurde. Um dies zu verdeutlichen, ein Beispiel in Abbildung 9.

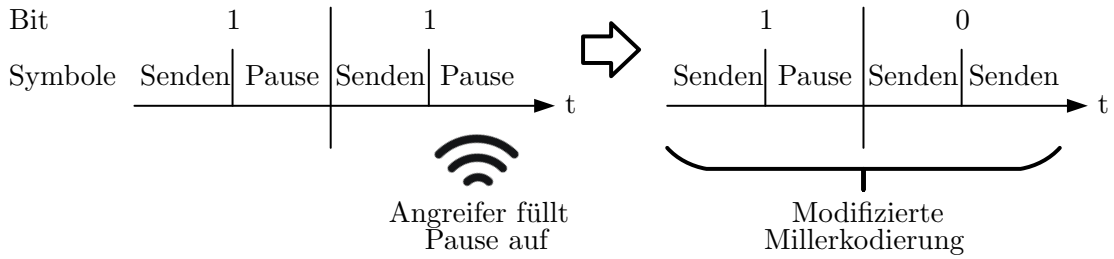

Abbildung 9: Das zweite Bit der Bitfolge 11 lässt sich auf 0 zu kippen.

Daher ist es bei ASK mit 100% Modulation *nur* möglich Bits von 1 nach 0 zu kippen und nicht umgekehrt, unter der Bedingung das vor dem zu kippenden Bit eine 1 kodiert ist.

Bei ASK mit 10% Modulation kommt die *Manchesterkodierung* zum Einsatz. Wenn Daten gesendet werden, ist die Sendeleistung bei 100%. Werden keine Daten gesendet, wird nach wie vor ein RF Feld generiert, allerdings mit der geminderten Sendeleistung von 82%. Wenn ein Angreifer ein RF Feld mit der Sendeleistung von 82% auffüllt um auf 100% Sendeleistung zu kommen, wird auch gleichzeitig ein RF Feld mit 100% Sendeleistung dadurch auf 82% Sendeleistung abgeschwächt. Ein Beispiel in Abbildung 10.

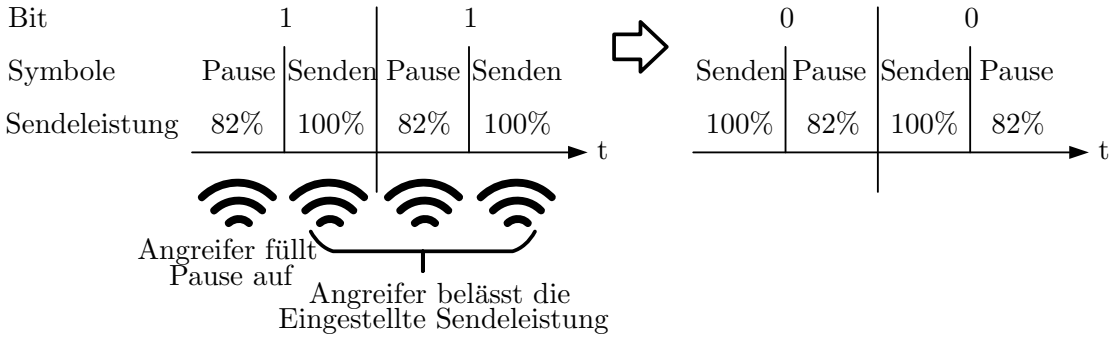

Abbildung 10: Alle Bits können gezielt gekippt werden.

Dadurch kann der Angreifer gezielt *alle* Bits manipulieren, die für den Decoder valide erscheinen.

E. Haselsteiner und K. Breitfuß kommen zu dem Entschluss, dass bei ASK mit 100%

Modulation nur *bestimmte* Bits kompromittiert werden können und bei ASK mit 10% Modulation *alle* Bits. Wie auch bei *Data Corruption* aus Kapitel 3.2 empfiehlt es sich das RF Feld während der Übertragung zu überprüfen auf einen möglichen Angriff. Des Weiteren bietet es sich an eine Verschlüsselung anzuwenden. Doch eine Verschlüsselung alleine wahrt zwar nur die Vertraulichkeit der Daten, aber nicht die Integrität. Daher sollte vor dem Verschlüsseln eine Checksumme an das Ende der Daten angehängt werden um zu verhindern, dass ein Angreifer Blöcke des verschlüsselten Datenstroms tauscht.

3.4 Data Insertion

Bei diesem Angriff nutzt der Angreifer die verzögerten Antwortzeiten eines NFC Geräts aus. In [HB06] beschreiben E. Haselsteiner und K. Breitfuß, wie ein Angreifer eine Nachricht in die Kommunikation zweier NFC Geräte einschleust. Wenn zwei NFC Geräte kommunizieren und eines davon eine längere Antwortzeit hat, kann der Angreifer dies ausnutzen, indem der Angreifer *vor* dem legitimen NFC Gerät antwortet. Sollte das legitime NFC Gerät antworten, *während* der Angreifer seine Nachricht übermittelt, so überlappen sich beide RF Felder, was korrupte Daten für den Receiver zur Folge hat. Ein möglicher Ansatz gegen diesen Angriff wäre z.B. möglichst zeitnah zu antworten ohne Verzögerung. So kann verhindert werden, dass der Angreifer die Verzögerung ausnutzt um schneller als das legitime NFC Gerät zu antworten. Wie aber auch bei *Data Modification* in Kapitel 3.3, empfiehlt es sich einen sicheren Kanal aufbauen mit einer Checksumme der Daten und Verschlüsselung.

3.5 Man-In-The-Middle

Bei einem *Man-In-The-Middle Angriff* kommunizieren zwei Parteien, typischerweise auch als *Alice* und *Bob* bezeichnet, über einen dritten, ohne das sie es wissen. Dabei ist der Angreifer dieser dritte und je nachdem, ob es sich um einen *aktiven* oder *passiven* Man-In-The-Middle Angriff handelt, wird der Angreifer als *Mallory* respektive *Eve* bezeichnet. E. Haselsteiner und K. Breitfuß beschreiben in [HB06] einen *aktiven* Man-In-The-Middle Angriff, bei dem Mallory in der Lage ist die Kommunikation zwischen Alice und Bob abzuhören *und* zu manipulieren. Um einen *aktiven* Man-In-The-Middle Angriff auf NFC anzuwenden, ist es zunächst relevant, wie beide NFC Geräte miteinander kommunizieren. Es wird davon ausgegangen, dass ein Man-In-The-Middle Angriff nicht auf Tags vollzogen wird, da Tags nur statische Inhalte ausliefern und sich daher nicht von einem Angreifer beeinflussen lassen.

Der *initiator* befindet sich im *Reader/Writer Mode* und generiert deshalb ein RF Feld, da er eine Verbindung zum *target* aufbauen möchte. Der Angreifer bemerkt dies und generiert selber ein RF Feld um die Daten zu korrumpieren, damit das *target* keine validen Daten erhält. An dieser Stelle ist es möglich, dass der *initiator* einen Angriff bemerkt und den Verbindungsaufbau beendet. Aber unter der Annahme, dass der *initiator* dies nicht prüft, wird der Verbindungsaufbau fortgesetzt. Im nächsten Schritt müsste der Angreifer eine Verbindung zum *target* aufbauen. Dies ist insofern ein Problem, weil der *initiator* nach wie vor das RF Feld generiert, da das *target* im *passive mode* arbeitet. Aus

diesem Grund ist es in der Praxis nahezu unmöglich die zwei RF Felder so abzustimmen, dass der Angreifer erfolgreich eine Verbindung zum *target* aufbauen kann.

Befinden sich stattdessen beide NFC Geräte im *Peer-To-Peer Mode*, scheint die Lage zunächst besser, da sich beide *NFC Geräte* abwechseln mit dem Generieren des RF Felds. Der *initiator* möchte eine Nachricht an das *target* senden, indem er ein RF Feld generiert. Der Angreifer korrumpiert erneut die Daten, damit das *target* keine validen Daten erhält. Auch diesmal könnte der *initiator* den Angriff erkennen, was dieser jedoch in diesem Beispiel nicht macht. Nachdem der *initiator* annimmt, dass die Nachricht angekommen ist, beendet der *initiator* sein RF Feld und wartet auf eine Antwort vom *target*. An diesem Punkt kann der Angreifer eine Nachricht an das *target* senden. Nur erwartet der *initiator* eine Antwort vom *target*. Stattdessen wird dieser aber die Nachricht vom Angreifer erhalten, die an das *target* gerichtet ist. Daher könnte der *initiator* an dieser Stelle merken, dass ein Angriff stattgefunden hat, und die ganze Kommunikation beenden. Für den Angreifer ist es nicht möglich, dass RF Feld so auszurichten, dass nur das *target* die Nachricht erhält.

Bedingt durch die oben genannten Gründe ist es nicht möglich einen *aktiven* Man-In-The-Middle Angriff auf unterster Ebene bei NFC durchzuführen. E. Haselsteiner und K. Breitfuß empfehlen den *passive communication mode*, da nur so sichergestellt werden kann, dass immer ein RF Feld aktiv ist und dies einen möglichen Angriff eines Angreifers vereitelt. Doch wie sieht die Situation auf höherer Ebene aus? In Kapitel 5.3 wird die Android App NFCProxy vorgestellt, die zeigt, dass ein Relay Angriff, eine Form eines Man-In-The-Middle Angriffs, möglich ist. In Kapitel 6 wird gezeigt, welche Maßnahmen ergriffen werden müssen, um sich vor einem Man-In-The-Middle Angriff auf höherer Ebene zu schützen und um erfolgreich einen sicheren Kanal aufzubauen.

4 Smartphone Betriebssysteme

In diesem Kapitel werden die Sicherheitsfeatures von Windows Phone 8 und Android 4.3 näher beschrieben, um zu zeigen, welche Möglichkeiten sich für Entwickler bzw. auch Angreifer ergeben. Des Weiteren wird das App Store Konzept beschrieben als primäre Bezugsquelle von Apps und welche Möglichkeiten zur Verfügung stehen, um Apps außerhalb der App Stores zu installieren. Im weiteren Verlauf wird auch gezeigt, welche Möglichkeiten zur App-zu-App-Kommunikation bestehen und wie sich beide Betriebssysteme in der NFC Standardunterstützung unterscheiden. Als letzter Punkt wird an einem Beispiel gezeigt, wie eine App NFC-Inhalte empfangen kann.

4.1 Sicherheitsmodell

Smartphone Betriebssysteme wie Windows Phone 8 und Android 4.3 besitzen ein striktes Sicherheitsmodell, welches Apps voneinander isoliert und erfordert, dass, verwendete Systemfunktionen von Apps explizit deklariert werden müssen. Durch Isolation von Apps kann verhindert werden, dass eine App A die Daten von einer anderen App B manipuliert. Durch Deklaration von Systemfunktionen können der App vom Betriebssystem Rechte eingeräumt werden, um diese Systemfunktion, wie z.B. den Zugriff auf die aktuelle GPS-Position, nutzen zu dürfen. Dadurch, dass eine App nur die nötigsten Rechte anfordert, kann zusätzlich die Angriffsfläche minimiert werden, da einer kompromittierten App unter Umständen die benötigten Rechte fehlen um merkbaren Schaden anzurichten.

Das Smartphone Betriebssystem Windows Phone 8 basiert auf dem NT-Kernel der von Microsoft auf die ARM CPU-Architektur portiert wurde. Bisher wurde der NT-Kernel primär in den Betriebssystemen Windows und Windows Server verwendet. Durch den Tausch des Kernels von Windows Phone 7, welcher auf Windows Embedded Compact basiert, auf den NT-Kernel, ergeben sich unter anderem sicherheitstechnische Vorteile, da der NT-Kernel viele Sicherheitsfeatures bereitstellt, wie z.B. Adress Space Layout Randomization (ASLR), Data Execution Prevenion (DEP) und UEFI mit Secure Boot. Das Sicherheitsmodell basiert auf der *Trusted Computing Base* (TCB), welches von A. S. Tanenbaum in [Tan09] auf S. 732-733 beschrieben wird. Daher ist in Windows Phone 8 *Chamber* die verwendete Terminologie für die Sandbox, in der sich die jeweilige isolierte App befindet. Microsoft unterscheidet in [Her13] zwischen zwei Chambers:

- Trusted Computing Base und
- Least Privilege Chamber.

In der TCB befindet sich der NT-Kernel, im *Least Privilege Chamber* (LPC) Betriebssystemkomponenten, Treiber und Apps. Windows Phone 8 ermöglicht es Entwicklern für Ihre Komponenten *Capabilities* zu definieren, die beschreiben, welche Rechte die Komponente benötigt. Der TCB ist ein *Fixed-Rights-Chamber*, d.h. die eingestellte Berechtigung in diesem Chamber ist fest vorgegeben. Da der NT-Kernel in der TCB ausgeführt wird, wird angenommen, dass der Kernel im Kontext des SYSTEM-Kontos

operiert, wie unter Windows und Windows Server. Im LPC werden die Capabilities *dynamisch*, abhängig von App und Herausgeber, verteilt. Dabei werden die Capabilities in drei Gruppen unterteilt, wie auf Folie 8 in [EC13] gezeigt wird: *Entwickler*, *Original-Equipment-Manufacturer* (OEM) und *System*. In der Gruppe *System* werden Capabilties definiert, die unter Umständen sicherheitstechnisch bedenklich und deshalb *nur* für Microsoft-Programmierer bestimmt sind. Betriebssystemkomponenten und Treiber werden typischerweise von diesen Capabilities Gebrauch machen. *OEMs* werden ebenso spezielle Capabilities bereitgestellt um Treiber als auch herstellerspezifische Apps zu entwickeln. In der letzten Gruppe *Entwickler* befinden sich Capabilities, die häufig verwendet werden von App-Entwicklern, die zugleich nicht die Sicherheit des Systems beeinflussen können, um einen merkbaren Schaden anzurichten. In Abbildung 11 wird das Sicherheitsmodell schematisch dargestellt.

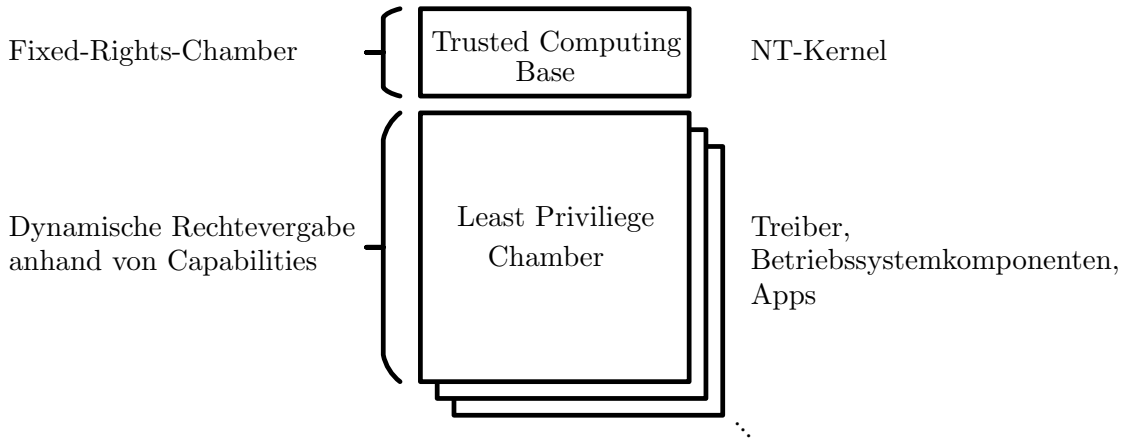

Abbildung 11: TCB und LPC von Windows Phone 8.

Durch das Installieren einer App aus dem Windows Phone Store wird die WMAppManifest.xml Datei vom Paket Manager daraufhin überprüft, welche Capabilities dort definiert wurden. Anhand dieser Capabilities wird für die App ein *neuer* Chamber eingerichtet mit nur den benötigten Rechten. Nachträglich lassen sich die Rechte eines Chambers nicht ändern. Das typische Angriffsziel eines Angreifers wird die TCB sein. Die Vergangenheit hat gezeigt, dass Fehler in OEM Treiber zur *Privilege Escalation* (Rechteerhöhung) eines Angreifers führen kann, wie in [WP7] dokumentiert ist. Bedingt dadurch, dass Treiber im LPC laufen und nur dem Treiber die nötigsten Rechte gegeben werden, wird die Angriffsfläche auf den TCB deutlich reduziert. Wie oben erwähnt, werden auch Betriebssystemkomponenten im LPC ausgeführt. Das Ziel, das sich Microsoft gesetzt hat, ist so viele Komponenten wie möglich im LPC auszuführen um die Angriffsfläche auf den TCB zu minimieren. Sollte z.B. eine System-App kompromittiert werden, muss dies nicht zwangsweise weitreichende Folgen haben, da die App nur die *nötigsten* Capabilities definiert hat, die die App auch benötigt. Des Weiteren müssen alle Binärdateien von Microsoft digital signiert sein, um ausgeführt werden zu können. Dieses sog. *Code Signing* (Codesignierung) verhindert das Modifizieren von z.B. Systemdateien

um ein anderes Verhalten herbeizurufen als auch das Einschleusen von Apps, die nicht aus dem Windows Phone Store stammen. In [Tan09] wird *Code Signing* in Kapitel 9.8.3 Codesignierung, S. 809-811, in den einleitenden Worten, als ein Ansatz *nur Software von zuverlässigen Anbietern laufen zu lassen*, beschrieben. Eine Ausnahme besteht jedoch für Smartphones, die entsperrt wurden als *Entwicklergerät*. Dazu mehr in Kapitel 4.2.

In Android 4.3 ist das Sicherheitsmodell dem von Windows Phone 8 sehr ähnlich. Die Terminologie für die isolierten Apps bei Android ist *Application Sandbox*. Wie auf Windows Phone 8, muss eine App unter Android 4.3 ihre benötigten Rechte deklarieren. Die Rechte unter Android werden als *Permissions* bezeichnet. Sicherheitstechnisch bedenkliche Permissions werden wie bei Windows Phone 8 auch nur bestimmten Benutzergruppen zur Verfügung gestellt, allerdings ist Google da *nicht* so restriktiv wie Microsoft. Vorteil dieses Vorgehens ist, dass Entwicklern eine mächtige API zur Verfügung steht, mit der vieles umgesetzt werden kann. Nachteil wiederum ist, dass dies Angreifer auch ausnutzen können. Ein Beispiel hierfür ist die SMS API. Unter Windows Phone 8 hat die Gruppe *Entwickler* keinen Zugriff auf die SMS API, wie in [EC13] auf Folie 8 präsentiert wird. Die Folge ist, dass nur Microsoft und ggf. Partner diese API verwenden können. Unter Android kann ein Entwickler die Permission deklarieren, um auf die SMS API zugreifen zu können. In Kombination durch Installation von Apps aus Fremdquellen bzw. Apps, die nicht über Google Play Store bezogen werden, wird ein Angriff ermöglicht, wie es z.B. jüngst *Eurograbber* gezeigt hat. Bei Eurograbber handelt sich um einen Trojaner, der gezielt Rechner und *Smartphones* infiziert um das Online-Banking eines Anwenders zu manipulieren und an das Smartphone gesendete mTANs zu entwenden, wie in [KB12] beschrieben wird. In Kapitel 5.1 wird gezeigt, welches Sicherheitsrisiko durch Installation von Apps aus Fremdquellen entstehen kann. Aber auch im Weiteren ähneln sich die Konzepte des Sicherheitsmodells von Windows Phone 8 und Android 4.3. Auch unter Android 4.3 existiert ein *Code Signing*. Das Modell unter Android 4.3 unterscheidet sich insofern, dass Apps vom Entwickler selbstsigniert werden und nicht von Google, wie in [Goo], im Abschnitt *Application Signing*, erklärt wird. Daher können auch bei Android 4.3 nur Apps ausgeführt werden, die über eine digitale Signatur verfügen. Wird eine App aus dem Google Play Store heruntergeladen und installiert, so wird anhand der deklarierten *Permissions* in der AndroidManifest.xml Datei ein neues Linux Benutzerkonto angelegt mit einer zufälligen User ID (UID), der Abschnitt *UserIDs and File Access* in [Goo] erklärt. In Abbildung 12 wird dies veranschaulicht.

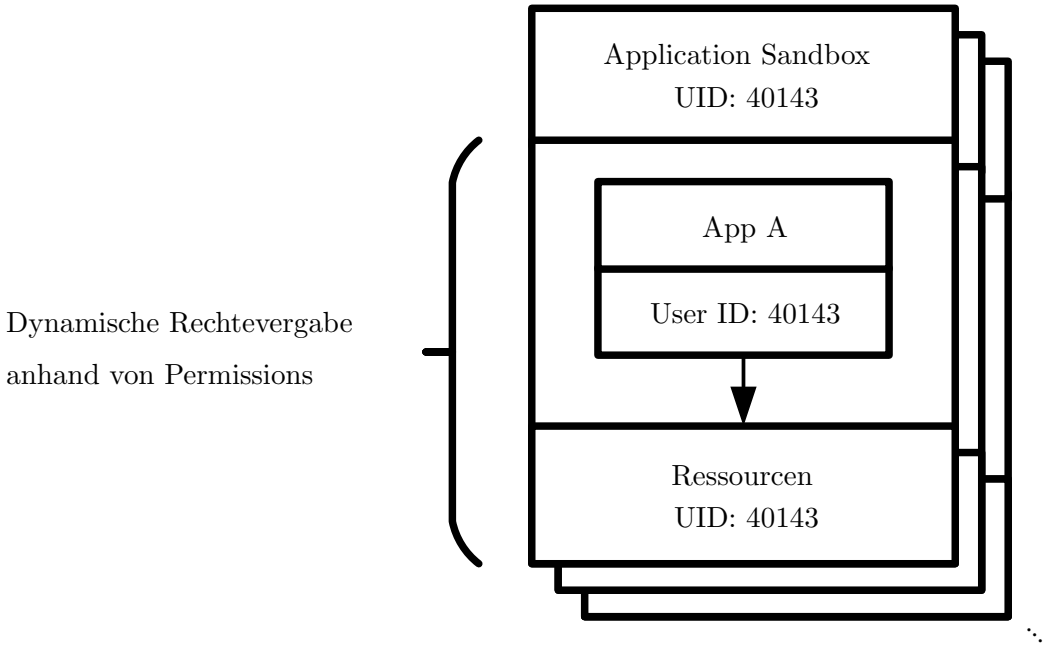

Abbildung 12: Application Sandbox in Android 4.3.

Sollte jedoch ein Herausgeber zwei Apps in *derselben* Application Sandbox ausführen wollen, ermöglicht Android 4.3 durch Festlegen des Wertes `sharedUserId` in der AndroidManifest.xml. Voraussetzung hierfür ist jedoch, dass beide Apps mit *demselben* Private Key signiert wurden. Mit neueren Versionen hat Google immer mehr Permissions eingeführt. Die Problematik, die dabei entsteht, ist jedoch, dass alte Apps, die beispielsweise bei einer alten Android Version für die entsprechende API *keine* Permission benötigt haben, in der neuen Android-Version eine solche benötigen. Dies würde typischerweise dafür sorgen, dass die alte App nicht mit der neuen Android-Version kompatibel ist. Um dem entgegenzuwirken, kann Android die `targetSdkVersion` der App abfragen und automatisch die fehlenden Permissions zur Application Sandbox hinzuzufügen, wie im Abschnitt *Using Premissions* in [Goo] dokumentiert ist. Dies geschieht jedoch nur bei Apps, deren `targetSdkVersion` 3 *oder* niedriger ist. Unter Windows Phone 8 ist das nicht möglich. Des Weiteren unterscheidet sich die *Application Sandbox* von Android 4.3 zum *Chamber* von Windows Phone 8 so, dass es möglich ist, dass sich Apps eines Herausgebers eine *Application Sandbox* teilen können. Dies hat den Vorteil, dass die Apps z.B. auf gemeinsame Dateisystem-Ressourcen zugreifen können. Unter Windows Phone 8 ist das nur bedingt möglich: Eine App kann einen *Background Agent* haben. Dieser Background Agent ist separater Code, der zyklisch aufgerufen wird und an eine App gebunden ist. D.h. die App und der Background Agent teilen sich ein gemeinsames Dateisystem, das zum Informationsaustausch genutzt werden kann. In Kapitel 4.3 werden mögliche Wege zur App-zu-App-Kommunikation auf Smartphones beschrieben. Weitere Unterschiede zwischen Windows Phone 8 und Android 4.3 bestehen auf Trei-

berebene. Treiber werden in Android im Kernelmodus ausgeführt und sind nicht wie in Windows Phone 8 *teilweise* im LPC ausgelagert und mit nur den nötigsten Capabilities ausgestattet. Daher kann eine Schwachstelle in einem OEM-Treiber, welcher im Kernelmodus ausgeführt wird, das gesamte System kompromittieren.

4.2 Bezugsquellen von Apps

Smartphone Betriebssysteme wie Windows Phone 8 und Android 4.3 haben eine zentrale Anlaufstelle für Apps, häufig auch allgemein als *App Store* bezeichnet. Somit wird es für Anwender einfach gemacht neue Apps aus einer sicheren und verlässlichen Quelle zu beziehen. Anwender können mithilfe von Rezessionen und Bewertungen in Betracht ziehen, ob die App ihren Anforderungen genügt. Des Weiteren bieten App Stores einen Update Mechanismus an, sodass der Anwender eine Benachrichtigung bekommt, dass eine neuere Version zur Verfügung steht. Dennoch ist es möglich, Apps aus Fremdquellen zu installieren, was ein potentielles Sicherheitsrisiko darstellt.

In Windows Phone 8 verfolgt Microsoft den Ansatz, dass Apps nur aus dem *Windows Phone Store* bezogen werden können. D.h. die meisten Kunden, die eine App installieren wollen, können diese *nur* aus dem Windows Phone Store beziehen. Eine Ausnahme stellen jedoch Entwickler dar. Nach einer Registrierung bei Microsoft als Entwickler kann dieser sein Smartphone als *Entwicklergerät* freischalten lassen, so dass Windows Phone 8 es zulässt Apps auf das Smartphone zu installieren, *ohne* dass diese Apps im Windows Phone Store veröffentlicht sein müssen. Dieses sog. *Sideloaden* von Apps ist auf maximal 10 Apps pro Smartphone beschränkt. Dies gilt jedoch nur für registrierte Entwickler, was zur Folge hat, dass der Großteil der Windows Phone Anwender nach wie vor auf den Windows Phone Store angewiesen sind. Wenn ein Entwickler eine App im Windows Phone Store veröffentlichen möchte, muss dieser die App von Microsoft zertifizieren lassen. Die App wird auf Schadsoftware, unerlaubte-API Aufrufe, unerlaubte Capabilities usw. geprüft. Besteht die App diese Tests, wird diese von Microsoft zertifiziert und im Windows Phone Store veröffentlicht.

In Android 4.3 ist die primäre Bezugsquelle von Apps der *Google Play Store*. Anwender sind jedoch *nicht* auf den Google Play Store als einzige Bezugsquelle angewiesen, sondern können in den Einstellungen die Installation von Apps aus Fremdquellen erlauben. Auch Entwickler müssen nicht ihr Smartphone von Google als Entwicklergerät freischalten lassen, sondern können dies manuell in den Einstellungen vornehmen. Daher unterscheidet sich das Modell von Google zu dem von Microsoft in der Weise, dass jeder Anwender in der Lage ist Apps aus fremden Quellen, die also *nicht* aus dem Google Play Store stammen, zu installieren. Die Probleme, die dabei entstehen können, werden in Kapitel 5 näher erläutert. Aber auch bei der Zulassung von Apps im Google Play Store unterscheidet sich Google von Microsoft. Neben dem Einhalten der obligatorischen Nutzungsbedingen, die in beiden App Stores gelten, wird im Google Play Store vom Entwickler lediglich verlangt, dass die App von ihm selbstsigniert ist, während im Windows Phone Store die App vor der Zertifizierung und Freigabe zuerst auf Konformität geprüft wird. Des Weiteren sind die App-Pakete im Google Play Store standardmäßig nicht verschlüsselt, während die App-Pakete im Windows Phone Store automatisch verschlüsselt

werden. Um App-Pakete zu verschlüsseln, muss der Entwickler dies explizit vorsehen.

4.3 App-zu-App-Kommunikation

Je nach Smartphone Betriebssystem variieren die Möglichkeiten für Entwickler die Kommunikation zwischen Apps zu implementieren.

In Windows Phone 8 hat Microsoft die Möglichkeiten von App-zu-App-Kommunikation stark eingeschränkt. Apps können ein URI-Schema registrieren, dass, beim Aufruf des URI-Schemas die App startet. Sollten mehrere Apps das gleiche URI-Schema registrieren, bekommt der Anwender eine Liste mit möglichen Apps zur Auswahl, die das URI-Schema registrieren. Unter Windows Phone 8 ist es nicht möglich, dass eine Drittanbieter-App für ein URI-Schema sich selber als Standard-App einträgt, wie es z.B. unter Windows und Windows Server möglich ist. D.h. sobald *mehr* als eine App dasselbe URI-Schema registrieren, muss der Anwender eine Entscheidung treffen, *welche* App geöffnet werden soll. Alternativ zum Registrieren eines URI-Schemas kann auch eine Datei mit Informationen im LocalFolder, dem Dateisystem der App, abgelegt werden. Dies jedoch ist insofern beschränkt, da nur der zugehörige *Background Agent* mit der App einen LocalFolder teilen kann um somit Ressourcen auszutauschen.

Unter Android 4.3 ist das, was unter Windows Phone 8 möglich ist, ebenso möglich. Nur sind die Möglichkeiten weniger beschränkt. Realisiert wird das unter Android 4.3 mithilfe von *Intents*. Eine App kann einen *Intent-Filter* festlegen, um zu definieren, für welche Inhalte sie sich interessiert. Eine App, die Inhalte erzeugt, kann anderen Apps, die sich für den Inhalt interessieren, diesen übergeben, damit diese ihn *weiterverarbeiten*. Um ein Beispiel zu nennen: Angenommen zwei Apps sind installiert, die das URI-Schema „tresor:“ registrieren. Der Anwender liest mit seinem Android-basierten Smartphone einen Tag aus, auf dem eine NDEF message mit dem genannten URI-Schema gespeichert ist. Der NFC-Dienst in Android 4.3 empfängt die NDEF message und stellt fest, dass zwei Apps das URI-Schema „tresor:“ registriert haben. In diesem Fall wird der Anwender vor die Wahl gestellt, welche App die NDEF message *weiterverarbeiten* soll. Unter Windows Phone 8 sind bestimmte URI-Schemata für das Betriebssystem reserviert, sodass eine App das URI-Schema nicht für sich beanspruchen kann. Unter Android 4.3 gibt es solche Einschränkungen nicht. Daher ist es beispielsweise auch möglich, dass eine App das HTTP URI-Schema implementiert. Des Weiteren bietet Android 4.3 *Remote Procedure Calls* (RPC) bei einem *gebundenen* Service an. D.h. eine App, die mit einem Service verknüpft ist, der im Hintergrund unabhängig arbeitet, kann über RPC Methoden dieses Service aufrufen.

4.4 Unterstützte NFC Standards

Standards werden von den Smartphone Betriebssystemherstellern implementiert. Je nachdem was das Unternehmen unterstützen möchte und wie viele Ressourcen in die Standardunterstützung von NFC investiert werden, unterscheidet sich die Unterstützung von NFC Standards in den Smartphone Betriebssystemen Windows Phone 8 und Android 4.3. In [JR13] wird eine Tabelle angezeigt, die darstellt, welche proprietären Features

als auch NFC Standards unterstützt werden. Ein Beispiel wäre das *Logical Link Control Protocol* (LLCP), das auf NFCIP-1 aufbaut. Mit LLCP soll es vereinfacht werden eine verbindungsorientierte bzw. verbindungslose Verbindung zweier NFC Geräte aufzubauen. Weder Microsoft, noch Google bieten in ihren Betriebssystemen eine API für Entwickler an. Daher ist es derzeit nicht möglich größere Datenmengen, wie z.B. ein Bild, zwischen einem Windows Phone 8 Smartphone und einem Android 4.3 Smartphone über NFC auszutauschen. Kleinere Daten, wie z.B. der Link zu einer Webseite, funktionieren, da beide *Simple NDEF Exchange Protocol* (SNEP) implementiert haben. Da NFC keinen NDEF Standard anbietet um gezielt eine App auf dem Smartphone zu öffnen, haben Microsoft als auch Google entsprechend einen proprietären Record in ihre Plattformen implementiert. Unter Windows Phone 8 bietet Microsoft den *LaunchApp* Record an, um gezielt eine App auf dem Smartphone zu öffnen oder aus dem Windows Phone Store herunterzuladen, falls die App nicht installiert ist. Unter Android 4.3 hat Google einen *Android Application Record* (AAR) implementiert. Das Verhalten vom AAR auf der Android-Plattform ist identisch zu dem von LaunchApp auf der Windows Phone-Plattform. Es gibt Möglichkeiten LaunchApp und AAR in einem Tag unterzubringen, wie es in [Jak12] beschrieben ist. Dennoch sollte auf interoperable Techniken gesetzt werden, da weder *LaunchApp* noch *AAR* standardisiert sind. Daher bietet sich an, stattdessen ein *URI Schema* zu verwenden, das die App auf dem jeweiligen Smartphone Betriebssystem startet.

4.5 NFC-Inhalte in einer App empfangen

Es gibt unterschiedliche Wege um NFC-Inhalte in einer App zu empfangen. Zunächst wird vorausgesetzt, dass der Sperrbildschirm *nicht* aktiv ist und der Bildschirm eingeschaltet ist, damit NFC-Inhalte empfangen werden können. Ein möglicher Weg, um NFC-Inhalte in einer App zu empfangen, wäre z.B., dass die App bereits offen und bereit ist NFC-Inhalte zu empfangen durch z.B. einen Tag. Ein anderer denkbarer Weg wäre z.B. durch Registrieren eines URI-Schemas in einer App. Wenn das Smartphone mit z.B. einem Tag in Kontakt kommt, der das URI-Schema überträgt, wird der Anwender gefragt, ob er die zugehörige App öffnen möchte, die das URI-Schema registriert. Falls keine App installiert ist, die das URI-Schema registriert, wird der Anwender gefragt, ob er im App Store nach einer passenden App suchen möchte. Des Weiteren kann es passieren, dass mehrere Apps installiert sind, die das URI-Schema registrieren. In diesem Fall, wird der Anwender gefragt, welche App geöffnet werden soll. In Abbildung 13 werden beide Möglichkeiten übersichtlich dargestellt.

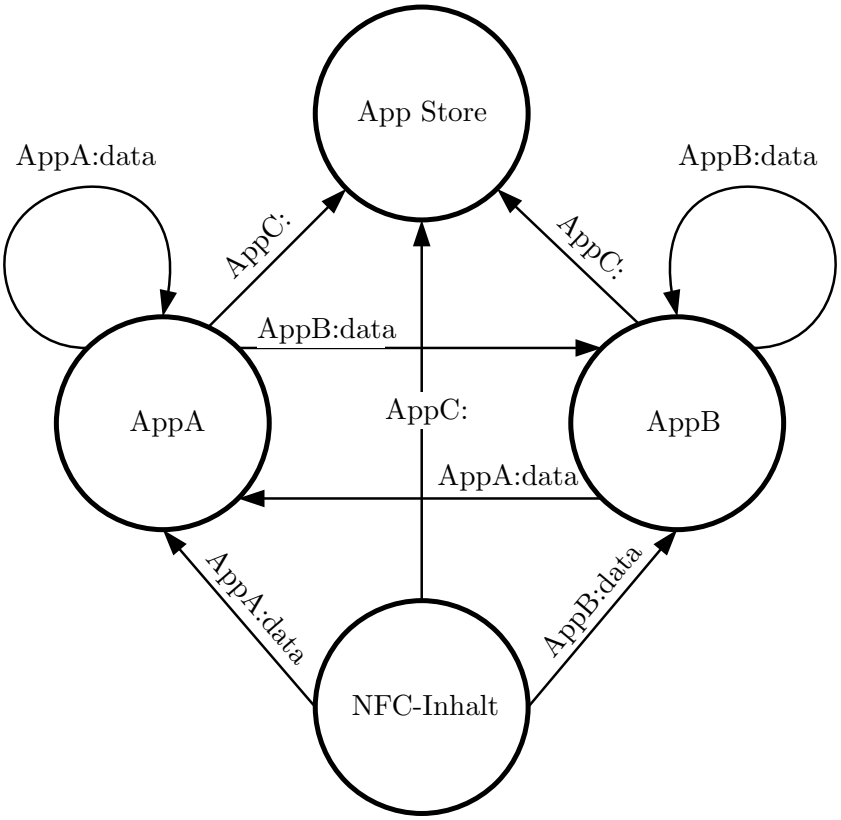

Abbildung 13: Unterschiedliche Möglichkeiten Apps zu starten über URI-Schemata.

AppA und *AppB* registrieren die Protokolle „AppA:“ bzw. „App2:“. Empfängt das Betriebssystem über NFC „AppA:data“, so wird der Anwender gefragt, ob er „AppA“ öffnen möchte, je nach verwendetem Betriebssystem. Ist keine App installiert, die das URI-Schema „AppA:“ registriert, so wird im App Store nach Apps gesucht, die das Protokoll „AppA:“ registrieren. *AppA* kann sowohl *AppB* durch Ausführen des URI-Schemas „AppB:data“ starten, als auch sich selber ggf. neustarten, indem es „AppA:data“ ausführt. Dies gilt analog für AppB. Beim Erhalt von „AppC:“ über NFC oder durch Ausführen des URI-Schemas durch die Apps *AppA* und *AppB*, wird umgehend im App Store nach einer App gesucht, die das besagte URI-Schema registriert.

In Kapitel 4.3 wurden bereits die Möglichkeiten erläutert, wie Apps untereinander kommunizieren können. Die Kommunikation über Protokolle ist einfach, plattformunabhängig, birgt aber einige Gefahren, wenn sensible Daten übertragen werden und keine Sicherheitsvorkehrungen getroffen werden. Im nachfolgenden Kapitel 5.1 wird eine naive Implementierung demonstriert, bei der vertrauenswürdige Informationen als Parameter über ein URI-Schema übertragen werden.

5 Sicherheitsschwachstelle: Social Engineering

Da Smartphone Betriebssysteme wie Windows Phone 8 und Android 4.3 robuste Schutzmechanismen anbieten, wie z.B. Isolation von Apps, ist es schwierig für Angreifer diese zu umgehen. Sollte dennoch ein Angreifer dies schaffen, helfen eine *Defense-in-Depth-Strategie*, wie es in [Tan09], S. 800ff., beschrieben ist, die Auswirkungen des Angriffs zu begrenzen. Statt nach Fehlern in den unteren Schichten von NFC oder in Betriebssystemkomponenten zu suchen, wird vielmehr auf höherer Ebene versucht das Verhalten von Menschen zu beeinflussen. Dieser Angriff wird auch als *Social Engineering* bezeichnet [SMK01]. Auch beim *Eurograbber* Angriff, der in Kapitel 4.1 bereits kurz angesprochen wurde, wird Social Engineering angewendet, um den Anwender zu täuschen. In [KB12] wird der grobe Ablauf von Eurograbber wie folgt beschrieben: Ein Anwender infiziert seinen Computer mit einem Trojaner. Beim nächsten Login für das Onlinebanking, wird der Trojaner aktiv. Der Trojaner modifiziert die Webseite und zeigt dem Anwender eine Meldung an zum Durchführen eines „security upgrades“. An dieser Stelle wird der Anwender darum gebeten seine Handynummer einzugeben und sein Smartphone Betriebssystem auszuwählen. Nach Eingabe der Daten, erhält der Anwender eine SMS auf seinem Smartphone. In der SMS wird der Anwender gebeten eine verlinkte App herunterzuladen und zu installieren um das „security upgrade“ abzuschließen. Parallel dazu wird der Trojaner am Computer aktiv und zeigt eine Nachricht, in der der Anwender aufgefordert wird, seinen Aktivierungscode einzugeben. Nachdem die App installiert und gestartet wurde, zeigt diese den geforderten Aktivierungscode an. An dieser Stelle sind die Angreifer in der Lage, Geld vom Konto der betroffenen Person unbemerkt auf andere Konten zu überweisen. In Kapitel 4 wurden die Schutzmechanismen von Smartphone Betriebssystemen vorgestellt. Bei Eurograbber wird das Vertrauen vom Anwender in die Bank missbraucht. Dies beginnt bei der Eingabe der Handynummer und der Wahl des Smartphone Betriebssystems. Beim Erhalt der SMS wird der Anwender aufgefordert eine App für das Android Betriebssystem außerhalb des Google Play Stores zu installieren. Durch das Vertrauen in die Bank ignorieren die Anwender typischerweise die vom Android Betriebssystem angezeigte Warnung, die vor der Installation von Apps aus Fremdquellen warnt.

Die Analogie zu NFC besteht darin, dass eingehende Informationen entwendet werden können. Bei Eurograbber operiert ein Trojaner im Hintergrund auf dem Smartphone und wartet darauf, bis eine mTAN SMS ankommt. Bei Ankunft einer mTAN SMS versendet der Trojaner diese weiter zu den Servern des Angreifers und löscht die SMS anschließend, ohne dass der Anwender etwas davon mitbekommt. NFC ist aber ein anderer Kommunikationskanal als SMS. Daher ist typischerweise die App, die NFC-Inhalte empfangen soll, bereits offen oder sie registriert ein URI-Schema, das beim Empfangen von NFC-Inhalten mit dem URI-Schema das Öffnen der App auslöst. In Abbildung 14 wird dies veranschaulicht.

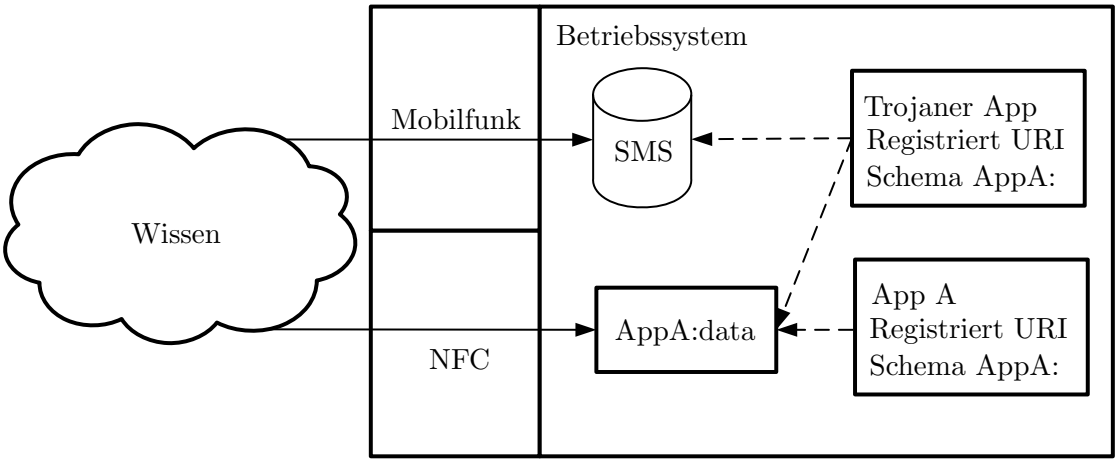

Abbildung 14: Eine Trojaner App, die SMS bzw. NFC-Inhalte abfängt.

5.1 URI-Schema-Übernahme durch Schadsoftware

Die erste Nachricht, die zwei NFC Geräte typischerweise austauschen, ist ein URI-Schema zum Öffnen der Anwendung über das empfangen URI-Schema und wird dann je nach Anwendungsfall im *Peer-To-Peer Mode* fortgesetzt. Als Beispiel können eine *Tresor App* und eine *Schlüssel App* genommen werden. Die *Schlüssel App* enthält den geheimen Schlüssel, der über NFC zur *Tresor App* übermittelt wird, um den Tresor zu öffnen. D.h. beim Berühren beider Smartphones übermittelt die *Schlüssel App* über NFC eine NDEF message mit einem *URI Record* an das andere Smartphone. Im *URI Record* befindet sich die URI „tresor:schluessel=Student123“. Beim Empfang der NDEF message sucht das Betriebssystem anhand des URI-Schemas „tresor:“ nach Apps, die das URI-Schema registriert haben. Unter der Annahme, dass die installierte *Tresor App* die einzige ist, die das URI-Schema registriert hat, wird bei Windows Phone 8 gefragt, ob der Anwender es zulassen möchte, dass die Tresor App geöffnet werden soll. Unter Android 4.3 hingegen wird ohne Einverständnis des Anwenders die App geöffnet. Beim Öffnen der App, werden die Parameter nach dem Doppelpunkt in der URI an die App weitergegeben. So ist die App in der Lage mithilfe des Werts des Parameters *schluessel* den Tresor zu entschlüsseln um dem Anwender die entschlüsselten Inhalte zu präsentieren. In Abbildung 15 wird der Ablauf verdeutlicht.

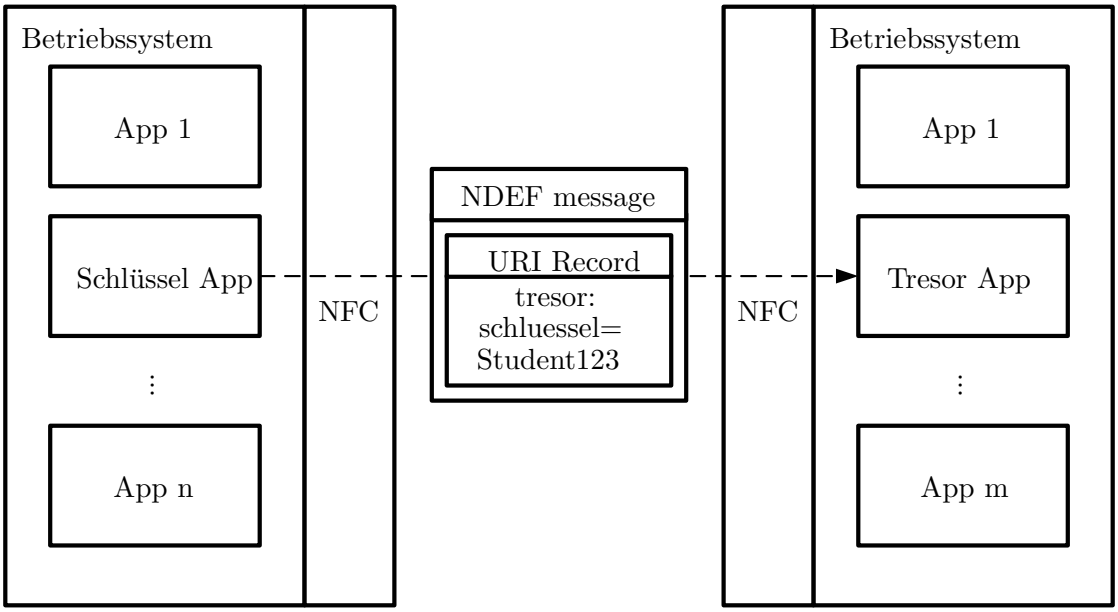

Abbildung 15: Übermitteln von Daten über ein URI-Schema mithilfe von NFC.

Doch was passiert, wenn eine weitere App das URI-Schema „tresor:“ registriert? Der Anwender wird vom Betriebssystem vor die Wahl gestellt, welche App geöffnet werden soll. Angenommen ein *Trojaner* befindet sich auf dem Smartphone, der dieses URI-Schema „tresor:“ registriert. Durch geschicktes Social Engineering ist es daher möglich, dass der Anwender bei der Auswahl, welche App geöffnet werden soll, den Trojaner auswählt. Typischerweise wird der Trojaner einen ähnlichen Namen haben, wie die legitime App. Im Fall der Tresor App könnte der Trojaner *Secure Tresor* oder einfach *Tresor v2* heißen. Ersteres würde dem Anwender „verbesserte Sicherheit“ suggerieren, während letzteres den Anwender glauben lässt, dass eine neuere Version installiert sei und diese besser sein müsse. Sollte der *Trojaner* gestartet werden, kann es sein, dass die andere NFC-basierte Anwendung die Kommunikation im *Peer-To-Peer Mode* fortsetzten möchte. Damit der *Trojaner* keine vertraulichen Informationen erhält, muss die Kommunikation im *Peer-To-Peer Mode* abgesichert werden.

Daher wird empfohlen bei sicherheitskritischen Daten, die Daten, die nach dem Doppelpunkt des URI-Schemas folgen, mit einer Checksumme zu versehen und die Daten mit der Checksumme zu verschlüsseln, um das Tauschen der verschlüsselten Blöcke zu vermeiden.

5.2 Automatisches Öffnen von Apps über NFC

Um eine App über NFC zu öffnen wurden bereits zwei mögliche Wege erläutert: Über ein URI-Schema oder über einen LaunchApp Record (Windows Phone 8) bzw. einen AAR (Android 4.3). Beide Betriebssystem verhalten sich jedoch anders beim Empfangen solcher NFC-Inhalte. Unter Windows Phone 8 wird der Anwender immer um Zustimmung

gebeten, ob er die App oder die URL im Internet Explorer öffnen möchte oder im Windows Phone Store nach einer passenden App suchen möchte. Unter Android 4.3 wird der Anwender grundsätzlich nicht gefragt, was ein Sicherheitsproblem darstellt. Es wird erst dann gefragt, sobald mehr als eine App ein URI-Schema registriert, um den Anwender auswählen zu lassen, welche App geöffnet werden soll. Tests im Labor ergaben jedoch, dass offensichtlich nicht immer gefragt wird, beispielsweise nach der Installation des Browsers Opera in der Version 15.0.1162.61541 aus dem Google Play Store. Empfängt das Smartphone über NFC eine URL, so wird *ohne* nachzufragen Opera geöffnet ohne dem Anwender die Wahl zu lassen, ob er nicht die URL eventuell in Googles eigenem Browser Chrome in der Version 27.0.1453.111 öffnen möchte. Es scheint also möglich zu sein, dass Apps ein URI-Schema für sich beanspruchen können, *obwohl* alternative Apps installiert sind, die ebenfalls mit dem URI-Schema umgehen können. Inwiefern das automatische Öffnen von Apps über NFC ein Sicherheitsrisiko darstellt, wird in zwei Beispielen beschrieben.

Im ersten Beispiel zeigt Collin Mulliner in [Mul08], S.60-61, wie ein einfacher Tag mit einem neuen Tag des Angreifers „überklebt“ werden kann. Eine Möglichkeit ist mit einem sog. *RFID-Zapper* durch zu hohe Spannung ein Bauelement in der Schaltung des RFID-Chips im Tag durchbrennen zu lassen. Eine andere Möglichkeit ist jedoch, den originalen Tag mit Alufolie abzuschirmen und mit dem Tag des Angreifers zu überkleben. Ein typisches Ziel von Angreifern ist es Tags von Dienstanbietern zu überkleben, da Anwender dem Dienstanbieter vertrauen und nichts Böses beim Tag vermuten.

Im nächsten Beispiel kann ein Angreifer einen Tag mit einer URL zu einer Webseite mit Schadcode unter einem Tisch verstecken. Sobald ein NFC Gerät mit entsperrtem Bildschirm sich über dem Tag befindet, wird der Tag ausgelesen und je nach Smartphone Betriebssystem wird die Webseite des Angreifers, die Schadcode enthält, automatisch geladen. Tests im Labor mit den Geräten Nokia Lumia 925, LG Nexus 4 und ASUS Nexus 7 ergaben, dass der Durchmesser eines Holztisches zwischen 2,0 und 2,5cm betragen kann. Wie in Kapitel 3.1 festgestellt wurde, beträgt eine mögliche Distanz zwischen einem Tag und dem Smartphone etwa 2,5 bis 3,0cm und ist demnach auch hardwareabhängig. Mit dem Holztisch zwischen dem Tag und dem Gerät beeinflusst ein weiterer Parameter die mögliche Distanz.

Die Beispiele unterscheiden sich in dem Punkt, dass im ersten Beispiel der Anwender bewusst den Tag mit seinem Smartphone ausliest, da der Anwender dem Dienstanbieter vertraut. Im zweiten Beispiel aber verläuft alles automatisch, *ohne* dass der Anwender das bewusst angestoßen hat. Wie im zweiten Beispiel schon genannt wurde, besteht die Gefahr bei Webseiten, die Schadcode enthalten. Sicherheitsschwachstellen können dadurch in den Browsern ausgenutzt werden um das Smartphone Betriebssystem zu kompromittieren. Im Gegensatz zu Android 4.3 fragt Windows Phone 8 den Anwender, ob dieser die Webseite öffnen möchte. Dabei wird dem Anwender nur der Hostname und die Top-Level-Domain (TLD) angezeigt, wie z.B. *hs-ulm.de*.

5.3 NFCProxy

Bei NFCProxy handelt es sich um eine Android App [Sou13], mit der es möglich ist die Kommunikation zwischen einem Reader und einer kontaktlosen Smartcard über einen Proxy durchzuführen [Lee12]. Die App NFCProxy unterstütz dabei zwei Betriebsmodi: *Relay Mode* und *Proxy Mode*. Ein Smartphone, das sich im *Relay Mode* befindet, liest über NFC eine kontaktlose Smartcard aus und verbindet sich über WLAN zu einem anderen Smartphone, welches sich im *Proxy Mode* befindet. Das andere Smartphone, welches sich im *Proxy Mode* befindet, befindet sich auf einem Reader. Über die WLAN Verbindung werden *Application Protocol Data Units* (APDU) ausgetauscht. APDUs sind die Kommunikationseinheit zwischen einem Reader und einer kontaktlosen Smartcard. Der Reader sendet eine *command APDU*, auf die eine kontaktlose Smartcard mit einer *response APDU* antwortet. Um dies zu veranschaulichen, ein Beispiel in Abbildung 16.

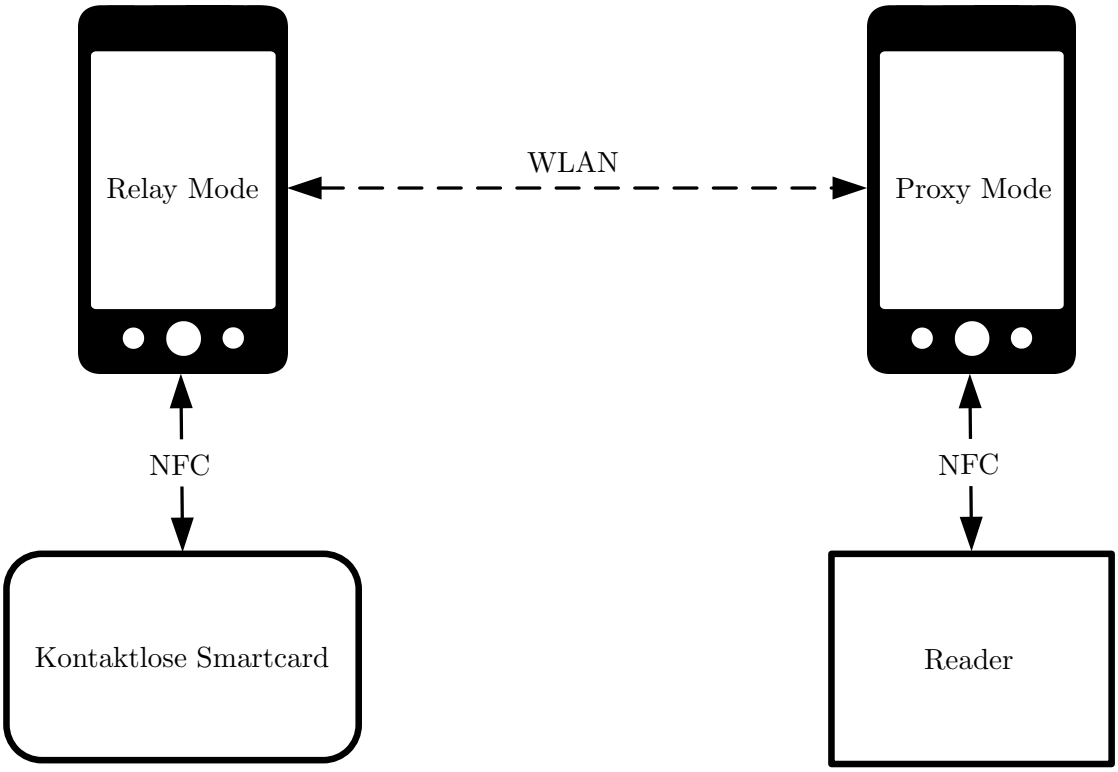

Abbildung 16: Typischer Aufbau bei der Verwendung von NFCProxy.

In Kapitel 3.5 wurde beschrieben, dass Man-In-The-Middle Angriffe auf unterster Ebene nicht möglich sind. NFCProxy zeigt jedoch, dass Man-In-The-Middle Angriffe auf Applikationsebene des ISO/OSI Referenzmodells ermöglicht werden.

5.3.1 Relay-Angriff

Ein Relay-Angriff [And08] ist eine Form eines Man-In-The-Middle Angriffs, bei dem die Nachricht eines legitimen Senders vom Angreifer an den legitimen Empfänger weitergeleitet wird. Dazu ein Beispiel in Abbildung 17.

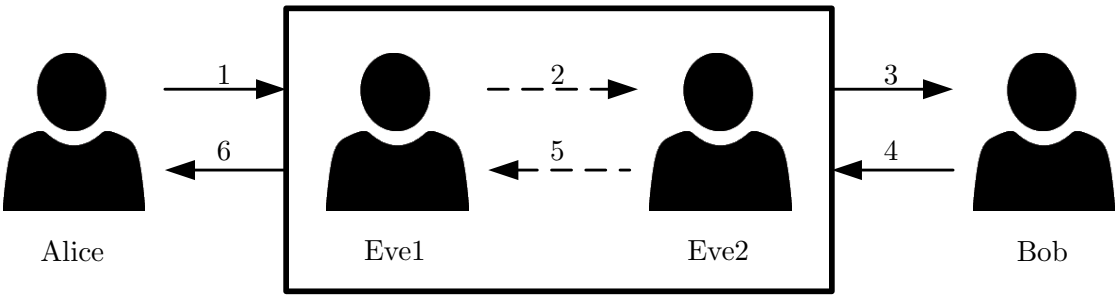

Abbildung 17: Aufbau eines Relay-Angriffs.

Daher entspricht ein Angriff mit NFCProxy einem Relay-Angriff. Ein Beispiel, bei dem ein Relay-Angriff mit NFCProxy durchgeführt werden kann, ist *Pull printing*. Pull printing ist eine Druckerfunktion, die den gesendeten Druckauftrag nicht sofort ausdruckt, sondern auf dem Druckserver belässt. Sobald der Anwender sich an einem Drucker mit seiner Smartcard authentifiziert, wird der Druckvorgang gestartet. Eines der Hauptmerkmale dieser Funktion ist, die Sicherheit zu erhöhen. Dadurch, dass die Person, die den Druckauftrag gestartet hat, sich lokal am Drucker anmelden muss, kann verhindert werden, dass in der Zwischenzeit jemand anderes die gedruckten Dokumente liest bzw. entwendet. Insbesondere für Unternehmen ist das eine praktische Funktion, um vertrauliche Unterlagen von Unbefugten fernzuhalten. Dieses Modell hat jedoch eine Schwachstelle: Die Hersteller solcher Systeme verlassen sich auf den Faktor *Besitz*, was der *kontaktlosen Smartcard* entspricht. Eine starke Authentifizierung verlangt aber neben dem Faktor *Besitz* zusätzlich den Faktor *Wissen*, in Form eines Passworts oder einer PIN. Da jedoch auf das Passwort oder die PIN verzichtet wird, ist es möglich mit *NFCProxy* einen Druckauftrag eines Anwenders mithilfe eines Smartphones im *Proxy Mode* abzuholen, sofern ein anderes Smartphone im *Relay Mode* auf der *kontaktlosen Smartcard* liegt. Um den Social Engineering Aspekt einzubringen, wäre ein denkbares Szenario der Arbeitsplatz eines Mitarbeiters. Unter der Annahme, dass der Mitarbeiter seinen Werksausweis im Geldbeutel hat und diesen immer auf den Tisch legt, kann der Angreifer durch Beobachten des Mitarbeiters erkennen, an welcher Stelle am Tisch sich der Geldbeutel am häufigsten befindet. Anhand dieses Wissens kann der Angreifer ein Smartphone im *Relay Modus* versteckt unter dem Tisch an entsprechender Stelle des Mitarbeiters anbringen. Die Durchführbarkeit des Angriffs hängt ab von Material und Durchmesser des Tisches als auch von dem der eingesetzten Hardware, wie in Kapitel 5.2 festgestellt wurde. Sollte dieser Mitarbeiter vertrauliche Dokumente mit *Pull printing* drucken, kann der Angreifer sich mit dem Smartphone im *Proxy Mode* am Drucker authentifizieren und den Druckvorgang starten, sofern sich zu dem Zeitpunkt der Geldbeutel mit dem Werksausweis des Mitarbeiters an der richtigen Stelle des Tisches befindet.

Um *Pull printing* gegen solche Angriffe zu schützen, wäre ein möglicher Lösungsansatz eine starke Authentifizierung zu verwenden. Beispielsweise wenn der Anwender einen Druckauftrag mit *Pull printing* absendet, zeigt das System dem Anwender eine zufällig generierte PIN an, die er, neben der Authentifizierung mit der kontaktlosen Smartcard zusätzlich eingeben muss.

6 Aufbau eines sicheren Kanals über NFC

In diesem Kapitel wird gezeigt, wie ein sicherer aufgebaut werden kann. Als Schlüsselaustauschverfahren kommt *Elliptic Curves Diffie-Hellman* (ECDH) zum Einsatz. Neal Koblitz [Kob87] und Victor S. Miller [Mil86] haben unabhängig voneinander etwa zeitgleich, das Prinzip der *Elliptic Curve Cryptography* (ECC) vorgestellt und vorhandene Verfahren, wie z.B. Diffie-Hellman, auf Basis von Elliptic Curves implementiert, wie Bruce Schneier in [Sch96], S. 480-481, beschreibt. Die Sicherheit von Verfahren, die auf ECC setzen, besteht darin, dass sich der diskrete Logarithmus in der Gruppe der Punkte der elliptischen Kurve schwierig berechnen lässt. Die Funktionsweise von ECC wird im Weiteren nicht näher betrachtet. Der Einsatz von ECC bei NFC-basieren Anwendungen rechtfertigt jedoch die Tatsache, dass deutlich kürzere Schlüssellängen ermöglicht werden, was durch die geringe Übertragungsrate ein entscheidender Faktor ist. Beispielsweise bietet ein Schlüssel mit der Länge von 160 Bit, bei ECC ein ähnliches Sicherheitsniveau wie ein Schlüssel des asymmetrischen Verschlüsselungsverfahren RSA mit 1024 Bit [Gir13].

6.1 Diffie-Hellman

Das Schlüsselaustauschverfahren Diffie-Hellman wurde von Whitfield Diffie und Martin Hellman 1976 entwickelt und veröffentlicht [DH76]. Das Besondere an dem Schlüsselaustauschverfahren ist, dass Alice und Bob sich auf einen geheimen Schlüssel einigen können, ohne dabei selber den geheimen Schlüssel übertragen zu müssen. Die Funktionsweise und die Mathematik von Diffie-Hellman beschreibt Bruce Schneier in [Sch96], S. 513-514. Zunächst müssen sich Alice und Bob auf zwei Primzahlen n und g einigen, sodass g die Primitivwurzel Modulo n ist. Dabei ist es nicht erforderlich, dass n und g geheim gehalten werden müssen. Der weitere Verlauf des Protokolls ist wie folgt:

1. Alice erzeugt per Zufall eine beliebige Zahl x und sendet X an Bob:
$$X = g^x \bmod n.$$
2. Bob erzeugt per Zufall eine beliebige Zahl y und sendet Y an Alice:
$$Y = g^y \bmod n.$$
3. Alice berechnet:
$$k = Y^x \bmod n.$$
4. Bob berechnet:
$$k' = X^y \bmod n.$$

Den geheimen Schlüssel repräsentieren k und k‘, die vollkommen identisch sind. Keiner, der die Kommunikation zwischen Alice und Bob abhört, kann den geheimen Schlüssel k bzw. k‘ rekonstruieren, außer dieser ist in der Lage den diskreten Logarithmus zu berechnen und die Werte x und y zu rekonstruieren. In Abbildung 18 wird ein Beispiel gezeigt mit dem Werten $g = 2$ und $n = 13$.

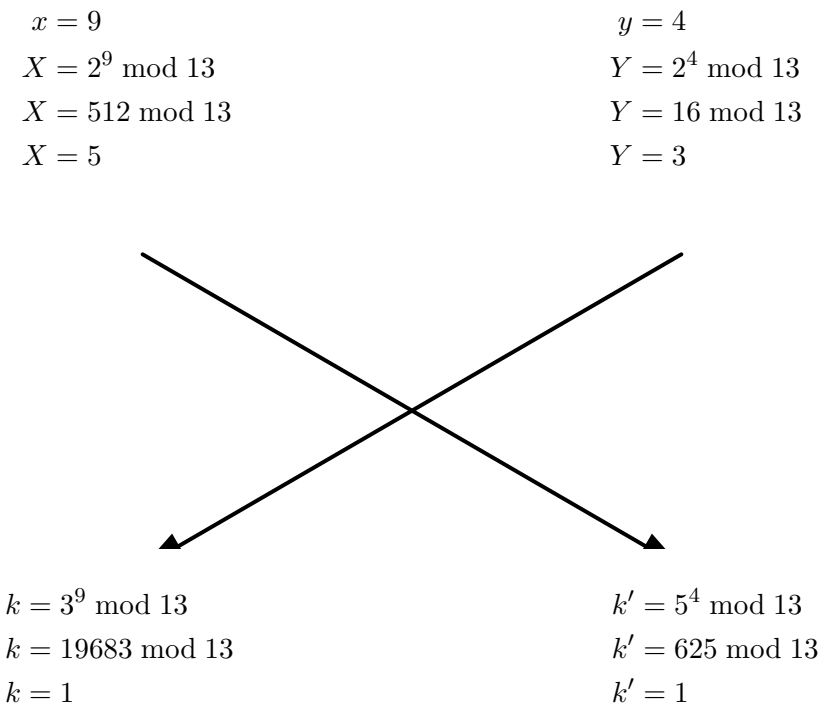

Abbildung 18: Alice und Bob erzeugen den geheimen Schlüssel $k = k' = 1$.

In der Realität jedoch werden wesentlich höhere Zahlen verwendet, um entsprechend längere Schlüssel zu erzeugen. Ein anderes abstraktes Beispiel in Abbildung 19 erklärt die Funktionsweise von Diffie-Hellman durch Mischen von Farben.

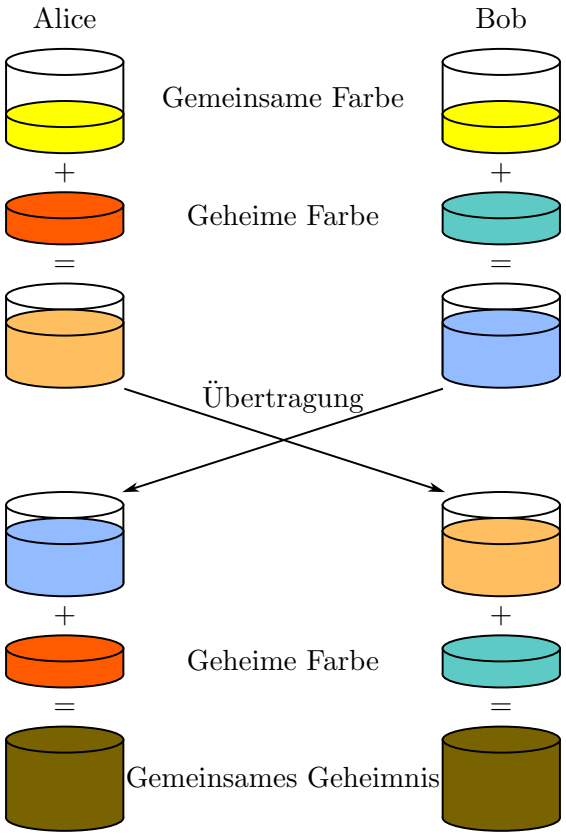

Abbildung 19: Diffie-Hellman beschrieben durch Mischen von Farben [Vin12].

Zunächst müssen sich Alice und Bob auf eine Farbe einigen. Nachdem sie sich auf eine Farbe geeinigt haben, mischen Alice und Bob jeweils eine geheime Farbe hinzu. Das Ergebnis, dieser Farbmischung wird an den anderen übertragen. Unter der Annahme, dass das Trennen von Farben eine teure Operation ist, kann ein Angreifer zunächst, die geheime Farbe nicht rekonstruieren. Nachdem Alice von Bob die Mischung erhalten hat und umgekehrt, fügen Alice als auch Bob jeweils die eigene geheime Farbe zu der erhaltenen Farbmischung hinzu. Als Ergebnis erhalten Alice und Bob die gleiche Farbe. Aus technischer Sicht betrachtet einigen sich Alice und Bob zunächst auf die Parameter g und n, übertragen dann ihr Teilgeheimnis X bzw. Y und bilden aus dem erhaltenen Teilgeheimnis den geheimen Schlüssel k bzw. k‘. Damit kann an dieser Stelle ein symmetrisches Verschlüsselungsverfahren angewandt werden. In Abbildung 20 wird der Ablauf des Schlüsselaustauschverfahrens Diffie-Hellman beschrieben.

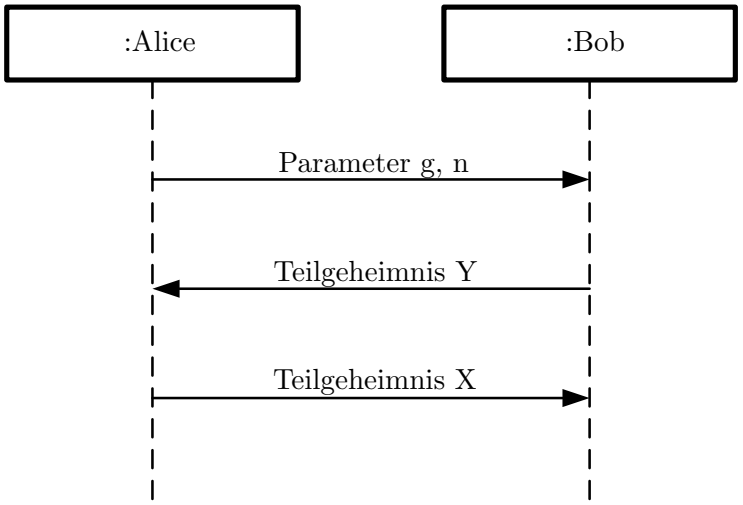

Abbildung 20: Handshake des Schlüsselaustauschverfahren Diffie-Hellman.

Wie die Android App NFCProxy jedoch zeigt, sind Man-In-The-Middle Angriffe auf der Anwendungsschicht des ISO/OSI Referenzmodells möglich. Daher ist es dringend notwendig, dass die Diffie-Hellman Teilgeheimnisse mit einer digitalen Signatur versehen werden. Andernfalls ist es möglich, dass ein *aktiver* Man-In-The-Middle, auch als Mallory bekannt, einfach zwei Mal einen Schlüsselaustausch jeweils mit Alice und Bob durchführt. In Abbildung 21 wird dieses Szenario beschrieben.

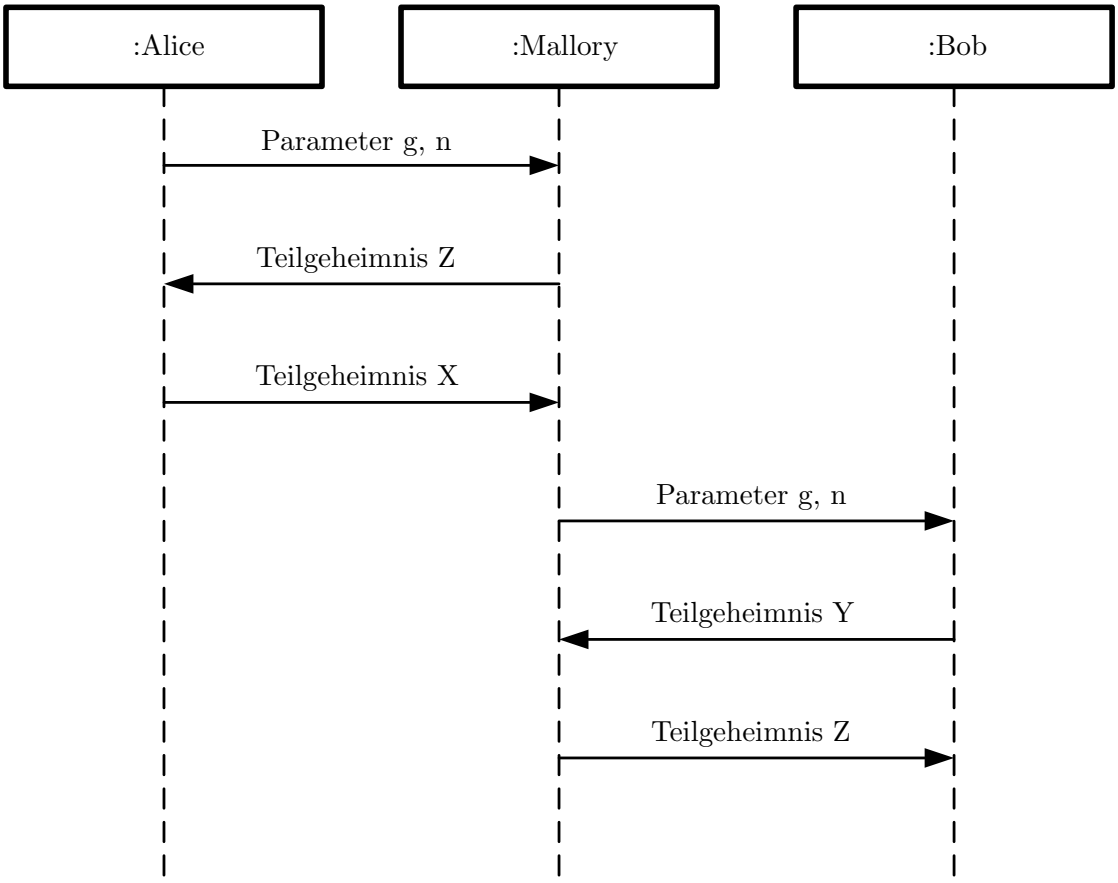

Abbildung 21: Man-In-The-Middle Angriff auf Diffie-Hellman.

Um einen *aktiven* Man-In-The-Middle Angriff zu vermeiden, müssen, wie schon oben erwähnt, die Diffie-Hellman Teilgeheimnisse mit einer digitalen Signatur versehen werden. Alice erzeugt eine digitale Signatur, indem sie einen Hash ihres Teilgeheimnisses erstellt und den Hash mit ihrem Private Key verschlüsselt. Nachdem Bob das Teilgeheimnis und den verschlüsselten Hash erhalten hat, entschlüsselt Bob mit dem Public Key von Alice den verschlüsselten Hash und bildet den Hash des erhaltenen Teilgeheimnisses um beide Hash-Werte miteinander zu vergleichen. Sind beide Hash-Werte identisch, wurden die Daten während der Übertragung nicht modifiziert. Doch bevor Bob die Signatur von Alice prüfen kann, muss Bob den Public Key erst von Alice erhalten. Wie kann beim Erhalt des Public Key von Alice aber nachgewiesen werden, dass dieser auch wirklich von Alice ist und nicht vom Man-In-The-Middle Mallory? Mithilfe eines Trust-Centers, wäre es möglich Alice und Bob eine eindeutige Identität zu vergeben und diese auch bei Bedarf beim TrustCenter zu validieren. Dazu müssten jedoch Alice und Bob ihren Private und Public Key auf dem Dateisystem abspeichern. Da stellt sich wiederum die Frage, insbesondere bei Smartphone Apps, ob das Smartphone-Betriebssystem einen ausreichenden Schutz für den Private Key auf dem Dateisystem vor unbefugtem Zugriff bietet?

6.2 Sicherheitsanforderungen an das Smartphone Betriebssystem

Sensible Daten von Apps müssen sicher gespeichert werden können. Wie in Kapitel 4.1 erläutert wurde, ist es nicht möglich, dass z.B. eine App A auf die Daten von einer App B zugreifen kann. Dennoch unterliegen Smartphone Betriebssysteme verschiedenen Angriffsvektoren. Ein typisches Ziel eines Angreifers ist es eine *Privilege Escalation* durch fehlerhafte Softwareimplementierung zu erreichen. Dadurch, dass die Rechte vom Angreifer erhöht werden, kann dieser je nach Berechtigungsstufe und der konsequenten Umsetzung der *Defense-in-Depth*-Schutzmaßnahmen vom Betriebssystem unterschiedlich großen Schaden anrichten. Ein erster Schritt z.B. wäre der Zugriff auf die Sandbox einer anderen App. Sollte dieser Zugriff gelingen, ist ein Angreifer möglicherweise in der Lage vertrauenswürdige Daten von der App zu kopieren oder zu manipulieren. Daher empfiehlt es sich, sensible Informationen, wie z.B. einen Private Key, nicht auf dem Dateisystem zu speichern [Cer13]. Sollte keine andere Möglichkeit bestehen, scheint zunächst Verschlüsselung als möglicher Lösungsansatz. Wenn ein Angreifer aber Zugriff auf die Sandbox hat und die Daten verschlüsselt sind, kann der Angreifer immer noch in die ausführbaren Dateien der App einsehen. Darin wird sich auch üblicherweise der geheime Schlüssel befinden, um die Dateien in der Sandbox zu entschlüsseln. Apps in Windows Phone 8 und Android 4.3 werden typischerweise in *Managed Code* umgesetzt, wie z.B. .NET bzw. Java. Da Managed Code zur Laufzeit übersetzt wird, befinden sich die ausführbaren Dateien in einem plattformunabhängigen Format, auch als *Common Intermediate Language* (CIL) bei .NET und *Bytecode* bei Java bezeichnet. Aus diesem plattformunabhängigen Format lässt sich mit diversen Programmen der originale Quellcode wiederherstellen. Daher ist es ratsam, keine Geheimnisse im Quellcode hardzucoden. Es gibt zwar Verschleierungsprogramme, die Klassen und Objekte in nicht-triviale Namen umbenennen, um das *Reverse Engineering* eines Angreifers zu erschweren. Dennoch ist das zwangsweise kein Schutz, da z.B. ein String, der im Code eine längere Zeichenfolge zugewiesen bekommt, für den Angreifer ein Indiz dafür sein kann, dass dies möglicherweise ein Private Key zum Entschlüsseln sein könnte. In Windows Phone 8 wird der CIL im Windows Phone Store in ein sogenanntes *Machine Dependent Intermediate Language* (MDIL) Format kompiliert [Ram12]. Zum Installationszeitpunkt wird nativer Code aus den MDIL-Binärdateien erzeugt. Aber auch nativer Code bietet an dieser Stelle keinen garantierten Schutz, da auch dieser mit Analyseprogrammen nach auffälligen Codeabschnitten durchsucht werden kann.

Absolute Sicherheit kann nicht erreicht werden, da es letztendlich nur um Schadensbegrenzung geht. Es muss davon ausgegangen werden, dass Smartphone Betriebssysteme kompromittierbar sind und das Dateisystem, welches einer App zur Verfügung steht kein vertrauenswürdiger Ort, für sensible Daten ist. In Kapitel 7 wird ein Ansatz für Windows Phone 8 gezeigt, wie ein Schlüsselpaar aus Private Key und Public Key erzeugt werden kann und sicher persistiert werden kann.

6.3 Möglicher Implementierungsansatz

In diesem Kapitel wird ein möglicher Implementierungsansatz beschrieben, wie die Kommunikation zwischen einem Reader und einer App abgesichert werden kann. Die Kommunikation erfolgt im Peer-To-Peer Mode. Der grobe Ablauf zum Aufbau eines sicheren Kanals könnte in etwa so aussehen: Der Reader sendet sein digital signiertes Teilgeheimnis an die App. Die App empfängt und überprüft die digitale Signatur des Teilgeheimnisses. Dies ist möglich, da der Public Key vom Reader in der App hardgecodet ist. Dadurch, dass beim TrustCenter der Private Key und Public Key auf dem Dateisystem gespeichert werden müssten, wird auf ein TrustCenter aus Sicherheitsgründen verzichtet. Die Konsequenz beim Empfang eines Public Keys vom Reader wäre, dass die Identität nicht geprüft werden kann. Aus diesem Grund wird der Public Key vom Reader in der App hardgecodet. Andernfalls bestünde auch die Gefahr, dass ein *aktiver* Man-In-The-Middle den Public Key vom Reader abfängt und seinen eigenen an die App weiter schickt. Bestätigt die Überprüfung der digitalen Signatur die Echtheit der Daten vom Reader, signiert die App mit ihrem Private Key ihr Teilgeheiminis und sendet die Daten an den Reader weiter. An dieser Stelle gibt es jedoch ein Problem: Woher weiß der Reader, dass die Daten wirklich von der App stammen? Der Reader als *target* könnte zu dem Zeitpunkt Daten von einem Angreifer empfangen. Aus diesem Grund muss die App einmalig beim ersten Start der App, ein Schlüsselpaar generieren und den Public Key sicher an den Reader übertragen.

6.3.1 Sichere Übertragung des generierten Public Keys der App

Beim ersten Start der App wird ein Private Key und Public Key Schlüsselpaar erzeugt. Dieses Schlüsselpaar muss jedoch sicher abgespeichert werden. In Windows Phone 8 besteht die Möglichkeit, das Schlüsselpaar z.B. im Cryptographic Service Provider (CSP) abzuspeichern. Der CSP ist eine Softwarekomponente, die es zulässt kryptografische Operationen durchzuführen und die sichere Speicherung von Schlüsselpaaren ermöglicht. Microsoft rät davon ab, Schlüsselpaare auf dem Dateisystem zu speichern und empfiehlt stattdessen diese in einem Container im CSP abzulegen [Mic]. Um ein Schlüsselpaar in einem CSP Container abzuspeichern, muss der Programmierer zunächst den Container benennen. Beim nächsten Start der App gibt der Programmierer den Bezeichner des Containers an, um auf das Schlüsselpaar zugreifen zu können. Im Gegensatz zu Windows 8.1 Pro Preview, kann eine App A unter Windows Phone 8 nicht auf den CSP Container der App B zugreifen, um das Schlüsselpaar von App B zu entwenden, sofern App A weiß, wie App B den Container benannt hat. D.h. dass jede App einen *eigenen* CSP Container in ihrem Chamber hat und nicht in der Lage ist den Chamber zu verlassen, um auf den CSP Container einer anderen App zuzugreifen. Da der CSP Container benannt werden muss, empfiehlt es sich die *PublisherHostId* Property von `Windows.Phone.System.Analytics.HostInformation` der Windows Phone Runtime abzufragen. Damit lässt sich ein Gerät für einen Herausgeber eindeutig identifizieren. D.h. wenn eine andere App diese API aufruft, bekommt sie einen anderen Rückgabewert. Daher bietet sich an, *PublisherHostId* als Bezeichner für den

CSP Container zu verwenden. Um jedoch diese API zu verwenden, muss die Capability ID_CAP_IDENTITY_DEVICE angefordert werden.

Angenommen es gelingt einem Angreifer aus der App A auf den CSP Container einer App B zuzugreifen und er hat zumindest lesenden Zugriff auf den Chamber der App B. Er wird versuchen herauszufinden, wie App B ihren CSP Container benannt hat, um das Schlüsselpaar entwenden zu können. Dadurch, dass die App B die *PublisherHostId* API verwendet, um den Container zu benennen, müsste der Angreifer Code im Kontext der App B ausführen, um den Rückgabewert der *PublisherHostId* zu erfahren. Dies funktioniert jedoch nicht, da wie in Kapitel 4.1 beschrieben wurde, alle ausführbaren Dateien dem *Code Signing* von Windows Phone 8 unterliegen.

Der Sicherheitsgewinn, der durch Verwendung der *PublisherHostId* Property entsteht, besteht darin, dass, auch wenn der Angreifer in der Lage ist auf den CSP Container der App zuzugreifen und den Quellcode zu analysieren, er nicht weiß, wie der CSP Container benannt wurde. Ein anderer Interoperabler Ansatz wäre, das Schlüsselpaar in einem *Secure Element* zu erzeugen und abzuspeichern. Wie bereits kurz in Kapitel 2.4.3 erwähnt wurde, ist ein Secure Element in der Lage, kryptografische Operationen durchzuführen und Dateien sicher zu speichern. Da in Kapitel 6.2 festgestellt wurde, dass sensible Daten *nicht* auf dem Dateisystem der App gespeichert werden sollten, bietet sich an dieser Stelle an, die vertrauenswürdigen Informationen von einem Secure Element zu erzeugen und dort sicher aufbewahren. Dadurch, dass ein Secure Element kryptografische Operationen durchführen kann, muss ein Private Key diesen Geschützen Bereich nie verlassen [Zef12]. Ein Problem besteht jedoch: Entwickler haben normalerweise keinen Zugriff auf die Secure Element Betriebssystem APIs. D.h. Zugriff muss vom Entwickler beim Betriebssystemhersteller angefordert werden.

Nachdem das Schlüsselpaar erzeugt wurde, wird der Anwender aufgefordert einen Code in der App einzugeben, welchen der Reader dem Anwender anzeigt. Dieser Code wird als Salt verwendet, indem der Salt, dem Public Key von der App angehängt wird und davon ein Hash gebildet wird. Dadurch, dass der Public Key vom Reader in der App hardgecodet ist und der Anwender einen Code in der App eingeben muss, entspricht diese einer *starken Authentifizierung*. Diese Kombination *Besitz* (Public Key vom Reader) und *Wissen* (Code, den der Anwender in die Appeingibt) schließt *zunächst* einen erfolgreichen Man-In-The-Middle Angriff aus, da dem Angreifer das Wissen über den Code fehlt. Der generierte Hash und der Public Key von der App wird mit dem hardgecodeten Public Key des Readers verschlüsselt und zum Reader übertragen. Der Reader entschlüsselt mit seinem Private Key den erhaltenen Public Key der App und den Hashwert. Der Reader fügt den Salt dem erhaltenen Public Key hinzu und generiert daraus den Hash. Anschließend überprüft der Reader beide Hashwerte ob diese identisch sind. Ist dies der Fall, fügt der Reader den Public Key der App der Liste der *vertrauenswürdigen Clients* hinzu. In Abbildung 22 wird der Ablauf nochmal verdeutlicht.

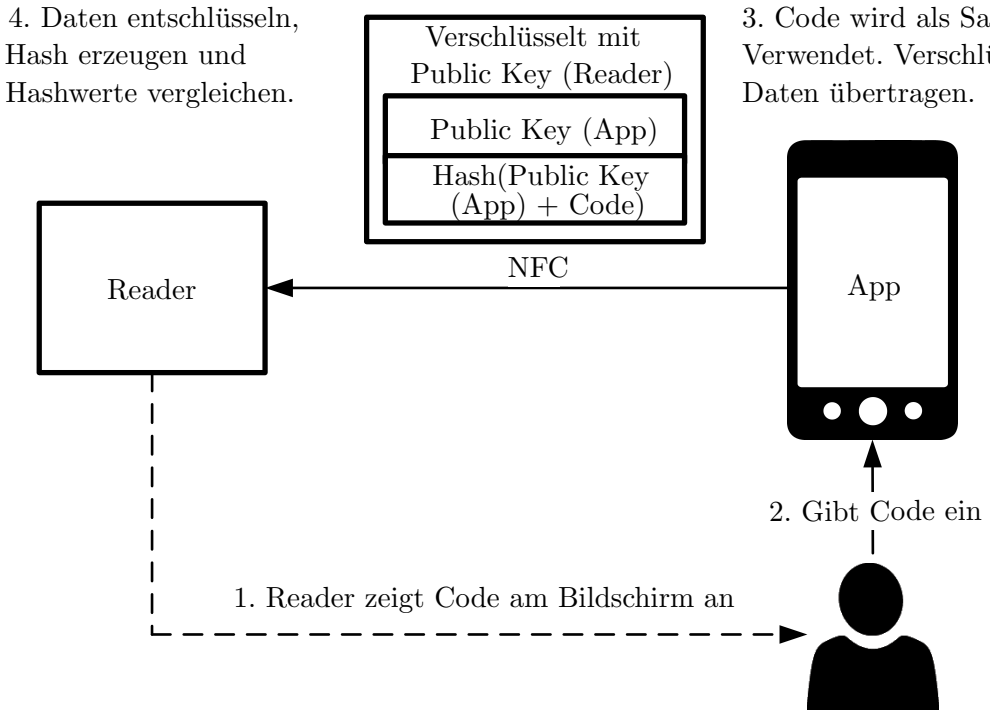

Abbildung 22: Starke Authentifizierung zur sicheren Übertragung des Public Keys.

Nachdem der Reader die App der Liste der vertrauenswürdigen Clients hinzugefügt hat, ist der Reader in der Lage, nach Erhalt eines digital signierten Teilgeheimnisses, zu prüfen, ob der Urheber des digital Signierten Teilgeheimnisses sich in der Liste der vertrauenswürdigen Clients befindet. Ist das der Fall, kann aus dem Erhalt des Teilgeheimnisses der geheime Schlüssel erzeugt werden. Andernfalls handelt es sich dabei um einen *aktiven* Man-In-The-Middle Angriff. Die Sicherheit von diesem Verfahren hängt jedoch von der Länge des Codes ab. Unter der Annahme, dass der Angreifer im Besitz des Public Keys vom Reader ist und die Codelänge sich auf 4 Zahlen beschränkt, gibt es $10^4 = 10000$ Möglichkeiten, die ein Angreifer durchprobieren kann um *seinen* Public Key dem Reader zu übertragen. Ein Möglicher Schutz gegen so einen *Bruteforce* Angriff wäre, z.B. statt 4 Zahlen, 4 Alphanumerische Zeichen zu verwenden. Die mögliche Anzahl an Kombinationen steigt deutlich auf $36^4 = 1679616$. Ebenso ist empfehlenswert die Gültigkeit des Codes zeitlich zu begrenzen. Nach Ablauf dieser Zeit wird ein neuer Code generiert. Sollte in dem gegebenen Zeitraum die Anzahl der gescheiterten Hashvalidierungen einen bestimmten Wert überschreiten, kann eine Sperrfrist gesetzt werden. Sollten die Datenmengen für das asymmetrische Verschlüsselungsverfahren zu groß sein, kann alternativ auf ein hybride Verschlüsselungsverfahren gesetzt werden. Dazu erzeugt die App einen zufällig generierten Code der mit dem Public Key des Readers verschlüsselt und an den Reader übertragen wird. Anschließend erfolgt der Datentransport mit symmetrischer Verschlüsselung.

6.3.2 Schlüsselaustausch über ECDH

Nachdem die App erfolgreich ihren Public Key an den Reader übertragen konnte, sind der Reader und die App in der Lage einen sicheren Kanal aufzubauen. Die Kommunikation beginnt damit, dass der Reader zunächst eine NDEF message mit einer URI an die App überträgt. Der Inhalt der URI enthält *NfcApp:HelloFromReader*. Das URI-Schema *NfcApp:* wird von der App beim Smartphone Betriebssystem registriert. Nachdem das Smartphone Betriebssystem die URI vom Reader erhält, wird je nach verwendetem Smartphone Betriebssystem der Anwender gefragt, ob er die mit dem URI-Schema verknüpfte App öffnen möchte. Dabei können mehrere Apps das gleiche URI-Schema registrieren, wie in Kapitel 5.1 gezeigt wurde. Nachdem die App geöffnet wird, überprüft die App die *erhaltenen* Parameter des empfangenen URI-Schemas. Als Antwort auf das *HelloFromReader* übertragt die App ihr Teilgeheimnis. Im nächsten Schritt generiert der Reader sein Teilgeheimnis und überträgt dieses. Üblicherweise müssen sich beide Kommunikationspartner auf gemeinsame Konfigurationsparameter einigen, wie es oben beschrieben wurde. Hier entfällt dies jedoch, da auf Codeebene beide Anwendungen die gleichen *Elliptic Curves Domain Parameters* verwenden und diese dadurch nicht mehr ausgetauscht werden müssen. Die App empfängt das Teilgeheimnis vom Reader und speichert dieses ab. Als nächstes erzeugt die App eine digitale Signatur von ihrem Teilgeheimnis und überträgt dieses verschlüsselt mit dem Public Key des Readers, an den Reader. Der Reader empfängt die digitale Signatur von der App und entschlüsselt die digitale Signatur mit seinem Private Key, um den Hashwert zu erhalten. Als nächstes erzeugt der Reader einen Hash von dem zuvor empfangenen Teilgeheimnis von der App und vergleicht beide Hashwerte. Sind beide Hashwerte identisch, setzt der Reader die Kommunikation fort und erzeugt eine digitale Signatur von seinem Teilgeheimnis, welches der Reader an die App weiter sendet. Die App empfängt die digitale Signatur vom Reader und überprüft diese auf Validität. Dadurch, dass in Kapitel 6.3.1 der Public Key der App an den Reader übertragen wurde und der Public Key vom Reader in der App hardgecodet ist, kann sowohl der Reader als auch die App an dieser Stelle die jeweilige digitale Signatur überprüfen. Stellt sich heraus, dass beide digitale Signaturen valide sind, sind beide in der Lage aus dem erhaltenen Teilgeheimnis, den geheimen Schlüssel zu erzeugen. Anschließend setzten der Reader und die App die Peer-To-Peer Kommunikation mit einem symmetrischen Verschlüsselungsverfahren, wie z.B. AES, beliebig fort. Das Kommunikationsprotokolls wird in Abbildung 23 zusammengefasst.

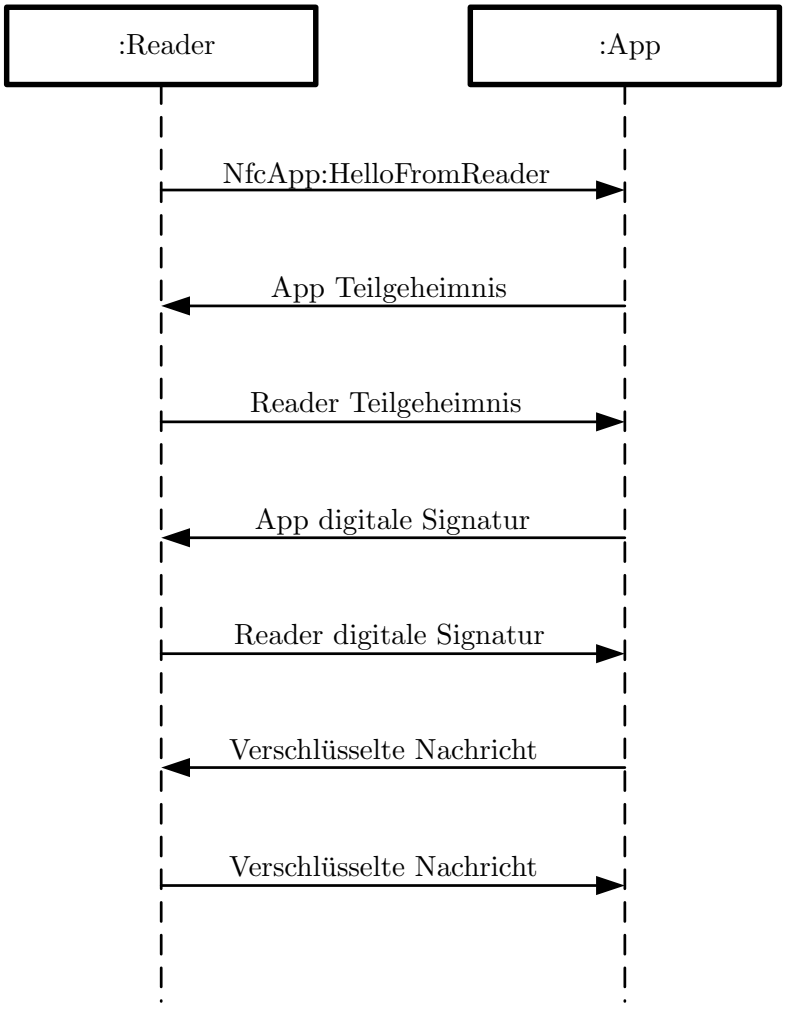

Abbildung 23: Kommunikationsprotokoll zwischen Reader und der App.

Um die Integrität der Daten zu wahren, sollte vor dem verschlüsseln eine Checksumme von den zu übertragenden Daten gebildet werden und ans Ende der Daten angehängt werden. Dadurch kann verhindert werden, dass ein Angreifer Blöcke des verschlüsselten Datenstroms tauscht.

7 Implementierung

In diesem Kapitel werden zwei Anwendungen zum Aufbau eines sicheren Kanals über NFC implementiert. Die erste Anwendung ist die Reader-Anwendung, die auf Windows 8.1 Pro Preview mit .NET 4.5 und der *Windows Runtime* (WinRT) realisiert wird. Die zweite Anwendung ist eine Windows Phone 8 App, die mit der *Windows Phone Runtime* (WinPRT) durch Installation des Windows Phone 8 SDK umgesetzt wird. WinRT und WinPRT bieten eine gemeinsame Schnittmenge an Klassen, die sowohl unter Windows 8.1 Pro Preview als auch unter Windows Phone 8 verfügbar sind. Die für diese Implementierung relevante Klasse `ProximityDevice` ist im WinRT/WinPRT Namespace `Windows.Networking.Proximity` für beide Plattformen verfügbar. Für beide Anwendungen wird der Quellcode-Stand vom 08.09.2013 von GitHub der Kryptographie Bibliothek *Bouncy Castle C#* verwendet, um eine möglichst aktuelle Version der Bibliothek für Windows 8.1 Pro Preview und Windows Phone 8 zu kompilieren [Git12]. Ebenso wird die *NDEF Library for Proximity APIs / NFC* in der Version 1.1.0.1 verwendet [Jak13]. Als Entwicklungsumgebung kommt sowohl für die Reader-Anwendung als auch die Windows Phone 8 App Microsoft Visual Studio 2013 Preview zum Einsatz. Als Hardware für die Reader-Anwendung wird der NFC-USB-Stick *Sensor ID Stick ID* mit dem NXP PN-533 Chipsatz verwendet. Die Implementierung wird wie in Kapitel 6.3 umgesetzt, mit dem Unterschied, dass sowohl in der App als auch im Reader der Public Key hardgecodet ist und dadurch die sichere Übertragung des Public Key der App entfällt.

7.1 Architektur

Bevor auf die Softwarearchitektur eingegangen wird, sollte zunächst beschrieben werden, welche funktionalen Komponenten existieren. Sowohl der Reader als auch die App beschränken sich auf eine: NFC. Daher lässt sich der etwaige Aufbau in Abbildung 24 beschreiben.

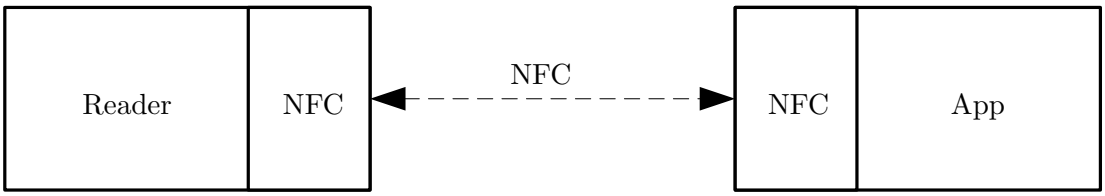

Abbildung 24: Funktionale Komponenten vom Reader und der App.

Um die Software beider Anwendungen auf einer abstrakten Ebene zu beschreiben, *welche* Aufgaben aufgeführt werden, lassen sich diese wie folgt unterteilen: *Senden*, *bearbeiten* und *empfangen*. In welchen Schritten diese Aufgaben durchgeführt werden, wird in Abbildung 25 gezeigt.

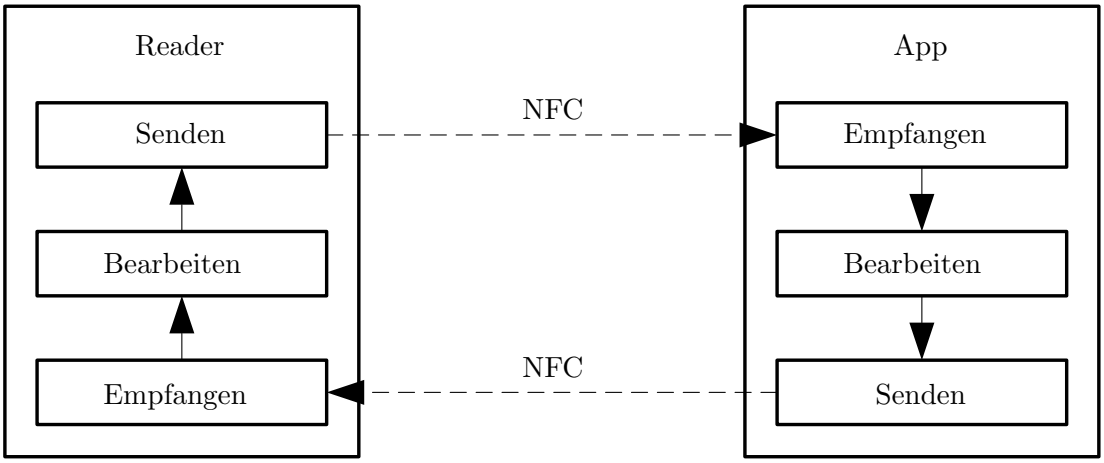

Abbildung 25: Abstrakte Beschreibung der Aufgaben vom Reader und der App.

Wie zu erkennen ist, ist der typische Ablauf *empfangen*, *bearbeiten* und *senden*. Die verwendeten Klassen in diesen Aufgaben wurden so umgesetzt, dass diese sowohl im Reader als auch in der App *ohne* plattformspezifische Anpassungen funktionieren. Die wesentlichen Klassen die Umgesetzt wurden, sind, `Aes`, `Checksum`, `EllipticCurveDiffieHellman` und `Nfc`. Ohne auf die Klassen an dieser Stelle näher einzugehen, können die Klassen wie folgt den Aufgaben zugeordnet werden: Die Klasse `Nfc` übernimmt das *senden* und *empfangen*. Hingegen übernehmen alle anderen Klassen das die Aufgabe *bearbeiten*.

7.2 Handshake

Doch bevor eine Kommunikation zwischen Reader und App beginnen kann, muss ein Handshake-Protokoll definiert werden. Im Wesentlichen wird aus der NDEF Library for Proximity APIs / NFC ein `NdefRecord` Objekt, geschachtelt in einem `NdefMessage` Objekt, erzeugt und übertragen. In dem `NdefRecord` Objekt wird neben dem *Payload*, welches die Nutzdaten in Form eines Byte-Arrays repräsentiert, eine *Id* vergeben, die als Sequenznummer dient. Anhand der Property *Id* kann die Anwendung ermitteln, in welchem Schritt des Protokolls sie sich befindet. Daher sind die vergebenen Sequenznummern beim Reader *gerade* und bei der App *ungerade*, wie in Abbildung 26 verdeutlicht wird.

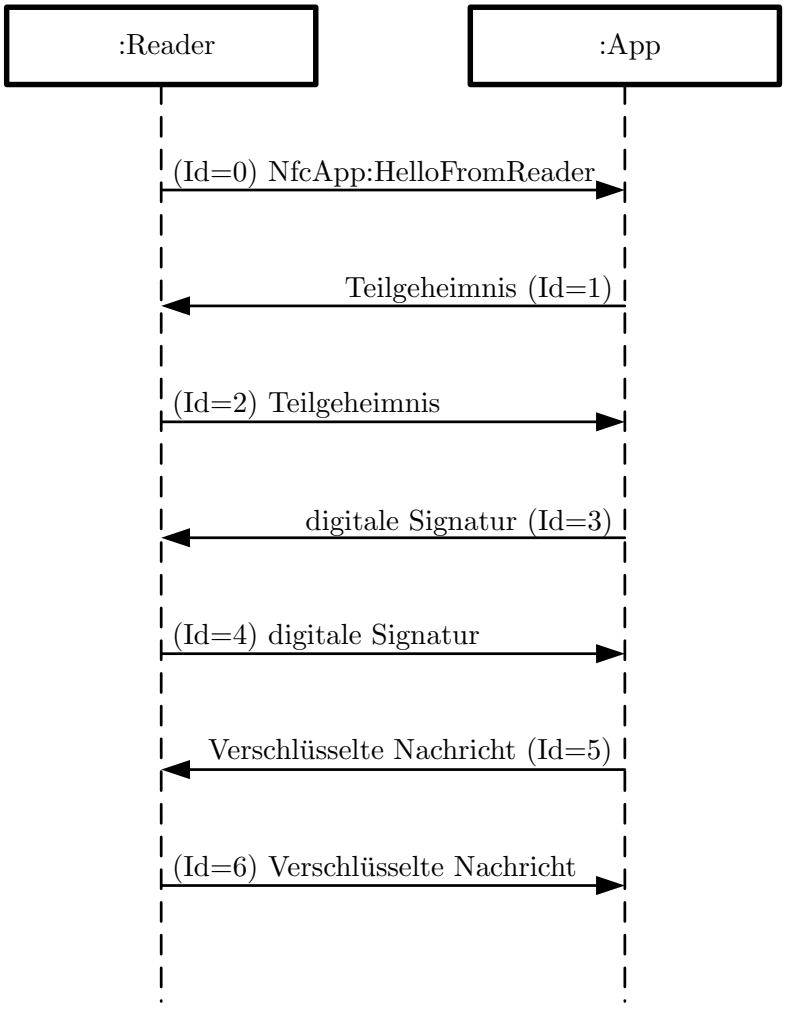

Abbildung 26: Handshake zwischen Reader und der App.

Des Weiteren wird auch gezeigt, *welcher* Inhalt *wann* versendet wird. Die erste Nachricht die versendet wird mit der Sequenznummer 0 ist ausnahmsweise ein `NdefUriRecord` Objekt mit der URI *NfcApp*:*HelloFormReader*. Die vom Reader versendete Nachricht wird vom Windows Phone 8 Betriebssystem abgefangen, bei dem der Anwender zunächst das Öffnen der mit dem URI-Schema verknüpften Anwendungen bestätigen muss. Warum diese Abfrage Sicherheitstechnisch relevant ist, wurde bereits in Kapitel 5.2 erklärt. Die weiteren Nachrichten die versendet werden sind das Teilgeheimnis mit der Sequenznummer 1 und 2. In Sequenznummer 3 und 4 wird die digitale Signatur des zuvor gesendeten Teilgeheimnisses übermittelt. In den letzten beiden Sequenznummern 5 und 6 tauschen beide eine symmetrische verschlüsselte Nachricht mit dem erzeugen geheimen Schlüssel aus. Im weiteren Verlauf wird auf die beteiligten Klassen zum erzeugen des Teilgeheimnisses, digital Signieren des Teilgeheimnisses und dem Verschlüsseln und Entschlüsseln von Daten mit einer Checksumme eingegangen.

7.3 Schlüsselverwaltung

Doch bevor beide Anwendungen Ihre Kommunikation beginnen, wird zunächst ein Schlüsselpaar erzeugt bzw. geladen. Wie schon in Kapitel 6.3.1 erwähnt wurde, handelt es sich beim *Cryptographic Service Provider* um eine Softwarekomponente mit der kryptografische Operationen durchgeführt werden und Schlüsselpaare sicher gespeichert werden können. Sowohl der Reader als auch die App nutzen CSP Container um ihr generiertes Schlüsselpaar sicher zu speichern. In Listing 1 wird gezeigt, wie die App ihr erzeugtes Schlüsselpaar in einem CSP Container abspeichert.

```
1 CspParameters cp = new CspParameters();
2 cp.KeyContainerName = HostInformation.PublisherHostId;
3
4 this.appRsa = new RSACryptoServiceProvider(cp);
```

Listing 1: Speichern und Wiederherstellen eines Schlüsselpaars aus dem CSP.

Mit der Klasse `CspParameters` ist es möglich über die Property *KeyContainerName* einen Namen für den CSP Container anzugeben, in dem das Schlüsselpaar gespeichert wird. Durch Angabe des Names des CSP Containers und Übergabe des `CspParameters` Objekts *cp* an den Konstruktor der `RSACryptoServiceProvider` Klasse, wird automatisch das Schlüsselpaar aus dem angegebenen CSP Container in das Objekt der Klasse `RSACryptoServiceProvider` geladen, sofern sich ein Schlüsselpaar im CSP Container befindet. Andernfalls wird ein neues Schlüsselpaar erzeugt und in dem CSP Container abgespeichert. Der Name für den CSP Container wird in der App über die Property *PublisherHostId* der Klasse `HostInformation` ermittelt. Wie bereits in Kapitel 6.3.1 erklärt wurde, ist unter anderem der Sicherheitsvorteil durch Nutzen dieser Klasse, dass ein Angreifer durch dekompilieren von Code nicht an den Namen des CSP Containers gelangen kann um das Schlüsselpaar zu entwenden.

Um wiederum einen Public Key hardcoden zu können, muss zunächst dieser exportiert werden. Um den Public Key zu exportieren, wird die Methode `ToXmlString()` mit dem Parameter `false` aufgerufen. Durch den Parameter `false` wird angegeben, dass nur der Public Key exportiert werden soll. Der Rückgabewert ist ein `string` , wie Listing 2 zeigt.

```
1 string readerPublicKey = readerRsa.ToXmlString(false);
```

Listing 2: Exportieren eines Public Key.

Dieser Rückgabewert wird dazu genutzt, um mit dem Debugger von Visual Studio den Wert zu entnehmen und diesen in der anderen Anwendung hardzucoden. Nachdem dieser in einer anderen Anwendung hardgecodet wurde, muss dieser aber auch wieder in ein `RSACryptoServiceProvider` Objekt geladen werden. Dazu wird die `FromXmlString()` Methode verwendet, wie in Listing 3 zu sehen ist.

```
1 this.readerRsa.FromXmlString(readerPublicKey);
```

Listing 3: Laden eines hardgecodeten Public Key.

Die `RSACryptoServiceProvider` Klasse aus dem .NET Framework Namespace `System.Security.Cryptography` bietet aber auch weitere Methoden an. So können beispielsweise `SignData()` oder `VerifyData()` genutzt werden um Daten zu Signieren oder digitalen Signaturen zu überprüfen, wie im weiteren Verlauf gezeigt wird.

7.4 Umsetzung der Peer-To-Peer Kommunikation

Um die Peer-To-Peer Kommunikation zwischen dem Reader und der App zu realisieren, wurde die Klasse `Nfc` erstellt. Diese Klasse ist ein Wrapper, um die Klasse `ProximityDevice` aus dem Namespace `Windows.Networking.Proximity` zu kapseln und um eine einfache Bedienung nach außen bereitzustellen. Die `ProximitDevice` Klasse ist in der WinRT und WinPRT verfügbar, sodass der gesamte Code der Klasse `Nfc` *portabel* ist. In Abbildung 27 zunächst ein Klassendiagramm.

Nfc
- FirstMessageToSend : NdefMessage - MessageId : long - ProximityDevice : ProximityDevice - SendsFirstMessage : bool + DeviceDepartedDelegate : DeviceDeparted + MessageReceivedDelegate : MessageReceived
- DeviceArrivedHandler(ProximityDevice) : void - DeviceDepartedHandler(ProximityDevice) : void - MessageReceivedHandler(ProximityDevice, ProximityMessage) : void - MessageTransmittedHandler(ProximityDevice, long) : void + Nfc() + Nfc(NdefMessage) + SendMessage(NdefMessage) : void

Abbildung 27: Klassendiagramm der Klasse Nfc.

Wie erkennbar ist, ist der Konstruktor überladen. D.h. es kann ein Objekt von der Klasse `Nfc` erzeugt werden, durch Übergabe von einem *oder* keinem Parameter. Dies ist ein wichtiger Punkt, von dem sich der Reader von der App unterscheidet. Der Reader *übergibt* einen Parameter, nämlich ein `NdefMessage` Objekt mit einem `NdefUriRecord` mit der URI *NfcApp*:*HelloFromReader*, wie Listing 4 zeigt.

```
1 this.nfc = new Nfc(new NdefMessage() { new NdefUriRecord() {
      Id = id, Uri = "NfcApp:HelloFromReader" } });
```

Listing 4: Erzeugen eines Objekts der Klasse Nfc im Reader.

Durch Übergeben eines `NdefMessage` Objekts wird die Variable `SendsFirstMessage` auf `true` gesetzt. Dadurch wird festgelegt, dass der Reader als erstes beginnt eine Nachricht zu senden. Des Weiteren wird der übergebene Parameter der Klassenvaria-

ble `FirstMessageToSend` zugewiesen. Zudem stellt die Klasse `ProximityDevice` unterschiedliche Events bereit, unter anderem das Event *DeviceArrived*, wenn ein anderes NFC Gerät erkannt wird. In Listing 5 wird dies veranschaulicht.

```
this.ProximityDevice.DeviceArrived += DeviceArrivedHandler;
this.ProximityDevice.DeviceDeparted += DeviceDepartedHandler
    ;
this.ProximityDevice.SubscribeForMessage("NDEF",
    MessageReceivedHandler);
```

Listing 5: Registrieren von Event Handlern bei der Instanz der Klasse ProximityDevice.

Da sowohl Reader als auch die App den *gleichen* Code der Klasse `Nfc` verwenden, werden beide auf das Event *DeviceArrived* reagieren und die Methode `DeviceArrivedHandler()` aufrufen. Wenn aber beide Anwendungen *gleichzeitig* anfangen zu senden, wird die andere Anwendung nichts verstehen können. Daher die Notwendigkeit festzulegen, wer als erstes eine Nachricht senden darf. In Listing 6 wird die `DeviceArrivedHandler()` Methode gezeigt.

```
private void DeviceArrivedHandler(ProximityDevice sender)
{
        if(this.SendsFirstMessage)
        {
                // ...
                this.SendMessage(this.FirstMessageToSend);
        }
}
```

Listing 6: Implementierung des DeviceArrivedHandler Event Handlers.

Im Fall des Readers, überträgt dieser über die `SendMessage()` Methode den beim Erzeugen der Klasse `Nfc` übergebenen Parameter.

Die `SendMessage()` Methode bedient die `PublishBinaryMessage()` Methode der Klasse `ProximityDevice`, wie in Listing 7 ersichtlich ist.

```
public void SendMessage(NdefMessage ndefMessage)
{
        this.MessageId = this.ProximityDevice.
            PublishBinaryMessage("NDEF", ndefMessage.
            ToByteArray().AsBuffer(), this.
            MessageTransmittedHandler);
}
```

Listing 7: Übermitteln einer Nachricht mit der SendMessage Methode.

Als erster Parameter wird angegeben, um *welchen* Nachrichtentyp es sich handelt, wie z.B. *NDEF*. Dieser Nachrichtentyp muss von einem *Empfänger* abonniert werden, damit

beim empfangen einer Nachricht des besagten Typs der Event Handler des Events *MessageReceived* auch aufgerufen wird. Beispielsweise wurde in Listing 5 mit der Methode `SuscribeForMessage()` der Nachrichtentyp *NDEF* definiert und `MessageReceivedHandler()` als Event Handler festgelegt. Als zweiter Parameter wird in der `PublishBinaryMessage` Methode die `IBuffer` Repräsentation von dem erhaltenen Parameter *ndefMessage* übergeben. Also dritter optionaler Parameter wird die `MessageTransmittedHandler()` Methode als Event Handler angegeben. Nachdem die Nachricht übermittelt wurde, wird dieser Event Handler aufgerufen, in dem dann das durch `PublishBinaryMessage()` generierte RF Feld mit der `StopPublishingMessage()` Methode wieder deaktiviert wird. In Listing 8 wird dies verdeutlicht.

```
1 private void MessageTransmittedHandler(ProximityDevice
    sender, long messageId)
2 {
3         // ...
4         sender.StopPublishingMessage(messageId);
5 }
```

Listing 8: Deaktivierung des generierten RF Feld.

Der bisherige Ablauf im Reader lässt sich mit einem Sequenzdiagramm wie folgt in Abbildung 28 zusammenfassen.

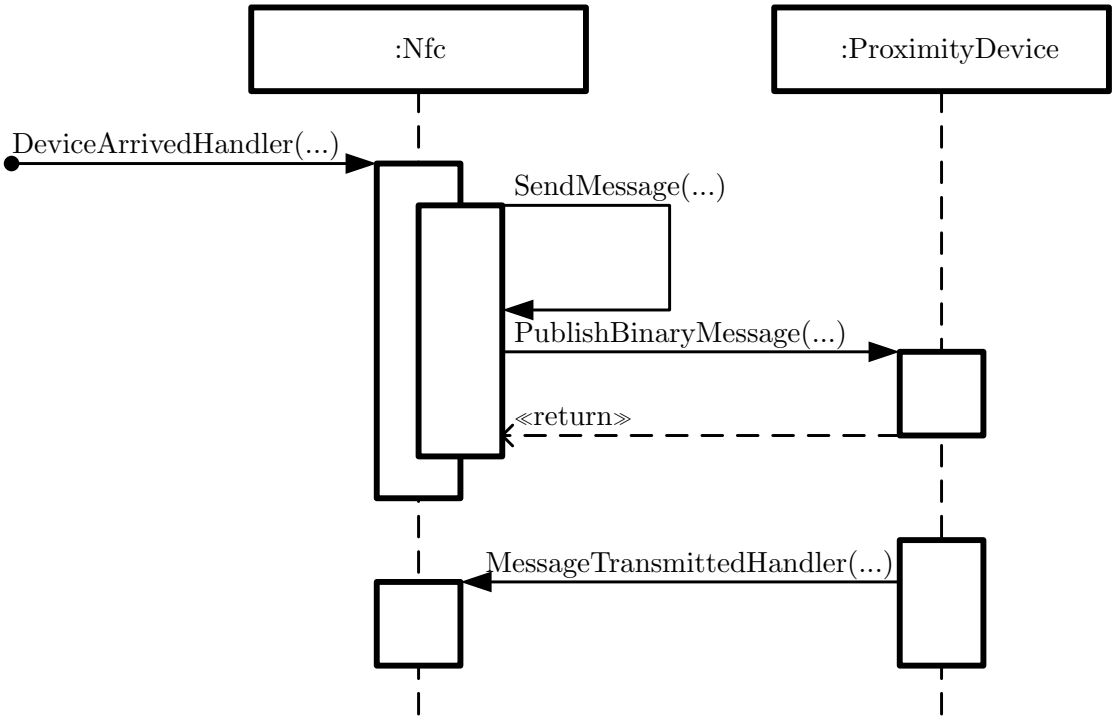

Abbildung 28: Ablauf im Reader bei einem DeviceArrived Event.

Nachdem die Nachricht vom Reader an die App übermittelt wurde, befindet sich die App in der „Aufgabe“ *empfangen*, die ebenso teil der Klasse `Nfc` ist. Beim Erhalt einer Nachricht, die mit dem abonnierten Nachrichtentyp übereinstimmt, wie in Listing 5 festgelegt wurde, wird der `MessageReceivedHandler()` Event Handler für das Event *MessageReceived* aufgerufen. Im Wesentlichen wird in dem Event Handler die erhaltene `IBuffer` Repräsentation von der Property *Data* des Objekts *message* zurück in die `NdefMessage` Repräsentation gewandelt, wie in Listing 9 zu sehen ist.

```
1 private void MessageReceivedHandler(ProximityDevice sender,
    ProximityMessage message)
2 {
3         byte[] rawMessage = message.Data.ToArray();
4         NdefMessage ndefMessage = NdefMessage.FromByteArray(
             rawMessage);
5
6         // ...
7
8         this.MessageReceivedDelegate(ndefMessage);
9 }
```

Listing 9: Empfangen einer Nachricht.

Anschließend wird das `ndefMessage` Objekt dem `MessageReceivedDelegate()` *delegate* übergeben. Mithilfe von delegates ist es möglich unter C# das *Observer Pattern* mit wenig Code umzusetzen. In Abbildung 27 sind zwei öffentlich zugängliche delegates definiert: `MessageReceivedDelegate()` und `DeviceDepartedDelegate()`. Diesen delegates können Methoden zugewiesen werden, die aufgerufen werden, wenn das jeweilige delegate aufgerufen wird. In Listing 10 wird gezeigt, wie ein Event Handler registriert werden kann.

```
1 this.nfc.MessageReceivedDelegate +=
    MessageReceivedFromReaderHandler;
2 this.nfc.DeviceDepartedDelegate += () =>
3 {
4         // ...
5 };
```

Listing 10: App registriert ihre Event Handler.

Analog dazu registriert der Reader die Methode `MessageReceivedFromAppHandler()` als Event Handler für das Event *MessageReceived*. Da die App aber eine Nachricht vom Reader erhalten hat, wird zunächst der `MessageReceivedFromReaderHandler()` Event Handler von der App aufgerufen, um die Nachricht zu *bearbeiten*. Anschließend überträgt die App selbst als *initiator* eine Nachricht an den Reader über die `SendMessage()` Methode. An dieser Stelle sei angemerkt, dass die Klasse `MainPage` in einer Windows Phone 8 App der typische Einstiegspunkt für einen Programmierer ist. Daher befinden

sich die Event Handler in dieser Klasse. Der Ablauf vom *MessageReceived* Event bis zum übertragen einer Nachricht lässt sich in Abbildung 29 zusammenfassen.

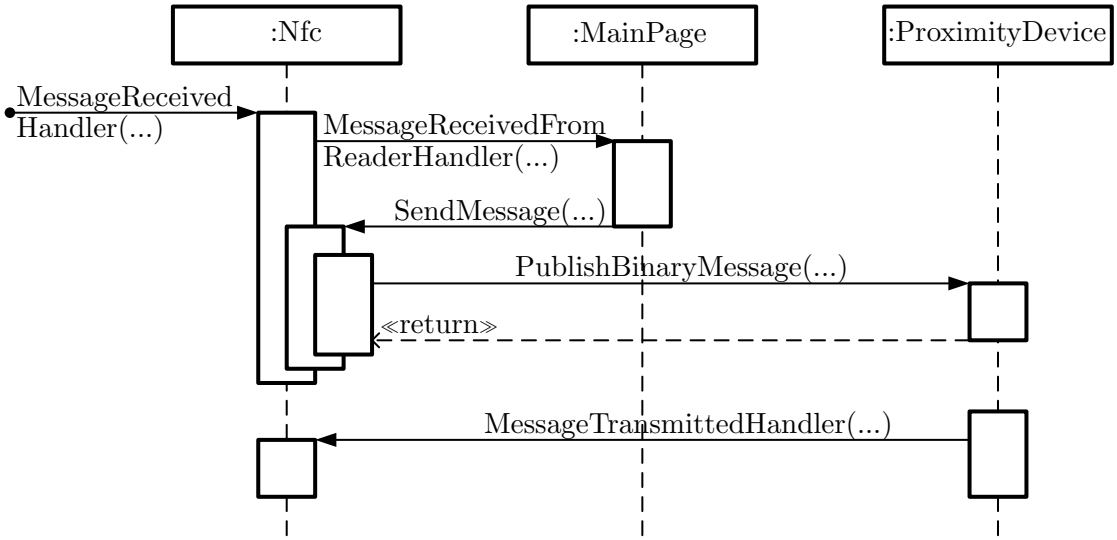

Abbildung 29: Ablauf in der App beim Erhalt einer Nachricht.

Es gibt jedoch einen Sonderfall: Die erste Nachricht die Übertragen wird sendet der Reader. Es wird ein `NdefMessage` Objekt mit einem `NdefUriRecord` Objekt mit dem Inhalt *NfcApp:HelloFromReader* übertragen. Der Sinn dieser URI die Übertragen wird ist lediglich die App zu starten. D.h. die URI wird zunächst vom Windows Phone 8 Betriebssystem abgefangen, um den Nutzer um Erlaubnis zu bitten, die mit dem URI-Schema registrierte App zu öffnen. Beim Öffnen der App werden die Informationen *nach* dem Doppelpunkt, also *HelloFromReader*, der App mitübergeben. Die Konsequenz daraus ist, dass statt der `MessageReceivedFromReaderHandler()` Methode *durch* das Event *MessageReceived*, die `OnNavigatedTo()` Methode mit dem übergebenen Parameter, aufgerufen wird. Um die Logik in der `MessageReceivedFromReaderHandler()` Methode möglichst einfach und Konsistent zu halten, wird innerhalb der `OnNavigatedTo()` Methode das `NdefMessage` Objekt mit dem `NdefUriRecord` Objekt *selbst* erzeugt und der `MessageReceivedFromReaderHandler()` Methode *manuell* übergeben, damit es erscheint, als wurde die Nachricht *von* der App über NFC empfangen. Vor der Übergabe wird beim `NdefUriRecord` Objekt die Property *Id* auf 0 gesetzt, damit die Logik innerhalb von `MessageReceivedFromReaderHandler()` erkennt, in welchem Schritt sich der Reader und die App befinden. In Listing 11 wird dies dargestellt.

```
1 protected override void OnNavigatedTo(NavigationEventArgs e)
2 {
3         if(NavigationContext.QueryString.ContainsKey("
              Message"))
4         {
5                 string message = NavigationContext.
```

```
                    QueryString["Message"];
6
7                   if(message == "HelloFromReader")
8                   {
9                           // ...
10                          byte[] id = { 0 };
11                          this.
                                MessageReceivedFromReaderHandler(
                                new NdefMessage() { new
                                NdefUriRecord() { Id = id, Uri =
                                "NfcApp:" + message } });
12                  }
13          }
14
15          base.OnNavigatedTo(e);
16  }
```

Listing 11: Empfangen der URI-Schema Parameter beim Öffnen der App.

Nach diesem Schritt, beginnt die App damit, ihr Teilgeheimnis an den Reader zu übertragen.

7.5 Übertragung des Teilgeheimnisses

Bevor ein Teilgeheimnis übertragen werden kann, muss dieses zunächst erzeugt werden. Mit der Klasse `EllipticCurveDiffieHellman` ist es möglich, ein Teilgeheimnis zu generieren und beim Erhalt eines Teilgeheimnisses einen geheimen Schlüssel daraus zu erzeugen. In Abbildung 30 wird zunächst ein Klassendiagramm gezeigt mit den verfügbaren Methoden der Klasse.

EllipticCurveDiffieHellman
- keyPair : AsymmetricCipherKeyPair + Teilgeheimnis : byte[]
+ EllipticCurveDiffieHellman() + GetSecretKey(byte[]) : BigInteger

Abbildung 30: Klassendiagramm der Klasse EllipticCurveDiffieHellman.

Wie zu erkennen ist, bietet die Klasse *keine* Methoden an, um die Parameter g und n auszutauschen, wie es beim *klassischen* Diffie-Hellman, wie es in Kapitel 6.1 beschrieben wurde, der Fall ist. Dies ist aber auch nicht Notwendig, da sowohl der Reader als auch die App die *gleiche* `EllipticCurveDiffieHellman` Klasse verwenden. Auf Codeebene werden mit der Bouncy Castle C# Bibliothek, die *gleichen* Parameter in Form von einer elliptischen Kurve erzeugt. Dadurch entfällt die Notwendigkeit diese *Elliptic Curve Domain Parameters* Parameter erst austauschen zu müssen, wie Listing 12 zeigt.

```
1  public EllipticCurveDiffieHellman()
2  {
3          X9ECParameters ecP = NistNamedCurves.GetByName("K
               -163");
4          ECDomainParameters ecSpec = new ECDomainParameters(
               ecP.Curve, ecP.G, ecP.N, ecP.H, ecP.GetSeed());
5
6          ECKeyPairGenerator g = new ECKeyPairGenerator();
7          g.Init(new ECKeyGenerationParameters(ecSpec, new
               SecureRandom()));
8          this.keyPair = g.GenerateKeyPair();
9
10         SubjectPublicKeyInfo pki =
               SubjectPublicKeyInfoFactory.
               CreateSubjectPublicKeyInfo(keyPair.Public);
11         this.Teilgeheimnis = pki.ToAsn1Object().
               GetDerEncoded();
12 }
```

Listing 12: Erzeugen eines Teilgeheimnisses.

Um den Code an dieser Stelle zu erläutern: Mit *K-163* als Übergabeparameter bei der `GetByName()` Methode der Klasse `NistNamedCurves` wird festgelegt, welche elliptische Kurve vom *National Institute of Standards and Technology* (NIST) verwendet werden soll. Anhand dieser elliptischen Kurve werden die *Elliptic Curve Domain Parameters* erzeugt. Um ein Schlüsselpaar zur erzeugen, werden diese Parameter und eine Zufallszahl übergeben. Aus dem resultierende Schlüsselpaar wird der Public Key der öffentlich zugänglichen Property *Teilgeheimnis* zugewiesen. Der Private Key in dem Schlüsselpaar entspricht der geheimen Zahl x bzw. y von Alice und Alice, wie in Kapitel 6.1 erklärt wurde.

Beim *Erhalt* eines Teilgeheimnisses kann der `GetSecretKey()` Methode das Teilgeheimnis übergeben werden, um den geheimen Schlüssel zu erzeugen, wie in Listing 13 zu sehen ist.

```
1  public BigInteger GetSecretKey(byte[]
       empfangenesTeilgeheimnis)
2  {
3          IBasicAgreement keyAgree = AgreementUtilities.
               GetBasicAgreement("ECDH");
4          keyAgree.Init(this.keyPair.Private);
5
6          AsymmetricKeyParameter temp = PublicKeyFactory.
               CreateKey(empfangenesTeilgeheimnis);
7          BigInteger secretKey = keyAgree.CalculateAgreement(
               temp);
```

```
8
9           return secretKey;
10  }
```

Listing 13: Funktionsweise der Methode GetSecretKey() der Klasse EllipticCurveDiffieHellman.

Zunächst wird ein `IBasicAgreement` Objekt erstellt, welches das Diffie-Hellman Schlüsselaustauschverfahren mit elliptischer Kurven repräsentiert. Dieses Objekt wird mit dem Private Key des zuvor im Konstruktor erzeugen Schlüsselpaars initialisiert. Um den geheimen Schlüssel zu erzeugen, wird der Methode `CalculateAgreement()` des `IBasicAgreement` Objekts das empfangene Teilgeheimnis übergeben.

Der praktische Einsatz dieser Klasse wiederspiegelt sich *zunächst* bei den Nachrichten mit der Sequenznummer 1 und 2, wie in Abbildung 26 bereits zu sehen war. Die App empfängt die Nachricht vom Reader mit der Sequenznummer 0. Die Instanz der Klasse `EllipticCurveDiffieHellman` wurde zuvor im Konstruktor der App erzeugt und der Klassenvariable *ecdh* zugewiesen. Mithilfe des *ecdh* Objekts wird auf die Property *Teilgeheimnis* zugegriffen und der Inhalt der Variable *appTeilgeheimnis* zugewiesen. Als nächstes wird ein `NdefMessage` Objekt erzeugt, dem ein `NdefRecord` Objekt mit der *Id = 1* als Sequenznummer und dem *appTeilgeheimnis* als Payload zugewiesen wird. In Listing 14 wird dies veranschaulicht.

```
1  if(ndefMessage[0].Id[0] == 0 && this.currentStage == 0)
2  {
3           byte[] appTeilgeheimnis = this.ecdh.Teilgeheimnis;
4
5           this.currentStage += 2;
6           byte[] B = { 66 };
7           byte[] id = { 1 };
8           this.nfc.SendMessage(new NdefMessage() { new
               NdefRecord() { Id = id, Payload =
               appTeilgeheimnis, Type = B, TypeNameFormat =
               NdefRecord.TypeNameFormatType.ExternalRtd } });
9   }
```

Listing 14: App überträgt sein Teilgeheimnis.

Anschließend wird das Teilgeheimnis der App über die `SendMessage()` Methode an den Reader übertragen. Der Reader empfängt die Nachricht mit der Sequenznummer 1 und speichert den Payload der Nachricht in der Variable *appTeilgeheimnis* ab, wie in Listing 15 zu sehen ist.

```
1  if(ndefMessage[0].Id[0] == 1 && this.currentStage == 1)
2  {
3           this.appTeilgeheimnis = ndefMessage[0].Payload;
4
```

```
5           byte[] readerTeilgeheimnis = this.ecdh.Teilgeheimnis
                ;
6
7           this.currentStage += 2;
8           byte[] B = { 66 };
9           byte[] id = { 2 };
10          this.nfc.SendMessage(new NdefMessage() { new
                NdefRecord() { Id = id, Payload =
                readerTeilgeheimnis, Type = B, TypeNameFormat =
                NdefRecord.TypeNameFormatType.ExternalRtd } });
11  }
```

Listing 15: Reader empfängt das Teilgeheimnis von der App und sendet seins weiter.

Ansonsten sind die Schritte die der Reader anwendet identisch zu denen der App, um *sein* Teilgeheimnis an die App zu übermitteln. Der einzige Unterschied ist jedoch, dass die *Id* der Nachricht 2 anstelle von 1 ist. Sobald die App das Teilgeheimnis des Readers erhält, muss die App die digitale Signatur ihres Teilgeheimnisses erzeugen und übermitteln.

7.6 Digitales signieren der Teilgeheimnisse

Teilgeheimnisse müssen digital signiert werden, um sich vor *aktiven* Man-In-The-Middle Angriffen zu schützen, wie in Kapitel 6.1 gezeigt wurde. Um digitale Signaturen zur erstellen und zu validieren, werden die Methoden `SignData()` bzw. `VerifyData()` der Klasse `RSACryptoServiceProvider` aus dem .NET Framework Namespace `System.Security.Cryptography` verwendet.

Nachdem die App eine Nachricht vom Reader mit der Sequenznummer 2 erhält, speichert die App den empfangenen Payload mit dem Teilgeheimnis in der Variable *readerTeilgeheimnis* ab, wie in Listing 16 verdeutlicht wird.

```
1   if(ndefMessage[0].Id[0] == 2 && this.currentStage == 2)
2   {
3           this.readerTeilgeheimnis = ndefMessage[0].Payload;
4
5           byte[] appSignature = this.appRsa.SignData(this.ecdh
                .Teilgeheimnis, new SHA256Managed());
6
7           this.currentStage += 2;
8           byte[] B = { 66 };
9           byte[] id = { 3 };
10          this.nfc.SendMessage(new NdefMessage() { new
                NdefRecord() { Id = id, Payload = appSignature,
                Type = B, TypeNameFormat = NdefRecord.
                TypeNameFormatType.ExternalRtd } });
```

```
11 }
```

Listing 16: App empfängt das Teilgeheimnis vom Reader und sendet ihre digitale Signatur weiter.

Anschließend wird das Objekt *appRsa*, die das Schlüsselpaar der App repräsentiert wie in Kapitel 7.3 gezeigt wurde, dazu verwendet um mit der `SignData()` Methode das Teilgeheimnis der App zu signieren. Dazu wird das Teilgeheimnis und eine Instanz der Klasse `SHA256Managed`, die als Hashfunktion verwendet wird, übergeben. Der Rückgabewert der `SignData()` Methode wird in dem Byte-Array *appSignature* zugewiesen. Danach wird ein `NdefMessage` Objekt erzeugt, dem ein `NdefRecord` Objekt hinzugefügt wird. Dem `NdefRecord` Objekt wird die *Id = 3* zugewiesen sowie der Property *Payload* der Inhalt der Variable *appSignature*. Anschließend wird das `NdefMessage` Objekt mit der `SendData()` Methode an den Reader übertragen.

Der Reader erhält die Nachricht mit der Sequenznummer 3 von der App und fügt zunächst den Inhalt des empfangenen Payloads der Variable *appSignature* hinzu, wie in Listing 17 gezeigt wird.

```
1  if(ndefMessage[0].Id[0] == 3 && this.currentStage == 3)
2  {
3          this.appSignature = ndefMessage[0].Payload;
4
5          bool isValid = this.appRsa.VerifyData(this.
               appTeilgeheimnis, new SHA256Managed(), this.
               appSignature);
6          if(isValid)
7          {
8                  byte[] readerSignature = this.readerRsa.
                       SignData(this.ecdh.Teilgeheimnis, new
                       SHA256Managed());
9
10                 this.currentStage += 2;
11                 byte[] B = { 66 };
12                 byte[] id = { 4 };
13                 this.nfc.SendMessage(new NdefMessage() { new
                        NdefRecord() { Id = id, Payload =
                       readerSignature, Type = B, TypeNameFormat
                        = NdefRecord.TypeNameFormatType.
                       ExternalRtd } });
14         }
15         else
16         {
17                 this.currentStage = 1;
18                 Console.WriteLine("Invalid Checksum\n");
19                 Debug.WriteLine("Invalid Checksum");
```

```
20          }
21 }
```

Listing 17: Reader überprüft die empfangene digitale Signatur der App und sendet ggf. seine weiter.

Als nächster überprüft der Reader die Gültigkeit der digitalen Signatur der App. Da in Kapitel 7.3 erklärt wurde, dass in dieser Implementierung der Public Key vom Reader in der App hardgecodet ist und umgekehrt, ist es an dieser Stelle für den Reader möglich, die Gültigkeit der digitalen Signatur der App zu überprüfen. Dazu wird die Methode `VerifyData()` des Objekts *appRsa* aufgerufen. Als Parameter werden das Teilgeheimnis der App, eine Instanz der Klasse `SHA256Managed` und die empfangene digitale Signatur der App übergeben. Ist der Rückgabewert `true`, setzt der Reader die Kommunikation fort, da die Verifikation der digitalen Signatur positiv verlaufen ist. Als nächstes führt der Reader dieselben Schritte wie die App durch: Er Erzeugt eine digitale Signatur von seinem Teilgeheimnis mit SHA256 als Hashfunktion und überträgt diese Nachricht mit der $Id = 4$ mit der `SendMessage()` Methode an die App.

Die App empfängt die Nachricht mit der Sequenznummer 4 und weist den erhaltenen Payload der Variable *readerSignature* zu. In Listing 18 wird dies verdeutlicht.

```
1  if(ndefMessage[0].Id[0] == 4 && this.currentStage == 4)
2  {
3          this.readerSignature = ndefMessage[0].Payload;
4
5          bool isValid = this.readerRsa.VerifyData(this.
               readerTeilgeheimnis, new SHA256Managed(), this.
               readerSignature);
6          if(isValid)
7          {
8                  // ...
9          }
10         // ...
11 }
```

Listing 18: App überprüft die empfangene digitale Signatur des Readers.

Ebenso wie der Reader, überprüft die App die empfangene digitale Signatur auf Gültigkeit und setzt bei positiver Validierung die Kommunikation verschlüsselt fort.

7.7 Übertragung einer verschlüsselten Nachricht

Um die *Vertraulichkeit* von zu übertragenen Daten zu wahren, müssen diese verschlüsselt werden. Daher wurde die Klasse `Aes` umgesetzt, die ein Wrapper um die `AesManaged` Klasse aus dem .NET Framework Namespace `System.Security.Cryptography` ist. Die Verschlüsselung die dabei zum Einsatz kommt ist *Advanced Encryption Standard* (AES), wie der Name schon vermuten lässt. In Abbildung 31 wird zunächst in Klassendiagramm gezeigt.

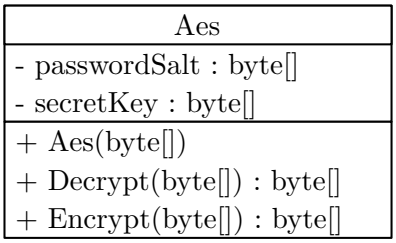

Abbildung 31: Klassendiagramm der Klasse Aes.

Wie zu erkennen ist, beschränkt sich die Klasse auf *zwei* wesentliche Methoden: `Encrypt()` und `Decrypt()`, die die Daten mit dem an den Konstruktor übergebenen *geheimen Schlüssel* verschlüsseln bzw. entschlüsseln. Dem Konstruktor wird ein Byte-Array als geheimer Schlüssel übergeben, wie in Listing 19 zu sehen ist.

```
1  public Aes(byte[] secretKey)
2  {
3          this.secretKey = secretKey;
4          string temp = null;
5
6          for(int i = 0; i < this.secretKey.Length; i += 3)
7          {
8                  temp += this.secretKey[i];
9          }
10
11         this.passwordSalt = Encoding.UTF8.GetBytes(temp);
12 }
```

Listing 19: Konstruktor der Klasse Aes.

Der geheime Schlüssel wird der Klassenvariable *secretKey* zugewiesen. Aus dem *secretKey* wird dann ein *Salt* gebildet, indem jede dritte Stelle des Byte-Arrays beginnend mit der Position 0 einem `string` hinzugefügt wird. Anschließend wird dieser `string` der `GetBytes()` Methode aus dem .NET Framework Namespace `System.Text.Encoding.UTF8` übergeben und der Rückgabewert der Klassenvariable *passwordSalt* zugewiesen.

Mit der `Encrypt()` Methode wird ein übergebenes Byte-Array mithilfe der Variable *secretKey* und *passwordSalt* verschlüsselt, wie in Listing 20 zu sehen ist.

```
1  public byte[] Encrypt(byte[] message)
2  {
3          using(AesManaged aes = new AesManaged())
4          {
5                  Rfc2898DeriveBytes deriveBytes = new
                      Rfc2898DeriveBytes(secretKey,
                      passwordSalt, 1000);
6                  aes.Key = deriveBytes.GetBytes(aes.KeySize /
                       8);
```

```
7                   aes.IV = deriveBytes.GetBytes(aes.BlockSize
                        / 8);
8
9                   using(MemoryStream ms = new MemoryStream())
10                  {
11                          using(CryptoStream cs = new
                                CryptoStream(ms, aes.
                                CreateEncryptor(),
                                CryptoStreamMode.Write))
12                          {
13                                  cs.Write(message, 0, message
                                        .Length);
14                          }
15
16                          return ms.ToArray();
17                  }
18          }
19  }
```

Listing 20: Verschlüsseln der Daten mit der Encrypt() Methode.

Zunächst wird eine Instanz der Klasse `AesManaged` erzeugt. Mithilfe der Instanz der Klasse `Rfc2898DeriveBytes` ist es möglich aus einem *geheimen Schlüssel*, einem *Salt* und einer *Iterationsanzahl* einen Hash zu erzeugen. Anschließend werden aus dem Objekt der Klasse `Rfc2898DervieBytes` mit der Methode `GetBytes()` der *Key* und *IV* Property der Instanz der Klasse `AesManaged` ein neuer Wert zugewiesen. Dadurch, dass der Property *IV* ein Wert zugewiesen wird, arbeitet die Klasse im *Cipher-block chaining* (CBC) Modus. Als nächstes wird ein Objekt vom Typ `MemoryStream` mit dem Variablennamen *ms* und ein Objekt vom Typ `CryptoStream` mit dem Variablennamen *cs* erzeugt. Im Konstruktor der Klasse `CryptoStream` wird das Objekt *ms* als *Zieldatenstream* angegeben und mit der Methode `CreateEncryptor()` des Objekts *aes* die Logik zum Verschlüsseln der Daten übergeben. Anschließend wird die Übergebene Variable *message* mithilfe der Methode `Write()` des Objekts *cs* in das Objekt *ms* geschrieben. Danach wird die Byte-Array Repräsentation des Objekts *ms* zurückgegeben.

Die Funktionsweise der `Decrypt()` Methode ist nahezu identisch zu der Funktionsweise der `Encrypt()` Methode. Der Unterschied ist jedoch, dass *statt* der `CreateEncryptor()` Methode die `CreateDecryptor()` Methode, die die Logik zum Entschlüsseln der Daten enthält, aufgerufen wird. In Listing 21 wird der Unterschied gezeigt.

```
1  public byte[] Decrypt(byte[] encryptedMessage)
2  {
3          using(AesManaged aes = new AesManaged())
4          {
5                   Rfc2898DeriveBytes deriveBytes = new
                        Rfc2898DeriveBytes(secretKey,
```

```
                    passwordSalt, 1000);
6                   aes.Key = deriveBytes.GetBytes(aes.KeySize /
                        8);
7                   aes.IV = deriveBytes.GetBytes(aes.BlockSize
                        / 8);
8
9                   using(MemoryStream ms = new MemoryStream())
10                  {
11                          using(CryptoStream cs = new
                                CryptoStream(ms, aes.
                                CreateDecryptor(),
                                CryptoStreamMode.Write))
12                          {
13                                  cs.Write(encryptedMessage,
                                        0, encryptedMessage.
                                        Length);
14                          }
15
16                          return ms.ToArray();
17                  }
18          }
19  }
```

Listing 21: Entschlüsseln der Daten mit der Decrypt() Methode.

Doch Verschlüsselung alleine reicht nicht aus. Zwar kann die Vertraulichkeit gewahrt werden, aber *nicht* die Integrität. Um die *Ingerität* der Daten zu wahren, muss den Daten eine Checksumme hinzugefügt werden vor dem verschlüsseln. Daher wurde die Klasse `Checksum` mit den Methoden `GetBytesWithChecksum()`, `IsValid()` und `GetOriginalData()` umgesetzt, wie in Abbildung 32 zu sehen ist.

Checksum
+ GetBytesWithChecksum(byte[]) : byte[]
+ GetBytesWithChecksum(string) : byte[]
+ GetOriginalData(byte[]) : byte[]
+ IsValid(byte[]) : bool

Abbildung 32: Klassendiagramm der Klasse Checksum.

Mit der `GetBytesWithChecksum()` Methode erhält der Anwender ein Byte-Array als Rückgabewert von seinen übergeben Daten. Dieses Byte-Array enthält die Nutzdaten, den Hashwert der Nutzdaten und die Länge des Hashwerts. In Abbildung 33 wird dies verdeutlicht.

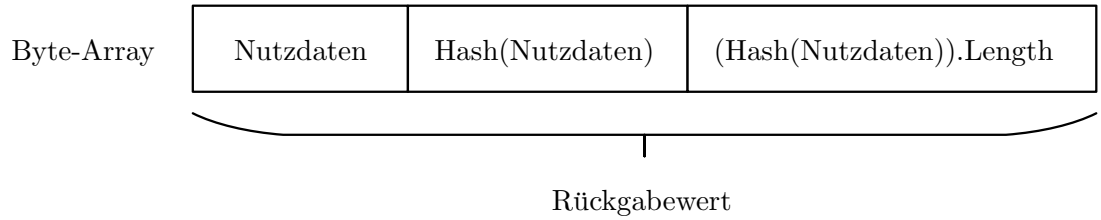

Abbildung 33: Aufbau des Rückgabewerts des Methode GetBytesWithChecksum().

Von den Nutzdaten wird ein Hash mit der Hashfunktion SHA256 berechnet. Im Fall, dass die Nutzdaten ein `string` statt dem erwarteten Byte-Array sind, wird die überladene Methode aufgerufen die einen `string` akzeptiert und den den `string` in ein Byte-Array konvertiert. Die Nutzdaten sowie der errechnete Hash wird einem neuen Byte-Array hinzugefügt. An der letzten Position des Byte-Arrays wird die Länge des Hashwerts hinzugefügt, wie in Listing 22 zu sehen ist.

```
1  public byte[] GetBytesWithChecksum(string message)
2  {
3          return this.GetBytesWithChecksum(Encoding.UTF8.
               GetBytes(message));
4  }
5
6  public byte[] GetBytesWithChecksum(byte[] data)
7  {
8          SHA256Managed sha = new SHA256Managed();
9          byte[] checksum = sha.ComputeHash(data);
10
11         int size = data.Length + checksum.Length + 1;
12
13         byte[] dataWithChecksum = new byte[size];
14         data.CopyTo(dataWithChecksum, 0);
15         checksum.CopyTo(dataWithChecksum, data.Length);
16         dataWithChecksum[size - 1] = (byte)checksum.Length;
17
18         return dataWithChecksum;
19 }
```

Listing 22: Funktionsweise der GetBytesWithChecksum() Methode.

Anschließend wird dieses Byte-Array zurückgegeben. Dieser Rückgabewert kann dann verschlüsselt und übertragen werden. Der Empfänger entschlüsselt die Daten und möchte dann zunächst mit der `IsValid()` Methode die Gültigkeit validieren. Dazu wird von den Nutzdaten ein Hash mit der Hashfunktion SHA256 gebildet und dieser mit dem empfangen Hash in dem Byte-Array verglichen. Sind beide identisch, gibt die Methode `true` zurück, wie in Listing 23 gezeigt wird.

```
1  public bool IsValid(byte[] bytesWithChecksum)
2  {
3          int checksumLength = (int)bytesWithChecksum.
               LastOrDefault();
4          byte[] data = this.GetOriginalData(bytesWithChecksum
               );
5
6          byte[] checksum = new byte[checksumLength];
7          int checksumBegin = data.Length;
8          int checksumEnd = checksumBegin + checksumLength;
9
10         int j = checksumBegin;
11         int k = 0;
12         for(; j < checksumEnd; j++, k++)
13         {
14                 checksum[k] = bytesWithChecksum[j];
15         }
16
17         SHA256Managed sha = new SHA256Managed();
18         byte[] calcChecksum = sha.ComputeHash(data);
19
20         if(checksum.Length == calcChecksum.Length)
21         {
22                 for(int i = 0; i < checksum.Length; i++)
23                 {
24                         if(checksum[i] != calcChecksum[i])
25                         {
26                                 return false;
27                         }
28                 }
29         }
30
31         return true;
32 }
```

Listing 23: Funktionsweise der IsValid() Methode.

Nachdem die Prüfung der Gültigkeit der Daten positiv verlaufen ist, möchte der Anwender wieder die ursprünglichen Daten haben. Daher wird der `GetOriginalData()` Methode das zuvor entschlüsselte Byte-Array übergeben. In der Methode wird anhand der angegebenen Länge des Hashwerts sowohl der Hashwert als auch der Wert der Länge aus dem Byte-Array entfernt um *nur* die Nutzdaten zu erhalten. Diese werden dann zurückgegeben. In Listing 24 wird dies veranschaulicht.

```
1 public byte[] GetOriginalData(byte[] bytesWithChecksum)
2 {
3         int checksumLength = (int)bytesWithChecksum.
             LastOrDefault();
4         int dataLength = bytesWithChecksum.Length -
             checksumLength - 1;
5
6         byte[] data = new byte[dataLength];
7         for(int i = 0; i < dataLength; i++)
8         {
9                 data[i] = bytesWithChecksum[i];
10        }
11
12        return data;
13 }
```

Listing 24: Funktionsweise der GetOriginalData() Methode.

Nachdem die App die Nachricht mit der Sequenznummer 4 vom Reader erhalten hat und die digitale Signatur vom Reader erfolgreich auf Gültigkeit überprüft werden konnte, überträgt die App eine AES-verschlüsselte Nachricht an den Reader. Welche Schritte dabei durchgeführt werden müssen, wird in Listing 25 gezeigt.

```
1 if(ndefMessage[0].Id[0] == 4 && this.currentStage == 4)
2 {
3         this.readerSignature = ndefMessage[0].Payload;
4
5         bool isValid = this.readerRsa.VerifyData(this.
             readerTeilgeheimnis, new SHA256Managed(), this.
             readerSignature);
6         if(isValid)
7         {
8                 this.secretKey = this.ecdh.GetSecretKey(this
                     .readerTeilgeheimnis);
9                 this.aes = new Aes(this.secretKey.
                     ToByteArray());
10
11                // ...
12
13                Checksum c = new Checksum();
14                byte[] dataWithChecksum = c.
                     GetBytesWithChecksum("Encrypted Hello
                     from App!");
15                byte[] encryptedMessage = this.aes.Encrypt(
                     dataWithChecksum);
```

```
16
17                  this.currentStage += 2;
18                  byte[] B = { 66 };
19                  byte[] id = { 5 };
20                  this.nfc.SendMessage(new NdefMessage() { new
                        NdefRecord() { Id = id, Payload =
                       encryptedMessage, Type = B,
                       TypeNameFormat = NdefRecord.
                       TypeNameFormatType.ExternalRtd } });
21          }
22          else
23          {
24                  this.currentStage = 0;
25                  Debug.WriteLine("Key Agreement Failed");
26          }
27  }
```

Listing 25: App überträgt eine verschlüsselte Nachricht.

Zunächst wird mit der Instanz der Klasse `EllipticCurveDiffieHellman` der *geheime Schlüssel* erzeugt, indem die `GetSecretKey()` Methode mit dem Teilgeheimnis vom Reader als Parameter aufgerufen wird. Dieser geheime Schlüssel wird der Klassenvariable *secretKey* zugewiesen. Anschließend wird eine Instanz der Klasse `Aes` mit der Klassenvariable *secretKey* als Parameter erzeugt. Vor dem verschlüsseln der Daten muss aber erst eine Checksumme an die Daten angehängt werden. Aus diesem Grund wird eine Instanz der Klasse `Checksum` erstellt. Aufgerufen wird die Methode `GetBytesWithChecksum()` der Klasse mit dem `string` *Encrypted Hello from App!* als Parameter. Der Rückgabewert dieser Methode wird dann anschließend der `Encrypt()` Methode der Instanz der Klasse `Aes` übergeben. Der Wert, den die Methode `Encrypt()` zurückgibt wird der Variable *encryptedMessage* zugewiesen. Wie auch bisher, wird dann ein `NdefMessage` Objekt erzeugt. In dieses `NdefMessage` Objekt wird dann ein `NdefRecord` hinzugefügt, dem zuvor der Inhalt der Variable *encryptedMessage* der Property *Payload* zugewiesen wurde sowie die *Id* auf 5 festgelegt wurde. Anschließend wird das `NdefMessage` Objekt über die `SendMessage()` Methode an den Reader übertragen.

Empfängt der Reader die Nachricht mit der Sequenznummer 5, erzeugt der Reader zunächst mit der `GetSecretKey()` Methode der Klasse `EllipticCurveDiffieHellman` den *geheimen Schlüssel*, durch Übergabe des Teilgeheimnisses der App als Parameter. Der Rückgabewert wird der Klassenvariable *secretKey* zugewiesen, wie in Listing 26 zu sehen ist.

```
1 if(ndefMessage[0].Id[0] == 5 && this.currentStage == 5)
2 {
3           this.secretKey = this.ecdh.GetSecretKey(this.
                appTeilgeheimnis);
4           Console.WriteLine("Secret Key: {0}", this.secretKey)
```

```
            ;
5
6           this.aes = new Aes(this.secretKey.ToByteArray());
7           Checksum c = new Checksum();
8
9           byte[] dataWithChecksumReceived = this.aes.Decrypt(
               ndefMessage[0].Payload);
10          string message = null;
11          if(c.IsValid(dataWithChecksumReceived))
12          {
13                  byte[] messageBytes = c.GetOriginalData(
                       dataWithChecksumReceived);
14                  message = Encoding.UTF8.GetString(
                       messageBytes);
15          }
16          else
17          {
18                  message = "Invalid Checksum";
19          }
20          Console.WriteLine(message + Environment.NewLine);
21          Debug.WriteLine(message);
22
23  // ...
24  }
```

Listing 26: Reader überprüft die Gültigkeit der digitalen Signatur der App.

Anschließend erzeugt der Reader eine Instanz der Klasse `Aes`, der er die Klassenvariable *secretKey* als Parameter übergibt. Der `Decrypt()` Methode wird die erhaltene Nachricht von der App als Parameter übergeben um die entschlüsselte Nachricht als Rückgabewert zu erhalten. Der Rückgabewert wird der Variable *dataWithChecksumReceived* zugewiesen. Die Variable *dataWithChecksumReceived* wird der Methode `IsValid()` der Instanz der Klasse `Checksum` übergeben, um zu überprüfen, ob die Daten während der Übertragung manipuliert wurden. Verläuft die Überprüfung positiv, wird anschließend mit der `GetOrginalData()` Methode durch Übergabe der Variable *dataWithChecksumReceived* die ursprüngliche Nachricht als Byte-Array wiederhergestellt. Um aus dem Byte-Array die originale Nachricht zu rekonstruieren, wird das Byte-Array der Methode `GetString()` aus dem .NET Framework Namespace `System.Text.Encoding.UTF8` übergeben.

Danach führt der Reader dieselben Schritte durch wie die App, um *seine* Nachricht mit einer Checksumme und Verschlüsselung zu übermitteln, wie in Listing 27 gezeigt wird.

```
1 if(ndefMessage[0].Id[0] == 5 && this.currentStage == 5)
2 {
```

```
3           // ...
4           byte[] dataWithChecksum = c.GetBytesWithChecksum("
               Encrypted Hello from Reader!");
5           byte[] encryptedMessage = this.aes.Encrypt(
               dataWithChecksum);
6
7           this.currentStage += 2;
8           byte[] B = { 66 };
9           byte[] id = { 6 };
10          this.nfc.SendMessage(new NdefMessage() { new
               NdefRecord() { Id = id, Payload =
               encryptedMessage, Type = B, TypeNameFormat =
               NdefRecord.TypeNameFormatType.ExternalRtd } });
11  }
```

Listing 27: Reader überträgt eine verschlüsselte Nachricht.

Der Reader unterscheidet sich von der App lediglich darin, dass er die Nachricht *Encrypted Hello from Reader!* statt *Encrypted Hello from App!* mit einer Checksumme versieht und verschlüsselt. Ebenso wird die Nachricht mit der *Id = 6* statt mit der Id = 5 versendet.

8 Zusammenfassung und Ausblick

Die in dieser Arbeit festgelegten Ziele wurden größtenteils umgesetzt. Zunächst wurde die Sicherheit von NFC auf unterster Ebene betrachtet. Dabei wurde festgestellt das NFC keinerlei Schutzmaßnahmen gegen Angreifer vorsieht. Zwar gibt es Angriffsszenarien die aus physikalischen Gründen nicht realisiert werden können, dennoch muss auf höherer Ebene die Kommunikation abgesichert werden. An dieser Stelle kam die Frage auf, welche Schutzmaßnahmen Smartphone Betriebssysteme anbieten. Smartphone Betriebssystemhersteller wie Microsoft mit Windows Phone 8 und Google mit Android 4.3 bemühen sich, eine sichere und vertrauenswürdige Plattform anzubieten. Allerdings muss davon ausgegangen werden, dass Smartphone Betriebssysteme kompromittierbar sind und daher vertrauenswürdige Informationen nicht auf dem lokalen Dateisystem der App gespeichert werden sollten. Des Weiteren erlaubt das Smartphone Betriebssystem Android 4.3 die Installation von Apps aus Fremdquellen, was zusätzlich ein Sicherheitsrisiko für den Anwender darstellt. Dargelegt wurde dies anhand des Eurograbber Angriffs, welcher die vermeintlich sichere Zwei-Faktor-Authentifizierung *ad absurdum* führt. Mithilfe eines Trojaners am PC, am Smartphone und durch geschicktem Social Engineering gelingt es den Angreifern Geld von den Konten der betroffenen Personen zu entwenden. Dadurch ist ersichtlich, dass der Mensch selbst die größte Sicherheitsschwachstelle ist. Aus diesem Grund ist es für einen Angreifer auch einfacher den Mensch als solches zu überwinden statt nach Sicherheitslücken in Betriebssystemen zu suchen. In diesem Kontext wurde die Android App NFCProxy vorgestellt, mit der es möglich ist zwei NFC Geräte über einen Proxy miteinander kommunizieren zu lassen. Als Beispiel wurde Pull Printing genannt, welches einen gesendeten Druckauftrag erst ausdruckt, sobald sich der Anwender mit seiner Smartcard am lokalen Drucker authentifiziert hat. Durch Social Engineering und der NFCProxy App kann ein Relay-Angriff durchgeführt werden, um die vertraulichen Dokumente ausdrucken zu lassen. Das grundsätzliche Problem ist dabei, dass solche Systeme Benutzerfreundlichkeit statt Sicherheit vorziehen. Es würde zunächst ausreichen neben dem Faktor „Besitz“ den Faktor „Wissen“ für zumindest grundlegende Sicherheit einzuführen, indem beim Anfordern des Druckauftrags z.B. ein 4-stelliger Code eingegeben werden muss. Voraussetzung hierfür ist, dass ein Man-In-The-Middle den Code nicht erfährt. Doch wie im Implementierungsansatz beschrieben wurde, reicht ein 4-stelliger Code alleine nicht zwangsweise aus, da diese anfällig gegen Bruteforce-Angriffe sind. Um solchen Angriffen entgegenzuwirken ist es ratsam z.B. die Gültigkeit der Codes zeitlich zu begrenzen und nach einer bestimmten Anzahl gescheiterter Authentifizierungsversuche eine Sperrfrist einzuführen. Daraufhin wurde das Schlüsselaustauschverfahren Diffie-Hellman mit elliptischen Kurven vorgestellt. Mit diesem Schlüsselaustauschverfahren ist es möglich einen geheimen Schlüssel zu erzeugen ohne dass dabei der eigentliche Schlüssel über die Kommunikationsverbindung ausgetauscht werden muss. Dazu werden Teilgeheimnisse ausgetauscht aus denen der geheime Schlüssel erzeugt werden kann. Das Problem ist jedoch, dass ein aktiver Man-In-The-Middle in der Lage ist zwei Mal den Schlüsselaustausch jeweils mit dem Reader und der App durchzuführen. Um dieses Problem zu umgehen, müssen die Teilgeheimnisse mit einer digitalen Signatur versehen werden und bei Erhalt auf Gültigkeit geprüft werden.

Damit die digitale Signatur der App überprüft werden kann, benötigt der Reader den Public Key der App. In dieser Arbeit wurde ein Ansatz vorgestellt, bei dem die App ihren Public Key sicher an den Reader übertragen kann, damit dieser die App als vertrauenswürdigen Client vormerken kann. Um das zu realisieren, muss dazu der Public Key vom Reader in der App hardgecodet werden. Andernfalls müsste dieser übertragen werden und es bestünde die Gefahr, dass ein aktiver Man-In-The-Middle seinen Public Key weiterleitet. Ein weiteres Problem an dieser Stelle ist die Prämisse, dass Smartphone Betriebssysteme kompromittierbar sind und dadurch Quellcode dekompiliert werden kann. Wenn ein Angreifer den Public Key des Readers erfahren sollte, ist dieser in der Lage selbst Nachrichten an den Reader zu senden und kann durch einen Bruteforce-Angriff versuchen den Code zu erraten. Allerdings kann die Gefahr eines Bruteforce-Angriffs in Grenzen gehalten werden, wie bereits erwähnt wurde. In der Implementierung wird auf das anfängliche Übertragen des Public Key der App verzichtet, indem der Public Key der App im Reader hardgecodet wird. Dennoch wird der Schlüsselaustausch über NFC mit Elliptic Curve Diffie-Hellman durchgeführt und anschließend eine Textnachricht mit AES-Verschlüsselung ausgetauscht. An dieser Stelle sei angemerkt, dass in der Implementierung auf eine Code-Abfrage, um Relay-Angriffe zu verhindern, verzichtet wird. Daher wird empfohlen, bei sicherheitskritischen Anwendungen, vor einer Transaktion einen Code abzufragen. Des Weiteren wird in der Implementierung auf den Cryptographic Service Provider zur sicheren Speicherung des Schlüsselpaars gesetzt. Allerdings ist dies eine Betriebssystem-API von Windows und Windows Phone und ist somit auf anderen Plattformen nicht verfügbar. Ein anderer, denkbarer Ansatz wäre, ein Secure Element zu verwenden, welches für die Erzeugung und sichere Speicherung von Schlüsselpaaren und anderen sicherheitskritischen Informationen konzipiert worden ist. Allerdings ist ein Problem an dieser Stelle, das der Zugriff auf das Secure Element von den Betriebssystemherstellern kontrolliert wird und nicht für jeden Entwickler frei verfügbar ist. Des Weiteren muss das Secure Element Hardwaremodul in dem eingesetzten Smartphone auch verfügbar sein. An dieser Stelle muss noch Untersucht werden, welche Möglichkeiten andere Plattformen zur sicheren Speicherung von Schlüsselpaaren anbieten und ob ein interoperabler Ansatz realisierbar wäre. Falls es ermöglicht werden sollte, dass ein Schlüsselpaar sicher auf dem lokalen Dateisystem der App gespeichert werden kann, kann auf das hardcoden des Public Key verzichtet werden, indem sowohl der Reader als die App ein TrustCenter verwenden um beim Erhalt eines Public Key die Vertrauenswürdigkeit prüfen zu können.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[Abd11] M. M. Abd Allah, *Strengths and Weaknesses of Near Field Communication (NFC) Technology*, Global Journal of Computer Science and Technology, Volume 11, Issue 3, Version 1.0, March 2011

[And08] R. Anderson, *Security Engineering – A Guide to Building Dependable Distributed Systems*, Second Edition, Wiley, S. 356-357, 2008

[Cer13] CERN Computer Security Team, How to keep secrets secret (Alternatives to Hardcoding Passwords), `https://security.web.cern.ch/security/recommendations/en/password_alternatives.shtml`, 2013, Zuletzt zugegriffen am 18.09.2013

[DH76] W. Diffie, M. E. Hellman, *New Directions in Cryptography*, IEEE Transactions on Information Theory, Vol. IT-22, No. 6, pp. 644-654, November 1976

[EC13] D. Evdokimov, A. Chasovskikh, Windows Phone 8 application security, Hack In Paris 2013, June 17-21, 2013

[Gir13] D. Giry, Keylength – NIST Report on Cryptographic Key Length and Cryptoperiod (2012), `http://www.keylength.com/en/4/`, June 2, 2013, Zuletzt zugegriffen am 18.09.2013

[Git12] GitHub, Bouncy Caste C# Distribution (Mirror), `https://github.com/bcgit/bc-csharp`, Zuletzt zugegriffen am 19.09.2013

[Goo] Google, Permissions, `http://developer.android.com/guide/topics/security/permissions.html`, Zuletzt zugegriffen am 18.09.2013

[HB06] E. Haselsteiner, K. Breitfuß, *Security in Near Field Communication (NFC) Strengths and Weaknesses*, Workshop on RFID Security RFIDSec, 2006

[Her13] D. Hernie, Windows Phone 8 Security deep dive, Tech Days 2013, March 5-7, 2013

[ISO14443] ISO, *Identification cards – Contactless integrated circuit cards – Proximity card*, Part 1-4, ISO/IEC 14443

[ISO18092] ISO, *Information technology – Telecommunications and information exchange between systems – Near Field Communication – Interface and Protocol (NFCIP-1)*, ISO/IEC 18092, Second edition , 2013-03-15

[Jak12] A. Jakl, How to Create Cross-Platform LaunchApp NFC Tags, `http://social.technet.microsoft.com/wiki/contents/articles/13955.how-to-create-cross-platform-launchapp-nfc-tags.aspx#Cross-Platform_LaunchApp_Tag`, Revision 2, 11.09.2012, Zuletzt zugegriffen am 18.09.2013

[Jak13] A. Jakl, NDEF Library for Proximity APIs / NFC, `http://ndef.codeplex.com/`, July 29, 2013, Zuletzt zugegriffen am 19.09.2013

[JR13] A. Jakl, M. Roland, *Near Field Communication (NFC) Developer Comparison*, Version 1.4, 19.09.2013, Zuletzt zugegriffen am 19.09.2013

[KB12] E. Kalige, D. Burkey, *A Case Study of Eurograbber: How 36 Million Euros was Stolen via Malware*, December 2012

[Kob87] N. Koblitz, *Elliptic Curve Cryptosystems*, Mathematics of Computation, Volume 48, Nr. 177, pp. 203-209, January 1987

[Lee12] E. Lee, NFC Hacking: The Easy Way, DEF CON 20, July 26-29, 2012

[Mic] Microsoft, Gewusst wie: Speichern von asymmetrischen Schlüsseln in einem Schlüsselcontainer, `http://msdn.microsoft.com/de-de/library/vstudio/tswxhw92.aspx`, Zuletzt zugegriffen am 19.09.2013

[Mil86] V. S. Miller, *Use of Elliptic Curves in Cryptography*, In Proc. of Advances in Cryptology — CRYPTO '85, LCNS 218, pp. 417-426, 1986

[Mul08] C. Mulliner, Attacking NFC Mobile Phones, EUSecWest, May 21-22, 2008

[NDEF] NFC Forum, *NFC Data Exchange Format (NDEF)*, Technical Specification, 2006-07-24

[Nok11] Nokia, Introduction to NFC, `http://developer.nokia.com/Develop/NFC/Documentation`, 2011-07-08, Zuletzt zugegriffen am 18.09.2013

[Poo] I. Poole, NFC Modulation & RF Signal, `http://www.radio-electronics.com/info/wireless/nfc/near-field-communications-modulation-rf-signal-interface.php`, Zuletzt zugegriffen am 18.09.2013

[Ram12] S. Ramaswamy, Deep Dive into the Kernel of .NET on Windows Phone 8, `http://channel9.msdn.com/Events/Build/2012/3-005`, Zuletzt zugegriffen am 19.09.2013

[Sch96] B. Schneier, *Applied Cryptography*, Second Edition, Wiley, 1996

[SMK01] J. Scambray, S. McClure, G. Kurtz, *Hacking Exposed: Network Security Secrets & Solutions*, Second Edition, McGraw-Hill Osborne Media, S. 561-563, 2001

[Sou13] Sourceforge, NFCProxy, `http://sourceforge.net/projects/nfcproxy/`, 2013-02-27, Zuletzt zugegriffen am 18.09.2013

[Tan09] A. S. Tanenbaum, *Moderne Betriebssysteme*, 3., aktualisierte Auflage, Pearson Studium, 2009

[Vin12] A.J. H. Vinck, *Introduction to public key cryptography*, S. 16, May 12, 2012

[WP7] WP7 Root Tools, Unlocks explained, `http://wp7roottools.com/index.php/guides/unlocks-explained`, 17.04.2012, Zuletzt zugegriffen am 18.09.2013

[Zef12] T. Zefferer, *Secure Elements am Beispiel Google Wallet*, Version 1.0, S. 7-9, 28.04.2012