BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

USING NEAR-FIELD COMMUNICATION (NFC) TO IMPROVE MESSAGING PRIVACY ON ANDROID OS

VYUŽITÍ NEAR-FIELD COMMUNICATION (NFC) KE ZLEPŠENÍ SOUKROMÍ ZPRÁV
V OS ANDROID

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE
Bc. JAROMÍR KARMAZÍN

SUPERVISOR
VEDOUCÍ PRÁCE
Ing. PAVEL OČENÁŠEK, Ph.D.

BRNO 2016

Master Thesis Specification/18427/2015/xkarma06

Brno University of Technology - Faculty of Information Technology

Department of Information Systems Academic year 2015/2016

Master Thesis Specification

For:	**Karmazín Jaromír, Bc.**
Branch of study:	Information Technology Security
Title:	**Using Near-Field Communication (NFC) to Improve Messaging Privacy on Android OS**
Category:	Security

Instructions for project work:

1. Research Near-Field Communication (NFC), its security properties, and possible security applications.
2. Research the possibilities of using NFC on a recent version of the Android operating system.
3. Design a mechanism that utilizes NFC in an instant messaging (IM) application to enhance the security and privacy of its users.
4. Evaluate the security and privacy enhancements brought by the new design.
5. Create a reference implementation of the new design.

Basic references:

- Murphy, L. M. *Android 2: Průvodce programováním mobilních aplikací*. ČR: Computer Press, 2011. ISBN 978-80-251-3194-7.
- Mednieks, Z., Dornin, L., Meike, B., Nakamura, M. *Programming Android*: *Java Programming for the New Generation of Mobile Devices*. O'Reilly, 2011. ISBN 978-1-1493-1300-5.
- Schiller, J. *Mobile Communications*, Addison-Wesley, 2003. ISBN 0-321-12381-6

Requirements for the semestral defense:
Items 1, 2 and 3.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor:	**Očenášek Pavel, Ing., Ph.D.**, DIFS FIT BUT
Beginning of work:	November 1, 2015
Date of delivery:	May 25, 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

[illegible]

Dušan Kolář
Associate Professor and Head of Department

Abstract

In this thesis, we build a simple instant messaging (IM) system that makes use of Near-Field Communication (NFC) for improved security and privacy. First, we analyze the NFC technology itself. Then, we research the NFC capabilities of Android, a popular operating system for smart mobile devices, as well as existing applications that use these capabilities. We also research the state of the art regarding secure IM, especially Signal Protocol and its open-source implementation for Android. Next, we design an IM system that uses Signal Protocol for message encryption and NFC for the exchange of cryptographic material, requiring neither the use of phone numbers nor a trusted third party. Finally, we show a reference implementation of the client and server applications and an evaluation of the security and privacy properties of this system.

Abstrakt

V této práci vytváříme jednoduchý systém pro rychlou textovou komunikaci (IM), který využívá technologii Near-Field Communication (NFC) ke zlepšení bezpečnosti a soukromí. V první části zkoumáme samotnou technologii NFC. Poté se věnujeme možnostem NFC v operačním systému Android pro chytrá mobilní zařízení a také existujícím aplikacím, které tyto možnosti využívají. Rovněž zkoumáme nejmodernější přístup k bezpečnému IM, zejména Signal Protocol a jeho open-source implementaci pro Android. Dále pak navrhujeme IM systém, který používá Signal Protocol pro šifrování zpráv a NFC pro výměnu kryptografického materiálu, aniž by vyžadoval použití telefonních čísel nebo důvěryhodnou třetí stranu. Nakonec předkládáme vzorovou implementaci klientské i serverové aplikace a zhodnocení vlastností tohoto systému z hlediska soukromí a bezpečí.

Keywords

Android, cryptography, instant messaging (IM), Near-Field Communication (NFC), privacy, security, Signal Protocol, smart phones.

Klíčová slova

Android, Near-Field Communication (NFC), bezpečnost, chytré telefony, kryptografie, rychlá textová komunikace (IM), Signal Protocol, soukromí.

Reference

KARMAZÍN, Jaromír. *Using Near-Field Communication (NFC) to Improve Messaging Privacy on Android OS.* Brno, 2016. 77 pages. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Očenášek Pavel.

Using Near-Field Communication (NFC) to Improve Messaging Privacy on Android OS

Declaration

I hereby declare that this thesis is the result of my original work under the supervision of Ing. Pavel Očenášek, Ph.D., and that I have acknowledged all literary sources, publications, Internet sources, and software libraries used.

....
Jaromír Karmazín
May 23, 2016

Acknowledgements

I would like to thank my supervisor Ing. Pavel Očenášek, Ph.D., for his willingness to discuss the initial idea behind this thesis, for allowing me to actualize it, and for giving me the opportunity to co-author a related article to be published at the International Conference on Human Aspects of Information Security, Privacy, and Trust.

Contents

List of abbreviations

AAR Android Application Record. 15

AID application identifier. 2, 3, 17, 29, 30, 34, 57, 59, 61

APDU application protocol data unit. 16, 17, 29, 30, 39, 46, 59–62

API application programming interface. 2, 14–17, 21, 28, 29, 34–37, 41–43, 46, 51, 59, 63–65

ASCII American Standard Code for Information Interchange. 29, 33, 64

ASK amplitude shift keying. 9–11

CA certification authority. 46, 49, 73

CLI command line interface. 71, 72

DDoS distributed denial of service. 46

DF dedicated file. 61

DNS Domain Name System. 46, 49

DoS denial of service. 11, 46, 47

ECDH Elliptic curve Diffie-Hellman. 23, 31–33

FCI File Control Information. 30

FCP File Control Parameters. 30

FMD File Management Data. 30

GCM Google Cloud Messaging. 48

GUI graphical user interface. 13

HCE host-based card emulation. 16, 17, 31

HKDF HMAC-based key derivation function. 33

HMAC hash-based message authentication code. 4, 33

HTTP Hypertext Transfer Protocol. 35, 36, 63, 73

HTTPS Hypertext Transfer Protocol, Secure. 27, 34, 36, 46, 47, 49, 63, 73

ID identifier. 23, 24, 32–35, 40, 42, 50, 65–67

IEC International Electrotechnical Commission. 10, 16, 17

IETF Internet Engineering Task Force. 7

IM instant messaging. 7, 26, 51, 69

IP Internet Protocol. 27, 46, 47

ISO International Organization for Standardization. 2, 10, 15–17, 29, 39, 59–62

JIS Japanese Industrial Standard. 15

JSON JavaScript Object Notation. 34, 36, 38, 43, 63

MIME Multipurpose Internet Mail Extensions. 15

MVC model-view-controller. 41

NDEF NFC Data Exchange Format. 14, 15

NFC near-field communication. 1, 2, 5, 7–12, 14–20, 25–31, 34, 40, 41, 46–48, 50, 51, 59, 65, 69

ORM object-relational mapping. 42

OTR Off-the-Record Messaging. 21, 22

PGP Pretty Good Privacy. 49

PIN personal identification number. 19

REST representational state transfer. 2, 41, 43, 71

RF radio frequency. 1, 9–11

RFC Request for Comments. 33

RFID radio-frequency identification. 9, 10

SHA-256 Secure Hash Algorithm 2, 256 bits. 29–31, 33

SIM subscriber identity module. 11

SQL Structured Query Language. 72

TCP Transmission Control Protocol. 27

TLS Transport Layer Security. 36, 46, 73

UI user interface. 15

UID unique identifier. 17

URL Uniform Resource Locator. 33, 36, 63, 72

XMPP Extensible Messaging and Presence Protocol. 21, 22

Chapter 1

Introduction

Mobile devices and Internet services have become an integral part of modern life. It should therefore come as no surprise that people use current technology even for matters that could be considered sensitive and private. However, a considerable part of the technology used today was not originally built for security or privacy, and this deficiency is only slowly being fixed. An example of a yet unmitigated threat is pervasive monitoring, which the Internet Engineering Task Force (IETF) recognizes as an attack on privacy. [20] This situation leads us to believe that there is a need for more privacy-preserving technology, including computer software.

Although some privacy-preserving software exists already, its popularity varies due to a large number of factors. In our view, one important factor is ease of use. Therefore, in order to improve the privacy and security of the general public, we need to create software that is not only privacy-preserving and secure, but easy to use as well.

To achieve this goal, we have chosen to analyze and make a novel use of a technology known as near-field communication (NFC). This technology, which can be found on many current smart phones, enables wireless exchange of data over distances of around 10 centimeters, either between two active devices or between one active device and a passive tag. A detailed explanation of this technology can be found in chapter 2 (Properties of NFC), where we will also examine the properties of NFC to find out how it can be used in a security- and privacy-oriented application.

As the platform for implementing our new piece of software, we have chosen Android, an operating system found on many contemporary smart phones and other mobile devices. In chapter 3 (Android's capabilities), we will examine recent versions of Android and how NFC can be used in applications made for this operating system.

The use case of our newly created Android application will be the exchange of textual messages between two users, also known as instant messaging (IM). In chapter 4 (Existing solutions), we will examine existing IM applications for Android and discuss the state of the art in terms of security and privacy. Also in chapter 4, we will list some existing applications for Android that use NFC.

In chapter 5 (Design), we will consider the potential security and privacy benefits of NFC for IM on Android, outline the requirements for a new application to reap these benefits, and then propose and design a system around this application. Two protocols will lie at the heart of this design: One through which two NFC-capable Android devices can exchange cryptographic material (Key exchange protocol), and another one that will allow these devices to communicate privately over an Internet connection using the exchanged data and existing cryptographic libraries (Message relay protocol).

The reference implementation of a client application for Android, as well as a server application, both required by the aforementioned protocols, will be described in chapter 6 (Implementation).

Finally, in chapter 7 (Evaluation), the security and privacy properties of the new design and its reference implementation will be evaluated and compared to other existing technologies.

This thesis is a continuation of a term project, which was defended in January 2016. Chapters 2 to 4 have all been taken from this term project, with minor corrections and updates. A modified version of these chapters was also submitted to the International Conference on Human Aspects of Information Security, Privacy, and Trust. [28]

Chapter 5 was taken from the term project as well, but has received significant updates as a result of prototyping and experimentation, which brought new findings and allowed for a more detailed design. The remaining chapters 6 and 7 are new additions, building upon the results of the term project and the updated design.

Chapter 2

Properties of NFC

NFC is a technology for short-range wireless communication. It is somewhat related to radio-frequency identification (RFID).

2.1 Physical properties

The RFID Handbook [21] describes, among other things, the physical characteristics of NFC.

The *near-field* of an antenna is defined in [21, section 4.2.1.1] as the area around the antenna where inductive coupling is possible. This area contains a magnetic field that can retroact upon the antenna that generated it. Beyond this area lies the so-called *far-field*, where electromagnetic waves are fully formed and separated from the antenna. As a rule of thumb, $r_F = \lambda/2\pi \approx 0.16\lambda$ can be used as the estimated radius of the near-field.

According to [21, section 3.4], NFC uses "high-frequency magnetic alternating fields in the frequency range of 13.56 MHz," a radio frequency, as a carrier for the transmitted data. Using the above definition of a *near-field*, its radius for this frequency range would be equal to approximately 3.5 meters. However, the typical communication range for NFC data transmission is usually cited as 20 cm [21, section 3.4] or 10 cm [21, section 11.6].

Per [21, section 11.6], "NFC is compatible with existing RFID standards," which means that some RFID readers or RFID transponders may be able to communicate with an NFC interface.

When two NFC interfaces (or an NFC interface and an RFID interface) communicate, they assume one of two roles: NFC initiator or NFC target. The NFC initiator is always the interface that starts the communication. [21, section 3.4]

There are two operational modes of NFC:

Active mode. [21, section 3.4.1]
In this mode, both interfaces take turns transmitting (the NFC initiator first) using an alternating magnetic field. Transmitted data are modulated onto the alternating magnetic field using amplitude shift keying (ASK), "similar to the data transmission

between RFID reader and transponder." When an NFC interface's transmitter is active, it transmits both power and data.

Passive mode. [21, section 3.4.2]
In this mode, only the NFC initiator provides a magnetic alternating field. The data sent from the NFC initiator to the NFC target is modulated using ASK. In the reverse direction, data is sent using *load modulation*, which is also known from RFID systems. The load modulation uses a 848 kHz subcarrier according to ISO/IEC 14443. [21, section 11.6]

Load modulation works by one NFC interface retroacting upon the magnetic field of another. Because of that, passive mode NFC is strictly limited to the *near-field*, i.e. distances below r_F. [21, section 3.2.1.2.1]

An NFC initiator in passive mode can emulate an RFID reader. This mode of operation is called "reader mode" or "reader-emulation mode".

An NFC target in passive mode can emulate an RFID transponder. This mode of operation is called "card mode" or "card-emulation mode".

2.2 Security of NFC

2.2.1 Access to RF interface

Where using NFC is an option, the very nature of the technology makes it a harder target for attacks than other technologies such as Bluetooth or Internet-based services, as illustrated in table 2.1.

Technology	Range	Third parties involved?
NFC	very short	no
Bluetooth	short	no
Wi-Fi ad-hoc	short	no
Wi-Fi infrastructure	short	yes
Internet	worldwide	yes

Table 2.1: Comparison of communication technologies by accessibility by an attacker.

In order for an attacker to eavesdrop on or interfere with NFC communication, they need to establish a physical presence near the two communicating devices, at a range shorter than what conventional wireless technologies permit. Furthermore, there is no infrastructure involved, so there is no trusted third party that the attacker can spoof.

Therefore, we speculate that security threats to NFC are inherently more limited than threats to other wireless technologies.

2.2.2 Passive attacks on the RF interface

As with any method of wireless communication, eavesdropping on the physical layer is possible if the reception is good. Per [38, section 7.5.1], "the distance can generally be greater than the standard reading distance” in the case of RFID tags. Haselsteiner and Breitfuß state in [24, section 3.1] that for NFC, eavesdropping can be done within about 10 meters of a device communicating in active mode and within about 1 meter of a device communicating in passive mode. However, they give these numbers only as rough estimates, reasoning that a “huge number of parameters” determine the actual radius of possible eavesdropping.

2.2.3 Active attacks on the RF interface

It is possible for an attacker in the vicinity of communicating devices to corrupt the transmitted data by also transmitting at the same frequencies with the correct timing and modulation. This is a denial of service (DoS) attack. [24, section 3.2]

It is also possible for an attacker to modify the transmitted data by transmitting a specific signal at the same time. Except when 100% ASK modulation with modified Miller encoding is used[1], all bits can be modified by an attacker. [24, section 3.3]

An attacker may insert messages into a legitimate communication, but only if there is a long enough pause between two legitimate messages. [24, section 3.4]

A Man-in-the-Middle attack is considered “practically infeasible to mount” by Haselsteiner and Breitfuß due to overlapping radio fields. [24, section 3.5] However, we would like to not rule out the possibility of an attack being developed in the future. For example, a sticker with two antennae and a shield separating them, placed on an NFC device, could be devised for such an attack. Alternatively, a crude attack may be successful if the devices’ NFC driver software or firmware contains security vulnerabilities. Applications should therefore always provide a mechanism for verifying the integrity of the communication.

2.2.4 Application-level security

For certain security-sensitive applications, such as mobile payment, the mobile device needs the level of security similar to a smart card. The application’s private data should be stored in a location where other applications, and perhaps even the user, cannot read it or tamper with it.

Secure NFC exists for these purposes. [21, section 11.6.1] It relies on the presence of a hardware module called the *secure element*, which can be a subscriber identity module (SIM) card, a secure memory card, or a smart card chip. In this setup, the security-sensitive applet runs in the secure element and communicates directly with the NFC device. An application controller, which may run in an insecure environment, serves only administrative purposes and cannot access the secure element’s data directly.

2.3 Applications of NFC

Possible applications of the NFC technology include: [21, section 13.4]

[1]This mode of transmission is only used at 106 kBaud speeds in active mode. Even then, some bits can still be changed by an attacker.

- access control (where the NFC device behaves like an access token),
- public transportation (where the NFC device behaves like a transport ticket),
- contactless payment (where the NFC device behaves like a contactless smart card),
- smart labels and smart posters,
- transmitting information between two NFC devices, e.g. media files, links, or contact information (see section 3.2.2 for details on one such mechanism), and
- pairing and configuration of devices for Wi-Fi, Bluetooth, etc. [24, section 2.3]

Chapter 3

Android's capabilities

3.1 Application architecture

For readers unfamiliar with the architecture of Android applications, we now present a quick overview of the most important terms, citing [27, section 2.3.1]:

> An Android application consists of several application components: *activities*, *services*, *broadcast receivers*, and *content providers*.
>
> *Activities* are components that create the user interface of the application. They roughly correspond to windows in classical windowed GUIs. The main difference is that when an *activity* is not visible to the user, it can be removed by the operating system to save resources, and later created again when the user wants to display it again. This means that an *activity* has limited lifetime and should only be used for interacting with the user.
>
> *Services*, on the other hand, can run persistently, even without user interaction. They are used to perform work in the background, using separate threads. They do not have a user interface, but they can present information to the user using *toasts* (transient messages), system notifications, or broadcast messages to other components. (. . .)
>
> *Broadcast receivers* enable [an] application to receive events generated by the system or other applications, even if the application is not running when the event is generated.
>
> *Content providers* allow sharing data among applications. (. . .)

Another two important terms that will be used in this thesis are *intents* and *intent filters*, as described in [5].

Intents are messaging objects that describe an operation to be performed. They can be used to start an *activity*, to deliver a broadcast to interested *broadcast receivers*, and to start or bind a *service*.

Intent filters specify the type of *intents* that a given application component would like to receive.

3.2 NFC capabilities

The Android Developers portal offers a guide to Android's NFC application programming interface (API) [6], an additional lesson about sharing files with NFC [10], as well as reference documentation for the entire android.nfc Java package [1] that makes up the NFC API.

In this section, we will summarize the possibilities of using NFC in Android applications based on the above sources. In addition, we will include our findings based on a prototype application and experimentation with two NFC phones (see appendix A for their specifications).

Along with each feature, we will mention the API level at which the feature becomes available. To find out which Android version number and code name each API level refers to, see table 3.1.

API level	Android version	Code name
9	2.3 – 2.3.2	Gingerbread
10	2.3.3 – 2.3.7	Gingerbread
11	3.0	Honeycomb
12	3.1	Honeycomb
13	3.2.x	Honeycomb
14	4.0.1 – 4.0.2	Ice Cream Sandwich
15	4.0.3 – 4.0.4	Ice Cream Sandwich
16	4.1.x	Jelly Bean
17	4.2.x	Jelly Bean
18	4.3.x	Jelly Bean
19	4.4 – 4.4.4	KitKat
21	5.0	Lollipop
22	5.1	Lollipop
23	6.0	Marshmallow

Table 3.1: Android API levels and their corresponding versions. Source: [2]

3.2.1 Reader/writer mode

Android applications can "read and/or write passive NFC tags and stickers." With API level 9, limited tag reading is supported, and starting from API level 10, comprehensive support for the reader/writer mode is available. This uses NFC in passive mode.

When the device's screen is unlocked and NFC is not disabled in its settings, the device scans for nearby NFC tags. Once a tag is discovered, Android reads it and then delivers the result to the application best suited to handle it.

Per the documentation, "Android has the most support for the NFC Data Exchange Format (NDEF) standard, which is defined by the NFC Forum."

In addition, the API exposes interfaces which allow raw communication with other tag technologies. In these cases, the application must implement its own protocol for communicating with the tag.

The supported technologies are:

- NFC-A (ISO 14443-3A),
- NFC-B (ISO 14443-3B),
- NFC-F (JIS 6319-4),
- NFC-V (ISO 15693), and
- ISO-DEP (ISO 14443-4).

Optionally, Android devices may support these additional NFC tag technologies:

- MiFARE Classic and
- MiFARE Ultralight.

The initial tag discovery and reading is handled by the Android operating system itself. Once the system gathers enough information about the tag, it creates an *intent* and routes it to an application. By default, the receiving application does not need to be in the foreground; it will be started by the operating system if necessary.

Each application can specify an *intent filter* to let the system know which tags it wants to handle. Applications may filter based on the NDEF records found on the tag (e.g. by the record's MIME type), by the type of NFC technology used by the tag, or simply accept any tag. Additionally, tags may contain so-called Android Application Records (AARs), which contain "the package name of an application embedded inside an NDEF record." Starting from API level 14, Android checks AARs and selects the application to use based on these records.

Applications can also enable the so-called *foreground dispatch mode*. In this mode, the application will be given priority when discovered tag information is dispatched. In order to use this mode, the application must be in the foreground.

Since API level 19, an application in the foreground can enable *reader mode*, which limits all NFC operation to reader/writer mode and disables the peer-to-peer mode (see section 3.2.2) and card emulation mode (see section 3.2.3).

3.2.2 Peer-to-peer mode

Since API level 14, Android offers functionality called Android Beam which utilizes the active mode of NFC. The original purpose of this mode is to send an NDEF message from one Android-powered device to another.

To make this happen, the sending device must have an application capable of sending data over Android Beam in the foreground and the receiving device must be unlocked. When these conditions are met, the transfer is started by the following workflow:

1. The devices are brought together within a close range.
2. The sending device displays a "Touch to Beam" user interface (UI).
3. The user confirms the action by tapping the screen of the sending device.
4. The data is sent over NFC.

As we found experimentally, if an application does not define any content to share, by default, the application itself is the shared content, meaning that the receiving device will either open the same application, if installed, or open its page in the Google Play Store.

NFC is not suitable for the transfer of large files because of its low data rate, combined with the fact that the sender and receiver need to be in close proximity for the whole duration of the transfer. According to [4], the "sane upper bound" for data transferred using NFC is "about 1KB (...), which can usually be exchanged within 300ms."

Since API level 16, it is possible to transfer large files between Android devices using the *Android Beam file transfer API*. [10] Android overcomes the speed limitations of NFC by only using it for the initial setup, then silently enabling Bluetooth, temporarily pairing the two devices, and performing the actual file transfer over Bluetooth. [17]

A similar approach to transferring large files is implemented in Samsung Galaxy S III, whose S Beam technology uses Wi-Fi Direct instead of Bluetooth to transfer large files after a connection is initiated over NFC. [17]

Since API level 21, it is possible to invoke Android Beam manually (see the `NfcAdapter` class and its method `invokeBeam`). This supports an alternative workflow:

1. The user selects the content to share and triggers an action which results in invoking Android Beam.
2. The device displays the Android Beam animation
3. The user brings the device within a close range of another compatible device.
4. The data is sent over NFC.

As of Android 6.0, the peer-to-peer mode of NFC can only be used for Android Beam, which only allows one-shot, simplex data transfers. There is no way to obtain any payload in response to a peer-to-peer request, let alone develop a custom communication protocol, using only the public API.

During our experiments, both devices used (see appendix A) had a separate switch for turning Android Beam on and off. However, we found that no matter the setting, the devices would always prefer peer-to-peer mode when coming within close range of each other. With Android Beam turned off, the same sound still played as it would with Android Beam enabled (which is a different sound from the one played when a passive tag is scanned), but nothing would happen. If we wanted to force one device to behave as a tag and the other device to behave as a reader, we had to use *reader mode*, as explained in section 3.2.1.

3.2.3 Card emulation mode

Since API level 19, Android provides an API to implement host-based card emulation (HCE). [4] This allows Android applications to respond directly to NFC readers, without involving the secure element described in section 2.2.4.

Supported are emulated cards "based on the NFC-Forum ISO-DEP specification (based on ISO/IEC 14443-4) and process application protocol data units (APDUs) as defined in the ISO/IEC 7816-4 specification." The only protocol stack with mandatory support from all Android devices with NFC HCE, according to [4], is the one outlined in table 3.2. On top of these protocols, the developer must implement their own protocol stack for sending and receiving those APDUs.

ISO 7816-4
ISO 14443-4
ISO 14443-3 type A
ISO 14443-2
ISO 14443-1

Table 3.2: Protocol stack for card emulation.

Card emulation only works when NFC is enabled and the screen is on. Unlike the reader/writer mode, this can work from the device's lock screen and does not require any application to be in the foreground.

The source [4] mentions that "in the first part of the exchange the HCE device will present its UID; HCE devices should be assumed to have a random UID. This means that on every tap, the UID that is presented to the reader will be a randomly generated UID. Because of this, NFC readers should not depend on the UID of HCE devices as a form of authentication or identification."

The specific HCE *service* to use is selected based on an application identifier (AID). Applications may define a static list of AIDs and AID groups they wish to use. These AIDs may either be registered using the procedure defined in the ISO/IEC 7816-5 specification, or they may be placed in the proprietary range. Proprietary AIDs must have each of the four most significant bits of the first byte set to 1, and should be at least 6 bytes long to reduce the risk of collisions.

New methods for finer handling of card emulation have been added in API level 21 (see the class CardEmulation in the android.nfc.cardemulation package). They allow dynamically registering and removing HCE *services* at run time, as well as controlling the preferred HCE *service* to use.

3.2.4 NFC secure element

As of Android 6.0, there is no public API for using the secure element, described in section 2.2.4, for card emulation. Android's host-based card emulation, described in section 3.2.3, does not use the secure element at all.

Chapter 4

Existing solutions

4.1 Android applications using NFC

We searched for the term “NFC” on [3] in order to get an idea about how NFC is used in existing Android applications. In this section, we will analyze the many results that turned up in the search, attempt to categorize them, and point out some of the more unusual applications.

4.1.1 NFC diagnosis and management applications

As one would expect, among all NFC applications, there are ones intended for managing and diagnosing the NFC subsystem itself, rather than making any actual use of it. These include simple applications for checking the availability of NFC on the device (e.g. *NFC Enabled?* by Espen “Rexxars” Hovlandsdal) and widgets for enabling and disabling the device’s NFC interface (e.g. *NFC Widget* by AIT APPs).

4.1.2 Tag reader/writer applications

The largest portion of applications using NFC focuses operations with NFC tags. Naturally, this would make use of passive NFC in reader emulation mode.

There are many applications that provide low-level support for NFC tags, such as displaying basic information about a tag (make, model, supported NFC technology, serial number, etc.), reading data from a tag, writing data to a tag, and tag cloning. It can be assumed that some knowledge of the NFC technology is required in order for the user to find any value in these applications, so average users are probably not the target audience. Examples of these applications are *NFC Tools* by wakdev, *NFC ReTag* by WidgApp Mobile Solutions, *NFC Reader* by Adam Nybäck, and *NFC TagWriter* by NXP Semiconductors.

Certain specialized applications allow extracting more information from NFC tags used in existing real-world systems:

- *Credit Card Reader NFC (EMV)* by Jullien Millau uses NFC “to read public data on an NFC banking card compliant with EMV [norms],” such as the contactless credit and debit cards made by Visa, American Express, and MasterCard. The application can show the card type, number, expiration date, number of remaining PIN entries, and in some cases the card holder's name and the card's transaction history.

- *Metro tickets of Moscow* by Dmitriy V. Lozenko uses NFC to read information off of such tickets. The information includes the ticket's type, number, date of issue, date of expiration, number of trips (both total and remaining), and the name of the last station (presumably the last station where the ticket communicated with a reader).
- *Octopus* by Octopus Cards Limited uses NFC for reading so-called "Octopus cards", which can be used for transportation, parking, retail shopping, and other facilities in Hong Kong, as well as for online payment using an NFC-enabled Android device. [13]
- *saldoBip NFC* by YANKO uses NFC to check the balance of a "bip! card", which is a prepaid ticket used by the public transportation company Transantiago operating in Santiago de Chile. [8]
- *ShareMoreTransport* by Share More Studio uses NFC to check the balance and transaction records on public transport cards used in various Chinese cities, such as Beijing, Shenzhen, Wuhan, Hongkong, Qingdao, and Xian.
- *T-money Balance Check* by RW MobiMedia uses NFC to check the balance on a South Korean contactless payment card for transportation and some convenience stores.

Another frequent use case is the use of NFC tags to trigger certain actions or change certain settings in the device that reads them. This would allow a user to place NFC stickers in certain places of interest, such as different rooms at their home, in their car, etc., and have the phone enable or disable wireless interfaces, change the ringtone volume, open certain applications, etc., based on the tag which is scanned. Examples of these applications are *Trigger* by Egomotion and *NFC Tasks* by wakdev. Another peculiar instance is *NFC Alarm Clock* by kamituel, which requires the user to scan an NFC tag in order to mute a ringing alarm.

A related concept is implemented by *WifiTap* by Andreas Rossbacher, which allows writing Wi-Fi credentials to, and reading those credentials from, a writable NFC tag. This can be useful for providing Wi-Fi access to guests at home, at an office, or at a public event.

There are a few applications which use the NFC reader mode in a novel way. For example:

- *NFC Developer* by Thomas Rorvik Skjolberg allows a software developer to prepare data for an NFC tag on a computer, transfer that data onto their Android phone using a QR code, and finally program an NFC tag using the phone, thus eliminating the need for a separate tag reader.
- *SmartPassLock NFC* by DreamOnline, Inc. adds another method of unlocking an Android device's screen, requiring a registered NFC tag to be present (the application's description only mentions support for Japanese IC cards).
- *Crypto NFC* by rolios allows users to write notes on their Android device, encrypt them, and keep the decryption key on an NFC tag.

4.1.3 Android Beam applications

Android Beam is a peer-to-peer service using NFC's active mode to exchange data between two Android devices, as was explained in section 3.2.2.

Some of the default Android applications allow sharing content, such as web pages, contacts, and YouTube videos, through Android Beam. [7] Since Android 4.1 Jelly Bean, it is also possible to share photos and videos from Gallery with the help of a transient Bluetooth connection. [17]

Applications like *NFC Transfer Beta* by Abhinava Srivastava and *NFC File Transfer* by apps4u@android use Android Beam to transfer arbitrary files. *SuperBeam* by LiveQoS offers the same functionality, but uses NFC or QR codes for pairing and Wi-Fi Direct for file transfer.

Share Tether NFC by Javi Pacheco uses Android Beam to "share tethering between two devices using NFC."

4.1.4 Miscellaneous NFC applications

Applications like *CardShake* by Tesla System Co., Ltd. or *Business Card Holder with NFC* by ATSolution allow users to exchange electronic business cards over NFC.

Certain printers by HP Inc. can print content received via NFC from an Android device with the *HP ePrint* application installed.

NFC Porter by IMA s.r.o. uses NFC for premises access control when used together with a compatible identification system.

PassWallet - Passbook + NFC by Above Mobile Limited allows storing tickets and coupons in Apple's Passbook file format and redeeming them through NFC readers.

4.2 Android applications for secure messaging

The Electronic Frontier Foundation maintains a useful list [19] which ranks messaging applications and tools based on their security. By examining the Android applications found on this list, we can get a good idea about the state of the art and determine what improvements can be made.

The list gives scores to applications based on the following criteria:

1. client-to-server encryption ("encrypted in transit"),
2. end-to-end confidentiality ("encrypted so the provider can't read it"),
3. end-to-end verification ("can you verify contact's identities"),
4. forward secrecy ("are past comms secure if your keys are stolen"),
5. published source code ("is the code open to independent review"),
6. documentation ("is security design properly documented"), and

7. existing audits ("has there been any recent code audit").

The first four points are of interest because they outline the basic security properties a state-of-the-art protocol should have designed in, while the last three points can help us decide whether we can draw inspiration, or even use the source code, from an existing project.

At the time of writing this thesis, the following Android applications are listed to fulfill all the criteria:

- *ChatSecure + Orbot*,
- *Signal*[1],
- *Silent Phone*[2], and
- *Telegram*.

It may be interesting to notice that client-to-server encryption is almost ubiquitous, but out of the most popular messaging applications (*Facebook chat*, *Google Hangouts*, *Skype*, *SnapChat*, *Viber*, *WhatsApp*), only few offer more security than that.

4.2.1 *ChatSecure*

ChatSecure is a client for Extensible Messaging and Presence Protocol (XMPP). [15] Users are identified by a JID[3] just like in any other XMPP client. *ChatSecure*'s end-to-end encryption uses the Off-the-Record Messaging (OTR) protocol as implemented by the otr4j library, whose source code can be found at [40]. The latest version of the OTR protocol is documented at [23].

4.2.2 *Signal*

Signal is a mobile messaging application. It uses the users' phone numbers as identifiers. For transporting messages, it provides servers and an API documented at [34]. Messages are secured using Signal Protocol, a cryptographic protocol previously known as Axolotl [33], which uses Double Ratchet Algorithm at its core.

Signal Protocol is inspired in part by OTR and shares some of its properties. Open Whisper Systems, the developers of *Signal*, improved Signal Protocol over OTR to make it more suitable for asynchronous messaging, a common use case for mobile communication, and to give it new security properties. Double Ratchet Algorithm allows keeping OTR's "future secrecy"[4], which makes it possible to limit the window of compromise even in long-lasting message sessions. [30]

[1]Formerly maintained as two separate applications known as *TextSecure* and *RedPhone*. [32]

[2]*Silent Text* has been merged into *Silent Phone* as of 28th September 2015. [11]

[3]A historical name for the network address of an XMPP entity. [39]

[4]"Future secrecy" is a term used by Open Whisper Systems to describe the property of a cryptographic communication system that constantly generates new ephemeral keys, allowing it to "heal" itself "even if any individual ephemeral key is compromised or otherwise found to be weak at any time." OTR and Signal Protocol are both protocols exhibiting this property.

4.2.3 *Silent Phone*

Silent Phone shares the protocol for encryption text messages with *Signal*, although these two services are not compatible. [11] Users are identified by usernames. [12]

4.2.4 *Telegram*

Telegram uses a custom encryption protocol called MTProto. Although not officially broken, this protocol's design quality has been called into question. In addition, encryption is not enabled by default, but has to be activated willfully by the user by using the application's "Secret Chat" option. [41]

4.2.5 Summary

All of the above applications support client-to-server encryption, end-to-end encryption with the option of verifying the other party, and forward secrecy.

There are two factors that differentiate these applications:

1. **Protocol.** Although all of the protocols are currently considered safe, *Telegram*'s MTProto does not receive much praise from the cryptographic community. Signal Protocol has improvements over OTR and is used by two otherwise independent applications on the list.
2. **Architecture.** ChatSecure uses the existing architecture of XMPP, which allows the use of custom user identifiers and servers, at the cost of having to add contacts manually. Both *Signal* and *Telegram* use their own servers with phone numbers as identifiers, which simplifies the process of finding contacts, but weakens user privacy: The service provider may link the users' phone numbers to their real-world identities, or read the users' contact lists to learn about their social network. Silent Phone also uses its own servers, but users create separate registrations with usernames.

These findings are summarized in table 4.1.

Application	Protocol	User identifier	Custom servers
ChatSecure	OTR	JID	supported
Signal	Signal Protocol	phone number	not supported
Silent Phone	Signal Protocol	username	not supported
Telegram	MTProto	phone number	not supported

Table 4.1: Comparison of secure messaging applications on Android.

4.3 Signal Protocol

Out of the three cryptographic protocols mentioned in section 4.2, Signal Protocol comes out as superior, to both MTProto and OTR. It also comes with an open-source library for Java/Android, hosted at [35], which makes it suitable for the creation of new secure messaging applications.

For these reasons, we analyze this particular protocol further.

4.3.1 Building a session

In order to be able to encrypt and decrypt messages sent between users, Signal Protocol requires a session to be established.

The following three ways of establishing a session are permitted by the README.md file from [35]:

- PreKeyBundles,
- PreKeySignalMessages, or
- KeyExchangeMessages.

The term "PreKeys" refers to public keys and associated IDs stored by the server.

4.3.2 Structure of a KeyExchangeMessage

By examining the source code of Signal Protocol available at [35], we can see that the serialized form of a KeyExchangeMessage consists of:

- A **version byte**,
- an unsigned integer **message ID**,
- a public **base key**,
- a public **ratchet key**,
- a public **identity key**, and
- a **base key signature**.

Normally, KeyExchangeMessages are generated by Signal Protocol's SessionBuilder class. Its source code confirms that the base key and the ratchet key are ephemeral, whereas the identity key is persistent for the device. All of these keys are public Elliptic curve Diffie-Hellman (ECDH) keys implemented in the org.whispersystems.libsignal.ecc package.

4.3.3 Length of a KeyExchangeMessage

Using the information from the previous section, the source code of Open Whisper Systems' Curve25519 implementation available at [31], and the encoding rules for Google's Protocol Buffers described at [14], we can calculate the length of a serialized KeyExchangeMessage.

The version byte is encoded separately from the Protocol Buffers structure and always takes up one byte at the beginning of the serialized message.

The message identifier (ID) is encoded by Protocol Buffers using a one-byte tag and a variable-length integer. For a 32-bit unsigned integer, the integer may be encoded as one, two, three, four, or five bytes.

However, in the current implementation, the ID is generated using the Java expression `(sequence << 5) | flags`[5], where `sequence` is a random number between 1 and 65534 (inclusive)[6]. The number 65534 is equal to $2^{16} - 2$. The resulting integer is thus always greater than or equal to 2^5 but less than 2^{21}, and can be encoded as one, two, or three bytes as a variable-length integer. As a result, the message ID always takes up between 2 and 4 bytes when encoded, including the one-byte tag.

The public keys are 33 bytes long when serialized[7]. Protocol Buffers add a one-byte tag and a one-byte length indicator, making the total encoded length equal to 35 bytes.

The base key signature is 64 bytes long[8]. Protocol Buffers add a one-byte tag and a one-byte length indicator, making the total encoded length equal to 66 bytes.

The individual parts of the serialized message are listed in table 4.2.

Component	Type	Decoded length	Encoded length
Version	Byte	1 B	1 B
Message ID	Unsigned integer	6–21 b	2–4 B
Base key	Public key	33 B	35 B
Ratchet key	Public key	33 B	35 B
Identity key	Public key	33 B	35 B
Base key signature	Signature	64 B	66 B
Total			174–176 B

Table 4.2: The components of a serialized KeyExchangeMessage.

The total length of the entire message is between 174 and 176 bytes for the current message version. If the sequence number is greater than or equal to 2^9, the length of the serialized message will be the maximum possible, i.e. 176 bytes. This will happen with 65023 out of the 65534 possible sequence numbers, or about 99.2% of the time.

[5]See the class org.whispersystems.libsignal.protocol.KeyExchangeMessage in [35].
[6]See the class org.whispersystems.libsignal.SessionBuilder in [35].
[7]See the constant org.whispersystems.libsignal.ecc.ECPublicKey.KEY_SIZE in [35].
[8]See the method org.whispersystems.curve25519.Curve25519#calculateSignature in [31].

Chapter 5

Design

5.1 NFC and secure messaging

In the previous chapter, we have listed examples of two categories of Android applications: ones using NFC, and ones intended for secure communication. None of the applications we found belonged to both of these categories. Therefore, our goal – using NFC for secure messaging – appears to be unprecedented.

In chapter 2, we identified some properties of NFC that could be beneficial to the security of a messaging application. Namely:

1. **Reduced attack surface,** see section 2.2.1. If a critical part of the security protocol, for example key agreement, can be moved to NFC, it may create an insurmountable barrier for attackers without physical access to the devices' vicinity.
2. **Required presence of both users.** This could be used for identity verification or for key signing. If NFC is required for this operation, users will be less likely to make mistakes such as relying on unauthenticated channels, because face-to-face communication does naturally not face the same authentication challenges as online communication.
3. **Resistance to Man-in-the-Middle attacks,** see section 2.2.3. If an application is built on this principle, the user will not have to take any steps to mitigate Man-in-the-Middle attacks – in fact, they will not even need to know that these attacks exist. Such an application would benefit in terms of ease of use and better security by default, thus taking some of the burden of security off of the users' shoulders.
4. **Secure element,** see section 2.2.4. Applications and data loaded into the secure element would be protected from unwanted access by malware in the phone, backup software, etc.

Later, in section 4.2, we found that existing secure messaging applications choose different approaches to identifying users. Phone numbers are convenient, but custom usernames offer greater privacy.

5.2 Requirements

Our goal in this chapter shall be designing an IM application with the following properties:

1. Users shall be able to exchange text messages over the Internet, provided they have physically met and performed a setup over NFC at least once.
2. All messages shall be end-to-end encrypted for confidentiality.
3. The messaging protocol shall provide forward secrecy and future secrecy.
4. The application shall use NFC to protect from Man-in-the-Middle attacks.
5. User identifiers shall not be linked to phone numbers.

5.3 Usage scenario

The system shall support the following scenario:

1. Alice installs a secure messaging application on her NFC-enabled Android device.
2. Alice locates Bob, who also has the same application installed on his device but who has not used it to communicate with Alice yet.
3. Alice and Bob enable "key exchange mode" in the application on their respective devices, then they tap their devices together to start the NFC exchange.
4. Alice's device reports success and shows Bob as a new contact, while Bob's device also reports success and shows Alice as a new contact.
5. At any point in the future, Alice may use the application to send messages to Bob and to receive messages from Bob, securely, over the Internet.
6. At any point, Alice may locate another person and begin the same procedure as with Bob starting from item 2.

5.4 Use of cryptography

Our design requires end-to-end encryption of messages. As we discovered in section 4.2, some existing Android applications already support this, and thanks to being open-source, the cryptographic protocols from these applications can be re-used in our application.

As we reasoned in section 4.3, Signal Protocol has the best properties of all the protocols we analysed, and it can be readily used in our design because an open-source library is available.

History has proven that designing cryptographic protocols is a complex task exceeding the scope of a Master's thesis. We shall therefore use the existing and audited cryptographic functions of Signal Protocol as much as possible, and refrain from designing our own.

5.5 Architecture

The messaging system shall consist of the following components:

Key exchange protocol.
This shall be a communication protocol used over the NFC interface for exchanging cryptographic material between users.

Message relay protocol.
This shall be a communication protocol used over the Internet for exchanging encrypted messages between users after the cryptographic material is established.

Client Application.
This shall be an application written for the Android operating system and using the NFC capability of its host device. It shall implement both of the protocols above. The remainder of its functionality falls outside the scope of this design.

Server Application.
This shall be an Internet service using the HTTPS/TCP/IP protocol stack for communication with the clients. It shall implement the message relay protocol as specified above. The remainder of its functionality falls outside the scope of this design.

The components are illustrated in fig. 5.1.

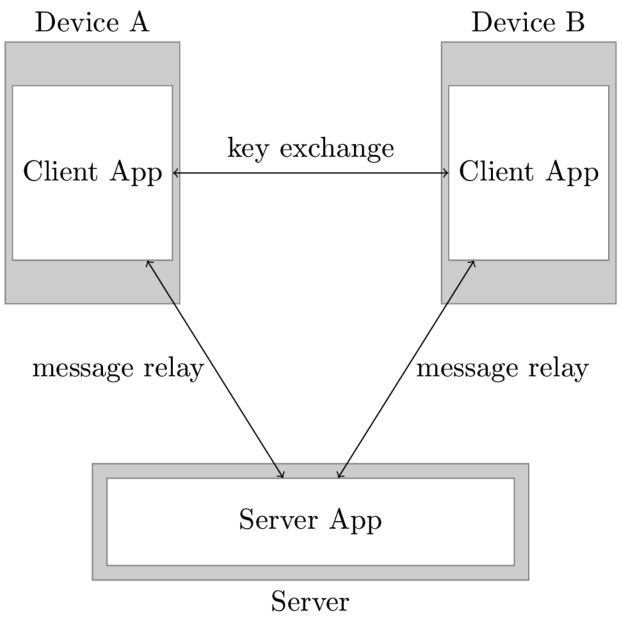

Figure 5.1: System architecture.

5.6 Naming

We chose the name "Touch and Chat" to refer to the design and the reference implementation. We registered the domain name touch-and-chat.eu for hosting an instance of the server application and created a Java package named eu.touch_and_chat.app for the client application.

5.7 Key exchange protocol

In this section, we will outline the considerations made for the design of the key exchange protocol.

5.7.1 Integration with Signal Protocol

The main purpose of the key exchange is to establish a new Signal Protocol session. The permitted ways of doing so were described in section 4.3.1.

Both PreKeyBundles and PreKeySignalMessages use PreKeys, which need to be stored by the server. Since we want to implement key exchange over NFC, we would prefer to avoid storing any related information on the server.

The remaining method simply involves one client sending a KeyExchangeMessage to the other and receiving another KeyExchangeMessage in response, as illustrated in fig. 5.2. This is more suitable for NFC, because there is no server storage involved and because these messages can be easily exchanged over an NFC link in a short time.

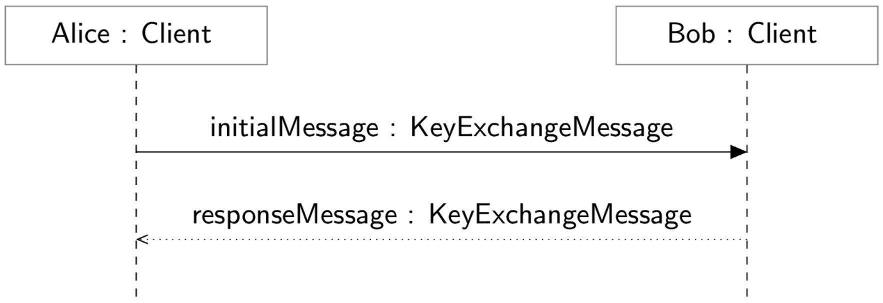
Alice : Client
Bob : Client
initialMessage : KeyExchangeMessage
responseMessage : KeyExchangeMessage

Figure 5.2: Using KeyExchangeMessages to establish a session.

We choose to use KeyExchangeMessages for the key exchange protocol, not only because the naming suggests so, but mainly because it puts very simple requirements on the protocol.

5.7.2 Integration with Android's NFC API

There are two possible methods of performing a two-way exchange of messages over NFC between two Android devices.

The first and logical method involves peer-to-peer mode on both devices. (See section 3.2.2 for the discussion on this topic.) However, the only way to use peer-to-peer mode with Android's current public API is Android Beam. As we mentioned earlier, this method provides only simplex communication. We can transfer one KeyExchangeMessage easily, but there is no documented way to obtain the response automatically. The only way would be to run Android Beam once more, in the opposite direction, but this would require cooperation with the user and could hamper user-friendliness.

The other method requires one device to run in card emulation mode (see section 3.2.3) and the other one to run in reader/writer mode (see section 3.2.1). This way, the devices can communicate in the same way an NFC reader would communicate with an ISO-DEP card with support for ISO 7816-4. In other words, the reader can send arbitrary command APDUs and receive arbitrary response APDUs from the emulated card.

We conclude that in order to make the key exchange user-friendly, we need to work around the limitations of Android's API and use a combination of card emulation mode and reader/writer mode.

To prevent the client devices from attempting peer-to-peer connections, one of the devices must be switched to *reader mode* (see section 3.2.1). The other device, on the other hand, must not be switched to *reader mode*, because the devices will not be able to detect each other if they are both in *reader mode*.

5.7.3 AID for card emulation

Before our key exchange protocol can be activated, Android needs a way to identify our application. This requires an AID, as was explained in section 3.2.3. We can register an AID or we can pick one in the proprietary range.

Since we are designing an experimental protocol, we created a proprietary AID using the following procedure:

1. We used the ASCII string "`touch-and-chat.eu`" as an application-specific value.
2. We used Secure Hash Algorithm 2, 256 bits (SHA-256) to calculate a hash of this string.
3. We truncated the hash output to the least significant 60 bits.
4. We prepended 4 bits with the value '1'.

The resulting 8-byte binary string is a valid AID in the proprietary range. Thanks to the hash function, our application-specific string is mapped to one of 2^{60} possible AIDs, and changing the string will result in a different AID from the same range with a roughly uniform probability distribution, so collisions with other AIDs should be unlikely.

In our case, the resulting AID consists of the following 8 bytes, in hexadecimal notation:

```
F5 58 4D A1 B3 02 B1 78
```

5.7.4 Integration with ISO 7816-4

Suppose we have the following setup: one device in card emulation mode, the other device in reader/writer mode, and both capable of generating and processing KeyExchangeMessages.

As far as the key exchange protocol is concerned, the first command APDU sent from the reader to the emulated card is always SELECT FILE (mentioned as SELECT AID in the

Android guide at [4]) containing the AID of the application, and this APDU is the first one delivered to the application.

According to [25, section 6.11], the response to SELECT FILE may or may not contain data. The data may be a File Control Information (FCI) template, a File Control Parameters (FCP) template, or a File Management Data (FMD) template depending on the options set in the command. These templates do not convey information useful to our protocol, so the response to SELECT FILE need not contain any data.

Signal Protocol implements both the serialization of a KeyExchangeMessage into an array of bytes and the reverse process. All we need to do in order to send these messages is to encapsulate them in valid APDUs. For the command APDU (reader to card), we can use the ENVELOPE command, which can be used to transmit "any data string which otherwise could not be transmitted by the available protocols." [25, section 7.2.1] The initial KeyExchangeMessage can be serialized into the data part of the command APDU, while the response KeyExchangeMessage can be serialized into the data part of the response APDU.

A KeyExchangeMessage is at most 176 bytes long in the current version, as we calculated in section 4.3.3. Thanks to this fact, we do not need to use extended length APDUs, which are not supported by some NFC adapters[1]. Standard APDUs have a limit of 255 payload bytes, which is sufficient in this case.

In conclusion, the key exchange can be managed with just two command-response pairs: one SELECT FILE for enabling the *service*, and one ENVELOPE to transmit both KeyExchangeMessages.

A concrete specification of this protocol is documented in appendix B and is also used in our reference implementation.

5.7.5 Integrity verification

To ensure the integrity of a run of the key exchange protocol, we need to verify that both KeyExchangeMessages were unchanged during transmission.

After completing a key exchange, both clients shall calculate a verification code using the scheme shown in fig. 5.3. Both KeyExchangeMessages shall be hashed using SHA-256, the hashes concatenated (with the initial KeyExchangeMessage's hash first), and the resulting concatenation hashed using SHA-256 again.

The verification code depends on the contents, the lengths, and the order of transmission of both KeyExchangeMessages. If the messages are transmitted without error or tampering, both clients should arrive at the same value of the verification code, which the user can then confirm. If the messages are corrupted during transmission, tampered with, or if they belong to a different run of the key exchange protocol, the verification codes should differ for each client. In that case, the results of the key exchange shall be discarded by the clients.

When presented to the user, the verification code shall be encoded as 64 hexadecimal digits with spaces between each pair.

[1]See the method android.nfc.tech.IsoDep#isExtendedLengthApduSupported in [1].

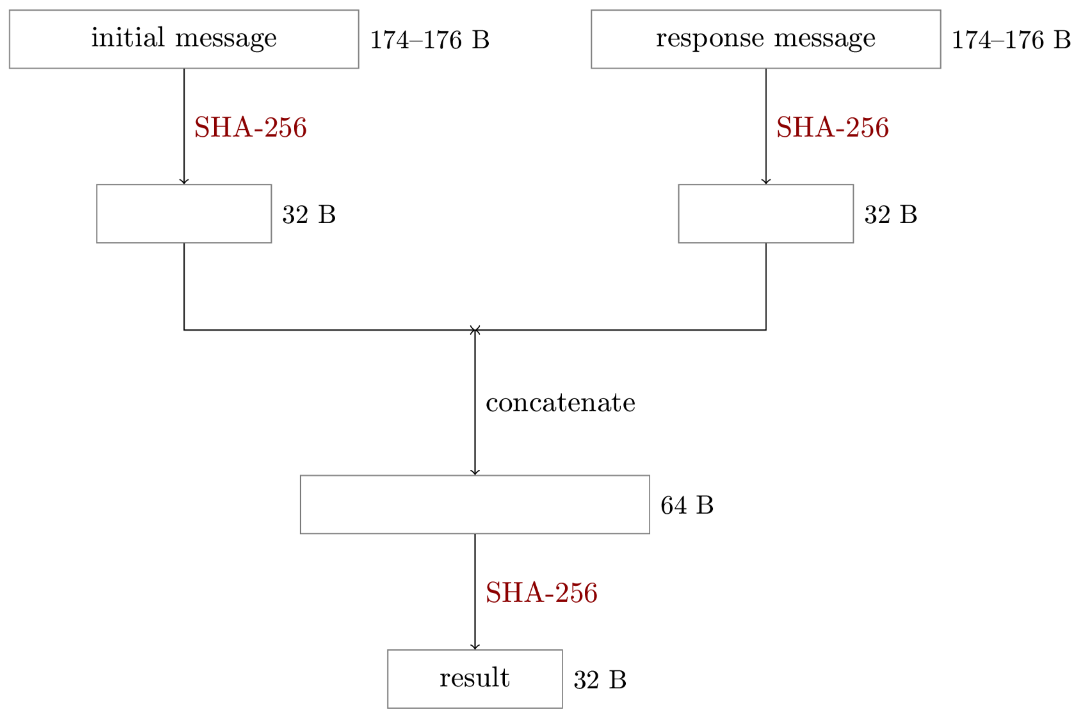

Figure 5.3: Calculation of the verification code.

5.7.6 Identifier for message relay

For the purposes of relaying messages over a server in the Internet, the clients need a way to identify each other. Two approaches could be used for this purpose:

Device identifiers. Each KeyExchangeMessage contains a public key (the identity key), which could be used. Alternatively, an ad-hoc identifier could be added to the protocol. HCE uses a device identifier but it is randomly generated, as was explained in section 3.2.3, so the NFC layer cannot be used to obtain such an identifier.

Session identifiers. The client could derive a shared value from the contents of the KeyExchangeMessages and use it as an identifier. The value could be derived using a hash function, in a fashion similar to the calculation of the verification code. Alternatively, we could exploit the fact that KeyExchangeMessages contain ephemeral public keys for an ECDH key agreement, as was found in section 4.3.2, so we can derive a shared secret from these values.

Because Android's NFC communication is not encrypted, an attacker can eavesdrop on the key exchange. Regardless of the method used for its generation, the identifier must be used in a way that does not allow an attacker to pose as one of the clients and hijack the message relay. Such a scenario would not result in loss of confidentiality because the attacker would still need the clients' private keys to decrypt the messages. However, an attacker could do the following with a successfully hijacked message relay:

- Collect metadata about the messages sent by the other client, e.g. size or time of sending.
- Collect encrypted messages sent by the other client, possibly as a step in an attack

on the encryption protocol.

- Send arbitrary messages to the other client, possibly as a step in an attack on the encryption protocol or on the client implementation.
- Cause a denial of service for the message relay session.

If ad-hoc device IDs are used, they must be registered with the message relay server beforehand. Otherwise, an attacker could learn a device ID through eavesdropping and attempt to be the first to register it, thus hijacking the message relay for the given device.

If public key-based device IDs are used, they do not need to be registered with the message relay server beforehand, but the server must verify that the client is in possession of the corresponding private key. This means that the server must support the same cryptographic primitives as the client.

If a hash-based session ID is used, it can be derived by the legitimate devices as well as an eavesdropping third party. There is no way for the server to differentiate these subjects, so the message relay may be hijacked easily.

The ECDH-based session ID should only be known by the two clients in possession of the private keys, and not by eavesdroppers who can only learn the public keys. This makes hijacking impossible by mere eavesdropping. The server does not need to know the devices' identifiers; clients simply have to prove knowledge of the same session ID in order to have messages relayed to each other.

Method	ID type	Can the ID be hijacked?
Public key	Device ID	Not if verified by the server.
Ad-hoc identifier	Device ID	Not if registered with the server beforehand.
Hash	Session ID	Yes.
ECDH	Session ID	Not if the session ID is kept private.

Table 5.1: Comparison of different methods to derive an identifier for message relay.

Based on the findings, as summarized in table 5.1, we choose ECDH-based session IDs generation.

Signal Protocol keeps a "root key" stored in each session. However, this root key is not the same for both clients in the session. Therefore, we cannot use this value to derive a session ID.

As was found in section 4.3.2, the KeyExchangeMessages each contain the following public keys:

- Base key (ephemeral).
- Ratchet key (ephemeral).
- Identity key (permanent).

We can calculate an ECDH agreement using the conveniently available method Curve.calculateAgreement in the package org.whispersystems.libsignal.ecc. We need to use ephemeral keys, so we can use the base key or the ratchet key. Since we are re-using keys used in a third-party cryptographic protocol, we must take precautions so that we do not weaken its security.

To generate the session ID, we use the base keys used in the key exchange, calculate an ECDH agreement, then use a HMAC-based key derivation function (HKDF) to generate a shared secret, and finally encode the result using Uniform Resource Locator (URL)-safe Base64 as desribed in RFC 3548 [26], without padding or line terminators.

An implementation of HKDF is available in Signal Protocol's package org.whispersystems.libsignal.kdf. This HKDF corresponds to RFC 5869 [29] and uses SHA-256 as its hash function. We use the ASCII string "`touch-and-chat.eu#session-id`" as the context-specific information to diversify the output, and we do not provide a salt.

The use of HKDF serves two purposes: to diversify our session ID from other shared secrets that may be generated from the same ECDH keys, and to prevent attackers from learning the original ECDH agreement value and using it in an attack on Signal Protocol. If our method of generating the session ID is found to be weak, it will enable session hijacking attacks, but it should not weaken the cryptography used to protect the message contents.

Base64 is used to make the session ID suitable for use in a URL, which will be useful when specifying the message relay protocol.

The concrete procedure of generating the session ID is described by the following Java code:

```java
import android.util.Base64;
import org.whispersystems.libsignal.ecc.Curve;
import org.whispersystems.libsignal.ecc.ECPrivateKey;
import org.whispersystems.libsignal.ecc.ECPublicKey;
import org.whispersystems.libsignal.kdf.HKDFv3;
import java.nio.charset.StandardCharsets;

class SessionIdGenerator {
    static byte[] CONTEXT = "touch-and-chat.eu#session-id"
                            .getBytes(StandardCharsets.US_ASCII);
    static String getSessionId(
            ECPublicKey theirPublicKey,
            ECPrivateKey ourPrivateKey
    ) {
        byte[] agreement = Curve.calculateAgreement(
                theirPublicKey,
                ourPrivateKey
        );
        byte[] bytes = new HKDFv3().deriveSecrets(
                agreement,
                CONTEXT,
                32
        );
        return Base64.encodeToString(
                bytes,
                Base64.NO_PADDING | Base64.NO_WRAP | Base64.URL_SAFE
        );
    }
}
```

5.7.7 Forward compatibility

This protocol is not designed with future upgrades in mind. However, future versions may use a different AID to differentiate themselves from past versions. Devices in reader mode may probe known AIDs to discover the most recent version of the protocol available.

5.8 Message relay protocol

This section gives an outline of the communication protocol used for exchanging messages over the Internet.

An inspiration for the message relay protocol is drawn from the TextSecure Server API described at [34]. This is the API used in *Signal*, designed to be used over a Hypertext Transfer Protocol, Secure (HTTPS) connection, with complex data structures serialized using JavaScript Object Notation (JSON).

However, we cannot use this API directly, since the following features are not suitable for our application:

- Verification of phone numbers. Our application does not use phone numbers. Adding them as a requirement would nullify one of the key privacy benefits.
- PreKeys. Our application does not use these.
- Contact list intersection. Our application uses NFC for contact discovery, making this feature not only unnecessary, but also less privacy-preserving.

Additionally, the following features can be left out for simplicity:

- Push service registration.
- Support for attachments.

For these reasons, we are going to design our own API, which will be better suited for our application.

5.8.1 API endpoints

The required functionality can be described by two simple functions:

- sendMessage(SessionId, Message)
- receiveMessages(SessionId) : Message[]

In the above function prototypes, SessionId is a private ID generated by two clients according to section 5.7.6, and Message represents an encrypted payload generated by Signal Protocol and opaque to the server.

These two functions would be enough for a working server, but their use would be inefficient: A client must poll for messages in each session it knows, which may result in a large number of requests being made each time the client wants to check for messages.

To make polling for messages more efficient, we add functions that allow a client to register any number of sessions and poll for all messages at once:

- registerDevice(Password) : UserId
- registerSession(UserId, Password, SessionId)
- sendMessage(UserId, Password, SessionId, Message)
- receiveMessages(UserId, Password) : (Message, SessionId)[]

Now the client must register its device by choosing a Password and receiving a UserId, and it must use these credentials in all other functions. The benefit is that the session ID does not need to be specified when polling for messages, and is instead provided by the server for each message so that its sender can be identified.

Still, the polling is inefficient: The server has no way of knowing which messages the client has already downloaded, and which ones it has not. There is no guarantee that a Hypertext Transfer Protocol (HTTP) client has received and completely processed a response, so if the received messages are lost in transmission or if the client crashes while processing them, the server must keep them stored so that it can try to deliver them again. As a result, the server must always send **all** applicable messages when polled. This list of messages will be constantly growing, reducing the efficiency of the protocol and possibly even causing overloads.

To combat the above problem, we add numbers to messages, picked from a monotonically increasing sequence for each recipient, so that the client can include the number of the last received message when polling and the server can only send messages newer than that.

Finally, we add the unregisterDevice function to complement registerDevice, and the block Session function to complement registerSession. The resulting protocol contains the following functions:

- registerDevice(Password) : UserId
- unregisterDevice(UserId, Password)
- registerSession(UserId, Password, SessionId)
- blockSession(UserId, Password, SessionId)
- sendMessage(UserId, Password, SessionId, Message)
- receiveMessages(UserId, Password, Number) : (Message, SessionId, Number)[]

A concrete specification of the final message relay API is documented in appendix C.

5.8.2 Protocol flow

The protocol places the following restrictions on the order of requests issued by a client:

- When using a UserId and Password in any request other than registerDevice, they must be first registered using registerDevice.
- When using a SessionId in sendMessage, the SessionId must be first registered using registerSession.

Otherwise, the protocol is stateless and its requests can be called in any order.

The recommended client behavior is the following:

- Call registerDevice as the first request after installation.
- Call registerSession when a new contact is added.
- Call blockSession when a contact is removed by the user.
- Call sendMessage for each message sent by the user.
- Call receiveMessages periodically to check for new messages.
- Optionally, call unregisterDevice before uninstallation.

5.8.3 Security considerations

Some data in the protocol (namely Password and SessionId) should not be disclosed to anyone other than the legitimate client and the server. The connection must be therefore protected using Transport Layer Security (TLS). Using plain HTTP instead of HTTPS would open a vulnerability which could allow attackers to hijack sessions.

5.8.4 Forward compatibility

New API endpoints can be added and new HTTP request headers, HTTP query parameters, HTTP response parameters, and items to existing JSON objects can be added to any API endpoint, as long as legacy clients can continue using the API while ignoring these additions.

The server may send a "200 OK" response with a body instead of a "204 No Content" response without a body. Clients expecting a "204 No Content" response should also expect the "200 OK" response with the same semantics, except the body of the response can be ignored.

In order to support compatibility-breaking changes, the URLs used by this protocol shall contain the component `/v1/`, short for "version one." If a change needs to be introduced that would result in breaking compatibility with legacy clients, a new set of functions can be created with a different version component in the URL, such as `/v2/`. A server may support multiple versions of the protocol in this way, and thus keep support for legacy clients while introducing new features.

5.9 Client application

The client application for Android needs to store the user's private key in a secure location. The security tips at [9] list the following options:

- Internal storage with default settings. Files created in this way are only accessible to the application that created them, which "is sufficient for most applications."

- KeyStore, which can protect the key “with a user password that is not stored on the device.” The documentation says that “while this does not protect data from a root compromise that can monitor the user inputting the password, it can provide protection for a lost device without file system encryption.”

As of version 3.13.1, *Signal* for Android stores its private keys in Android’s SharedPreferences framework, which uses the internal storage as mentioned above.

The implementation should use one of the solutions outlined in this section.

5.10 Server application

Since the server’s API is publicly accessible and the number of registrations per client is theoretically unlimited, the server should implement rate limiting as a protection against abuse.

Chapter 6

Implementation

This chapter will detail our reference implementation of the client application and the server application.

6.1 Client application

The minimum Android version required by the client application is version 4.4 because host-based card emulation is not available in earlier versions.

The target Android version for the client is version 6.0 because this was the latest version at the time of implementation.

A user manual for the client application can be found in appendix D.

6.1.1 Used libraries

The client application is compiled with the following third-party libraries:

Signal Protocol. [35]
: A key component of the whole application. Used for generating keys, maintaining sessions, and message encryption.

Jackson databind. [1]
: Used for converting between plain Java objects and JSON, which is used for building request bodies and parsing response bodies in the message relay protocol.

OkHttp. [2]
: Used for communicating with the message relay server.

All these libraries are also used by *Signal* for Android. Signal Protocol is included because our design is built on top of it. The other two libraries were found suitable for our implementation, so for simplicity, we replicated the choices of libraries for *Signal*.

[1]Project home page: `<https://github.com/FasterXML/jackson-databind>`
[2]Project home page: `<https://square.github.io/okhttp/>`

6.1.2 Packages

The client application is divided into the following Java packages:

eu.touch_and_chat.app
The base package of the whole application. Contains standard Android application components such as *activities*, *services*, and *list adapters*, as well as general purpose classes.

eu.touch_and_chat.app.iso7816_4
Contains classes for building and parsing APDUs according to ISO 7816-4.

eu.touch_and_chat.app.key_exchange
Contains classes implementing the key exchange protocol.

eu.touch_and_chat.app.message_relay
Contains classes implementing the message relay protocol.

eu.touch_and_chat.app.storage
Contains classes related to storing persistent application data, such as cryptographic keys, session data, contact information, and messages.

eu.touch_and_chat.app.util
Contains utility classes, i.e. classes that provide static methods with re-usable functionality.

6.1.3 Integration with Signal Protocol

Generating the identity key pair and registration id. The file README.md at [35] requires a client to "generate its identity keys, registration id, and prekeys" at install time. This information is somewhat inaccurate because:

1. PreKeys are not needed if sessions are built using KeyExchangeMessages and
2. the identity keys and registration ID may be generated later than at install time, as long as they are generated before any other library functions are used.

Since Android does not provide a way to run code at install time, our implementation uses lazy generation, i.e. it generates the values at the time they are first used. This is implemented in the class StorageHelper.

PreKeys are not generated.

Storing persistent data. The file README.md at [35] requires a client to implement four storage interfaces: IdentityKeyStore, PreKeyStore, SignedPreKeyStore, and SessionStore.

Alternatively, as can be discovered in the documentation of the class SessionBuilder, the client can implement the SignalProtocolStore interface, which is a wrapper for all four interfaces mentioned above.

We created a single class, StorageHelper, for storing persistent data. This class implements SignalProtocolStore and adds additional methods not required by Signal Protocol.

Since we are not using PreKeys, we implemented PreKeyStore and SignedPreKeyStore with method stubs that throw an UnsupportedOperationException. As long as we only use KeyExchangeMessages for session establishment, these methods will never be called.

The identity key pair and local registration ID are stored using Android's SharedPreferences mechanism. Remote clients' identities are stored in an SQLite database. Session records are stored as monolithic files in internal storage. All these choices mimic the implementation of *Signal* for Android.

Addressing. Signal Protocol identifies the parties in a session using a SignalProtocolAddress, which consists of a string name and an integer device ID. Since our current design does not support multiple devices per client, we always set the device ID to 0 and do not expect any other values. For the name, we use the session ID generated during key exchange. To make sure the name does not clash with *Signal*'s addresses, should these two projects ever merge, we add the prefix "`nfc-paired:`" and always expect this prefix to be present.

Session establishment. Normally, sessions in Signal Protocol should be built using SessionBuilder. The constructor of this class requires an implementation of SignalProtocolStore and the recipient's SignalProtocolAddress as parameters. The recipient's address is used as a key for storing information about the session being built.

Unfortunately, during a key exchange over NFC, we do not know the identity of the other party. Referring back to section 5.7.6, we can see that an ID is assigned to the session only after it has been built, because it is generated from the contents of two KeyExchangeMessages.

To deal with this, we reimplemented Signal Protocol's SessionBuilder in our own class called KeyExchangeEngine. Our implementation generates KeyExchangeMessages and creates sessions just like the library implementation, but does not store any intermediate or resulting session information. All intermediate information can be kept in transient memory because the key exchange over NFC is near-instantaneous. Instead of storing the resulting session persistently, we create an instance of PendingSession, which contains the session record to be saved and its ID, as well as its verification code, generated according to the design in section 5.7.5, and we pass this object to VerifyContactActivity. The session is only saved after the user verifies it.

6.1.4 Observations

KeyExchangeMessages generated by the application were observed to always be between 175 and 176 bytes long, which is consistent with the calculations in section 4.3.3. We also manually dissected one message to verify that it is indeed encoded in the way we described.

6.1.5 Unsolved problems

Even though we explicitly disable Android Beam in every *activity* of the application, we have encountered situations where the Android Beam animation was still shown, especially when holding the two devices together for too long while *activities* were changing.

The state of the NFC adapter as reported by Android appears to be inconsistent. Moreover, the documentation does not specify what methods can be called on an NFC adapter that is disabled, e.g. whether *reader mode* should be disabled. We ran into various kinds of problems where the device would fail to detect the changed state of the NFC adapter, fail to re-enable *reader mode*, scan one tag multiple times, or hang for several seconds because of internal issues with an inactive NFC adapter. We attempted to work around most of these problems, but some related bugs may still be present in the final implementation. We suspect that some of these bugs may depend on the version of the operating system.

6.2 Server application

The server software, programming language, framework, and other technologies can be used completely arbitrarily when implementing the server application. We chose the following setup because of our previous experience with it:

- Server software: Apache
- Programming language: PHP
- Framework: Symfony
- Database: MySQL for production, SQLite for development

We used Symfony REST Edition as a basis for the project.

A guide for installing the server application can be found in appendix E.

6.2.1 Architecture

The architecture of the server application follows the model-view-controller (MVC) pattern. Its classes are organized into the following main types:

Model entities.
Used for defining persistent data kept by the server. These form the model layer of the application. An additional class named Model is used for high-level operations with these entities.

REST entities.
Used for decoding the request bodies and encoding the response bodies in the API. These form the view layer of the application.

Controllers.
Used for interacting with the user. These form the controller layer of the application.

Additionally, automated functional tests were implemented to ensure proper functionality of the server.

6.2.2 Model entities

We created the following entity types to be handled by the application:

Device
: This corresponds to a client device registered through the API. The ID is generated by the server, while the password is provided by the client. The password is encoded in storage using bcrypt.

Session
: This corresponds to a session to relay messages between two client devices. The ID is determined by the client devices. A session may be blocked explicitly on client request, implicitly when a related client device deletes its registration, or if more than 2 client devices try to register the session. Once blocked, a session may never be unblocked again.

DeviceSession
: This represents a many-to-many relation between client devices and sessions. In practice, a session can never be associated with more than two devices.

Messages
: This represents an encrypted message sent from one client device to another through a session. A message's recipient may be unknown in the recipient has not registered the session yet. Messages with a known recipient are assigned numbers from an increasing sequence for each given recipient, so that the recipient may request not to receive already received messages when polling for new messages (see appendix section C.6).

These entities are persisted into a relational database through Doctrine ORM.

Entity attributes and relationships between entities can be seen in fig. 6.1.

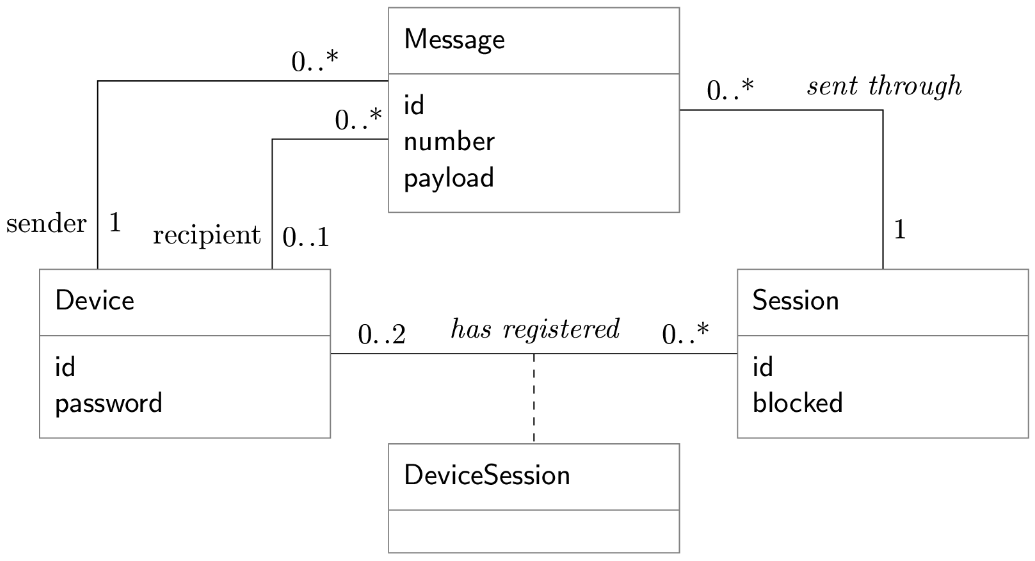

Figure 6.1: Entity-relationship diagram of persistent data on the server.

6.2.3 REST entities

One REST entity was added for each request body and each response body used in the API. These are:

DeviceRegistrationRequestBody
Corresponds to the body of the request in appendix section C.1.

DeviceRegistrationResponseBody
Corresponds to the body of the response to the request in appendix section C.1.

SendMessageRequestBody
Corresponds to the body of the request in appendix section C.5.

PollMessagesResponseBody
Corresponds to the body of the response to the request in appendix section C.6.

These objects can be converted to and from JSON using Johannes Schmitt's Serializer[3] library.

6.2.4 Controllers

Three controllers were implemented in the server application, one for each logical part of the API:

DeviceController
This handles the registration and unregistration of client devices. See appendix sections C.1 and C.2.

SessionController
This handles registering and blocking sessions. See appendix sections C.3 and C.4.

Additionally, it contains the method removeTestSessions, accessible in testing mode only, to remove leftover sessions, created by automated tests, from the database.

MessageController
This handles sending and receiving messages. See appendix sections C.5 and C.6.

6.2.5 Automated tests

To ensure correct behavior of the server, we implemented functional tests for each of the server's controller classes. These tests cover all documented uses of the API and can be run with PHPUnit[4].

Because the server application does not contain complex units that could be tested separately, no unit tests were implemented.

[3]Project home page: <http://jmsyst.com/libs/serializer>
[4]Project home page: <https://phpunit.de/>

6.2.6 Deployment

For the purposes of demonstrating our implementation, the server application has been set up to run at <https://touch-and-chat.eu>. We are only going to maintain this instance as long as needed for the defense of this thesis. Both the domain and the web hosting are going to expire on April 4th, 2017, after which the server will likely become unavailable.

Chapter 7

Evaluation

7.1 Attack surface

The system consists of two entities (the Client device and the Message relay server, not counting the user) and three interfaces (the User interface, the Key exchange channel, and the Message relay channel). Together, these constitute five points of attack.

7.1.1 User interface

The following attacks can be carried out against the user and their use of the client application:

Malicious code delivery. The user can be tricked into using a modified version of the client application, which may contain added security vulnerabilities, e.g. backdoors, or connect to an illegitimate server.

Key exchange verification bypass. The user may fail to compare verification codes shown by the application when adding a new contact. A compromise of the integrity of the key exchange may thus go undetected.

7.1.2 Client device

The following attacks can be carried out against the device running the client application:

Device theft. Unless the user activates device encryption and uses a strong screen lock, a thief may examine the contents of a stolen device's memory, read stored messages as well as newly received messages, and send messages under the device's original owner's identity.

Device compromise. A backdoor, rootkit, or unauthorized system-level access to the device due to a vulnerability may enable attackers to access the memory of the device, even if encryption is used. The attacker may read stored messages as well as newly created messages, send messages under the legitimate user's identity, and monitor the user's behavior.

7.1.3 Key exchange channel

The following attacks can be carried out against the key exchange protocol by an attacker with access to an NFC interface in the vicinity of two devices at the time they perform a key exchange:

NFC eavesdropping. The attacker may learn the contents of the APDUs transmitted, including the devices' identity key and the ephemeral keys used for the key exchange.

NFC Man-in-the-Middle. If the attacker overcomes the obstacles cited in section 2.2.3, they may create separate sessions with each of the legitimate devices. If the users fail to verify the integrity of the key exchange, they may unknowingly send messages to the attacker, who will then be able to decrypt these messages. The attacker may also re-encrypt and forward these messages to their original recipients. Forwarding can be done using legitimate API calls to the message relay server and does not require the server to be under the attacker's control.

NFC DoS. The attacker may transmit at the same time as the legitimate devices, creating interference in the magnetic field and preventing the key exchange from completing successfully.

7.1.4 Message relay channel

The following attack can be carried out against the message relay server from any node in the Internet:

DoS against the server. If the attacker overwhelms the message relay server with traffic, using either a regular DoS or a distributed denial of service (DDoS) attack, legitimate clients may not be able to use the message relay.

The following attacks can be carried out against the message relay protocol by an attacker with access to a privileged Internet node, e.g. a router between the client and the server, or a Domain Name System (DNS) server used by the client:

DNS spoofing. The attacker may provide the client with false information about the server's Internet Protocol (IP) address, causing the connection to be routed to the incorrect server. This may either be a DoS attack or, if the server is controlled by the attacker and logs incoming requests, this attack may be used to steal the client's authentication credentials for the message relay protocol.

HTTPS Man-in-the-Middle. If the user bypasses certificate verification on their device, or if the attacker manages to acquire a valid certificate for the message relay server, e.g. through a compromised certification authority (CA), they may insert themselves into the TLS connection and read, modify, insert, or drop data as they see fit. Through this, the attacker could gain access to the client's authentication credentials for the message relay protocol.

Using the knowledge of the client's credentials, an attacker can then hijack the client's message relay sessions, possibly causing further DoS, metadata leaks, and encrypted message leaks.

7.1.5 Message relay server

The following attacks can be carried out by an administrator in control of the message relay server or by an attacker having compromised the server:

DoS. The attacker may drop or refuse any or all connections or individual requests, or they may block legitimate sessions, making the message relay unusable by any number of clients.

Response modification. The attacker may alter any response to any request issued by the client, even in ways unexpected by the client. This may be used to exploit vulnerabilities in the client software, to increase the user's data bill, or simply to crash the client software.

Metadata analysis. The attacker may collect information about clients' source IP addresses, dates and times of requests, volume of requests, etc., and they may also use the registered sessions to construct a social network of the users.

Cryptographic break. The server has access to all encrypted messages exchanged among its clients, so if a cryptographic weakness is found in the Signal Protocol, an attacker in control of the server may try to use this weakness to decrypt messages.

7.2 Comparison to *Signal*

Both *Touch and Chat* and *Signal* use Signal Protocol for encrypting messages end-to-end and an HTTPS service to relay messages.

Signal uses the server for key exchange, which enables the server to carry out Man-in-the-Middle attacks on all of its clients. In contrast, *Touch and Chat* uses a local NFC connection for key exchange. Attackers can therefore only carry out Man-in-the-Middle attacks against devices in their physical vicinity, which greatly limits the number of affected devices, and they are forced to use a wireless interface which makes these attacks more difficult than if the communication was routed through the attacker's device over wires.

Signal uses phone numbers to identify users, to discover contacts, and to limit abuse. In contrast, *Touch and Chat* uses randomly-generated user identifiers in the message relay protocol, but does not disclose these identifiers to other users; contacts are discovered by physical presence through NFC; client abuse is prevented because clients can only receive messages from their contacts, and server abuse must be prevented through rate limiting. As a result, users of *Touch and Chat* are still protected from unsolicited messages, with the added benefit that their phone number is shared with neither the server nor their contacts. This makes *Touch and Chat* more suitable for users who are opposed to sharing their phone number with untrusted people, yet still have a need to communicate with those people.

Many features found in *Signal* are missing in *Touch and Chat*. See the following section for examples.

7.3 Possible improvements

While our design and our reference implementations are complete enough to demonstrate the concept of this thesis, many features were deemed unessential, and thus omitted, even though they could add other qualities to our project. Some of these features will be listed in this section.

Standard messaging application functionality. These features, commonly found in mobile messaging applications, would undoubtedly make using the application a better experience. These features include:

- Automatic receiving of new messages. For example, *Signal* does this efficiently by utilizing Google Cloud Messaging (GCM).
- Notifications about new messages.
- Message delievery reports.
- User avatars.
- Support for attachments such as pictures.
- Message backup.
- Storage encryption. This is not strictly necessary since Android provides full device encryption, but this could be useful for users who do not want to encrypt their entire device or for those who want another layer of protection.

Rather than add those features to our application, it would perhaps be more reasonable to instead add NFC key exchange functionality into an application that already has all those features, such as *Signal*.

Removing identity keys. Signal Protocol uses identity keys to sign key material used for establishing a session. This is technically not necessary when exchanging keys over NFC, provided that the users check the integrity of the key exchange. By dropping the identity keys, all sessions would be based solely on ephemeral keys and no permanent identification would be transmitted over NFC. This would provide better anonymity among users. In addition, attackers eavesdropping on the key exchange would be unable to learn the subjects' identity keys.

Known device detection. The current design of the key exchange protocol does not detect if the same two devices have participated in the protocol in the past, and thus allows them to create duplicate sessions. This could be avoided by comparing the identity keys from the key exchange messages with known keys in the database of contacts.

The two above points are mutually exclusive, because it is not possible to detect known devices without a permanent identifier. To keep the users anonymous while preventing duplicate sessions, a more complex key exchange would have to be designed. For example, the devices could compare their databases of known session IDs using Bloom filters.

Short authentication strings. To make the key exchange integrity verification more user-friendly, shorter fingerprints could be used. However, simply truncating the hash value could enable attackers to pre-compute messages with hash collisions and perform undetectable Man-in-the-Middle attacks. This can be mitigated by using hash commitment, as is described, for example, in [36].

Mediated contacts. The current design requires users to meet in person before they can exchange messages. While this allows for very secure authentication, it may be restrictive for users who are physically distant or do not usually meet each other in person. A more complex design could be created in a manner similar to the *web of trust* found in Pretty Good Privacy (PGP), where two users could establish a secure session using a chain of trusted mediators instead of physically meeting each other.

Feedback about session status. Our reference implementation expects users to correctly verify the integrity of the key exchange and to provide the same answer to their respective devices. If one user answers "yes" but the other user answers "no", an incomplete session will be created. The user who answered "yes" will see the session and will be able to send messages to it, but these messages will never be delivered. A new feature could be added to the message relay protocol so that answering "no" would preemptively block the new session and prevent the other user from using it.

Client-side message deletion. The current implementation stores all sent and received messages indefinitely, in a decrypted state. While this may be a convenience for the user, it also leads to greater damage in case the device's storage is compromised. Deleting messages after the user has read them or after a certain period of time would limit the number of messages leaked due to such compromise.

Server-side message deletion. There is no need for the server to keep messages that have been successfully delivered. The message relay protocol could be amended with a confirmation from the client that certain messages have been delivered, so that the server can safely delete them. Alternatively, the existing protocol could be kept mostly unchanged, but the server could silently delete messages whose number is less than or equal to the number used in the client's last polling request. This would require only a slight change in the protocol specification, which currently assumes that messages are never deleted.

Certificate pinning. By default, the HTTPS client is configured to accept connection to any server with a valid certificate chain containing one of the operating system's trusted CAs. However, an attacker may create their own server with a valid certificate, then use DNS spoofing to route the client's connection to their own server. With certificate pinning, the client would only establish connections to a server holding a specific certificate, thus mitigating this type of attack.

Rate limiting. This is necessary to prevent abuse in large-scale deployment.

7.4 Suggested improvements to Android

The lack of support for arbitrary peer-to-peer communication over NFC has been an impediment to our implementation and has caused the user interface to be less user-friendly than what we would consider optimal. We suggest that Android adds the following functionality: If two compatible devices come into each other's proximity and are running the same application and *activity* in the foreground, the two instances of the application should be able to establish a peer-to-peer NFC link for two-way communication. We believe that this workflow is natural to certain types of actions, e.g. exchanging contacts or linking game accounts, and that this feature would not only be convenient for developers, but user-friendly as well.

Ideally, Android should also provide a way for the developers to determine whether their application is currently running the "initiator" or "target" role, in order to break symmetry.

Optionally, Android could also add link-level security to these peer-to-peer connections. For example, NFC-SEC [18] could be implemented to add sequence integrity, confidentiality, data integrity, and origin authentication to the transmitted data, as well as to generate a shared secret between the two peers. If this feature is built into the system, it will be possible for developers to build simpler key agreement protocols over NFC, as well as to transmit user data with a reasonable level of confidentiality and integrity.

7.5 Suggested improvements to Signal Protocol

Several design choices of Signal Protocol were found unsuitable for our implementation and had to be worked around. The following modifications would make these workarounds unnecessary, resulting in more efficient and more secure code, while maintaining compatibility with previous versions.

Shared secret derivation. Signal Protocol generates a shared secret internally when establishing a session, but does not make this shared secret available to users of the library. Our implementation required a derived shared secret to be used as the session ID, so we resorted to re-using keys from the KeyExchangeMessages and calling undocumented cryptographic methods, as was described in section 5.7.6. A better solution could be used if Signal Protocol exposed a method to derive arbitrary shared secrets from the root key of a newly established session.

Anonymous sessions. The SessionBuilder class is used for creating KeyExchangeMessages, building sessions, and storing the resulting data. All three of these actions are coupled together and cannot be run separately. For reasons explained in section 6.1.3, the functionality of SessionBuilder had to be reimplemented so that keys can be generated first, without a storage or an address, and the session and identity information can be stored later. We suggest that Signal Protocol's SessionBuilder is refactored so that storage is decoupled from creating messages and sessions.

Chapter 8

Conclusion

In the beginning of this thesis, we researched the state of the art regarding NFC, a young technology for connecting devices over short distances, in relation to Android and IM. We used these findings to defend a term project and we also submitted them to an international conference. [28]

We then used this knowledge to design and implement a basic text messaging system that combines the advantages of Signal Protocol, a state-of-the-art secure instant messaging protocol, and NFC.

The client application, named *Touch and Chat*, maintains a list of contacts that can be populated by physically tapping compatible devices with NFC enabled. The application automatically exchanges cryptographic keys and asks the users to verify that the communication was not tampered with. Afterwards, the application works just like a typical IM application, using an Internet server to relay messages to other users. The messages are encrypted using Signal Protocol.

The server application, temporarily hosted at `<https://touch-and-chat.eu/>`, offers a simple API for registering accounts, setting up sessions, and relaying encrypted messages. The server never sees any plaintext messages or the clients' identity keys.

Compared to *Signal*, an IM application and the parent project of Signal Protocol, our implementation provides only a minimum of features, but improves upon security by making Man-in-the-Middle attacks more difficult to mount. It also improves user privacy by removing the need to use phone numbers as identifiers.

We believe that our approach brings innovation into the field of mobile communication, as we have not yet encountered an application that would combine NFC with instant messaging, nor have we found an application with the same privacy properties as our design offers.

Further work on this project could either aim to extend the reference implementation into a full-featured IM system, or alternatively, our system could be merged with an existing one, such as *Signal*, to give the users of the existing system the aforementioned security and privacy benefits while also increasing the application's user friendliness. Other applications could also adopt the idea of exchanging cryptographic material over NFC.

Bibliography

[1] *android.nfc* [online]. Android Developers [cited 2016-05-16]. Available at: <https://developer.android.com/reference/android/nfc/package-summary.html>.

[2] *Codenames, Tags, and Build Numbers* [online]. Android Open Source Project [cited 2016-05-16]. Available at: <https://source.android.com/source/build-numbers.html>.

[3] *Google Play Store* [online]. [cited 2016-01-01]. Available at: <https://play.google.com/store/>.

[4] *Host-Based Card Emulation* [online]. Android Developers [cited 2016-05-16]. Available at: <https://developer.android.com/guide/topics/connectivity/nfc/hce.html>.

[5] *Intents and Intent Filters* [online]. Android Developers [cited 2016-05-22]. Available at: <https://developer.android.com/guide/components/intents-filters.html>.

[6] *Near Field Communication* [online]. Android Developers [cited 2016-05-16]. Available at: <https://developer.android.com/guide/topics/connectivity/nfc/index.html>.

[7] *NFC Basics* [online]. Android Developers [cited 2016-05-16]. Available at: <https://developer.android.com/guide/topics/connectivity/nfc/nfc.html>.

[8] *Qué es la Tarjeta bip!* [online]. [cited 2016-05-16]. Spanish, Available at: <http://www.tarjetabip.cl/como-funciona.php>.

[9] *Security Tips* [online]. Android Developers [cited 2016-05-16]. Available at: <https://developer.android.com/training/articles/security-tips.html>.

[10] *Sharing Files with NFC* [online]. Android Developers [cited 2016-05-16]. Available at: <https://developer.android.com/training/beam-files/index.html>.

[11] *What is Silent Phone?* [online]. 2015-09-17 [cited 2016-03-04]. Original has been modified since citation, Available at: <https://web.archive.org/web/20160304091202/https://support.silentcircle.com/customer/en/portal/articles/2118686-what-is-silent-phone->.

[12] *How do I download and activate Silent Phone on my device?* [online]. 2015-11-06 [cited 2016-05-16]. Available at:

<https://support.silentcircle.com/customer/en/portal/articles/2118898-how-do-i-download-and-activate-silent-phone-on-my-device->.

[13] *Octopus Hong Kong : Where Can I Use It?* [online]. 2016-03-11 [cited 2016-05-16]. Available at: <http://www.octopus.com.hk/get-your-octopus/where-can-i-use-it/en/index.html>.

[14] *Protocol Buffers : Encoding* [online]. Google Developers, 2016-04-14 [cited 2016-05-16]. Available at: <https://developers.google.com/protocol-buffers/docs/encoding>.

[15] Ballinger, C. *ChatSecure* [online]. [cited 2016-05-16]. Available at: <https://chatsecure.org/>.

[16] Berners-Lee, T.; Fielding, R. T.; Nielsen, H. F. *RFC 1945 : Hypertext Transfer Protocol – HTTP/1.0* [online]. Internet Engineering Task Force, May 1996 [cited 2016-05-16]. Available at: <https://tools.ietf.org/html/rfc1945>.

[17] Dobie, A. *Jelly Bean feature: Sending photos and videos over Android Beam* [online]. Android Central, 2012-07-05 [cited 2016-05-16]. Available at: <http://www.androidcentral.com/jelly-bean-feature-sending-photos-and-videos-over-android-beam>.

[18] ECMA. *ECMA-385 : NFC-SEC : NFCIP-1 Security Services and Protocol.* 4th edition. ECMA, June 2015. Also approved as ISO/IEC 13157-1. Available online at: <http://www.ecma-international.org/publications/standards/Ecma-385.htm>.

[19] Electronic Frontier Foundation. *Secure Messaging Scorecard* [online]. 2016-04-05 [cited 2016-05-16]. Available at: <https://www.eff.org/node/82654>.

[20] Farrell, S.; Tschofenig, H. *RFC 7258 : Pervasive Monitoring Is an Attack* [online]. Internet Engineering Task Force, May 2014 [cited 2016-05-16]. Available at: <https://tools.ietf.org/html/rfc7258>.

[21] Finkenzeller, K. *RFID Handbook.* Third edition. Wiley, 2010. ISBN 978-0-470-69506-7.

[22] Franks, J.; Hallam-Baker, P. M.; Hostetler, J. L.; et al. *RFC 2617 : HTTP Authentication : Basic and Digest Access Authentication* [online]. Internet Engineering Task Force, June 1999 [cited 2016-05-16]. Available at: <https://tools.ietf.org/html/rfc2617>.

[23] Goldberg, I.; the OTR Development Team. *Off-the-Record Messaging Protocol version 3* [online]. 2016-03-09 [cited 2016-05-19]. Available at: <https://otr.cypherpunks.ca/Protocol-v3-4.1.1.html>.

[24] Haselsteiner, E.; Breitfuß, K. Security in near field communication (NFC). *Workshop on RFID security [RFIDSec 06].* 2006. Available online at: <http://events.iaik.tugraz.at/RFIDSec06/Program/papers/002%20-%20Security%20in%20NFC.pdf>.

[25] ISO. *ISO 7816 Part 4 : Interindustry Commands for Interchange.* 2013-04-15. Available online at: <http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816-4.aspx>.

[26] Josefsson, S. *RFC 3548 : The Base16, Base32, and Base64 Data Encodings* [online]. Internet Engineering Task Force, July 2003 [cited 2016-05-18]. Available at: <https://tools.ietf.org/html/rfc3548>.

[27] Karmazín, J. *Synchronization and Backup of Data under Android OS*. Brno, 2013. 59 pages. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Očenášek Pavel.

[28] Karmazín, J.; Očenášek, P. *The state of Near-Field Communication (NFC) on the Android platform*. In: Human Aspects of Information Security, Privacy, and Trust. Lecture Notes in Computer Science #9750. Springer Verlag, 2016, pages 1–7. ISBN 978-3-319-39380-3 (accepter paper).

[29] Krawczyk, H.; Eronen, P. *RFC 5869 : HMAC-based Extract-and-Expand Key Derivation Function (HKDF)* [online]. Internet Engineering Task Force, May 2010 [cited 2016-05-18]. Available at: <https://tools.ietf.org/html/rfc5869>.

[30] Marlinspike, M. *Advanced cryptographic ratcheting* [online]. Open Whisper Systems » Blog, 2013-11-26 [cited 2016-05-16]. Available at: <https://whispersystems.org/blog/advanced-ratcheting/>.

[31] Marlinspike, M. *WhisperSystems/curve25519-java : Pure Java and JNI backed Curve25519 implementation* [online]. GitHub, 2015-05-05 [cited 2016-05-16]. Available at: <https://github.com/WhisperSystems/curve25519-java>.

[32] Marlinspike, M. *Just Signal* [online]. Open Whisper Systems » Blog, 2015-11-02 [cited 2016-05-16]. Available at: <https://whispersystems.org/blog/just-signal/>.

[33] Marlinspike, M. *Signal on the outside, Signal on the inside* [online]. Open Whisper Systems » Blog, 2016-03-30 [cited 2016-05-16]. Available at: <https://whispersystems.org/blog/signal-inside-and-out/>.

[34] Marlinspike, M.; mx4; Kai, L.; et al. *WhisperSystems/TextSecure-Server Wiki : API Protocol* [online]. GitHub, 2014-10-06 [cited 2016-05-16]. Available at: <https://github.com/WhisperSystems/TextSecure-Server/wiki/API-Protocol>.

[35] Marlinspike, M.; R, J.; Tarek. *WhisperSystems/libsignal-protocol-java : Signal protocol library for Java/Android* [online]. GitHub, 2016-03-30 [cited 2016-05-16]. Available at: <https://github.com/WhisperSystems/libsignal-protocol-java>.

[36] Miers, I.; Green, M.; Rescorla, E. *Short Authentication Strings for TLS* [online]. Internet Engineering Task Force, 2014-02-14 [cited 2016-05-16]. Available at: <https://tools.ietf.org/html/draft-miers-tls-sas-00>.

[37] Nottingham, M.; Fielding, R. *RFC 6585 : Additional HTTP Status Codes* [online]. Internet Engineering Task Force, April 2012 [cited 2016-05-16]. Available at: <https://tools.ietf.org/html/rfc6585>.

[38] Oertel, B.; Wölk, M.; et al. *Security Aspects and Prospective Applications of RFID Systems*. Bonn: Bundesamt für Sicherheit in der Informationstechnik, 2005. Available online at: <http://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/RFID/RIKCHA_englisch_pdf.pdf?__blob=publicationFile>.

[39] Saint-Andre, P. *RFC 7622 : Extensible Messaging and Presence Protocol (XMPP): Address Format* [online]. Internet Engineering Task Force, September 2015 [cited 2016-05-16]. Available at: <https://tools.ietf.org/html/rfc7622>.

[40] Steiner, H.-C.; cobratbq; Wienke, J.; et al. *otr4j/otr4j : Off-The-Record messaging encryption written in pure Java* [online]. GitHub, 2016-02-01 [cited 2016-05-19]. Available at: <https://github.com/otr4j/otr4j>.

[41] the grugq. *Underground Tradecraft : Operational Telegram* [online]. 2015-11-18 [cited 2016-05-16]. Available at: <http://grugq.tumblr.com/post/133453305233/operational-telegram>.

Appendices

List of appendices

Appendix A

Used devices

Two smart phones running Android were used for experimenting, prototyping, development of the reference implementation, and testing. Table A.1 contains the specifications of these devices.

	Device A	Device B
Model number	LG-H340n	LG-D280n
Android version	5.0.1	4.4.2
Android security patch level	2015-12-01	N/A
Kernel version	3.10.49	3.4.0+
Build number	LRX21Y	KOT49I.A1404543497
Software version	V10e-230-01	V10c-230-01

Table A.1: Device specifications.

Appendix B

Key exchange protocol

When two client devices establish an NFC connection, the initiator may start a key exchange. Both the initiator and the target, if compatible and willing to participate in the exchange, must then communicate according to the protocol described in this appendix.

B.1 Architecture

The key exchange protocol is designed to be run by two devices over NFC, where the initiator uses reader mode and the target uses host-based card emulation mode. The reasons for this choice were explained in section 5.7.2.

As a result, the protocol is built on top of the protocol stack explained in section 3.2.3. Specifically, it consists of command and response APDUs compliant to ISO 7816-4 [25].

However, this protocol is not limited to the configuration mentioned above. As long as the two client devices are able to exchange ISO 7816-4 APDUs, they can participate in this protocol. For example, if a future update to Android API allows exchanging APDUs over a peer-to-peer NFC connection, it should be possible to adapt this protocol to such a situation.

B.2 AID

The AID used for the protocol defined in this appendix consists of the following 8 bytes, in hexadecimal notation:

```
F5 58 4D A1 B3 02 B1 78
```

The process of generating this AID was explained in section 5.7.3.

B.3 Protocol flow

The transmission of command APDUs from the initiator to the target, as well as the transmission of response APDUs from the target to the initiator, must follow the exact sequence shown in fig. B.1.

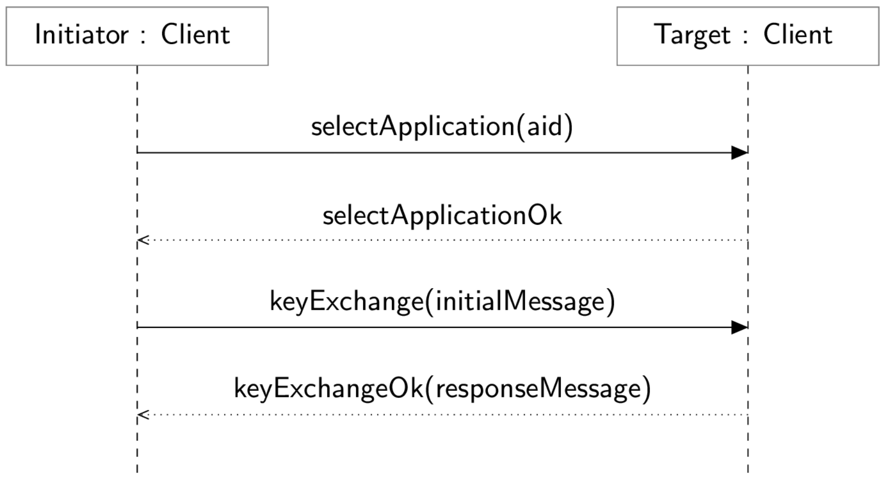

Figure B.1: Sequence of messages in the key exchange protocol.

The contents and encoding of each message are defined in appendix section B.4.

Once the protocol is started, any deviation from it is considered an error and should result in the protocol being terminated.

If the sequence is completed without deviation or error, the protocol can be considered to be completed. After the clients verify the integrity of the protocol, they can use the exchanged data to build a secure communication session.

B.4 Message definitions

Each message from the initiator to the target is an ISO 7816-4 command APDU, and each message from the target to the initiator is an ISO 7816-4 response APDU.

The class byte (CLA) in each command APDU is always set to the hexadecimal value `00`, which means, according to section 5.4.1 of ISO 7816-4:

- that the command and response are structured and encoded according to ISO 7816-4,
- that no secure messaging[1] is used, and
- that the logical channel used is always #0, as Android does not support more logical channels.

[1]This refers to authenticated communication between the reader and the card. This is a different concept from the security of user-to-user instant messaging, which is the topic of this thesis.

The exact contents and encoding of each protocol message are defined in the following text. Other commands can be implemented as needed, but should only return static or random data to preserve user privacy.

selectApplication(aid)

This is a SELECT FILE command APDU as defined in section 6.11 of ISO 7816-4. Android recognizes this command and uses it to route the first APDU to the correct application.

CLA	*INS*	*P1*	*P2*	*Lc*	*Data*
`00`	`A4`	`04`	`00`	*Lc*	*aid*

The instruction parameters *P1* and *P2* determine that the file is selected using a *DF name*, which is mapped to AIDs by Android.

aid is the AID.

Lc is the length of the AID in bytes.

The *Le* field is absent, meaning that no data are expected in the response body.

selectApplicationOk

This is a response APDU with the status words indicating “normal processing, no further qualification” according to section 5.4.5 of ISO 7816-4.

SW1	*SW2*
`90`	`00`

No data is contained in the response body.

keyExchange(initialMessage)

This is an ENVELOPE command APDU as defined in section 7.2 of ISO 7816-4.

CLA	*INS*	*P1*	*P2*	*Lc*	*Data*	*Le*
`00`	`C2`	`00`	`00`	*Lc*	*initialMessage*	`00`

initialMessage is the serialized form of a KeyExchangeMessage (see [35]) generated by the initiator. Its length is between 174 and 176 bytes, as was calculated in section 4.3.3.

Lc is the length of *initialMessage* in bytes, always between 174 and 176 decimal (between `AE` and `B0` hexadecimal).

Le is present and set to zero, meaning that a data payload is expected in the response body and can be up to 256 bytes long.

keyExchangeOk(responseMessage)

This is a response APDU with the status words indicating “normal processing, no further qualification” according to section 5.4.5 of ISO 7816-4.

Data	*SW1*	*SW2*
responseMessage	90	00

responseMessage is the serialized form of a KeyExchangeMessage (see [35]) generated by the target as a response to the KeyExchangeMessage received from the initiator. Its length is between 174 and 176 bytes, as was calculated in section 4.3.3.

Appendix C

Message relay protocol

In order for clients to send and receive messages, they must use a server for message relay.

Similarly to the *Signal* Server API Protocol described in [34], the protocol for communication between the clients and the server is described using an API, designed to be used with HTTPS for transport and protection of data in transit, and with JSON for data representation.

Each section of this appendix corresponds to one type of HTTP request that the client can issue. The specification for each request contains:

- An HTTP snippet containing:
  - The HTTP method to use.
  - The request URL.
  - All required request headers.
  - If required, the format of the request body, separated by a blank line.
- Specific HTTP response codes that can be used by the server.
- If response “200 OK” is legal, the format of the response body.

White space in the request and response bodies (outside of JSON string literals) is meaningless. It may be added arbitrarily by either party and must be ignored by the other.

Unrecognized HTTP headers, as well as items with unrecognized keys within JSON objects, should be ignored by the recipient for forward compatibility.

Angular brackets denote placeholders. The `⟨basicAuth⟩` placeholder stands for an HTTP basic authentication cookie as described in [16, section 11.1]. Other placeholders stand for values whose meaning is described below each snippet.

Where a JSON array is used, one item is given as an example. In practice, the length of the array can be any number greater than or equal to zero. The array items, if present, shall all have the same structure.

The following codes can be used by the server in response to any request, with the following meanings:

400 Bad Request
The request was not formatted according to the API specification.

401 Unauthorized
The client supplied incorrect credentials, or did not supply credentials for a request that requires them.

429 Too Many Requests [37]
Rate limit was exceeded. The client must wait for a short period of time and should then try the same request again.

500 Internal Server Error
There are problems with the server. The client may try the same request again later.

503 Service Unavailable
There are problems with the server. The client may try the same request again later.

C.1 Registering a device

The client registers with the server to create authentication credentials to use in all future requests.

This request must be used and a successful response must be received before any other API functions are used. Interaction with the user is not needed.

Request

```
POST /nfc-paired/v1/device/register

{
    "password": "⟨password⟩"
}
```

`⟨password⟩` is a 16-byte ASCII string randomly generated by the client. The same password must be used to authenticate all further requests by the same client. The allowed characters are octets from 33 (`!`) to 126 (`~`), inclusive, in US-ASCII. This is a subset of the characters allowed by [22] and [16].

Response

200 OK
The device was successfully registered.

```
{
    "userId": "⟨userId⟩"
}
```

⟨userId⟩ is a new identifier assigned by the server to the client's device. The same identifier must be used to authenticate all further requests by the same client.

C.2 Unregistering a device

The client can use this request to delete their registration with the server. This will permanently disable the credentials used with the request and block all sessions the device may have registered.

After deleting a device's registration, the client may not use any other API functions until it registers the device again.

Request

```
POST /nfc-paired/v1/device/unregister
Authorization: ⟨basicAuth⟩
```

Response

204 No Content
The registration was successfully deleted.

C.3 Registering a session

The client requests the server to receive messages sent with the given session ID, except those sent by the client themself.

This request can be used by a client who has successfully completed the key exchange over NFC with another client.

Request

```
PUT /nfc-paired/v1/session/⟨sessionId⟩
Authorization: ⟨basicAuth⟩
```

⟨sessionId⟩ is an arbitrary string generated by two clients who have completed the key exchange over NFC. This string is assumed to be known only by those two clients.

Response

204 No Content
The session has been successfully registered or it was already registered for the current device.

403 Forbidden
: The session is blocked. The clients should repeat the key exchange to generate another session ID.

C.4 Blocking a session

The client requests the server to stop dispatching any messages sent with the given session ID at any point in the future.

This request can be used by any client with the knowledge of a session ID. This may be used to block communication with the other device.

Request

```
DELETE /nfc-paired/v1/session/⟨sessionId⟩
Authorization: ⟨basicAuth⟩
```

`⟨sessionId⟩` can be any session ID that the client knows and wishes to block.

Response

204 No Content
: The session has been successfully blocked or it was already blocked before the request was submitted.

C.5 Sending a message

The client requests the server to dispatch the given message to another client who has registered the same session ID.

This request can be used for each message that a client wishes to send.

Request

```
POST /nfc-paired/v1/session/⟨sessionId⟩/message
Authorization: ⟨basicAuth⟩

{
    "body": "⟨messageBody⟩"
}
```

`⟨messageBody⟩` is an arbitrary message payload encoded using Base64 without spaces, padding, or line terminators. The format of the payload can be anything agreed to by the two clients, possibly a message encrypted using Signal Protocol.

Response

204 No Content
The message was successfully received for dispatch.

403 Forbidden
The given session ID is blocked.

404 Not Found
The given session ID is not registered by this device.

C.6 Polling for messages

The client requests messages dispatched from the server through any of the session IDs previously registered by this client.

Messages are numbered sequentially for each recipient as they are dispatched. In complete sessions (which both parties have already registered), messages are dispatched immediately. In partial sessions (which only the sending party has registered so far), messages are dispatched the moment the other party registers the session.

Request

```
GET /nfc-paired/v1/messages?after=⟨lastNumber⟩
Authorization: ⟨basicAuth⟩
```

`⟨lastNumber⟩` is the highest number assigned by the server to any message received from a previous poll by the same client. If this is this client's first poll, this parameter shall be zero (0).

Response

200 OK
New messages (if any) are contained in the response body.

```
[
    {
        "number": ⟨messageNumber⟩,
        "sessionId": "⟨sessionId⟩",
        "body": "⟨messageBody⟩"
    }
]
```

`⟨messageNumber⟩` is the number assigned by the server to this message for this particular recipient.

`⟨sessionID⟩` identifies the session through which the message was dispatched. This allows the client to identify the sender of the message and select the correct session to decrypt the payload.

`⟨messageBody⟩` is the Base64-encoded message payload created by the sender.

Appendix D

User manual

D.1 Overview

Touch and Chat is an IM application for devices with NFC support and the Android operating system. Contacts can be added by tapping devices together. Messages can then be exchanged over an Internet connection.

D.2 Requirements

In order to run *Touch and Chat*, your device must have NFC support and must be running Android version 4.4 or later. Your device must also be able to connect to the Internet, although a permanent connection is not required.

D.3 Adding a contact

Start *Touch and Chat*, then touch the floating "plus" button on its main screen to enter the "Add contact" screen.

If NFC is not enabled on your device, the application will instruct you to enable it through your device's settings. You must enable NFC at this point to proceed.

Tap the device you wish to add as a new contact. The device must have *Touch and Chat* installed, NFC enabled, and its screen turned on. Do not launch the "Add contact" screen on the other device or the process will not start.

Check the screen of your devices. You should see a screen titled "Verify contact" on both of them. If there was an error, you can try repeating the whole process.

On the "Verify contact" screen, you will see a verification code. Check that it is the same on both devices.

If the verification codes differ, it may be because of other devices in range, errors in transmission, or disruptions by a malicious party. Touch the "No" button to cancel the operation. You can then try adding the new contact again.

If the verification codes match perfectly, touch the “Yes” button. A new contact will then be added to both devices.

D.4 Sending messages

In order to send a message, you must be connected to the Internet and have at least one contact.

Start *Touch and Chat*, then select a contact from the list.

You will see a screen titled “Conversation” with an input box at the bottom containing the text “Type your message. . . ” You can type your message in this box, then send it by touching the button next to the box.

Once sent, your messages will appear at the bottom of the screen.

D.5 Receiving messages

In order to receive messages, you must be connected to the Internet and have at least one contact.

Start *Touch and Chat*, then select a contact from the list.

On the screen titled “Conversation”, touch the icon of two arrows forming a circle (its text is “Sync with the server”). If there are any new messages for you from this contact, they will appear at the bottom of the screen.

D.6 Troubleshooting

If you see the message “Sync failed”, there may still be unsent and unreceived messages. To make sure they are all sent and received, connect to the Internet, then use the “Sync with the server” button as described in the previous section.

D.7 Disclaimers

This application requires availability of the *Touch and Chat* server. For presentation purposes, this server was set up at <https://touch-and-chat.eu>, but it will likely not be maintained in the future, and it may be shut down.

This application is a proof of concept. You can use its core functionality, but keep in mind that not many extra features are implemented.

Appendix E

Server installation guide

This is a guide for installing the reference implementation of the message relay server for *Touch and Chat*.

E.1 Requirements

The server application is based on Symfony REST Edition. As a result, its recommended method of installation is via Composer, a dependency manager for PHP. You can download Composer from its home page at <https://getcomposer.org/>.

To run the server, you need a web server, support for the PHP language version 7, and a MySQL or SQLite database. The recommended web server to use is Apache.

E.2 Initial setup

Extract the source files to a directory accessible by your web server. Point the server's document root to the www directory inside the project root.

The rest of this guide assumes that all commands are executed with the working directory set to the project root.

Use the following command to install all dependencies and interactively enter configuration parameters, mainly for accessing the database:

```
php composer.phar install
```

If you need to change the parameters later, they can be found in the file app/config/parameters.yml.

Ensure that your installation of PHP is correctly configured for the command line interface (CLI) by running the command:

```
php app/check.php
```

Also ensure that your installation of PHP is correctly configured on the server by accessing the URL `/config.php` from a web browser.

Use the following command to create the database tables required by the server:

```
php app/console doctrine:schema:create
```

E.3 File permissions

Make sure the following access is permitted for the web server:

- www – read all contents.

Make sure the following access is permitted for the process running PHP, both from the CLI and from the web server:

- app – read all contents.
- app/cache – read and write contents.
- app/data – read and write contents.
- app/logs – read and write contents.
- src – read all contents.
- www – read PHP files.

E.4 Deployment to production

If you want to prepare the server in a staging environment and deploy it into production environment later, you can use these steps. You only need file access and database access on the production server. Shell access on the production server is not required.

- Perform the installation and setup on the staging server.
- Use the following command to generate the SQL statements for creating the database tables:

  ```
  php app/console doctrine:schema:create --dump-sql
  ```

- Run the resulting SQL statements on the production server.
- Transfer the following directories to the production server:
  - app
  - src
  - vendor
  - web

  Include all contents except the following files and directories:

  - app/cache
  - app/config/parameters.yml.dist
  - app/console
  - app/data (unless you use an SQLite database and intend to transfer it as a file)
  - app/logs
  - app/phpunit.xml.dist

- Make sure the following directories exist on the production server and have the correct permissions set, even if they are empty:
  - app/cache
  - app/data
  - app/logs

E.5 Use with the *Touch and Chat* client application

In the reference implementation, the *Touch and Chat* client application is configured to communicate with the message relay server located at <https://touch-and-chat.eu>. If you want to deploy your own server on a different domain, you will need to edit the source code of the client application so that it uses your server instead.

Your server instance must use a valid TLS certificate for the domain it runs on, signed by a root CA trusted by Android. Always use HTTPS for the client-to-server communication. Using plain HTTP is not secure and may allow various types of attacks, such as session hijacking.