TU WIEN Informatics

Security and Privacy of Secure Messaging Services

A Case Study of Wire

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Andreas Boll, BSc
Matrikelnummer 0825205

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Mitwirkung: Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc

Wien, 3. März 2020

Andreas Boll	Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Security and Privacy of Secure Messaging Services

A Case Study of Wire

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Andreas Boll, BSc
Registration Number 0825205

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Assistance: Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc

Vienna, 3rd March, 2020

Andreas Boll	Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Andreas Boll, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. März 2020

Andreas Boll

Kurzfassung

Ende-zu-Ende Verschlüsselung wurde zu einer Voraussetzung für sicheren Nachrichtenaustausch, der sich seit Einführung des *Double Ratcheting* Algorithmus zur Ende-zu-Ende Verschlüsselung durch Signal verbessert hat. Obwohl Metadaten für bestimmte Aufgaben, beispielsweise für die Zustellung der Nachrichten, erforderlich sind, sind diese oft nicht Ende-zu-Ende verschlüsselt. Ein weiteres Problem ist, dass zahlreiche mobile Kommunikationsdienste Telefonnummern zur eindeutigen Identifikation verwenden. Jedoch wird es immer schwieriger anonyme Prepaid-Karten zu erwerben. Zudem funktioniert die Kontaktsuche in den meisten Fällen durch Hochladen des Telefonbuchs, wodurch sensible Daten preisgegeben werden. Mit der Motivation einen sicheren Kommunikationsdienst zu finden, welcher keine dieser Probleme aufweist, befasst sich diese Arbeit mit der Evaluierung von Sicherheit und Datenschutz von sicheren Kommunikationsdiensten. Dazu wurde der Dienst Wire analysiert und mit anderen Lösungen, insbesondere Signal, verglichen. Die Hauptfragen, welche in dieser Arbeit beantwortet wurden, sind (1) wie kann die Sicherheit des Wire Protokolls evaluiert werden, (2) wie verhält sich Wire bezüglich Vertrauensaufbau, Unterhaltungssicherheit und Datenschutz des Nachrichtentransportes im Vergleich zu Signal und (3) wie viele Metadaten fallen bei Wire an? Für diese Arbeit wurde ein Testaufbau mit einem eigenem Wire Server, welcher ohne AWS Abhängigkeiten auskommt, aufgebaut. Mit diesem Aufbau wurden das Wire Protokoll, die REST-API, sowie die Datenbank insbesondere im Hinblick auf die anfallenden Metadaten ausführlich analysiert. Das Wire Protokoll wurde hinsichtlich Vertrauensaufbau, Unterhaltungssicherheit und Datenschutz des Nachrichtentransportes evaluiert. Für ein besseres Verständnis des Wire Protokolls wurde ein Plugin für Pidgin erstellt, welches die wichtigsten Funktionalitäten des Wire Protokolls unterstützt, um Nachrichten Ende-zu-Ende verschlüsselt auszutauschen. Auch die Produktionsumgebungen der offiziellen Server von Wire und Signal wurde mit Fokus auf TLS Sicherheit, HTTP Sicherheitsfunktionalitäten sowie der Sicherheit von Browsercookies analysiert. Zusammenfassend hat Wire ein gutes Sicherheitsniveau, jedoch gibt es Verbesserungsbedarf. Speziell beim Vertrauensaufbau und dessen Benutzungsfreundlichkeit sollte nachgebessert werden. Des Weiteren fallen bei Wire zu viele Metadaten an, welche, wo möglich, reduziert werden sollten.

Abstract

End-to-end encryption has become a requirement for secure messaging, which has improved a lot since Signal introduced the Double Ratcheting algorithm for end-to-end encryption. Although metadata is often needed by service providers to fulfill their tasks i.e. forward messages, it is usually not end-to-end encrypted. Another problem is that most mobile messaging apps depend on phone numbers as unique identifiers. However, it is increasingly difficult to acquire anonymous prepaid cards. Further, contact discovery often works via upload of the address book to the server, exposing sensitive data. Motivated to find a messaging service that does not have the above-mentioned drawbacks, this thesis shows how to evaluate the security and privacy of secure messaging services. For this, a case study of Wire was conducted and compared to other services i.e. Signal. The main questions answered in this thesis are (1) how can the security of the Wire protocol be evaluated, (2) how does Wire perform in trust establishment, conversation security and transport privacy compared to Signal and (3) how much metadata does Wire expose? To do this, a test setup with a self-hosted Wire server without AWS dependencies was built to inspect the Wire protocol, the REST API and the database, particularly for metadata. The Wire protocol was evaluated regarding trust establishment, conversation security and transport privacy. To help understanding the Wire protocol, a Pidgin plugin was developed which implements most features of Wire's protocol to support end-to-end encrypted messaging. Further, the production environments of Wire's and Signal's official servers were analyzed with a focus on TLS security, HTTP security headers and cookie security. To conclude, Wire has a good security level but has room for several improvements. Especially trust establishment and its usability should be advanced. Furthermore, Wire does expose a lot of metadata which should be reduced.

Contents

CHAPTER 1

Introduction

Nowadays, end-to-end encryption has become an indispensable requirement for online communication. However, this is still not universally supported and not all information is secured by end-to-end encryption. A certain kind of data (metadata) is often needed by the service providers to fulfill their tasks i.e. forward messages from a sender to one or more receivers. This metadata is usually not end-to-end encrypted but only transport encrypted between client and server. Another problem is that most mobile internet-based communication services use push notification services i.e. Firebase Cloud Messaging (FCM) or Apple Push Notification service (APNs). Thereby, the communication relies on a few big corporations which have access to metadata and potentially even communication content if the messages do not use some form of encryption. Besides the metadata problem, messaging services often depend on a unique phone number as identifier i.e. Signal. However, in increasingly more cases phone numbers cannot be acquired anonymously since more and more nations (i.e. recently also Austria) disallow the purchase of anonymous phone numbers i.e. anonymous prepaid cards. Other messaging services (i.e. Wire) offer email as alternative identifier solution, which can be acquired anonymously and changed at any time; in contrast to phone numbers, where a number change brings additional costs. Further, there is also a potential loss of sensitive private data i.e. the address book on mobile devices. Messenger services often continuously upload this private information to their own servers without asking for permission. Another problem for current messaging services is that trust establishment in their end-to-end encryption is, in most cases, only poorly implemented or not supported at all.

The aim of this work is to find answers to the following research questions in regard to messaging services:

- How can the security of the Wire protocol be evaluated?
- How does Wire perform in trust establishment, conversation security and transport privacy compared to Signal?

- How much metadata does Wire expose?

1.1 Structure of the Work

This thesis is structured into the following chapters: Chapter 2 shows the current state of the art of end-to-end encrypted messaging services. Chapter 3 describes the necessary background for this thesis. Chapter 4 describes the implementations which were conducted for this thesis. Chapter 5 explains the security analysis and its perquisites. Chapter 6 lists all results which have been found during the security analysis. The downsides of the results and possible improvements are discussed in chapter 7 and finally chapter 8 summarizes the conclusions of this thesis.

CHAPTER 2

State of the Art

This chapter describes the state of the art of end-to-end encrypted messaging services. It is split into the evolution of end-to-end encryption of messaging services 2.1 and the security analysis of end-to-end encryption of those messaging services 2.2.

2.1 The Evolution of End-to-End Encryption in Messaging Services

Electronic, internet-based communication such as text-based messages has been existing for several decades (i.e. email). However, security and privacy were not at the forefront of the original developers' mind.

Email. Email is a decentralized, asynchronous communication protocol and one of the oldest digital communication protocols in the internet. Even though there are standardized protocols to enable end-to-end encryption i.e. *OpenPGP* [7] and *Secure / Multipurpose Internet Mail Extensions (S/MIME)* [44], they are rarely used in practice and most emails are transmitted in plain text without any end-to-end encryption. Furthermore, OpenPGP and S/MIME do not support *forward secrecy*. A more commonly used encryption method is transport encryption which encrypts all, or at least some, of the routes of the communication (i.e. sender to server, server to server and server to receiver). In the recent past, transport encryption protocols such as *StartTLS* (opportunistic encryption) [40] [18] and *TLS* (strict encryption) [10] have been used extensively to support transport encryption for emails.

XMPP. Another commonly used decentralized communication protocol is the *Extensible Messaging and Presence Protocol (XMPP)* [1] which was originally designed as a

[1] `https://xmpp.org`, Accessed: 22-February-2020

synchronous communication protocol. XMPP was standardized, for the first time in 2004, by the Internet Engineering Task Force (IETF) [2] [49] [50] and was superseded by a newer version [51] [52] in 2011. As is stated in its name, XMPP is an extensible protocol which can be achieved by using the so called *XMPP Extension Protocol (XEP)*. Continuous development of extensions for XMPP also included extensions for asynchronous communication. XMPP also supports transport encryption and end-to-end encryption. Although the XMPP standard [51] stipulates the support of the transport encryption protocol StartTLS for opportunistic transport encryption, the usage of transport encryption is not obligatory. Consequently, a continuous transport encryption of messages cannot be guaranteed. Multiple solutions concerning end-to-end encryption have been developed over the years i.e. OpenPGP based encryption, Off-the-record (OTR) [4] encryption and, most recently, the *OMEMO Multi-End Message and Object Encryption* (OMEMO) protocol [3]. However, no consensus for obligatory implementation of one of these protocols has been reached.

Signal. Signal [4] (formerly known as TextSecure) offers messaging with end-to-end encryption for mobile and desktop devices. However, desktop devices are classified as secondary devices and cannot be used as a standalone, instead they initially have to be paired with a mobile device. Other than email and XMPP, Signal enforces the use of end-to-end encryption. It uses its own messaging protocol *libsignal-protocol*, which implements the *Double Ratchet* algorithm [42] (formerly known as Axolotl Ratchet). The Double Ratchet algorithm was also developed by Signal and is based on OTR [31]. It combines the asymmetric OTR ratcheting (Diffie-Hellman ratcheting) with a symmetric key ratcheting as can be seen in figure 2.1. Hence, for each message a new key is derived. The Signal protocol uses the Double Ratchet algorithm in combination with so called *prekeys* and thereby enables end-to-end encryption for asynchronous communication with security properties such as forward secrecy and deniability [32].

In addition to end-to-end encryption for messaging between two parties based on the Double Ratchet algorithm, the Signal protocol can also encrypt group conversations with more than two parties. To accomplish this, the Signal protocol falls back on ideas from *mpOTR* [14] and uses the already established two-party communication channel for pairwise key exchanges of the group encryption key.

Unlike email or XMPP communication, Signal does not support federation. Federation means that the communication network consists of several independent interconnected smaller networks and which creates a decentralized communication system. This allows, among other things, the integration of self-hosted servers into the communication network which in turn has the potential to increase the trust in the server and subsequently the metadata accessible by the server. However, the developers of Signal still have

[2] `https://ietf.org`, Accessed: 22-February-2020
[3] `https://conversations.im/omemo/`, Accessed: 22-February-2020
[4] `https://signal.org/`, Accessed: 22-February-2020

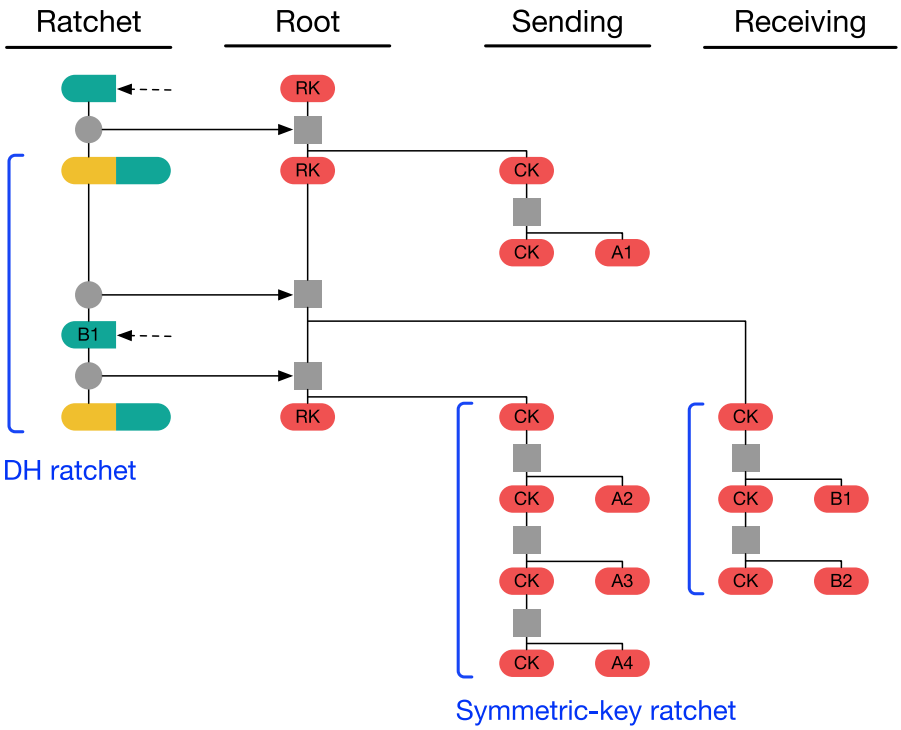

Figure 2.1: Double Ratchet algorithm [42]

no intention of supporting federation [36]. Another drawback of Signal is that phone numbers are still required for identification although the service itself is web based.

Signal's solution for end-to-end encryption of 1:1 conversations as well as group conversations has been adopted gradually by other messaging services i.e. WhatsApp [37], Facebook Messenger [34], Google Allo [35] and most recently by Skype [38]. However, Facebook Messenger, Google Allo and Skype do not enable end-to-end encryption by default.

XMPP & OMEMO. To offer end-to-end encryption based on Signal's Double Rachet algorithm for XMPP as well, the above-mentioned *OMEMO Multi-End Message and Object Encryption* protocol has been developed and standardized in XEP-0384 [55]. One of the advantages of OMEMO compared to OTR is that the end-to-end encryption also works for asynchronous environments (i.e. mobile communication) and still supports the security properties forward secrecy and deniability. Furthermore, OMEMO supports multiple devices per account and is better integrated into XMPP, which proves to be an advantage, especially for key management. To manage the keys, in particular the numerous prekeys, OMEMO uses the already existing XMPP extension *Personal Eventing Protocol (PEP)* [53] otherwise an additional key server would be required. Presently,

OMEMO has been implemented for many XMPP clients i.e. Conversations [5] (Android), Gajim [6] [7] (platform independent Python client), Pidgin [8] [9] (also platform independent, written in C) and many more [10].

Wire. The messaging service Wire [11] also supports end-to-end encryption by using *Proteus*, Wire's own implementation of the Double Ratchet algorithm. Proteus can encrypt 1:1 as well as group conversations. Like in Signal, end-to-end encryption is strictly enforced in Wire and thus not optional. Communication between client and server mostly happens with the HTTPS protocol by using a REST interface on the server. Furthermore, Wire uses *HTTP Strict Transport Security (HSTS)* [17] to enforce encrypted HTTP communication [66]. Besides communication via HTTPS, Wire clients also use the WebSocket protocol [12] to receive real time push notifications. Furthermore, mobile Wire clients can additionally use external push notification providers i.e. *Firebase Cloud Messaging (FCM)*, formerly known as *Google Cloud Messaging (GCM)*, or *Apple Push Notification service (APNs)*. External push notification providers have the advantage of requiring less battery, especially if multiple apps on the mobile device use push notifications. However, mobile Wire clients can also be used without such external push notification services. In that case, the mobile client would only use WebSockets for push notifications like web app and desktop apps. Such usage is required for Custom Android ROMs i.e. LineageOS [12] or GrapheneOS [13] which have no pre-installed Google Apps and therefore cannot use FCM for push notifications.

Besides encrypting messages Wire also supports some other privacy features [14] such as a people-search feature to find other contacts, making the upload of the user's address book optional. Furthermore, other than for Signal, Wire does not require a phone number for identification because Wire also supports email as alternative identification.

In addition, Wire clients are open source and available for all major platforms i.e. Android, iOS, Windows, MacOS, Linux and also as a universal web app [57]. In spring 2017 Wire started the release of parts of the server source code as open source and announced to release the entire source code as open source [59] which was achieved by the end of 2017 [61]. Furthermore, there are plans to support self-hosted servers as well as to support federation [59].

[5] `https://conversations.im`, Accessed: 22-February-2020
[6] `https://gajim.org`, Accessed: 22-February-2020
[7] `https://dev.gajim.org/gajim/gajim-plugins/-/wikis/OmemoGajimPlugin`, Accessed: 22-February-2020
[8] `https://pidgin.im`, Accessed: 22-February-2020
[9] `https://github.com/gkdr/lurch`, Accessed: 22-February-2020
[10] `https://omemo.top`, Accessed: 22-February-2020
[11] `https://wire.com`, Accessed: 22-February-2020
[12] `https://lineageos.org`, Accessed: 22-February-2020
[13] `https://grapheneos.org/`, Accessed: 22-February-2020
[14] `https://wire.com/en/security/`, Accessed: 22-February-2020

Beyond that, external security audits take place on regular bases to search for vulnerabilities in Wire. The first audit mainly focused on the end-to-end encryption components Proteus and Cryptobox [62]. Further audits focused on the apps for Android, iOS as well as the web app [64]. Security audits of parts of the desktop clients for Windows, MacOS and Linux [15], as well as the server implementation [16] have not taken place yet.

2.2 Security Analysis of End-to-End Encryption in Messaging Services

This section lists related work from past years regarding evaluation of the security of end-to-end encryption in messaging services.

OTR. The Off-the-Record (OTR) protocol for secure online communication has been published for the first time in 2004 by Borisov et al. [4]. The authors stated that end-to-end encryption based on OpenPGP is not suitable for most social online communication. Instead a protocol like OTR should be used. As main arguments for the usage of OTR the security properties *perfect forward secrecy* and *repudiability*, also known as *deniability*, have been mentioned. OpenPGP and S/MIME miss those security properties. OTR achieves perfect forward secrecy by using very short-lived keys in combination with the Diffie-Hellman key exchange. Thus, an attacker, who has gained access to a decryption key, can not decrypt old messages. To achieve deniability for individual messages, OTR uses *message authentication codes (MACs)* for each message instead of digital signatures which in turn can render content of messages unverifiable for third parties.

Additionally, another advantage of OTR is the independence of its underlying messaging protocol. Hence, OTR can be used on all messaging protocols, especially those popular in 2004 which were widely used but did not support end-to-end encryption themselves i.e. AIM, ICQ, MSN and many others. Borisov et al. [4] have published the generic library *libotr* that implements the OTR protocol as reference implementation. As part of this reference implementation, a plugin for the platform independent multi-protocol messenger Pidgin, originally known as GAIM instant messenger in 2004, was published. This OTR plugin, in combination with *libotr*, enabled end-to-end encryption for all messaging protocols supported by Pidgin.

GOTR. Since OTR only supported direct communication between two members (1:1 conversations) Bian et al. [3] suggested *Group OTR (GOTR)* which is an extension of the original OTR protocol. This extension supports end-to-end encryption of group chats based on OTR. The authors suggest that the creator of the group chat is chosen to stand in as a virtual server and to perform pairwise key exchanges with each member of the group respectively. The key exchange itself is identical to key exchanges for OTR 1:1

[15] `https://github.com/wireapp/wire-desktop`, Accessed: 22-February-2020
[16] `https://github.com/wireapp/wire-server`, Accessed: 22-February-2020

conversations. In addition to key exchanges, the virtual server is also responsible for distributing and forwarding all messages to other members of the group.

With this design the authors avoided using complex key exchange protocols for group conversations. However, this approach has some major disadvantages: it causes additional network an computing overhead for the virtual server, the virtual server is a single point of failure in case of outage or an attack (i.e. a denial of service), it is impossible to determine whether the virtual server has modified the message during forwarding or not, and GOTR ignores the case of group members changing i.e. a member leaves or a new member enters the group.

Goldberg et al. [14] also point out those shortcomings of GOTR's design. The original intention on OTR was to map a real life, private conversation between two people into the digital world. GOTR is far removed from this approach as it makes use of a virtual server, which would translate to a game of "Chinese Whispers" or "Telephone" in real life, including possible modifications to messages (see above). Further, the authors state that it is not trivial to extend OTR to group conversations because the used cryptographic primitives had been chosen for exactly two conversation members. This especially has an impact on the use of message authentication codes (MACs) concerning the deniability of communication because they only work for deniable conversations of two members.

mpOTR. This is why Goldberg et al. [14] suggest a different approach for end-to-end encryption of group conversations: the so-called *multi-party off-the-record (mpOTR)*. Instead of MACs they return to using digital signatures to support deniability. In order to ensure both forward secrecy and deniability for group conversations, short-lived key pairs are used instead of long-lived key pairs such as are used by OpenPGP signatures (a pair of public and private key). These short-lived key pairs are generated by each member of the group at the beginning of each session. Subsequently all members of the group conversation exchange their public key via pairwise key exchange. In order to exchange short-lived signature keys, the key exchange creates a confidential, authentic and deniable communication channel between each participant respectively. After this initial exchange, the short-lived signature keys of all group members are used to derive a common key for the encryption of all following by using a group key agreement protocol. With this last step the end-to-end encrypted group conversation is created and the actual messaging can begin. This complex procedure has to be repeated whenever a member leaves the group conversation or a new member enters the conversation.

Thus, mpOTR supports secure end-to-end encryption with the required properties confidentiality, authenticity and deniability for an arbitrary number of group members. Compared with GOTR, mpOTR does not need a centralized component, rather it employs a complex key exchange protocol instead.

GOTR 2013. Liu et al. [28] have found some shortcomings in mpOTR compared to the original OTR concerning deniability, which is not as strong as it was in OTR. Under certain circumstances it was shown that not only group members, but also third parties,

could trace the origin of a particular message to a specific group member. Hence the authors suggested an improved OTR for groups (GOTR) based on mpOTR, not to be confused with the GOTR by Bian et al. [3].

Signal. The messaging service Signal, formerly known as TextSecure, has been analyzed by Frosch et al. [13] with regards to authenticity and confidentiality. Besides some minor shortcomings the authors found that the protocol is vulnerable to unknown key-share attacks. In cooperation with the developers of Signal, the authors contributed to the solution of this vulnerability which was fixed in a follow-up release.

A more in-depth security analysis of Signal's messaging protocol *libsignal-protocol* was formally performed by Cohn-Gordon et al. [8]. Among other analyses, they looked at the key agreement protocol and the key exchange protocol. During the course of their work, the authors have created a formal description of the abstract protocol of Signal's core based on Signal's implementation as well as a security model for ratcheting. This model could prove the security of several security properties of Signal (i.e. confidentiality, authenticity and forward secrecy of the messaging key, which is used for encrypting messages). Furthermore, the security analysis did not find any major shortcomings in the design of Signals messaging protocol.

Unger et al. [56] have analyzed and rated several solutions, such as TextSecure, for secure communication in terms of security properties. In order to accomplish this, they developed a framework for the evaluation of secure messaging services pertaining to security, usability and ease-of-adoption of solutions. Consequently, the authors have identified the following three major challenges: trust establishment, conversation security and transport privacy. Current solutions, which have strong security and privacy properties for trust establishment, usually perform poorly at the aspects usability and ease-of-adoption. On the other hand, as soon as trust is established, conversation security can be solved without interaction with the user, thus increasing usability and ease-of-adoption. Transport privacy however remains a major challenge as solutions focusing on this come with heavy losses in their performance.

Rottermanner et al. [47] have analyzed some of the most widely used mobile messaging services, such as TextSecure, focusing on privacy and data protection. The study evaluated required app permission and transmission of sensitive private as well as app functionality to protect data saved on the device in case of loss or theft of said device. Because most research in the area of end-to-end encrypted messaging recently focused on 1:1 conversations and less on group conversations, Rösler et al. [48] analyzed the group communication protocols of Signal, WhatsApp and Threema. Amongst other things, they showed that the security property *future secrecy* of group conversations is not as strong as in 1:1 conversations.

Wire. The privacy white paper [65] of Wire describes which information and metadata are saved on Wire's server. Some information is optional or could be disabled by the users i.e. the transmission of usage statistics or crash reports. The upload of the address

book, which can be used for automatic contact, discovery is also optional. Similar to Signal, Wire only transmits hashes of the phone numbers to the server. On the server the hashes are only held in ephemeral memory as long as necessary for processing and are not saved permanently.

The used cryptographic algorithms and the process of registration, login and exchange of messages are also described in Wire's security white paper [66]. Text messages are end-to-end encrypted with Proteus and all other message types, such as images, videos or other files, are encrypted symmetrically. To accomplish this, the symmetric encryption keys as well as the corresponding metadata of the files are encrypted by Proteus to ensure that forward secrecy can also be supported for non text-based messages.

During the first external security audit of Wire the companies Kudelski Security and X41 D-Sec GmbH have found some vulnerabilities albeit not major [21]. The security audit focused on the cryptographic components Proteus and Cryptobox which are essential for Wire's secure end-to-end encryption. The vulnerabilities found were fixed in a release published shortly after the audit [62].

The following external security audit, also executed by Kudelski Security and X41 D-Sec GmbH, analyzed the client implementations of Android [22], iOS [23] and web app [24]. Like for the first audit, all vulnerabilities found were fixed immediately [64].

S/MIME and OpenPGP. *Efail* [43] introduced several attacks on email's end-to-end encryption protocols S/MIME and OpenPGP. Some of those attacks succeeded by combining vulnerabilities in the encryption protocols and in email clients respectively. Other reasons for successful attacks were the lack of authenticated encryption in the encryption protocols S/MIME and OpenPGP and vulnerable email clients due to incorrect MIME parsing potentially causing data leakages i.e. HTML forms, HTML source attributes and external images. The incorrect MIME parsing enables merging of encrypted and unencrypted parts of emails. Therefore, unencrypted parts can be modified by the attacker and can be used to leak data.

In contrast to S/MIME, OpenPGP developed a feature to secure the integrity of end-to-end encrypted emails called modification detection code (MDC) which prevents the above-mentioned modification attacks and was standardized in 2007 [6]. However, this security feature can be circumvented in many email clients, because they ignore wrong or missing MDC check sums and therefore show the unauthenticated but encrypted email instead of a decryption failure message. Displaying such emails is enough to transmit the decrypted email to the attacker through the above-mentioned data leakages.

In summary, the described vulnerabilities result in an attacker successfully circumventing end-to-end encrypted emails, usually by employing man-in-the-middle (mitm) scenarios. The attacker achieves this by modifying encrypted emails which the victim decrypts and subsequently gets leaked back to the attacker by the email client using one of several data leakages.

CHAPTER 3

Background

This chapter describes the background of Wire including a feature overview 3.1, several selected features of Wire 3.2, Wire's architecture 3.3, Wire's backend 3.4, and Wire's clients 3.5. Further, the background of Wire's libraries 3.6, Coax 3.7 and Pidgin / libpurple 3.8 is described as well.

3.1 Wire

Wire offers its clients two distinct account types: Wire Personal and Wire Pro. Wire Personal caters to individual users, is free of charge and offers secure, privacy- focused messaging to anybody with an account. All messages are end-to-end encrypted and can be sent and received on apps designed for mobile, desktop and web. Wire Pro (liable to costs) is geared towards business clients looking for easy and secure communication across internal teams, as well as external clients and partners. The latter can be invited to secure guest rooms. It offers special features such as end-to-end encrypted video conference calls and full administrator controls. Both account types can communicate with each other, are GDPR compliant, offer secure audio conference calls, file sharing and one on one video calls, among other features which are described below.

3.1.1 Feature Overview

The following section focuses on Wire supported features, table 3.1 contains a short overview:

Conversation types. Wire supports three types of conversation: one on one (1:1), group, and team conversations. In 1:1 conversations two users can exchange messages between each other. By default, users have 1:1 conversations with each individual contact member. Wire also supports group conversations with up to 500 members. Initially,

Features	Wire
1:1 conversations	✓
Group conversations	up to 500 members
Text messages	✓
+ Markdown formatting	✓
Link previews	✓
Send images	up to 25 MB
+ Send GIFs from Giphy	✓
Send audio, video messages or other files (i.e. PDF)	up to 25 MB
Share YouTube, Vimeo, SoundCloud, Spotify content	✓
Share location	✓
Get attention via ping or mentions	✓
Likes	✓
Delivery and read receipts	only in 1:1 and team
Ephemeral / self destructing messages	✓
Edit or delete messages	✓
Quote messages	✓
Group conversation roles and management	✓
Audio and video calls	✓
Audio conferences	up to 10 members
Screen sharing	up to 10 members
Register with email or phone number	✓
Unique username, arbitrary profile name and picture	✓
Search for people	✓
Optional phone book (hashes only) upload	✓
Support for multiple devices	up to 8 devices
+ Self-messaging	✓
Push notifications via WebSocket	✓
+ via FCM, APNs on mobile devices	✓
Open source apps for all popular platforms	✓
+ Android, iOS	✓
+ Windows, MacOS, Linux	✓
+ Web app	✓
All conversations end-to-end encrypted	✓
+ uses Double Ratcheting algorithm	✓
+ asynchronous by using prekeys	✓
+ forward and backward (future) secrecy	✓
Verify device fingerprints	✓
Transport encryption	TLS 1.2
+ Apps use certificate pinning	✓(web app ✗)
Pro features	only for Wire Pro users
+ Team conversations	up to 500 members
+ Guest room for ephemeral users	✓
+ Video conferences	up to 4 members
+ Send files	up to 100 MB

Table 3.1: Feature overview

groups were limited to 128 members, however several optimization of the underlying code resulted in increasing the number of participants [1]. Team conversations are group conversations with certain more advanced features e.g. dedicated team management web app, delivery receipts or guest accounts. This conversation type is only available for Wire Pro users.

Conversation content. Wire supports various content which can be sent as messages within any type of conversation described above. The most essential content type are text messages, which can be formatted by using simple Markdown formatting. Users have the choice to format text as bold, italic, monospace, code snippets, bullet lists, numbered lists, block quoting, heading and sub-heading.

URL links inside text messages are automatically detected and a preview image of the website is attached to the message. For privacy reasons, these link previews are generated on the sender's device. This prevents a malicious sender from gaining access to all IP addresses of all receiver devices as each device would automatically open the URL to generate a preview. Even if the sender has no malicious intent, IP addresses of all conversation members could be leaked to a third-party. However, this solution does not rule out potential leakage of the IP address when the recipient opens the URL manually. While link previews can be disabled on iOS and all desktop apps, this is not an option available on Android and the web app.

In addition to text-based messages, images and animated images (i.e. GIF images) can be sent as well. Wire has integrated the third-party service Giphy into its apps for easier access to a variety of GIF images. Audio and video messages as well as other files smaller than 25 MB can also be sent via Wire. Conversation content support is also provided for YouTube, SoundCloud, Spotify and Vimeo content. Additional features include sharing of a device's location and ping messages to attract the receiver's attention. This can also be accomplished by mentioning a specific username.

Other conversation features. Various additional features regarding message exchange are available inside conversations (i.e. ephemeral messages, editing and deleting messages, quoting a sent message, reactions and delivery and read receipts). Ephemeral messages, also known as timed messages, are removed from the recipient's device once the previously set timer runs out. The timer can be set at 10 seconds, 5 minutes, 1 hour, 1 day, 1 week or 4 weeks. The countdown starts as soon as the message is displayed on the recipient's device. To correct misspelled/unintentional messages, Wire supports editing and deleting of messages. Past messages can be quoted to facilitate following older or certain aspects of multiple conversations. Reacting to messages by liking them is also possible. Confirmations such as "delivered" and "read" make it easier to keep track of conversations by verifying that the recipient received/read a specific message. However, delivery confirmations are only supported for 1:1 and team conversations but

[1] `https://medium.com/wire-news/wire-for-web-2019-07-47a13a06ea7c`, Accessed: 22-February-2020

not for group conversations. The same applies to read confirmations but those are opt-in for 1:1 and opt-out for team conversations. A recently added feature is role management including the introduction of the administrator role for group conversations. Formerly, this feature was only available for Wire Pro users. Now group conversations can only be managed by group administrators i.e. adding and removing users to and from the group and managing user roles of the group. In the past, every group user had the permission to add and remove users and thus group conversations were hard to manage.

Audio / video calling. Wire supports audio and video calling using WebRTC technologies. However, audio conference calls are limited to 10 participants and video conference calls are limited to 4. Self-evidently, no limitation of participants applies in 1:1 conversations. The latter can only be initiated by Wire Pro users. For users behind firewalls or NAT, Wire uses STUN [46] and TURN [30] servers to initiate the call. The codec OPUS is used for audio while VP8 codec is used for video. Wire provides the option of using OPUS with constant bit rate instead of variable bit rate if the user prefers it [58]. Another feature for audio and video calling is support for screen sharing. Users can share their screen during a call, however, this feature is currently limited to the desktop clients on Windows and MacOS and the web app on Firefox.

Wire account. Users can register for a Wire account with their email address or their phone number. To log in, a user can provide either the email address and associated password or use the registered phone number to receive a login code via SMS.

Profile features. Each Wire account can set certain profile attributes i.e. a unique username, an arbitrary profile name, an optional profile picture, and choose a profile color from a limited set of 6 colors. Each profile attribute can be changed via profile settings in all Wire apps. Additionally, there is a backup feature to export the history of all conversations.

Contact discovery. There are two ways to find other contacts in Wire. Either use the people-search function to search for usernames and profile names or use the optional address book upload. The second approach hashes all phone numbers and email addresses in the user's phone book and uploads it to the Wire backend to find matching contacts.

Multiple devices. Wire supports up to 8 devices per account where 7 devices can be permanent and one device slot is for temporary devices. The temporary device is intended for ephemeral sessions via the web app. Each new temporary session replaces the existing temporary device. It is also possible to share messages between devices registered on the same account. For further information, read the section on self-messaging 3.2.6.

Multi account. A single Wire client can support up to 3 accounts at the same time. The android app is currently limited to 2 accounts. iOS and the desktop apps can support 3 accounts. Only the web app is limited to one active account within the same browser.

Push notification. Wire supports two kinds of push notifications: the first is via WebSocket and the second via external push notification providers i.e. Firebase Cloud Messaging (FCM) and Apple Push Notification service (APNs). Push notifications via WebSocket are provided from Wire's own backend service. External push notifications are additionally encrypted to hide metadata from the external push notification providers between Wire's backend and the Wire clients.

End-to-end-encryption. All messages between Wire clients are end-to-end encrypted using Proteus. Proteus is an implementation of the Double Ratchet algorithm and supports asynchronous messaging using prekeys. Additionally, it supports perfect forward secrecy and backward secrecy (also known as future secrecy). Audio and video calls are end-to-end encrypted with SRTP [1] and DTLS [45] and authenticated via Proteus messages. Device verification can be performed by comparing public key-fingerprints in hexadecimal representation. If needed, a cryptographic session between two devices can be manually reset.

Transport encryption. All communication between clients and server are encrypted by using TLS 1.2 [10] as transport encryption. All clients, except the web app, use certificate pinning to strengthen the transport encryption against possible man-in-the-middle attacks.

Clients for multiple platforms. Wire has clients for all popular operating systems including Android, iOS, Windows, MacOS, Linux and a platform independent web app for web browsers.

Wire Pro. Wire provides a number of additional features for business customers for a fee. These business accounts are called Wire Pro while the regular, free-of-charge account is called Wire Personal. Wire Pro users can create team conversations that include an assigned team administrator. For easier management of large teams Wire Pro users have access to a dedicated team management web app. Team conversations can also use a feature called "guest rooms" for ephemeral users. Ephemeral users are special insofar as they do not need to create an account and thus do not need credentials i.e. email address with password or phone number. These ephemeral users, also known as guest users, get invited via link from a Wire pro user. With this link, which contains an authentication token, guest users open the Wire web app and automatically join the corresponding guest room where the link was created. Such guest users are only validated for 24 hours and will be deleted afterwards. Another feature for Wire Pro is video conference calls with up to 4 members. Further, the limited size for file sharing i.e. sharing videos, PDF, ZIP or other files is increased from 25 MB to 100 MB for Wire Pro users. All the Pro features described here are open source and are included in every Wire client, although they are only usable by Wire pro accounts.

3.2 Selected Features

The features most relevant for this thesis are described in the following section i.e. end-to-end encryption, transport encryption, registration and authentication handling.

3.2.1 End-to-End Encryption

End-to-end encryption of messages ensures confidentiality, authenticity and integrity between clients. Thus, the server cannot read or change the content of sent messages. More information about Wire's end-to-end encryption can be found in Wire's security whitepaper [66].

Proteus. For end-to-end encryption of messages, Wire uses the Double Ratchet algorithm. In order to use this algorithm, Wire has developed its own implementation Proteus which is written in Rust, see also section 3.6.1. The security properties of this algorithm are forward secrecy and backwards (future) secrecy.

Cryptographic primitives. Proteus uses the cryptographic primitives ChaCha20 for encryption, HMAC-SHA256 for authentication and integrity, and Curve25519 as the elliptic curve Diffie-Hellman key exchange. The implementation of these cryptographic primitives is provided by libsodium [2].

Prekeys. To support forward secrecy for asynchronous messaging Wire uses prekeys. For each new Proteus session, a client fetches a prekey from the server. The server then removes this used prekey from the pool of available prekeys. The last prekey serves as a fallback prekey in case the prekeys pool gets exhausted. However, the usage of this fallback prekey weakens the perfect forward secrecy property especially if it is used for more than one Proteus session or for a longer period. Thus, clients should keep the prekeys pool large enough to ensure strong forward secrecy.

Device verification. Wire clients allow comparing public key-fingerprints by using the hexadecimal representation. After initial verification of those device key-fingerprints a prominent notification for newly added devices will be generated to inform the user.

Asset encryption. To reduce the computational work and to save network bandwidth assets, larger binary files i.e. images, audio, video or other files with up to 25 MB, or 100 MB for Wire Pro users, are not directly encrypted with Proteus but are symmetrically encrypted by using the cipher AES-256-CBC. SHA-256 checksums of the ciphertext, together with the symmetric encryption key, are exchanged via end-to-end encrypted Proteus messages between clients. The (large) ciphertext itself is permanently stored on the server and can be downloaded by Wire clients as needed. As a result, the asset encryption inherits the important security property forward secrecy because the

[2] `https://github.com/jedisct1/libsodium`, Accessed: 22-February-2020

symmetric key was sent via the Proteus protocol. This optimization to reduce the overhead has been proposed in [33].

WebRTC. 1:1 audio / video calls are end-to-end encrypted with the Secure Real-time Transport Protocol (SRTP) [1]. Datagram Transport Layer Security DTLS [45] is used to initiate the SRTP session i.e. exchange the encryption algorithm, keys and parameters [39]. This DTLS negotiation is encrypted via Proteus messages to properly authenticate the SRTP session and thus mitigates possible man-in-the-middle attacks. On phone to phone calls SRTP and KASE (Key Agreement Signaling Extension) is used to accelerate the call initialization. KASE is a key agreement protocol developed by Wire [3] and makes use of libsodium's key exchange API [4]. Proteus also authenticates the key negotiation KASE to mitigate possible man-in-the-middle attacks as it does for DTLS. Conference calls are implemented as 1:1 calls between each member pair thus all WebRTC streams are individually encrypted with separate encryption keys.

3.2.2 Transport Encryption

Wire uses transport encryption to secure the communication between clients (apps) and servers for confidentiality, authenticity and integrity. This is required because not all data are end-to-end encrypted i.e. metadata the server needs for service operation. The metadata, together with the end-to-end encrypted messages, get encrypted with the standard transport encryption protocol TLS. The transport encryption between clients and server is specified in Wire's security whitepaper [66].

TLS 1.2. Wire currently requires TLS 1.2 [10] for transport encryption and allows only ciphers with perfect forward secrecy to be used.

Certificate pinning. Wire uses certificate pinning for transport security hardening on all apps except the web app. Certificate pinning helps mitigating man-in-the-middle attacks, where the attacker has gained access to a certificate issued by a trusted certification authority i.e. a malicious or compromised CA or a local root certificate installed by the attacker. Wire apps implement certificate pinning by pinning the public key of Wire's leaf certificates.

CAA. Wire makes use of Certification Authority Authorization (CAA) [15] to allow DigiCert as the only certificate authority that can issue certificates for the backend and web app domains. CAA is a DNS record and is used to verify that certificates of certain domains are signed by an explicitly allowed certificate authority.

[3] `https://github.com/wireapp/wire-audio-video-signaling/tree/master/src/kase`, Accessed: 22-February-2020

[4] `https://download.libsodium.org/doc/key_exchange/`, Accessed: 22-February-2020

HSTS. Wire uses HSTS [17] to mitigate TLS stripping attacks, which is particularly important for Wire's web app because of its absence of certificate pinning.

Push notifications. External push providers i.e. FCM and APNs are used to send notifications on mobile devices. However, notifications containing end-to-end encrypted messages and metadata are additionally encrypted to keep metadata hidden from the push providers. The push notifications are encrypted using the cipher AES-256-CBC for encryption and use HMAC-SHA256 as message authentication code (MAC) for authentication and integrity. The mobile apps generate both required keys for encryption and authentication on first login and upload them to Wire's backend via the REST API.

3.2.3 Registration and Authentication

This subsection describes how registration and authentication are handled in Wire. Further details can be found in Wire's security whitepaper [66].

User registration. Wire supports registration of an account either via email address or phone number. For verification of the provided email address a verification code is generated by the backend and is sent to that email address. In case of registration via phone number the backend also randomly generates a verification code which is sent to the phone number provided via SMS. At the time of registration, a unique internal user ID (UUIDv4 [26]) gets generated by the backend.

Passwords. In case of registration via email address, a password is required as login credential. Passwords are stored in encrypted form in the backend by using the password-based key derivation function scrypt [41] with default parameters $N = 2^{14}$, $r = 8$, $p = 1$ and a random 32 Byte salt. The random salt is taken from the pseudo-random number generator */dev/urandom*.

Login. Login is possible either by providing the registered email address and corresponding password or or by providing phone number and a login code, which is transmitted via SMS for each login. Wire uses OAuth 2.0 [16] with Bearer tokens [19] for authentication. There are two distinct types of tokens: first, there are long-lived user tokens which can be valid for either one year for permanent devices or one week for session-based devices i.e. the web app. Second, there are short-lived access tokens which are valid for 15 minutes. After successful login, a user token and an access token are generated by the backend. Access tokens are used to authenticate each request to the backend API. The long-lived user tokens are used to refresh those short-lived access tokens. If necessary, the user token is refreshed by the server when the access token is refreshed.

Client registration. It is possible to use one Wire account on up to 8 clients (devices) at the same time. Each new client registration generates a *user.client-add* notification as

described in table 3.2 which all other clients receive. Furthermore, an e-mail notification is sent if the user has registered an email address.

3.2.4 Notifications

Notifications are used to deliver events and messages of any kind between the server and clients. Each notification is of a specific type in order for clients to process them. A list of most notification types and definitions is summarized in table 3.2.

Delivery of notifications uses either push notifications via WebSocket, an external push provider (i.e. FCM or APNs) or by a client synchronization call via the notification queue API `GET /notifications`. The notification queue is used to fetch potentially missed messages in cases where the device was offline or the push notification via external push provider was lost. Notifications are held in this notification queue for a period of 4 weeks. If the client does not retrieve the notification within this time frame it will be deleted by the server.

Notification type	Description
`conversation.create`	a new conversation has been created
`conversation.member-join`	a new user joins a group chat
`conversation.member-leave`	a user leaves a group chat
`conversation.member-update`	a conversation member has been updated
`conversation.otr-message-add`	a new end-to-end encrypted message is available
`conversation.rename`	the conversation has been renamed
`conversation.typing`	a user is currently typing a new message
`user.activate`	the user account has been successfully activated
`user.client-add`	a new device has been added to the account
`user.client-remove`	a device has been removed from to the account
`user.connection`	a new (auto-)connection between two users has been created
`user.contact-join`	a new contact join request is available
`user.delete`	a user has been deleted
`user.identity-remove`	an identity (email or phone) has been removed from own account
`user.properties-clear`	all user properties have been cleared
`user.properties-delete`	a user property has been deleted
`user.properties-set`	a user property has been set
`user.update`	a user profile has been updated

Table 3.2: Notification types

3.2.5 People-Search and Address Book Upload

To connect with other people users can either use the people-search feature or upload their address books to the backend.

In order to use the people-search function, users ("searchers") type a person's chosen profile name or username into the app's search input field, which returns a list of all matching profile names and usernames. The searcher can then select the correct contact and send a connection request. Communication can commence, once the contact has accepted this connection request. However, any connection request can be ignored by the user being contacted. Once accepted, connections can be blocked at any time, resulting in the backend to drop 1:1 messages from blocked connections. Also, blocked connections can be unblocked again and delivery of 1:1 messages resumes. Blocked or unblocked users will not be notified that they were blocked or unblocked.

The above-mentioned search API `GET /search/contacts` used by Wire clients, uses the search engine Elasticsearch [5] to search for users by employing a specific search query. This query is applied to a search index which includes user data such as: user ID, profile name, normalized profile name (*normalized*) and the unique username (*handle*). Normalized profile names are the result of transliterating the profile name into lowercase Latin letters. The search input provided by the searcher is transliterated into a normalized input as well. Subsequently, Elasticsearch searches for this normalized input within the search index fields *handle* and *normalized* using a *phrase prefix matching* [6] query. The search query prioritizes the field *handle* over the field *normalized*. Additionally, the search query uses *phrase match* to find exact matches which are given preference over prefix matches of *handle* or *normalized*.

A simplified example for a search query, as sent by brig to the search engine Elasticsearch, can be found in listing 3.1. The search input "Alice" is normalized into "alice" by brig. The query consists of two parts: phrase prefix matching and phrase matching. *Multi_match* allows to search in multiple fields i.e. *handle* and *normalized*. The caret ^2 in the field *handle^2* indicates that this field is twice as important as the field *normalized*. The fields *handle^3* and *normalized^2* from the phrase matching query are therefore higher weighted than the fields *handle^2* and *normalized* from the phrase prefix matching query.

The results of the search query contain the user data from the search index and include a search score for each user found. The search score is used to rank the returned users by assigning higher scores to better matches.

In addition to the search API, Wire clients can also use the user handle API `GET /users/handles/` to find a full matching username. The user handle API queries the Cassandra database of brig to return the corresponding user ID for the matching

[5] `https://www.elastic.co/elasticsearch`, Accessed: 22-February-2020

[6] `https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-multi-match-query.html#type-phrase`, Accessed: 22-February-2020

username, if it exists. With this returned user ID, Wire clients can use the user API `GET /users` to retrieve more details i.e. username, profile name, profile picture, profile color.

Listing 3.1: Simplified Elasticsearch query for search input “Alice”

```
{
    "multi_match": {
        "fields": [
            "handle^2",
            "normalized"
        ],
        "operator": "or",
        "query": "alice",
        "type": "phrase_prefix",
    }
},
{
    "multi_match": {
        "fields": [
            "handle^3",
            "normalized^2"
        ],
        "operator": "or",
        "query": "alice",
        "type": "phrase",
    }
}
```

The second approach to connect with other people is to use the address book upload feature. It uses *SHA256* hashes of email addresses and phone numbers in the users' address book and uploads them to the Wire backend using the onboarding API `POST /onboarding/v3`. Neither the email addresses and phone numbers themselves, nor other data in the address book (i.e. names, addresses, birthdays or notices) are uploaded to the backend. These uploaded hashes are compared to the brig table *user_keys_hash* with its stored hashes of all Wire users. On matching hashes, the corresponding users get automatically connected, which means that no connection requests have to be accepted.

Uploaded hashes are not stored on the backend but only processed during the upload itself. This implies that if a contact from the address book was not using Wire at the time of uploading, but starts to use Wire after the upload, they will not be automatically connected. Therefore, the address book has to be uploaded again in order to connect both users automatically. Alternatively, the user could use the people-search feature to connect with this contact, although this would create a connection request which has to be accepted by the contact as already described above.

3.2.6 Self-Messaging

It is possible to create new group conversations without adding any members to the group, thus the creator of the group is the only member. Hence this feature could be

used to establish end-to-end encrypted self-messaging, which is especially useful if the user uses multiple devices with the same Wire account. This facilitates the sharing of sensible information, pictures or other data between multiple devices, belonging to the same user, without giving up end-to-end encryption, sharing information with another person by abusing a 1:1 conversation for self-messaging, and having to create multiple accounts i.e. one for each device.

3.3 Wire Architecture

The messaging service Wire is heavily based on a traditional client-server architecture, in combination with a peer-to-peer (p2p) architecture. For clients, apps are available for several platforms i.e. mobile apps for Android and iOS, desktop apps for Windows, MacOS and Linux, as well as a platform independent web app for web browsers. The server side consists of several service components which can be grouped into internal and external services respectively. The communication between clients and server is handled by the protocols HTTPS and WebSocket. Wire's mobile clients for Android and iOS use additional external push notification services, i.e. FCM on Android and APNs on iOS, for communication. In addition to the client-server architecture, Wire uses the peer-to-peer technology WebRTC on the client-side for peer-to-peer communication between clients.

Wire's backend has been designed with strong scalability and high performance in mind. Most services of the backend have been written in the functional programming language Haskell. The Haskell services behind the backend's REST API are all stateless and support running multiple instances of those services at the same time and therefore services scale very well. Additionally, Haskell services provide an internal REST API for communication between backend components inside an internal network. Authentication is handled by Oauth 2.0, which also enables stateless REST authentication for good scalability.

Although most of the services in Wire's backend are stateless, the backend is not entirely stateless as it also features some stateful parts i.e. the WebSockets. The Haskell service, which handles the WebSocket connection, is designed to allow running of multiple instances of this service in order to handle numerous stateful connections. State information about the WebSocket connections, as well as external push notification endpoints, are stored in Redis, a key-value database. Conversely, persistent information is stored in Cassandra, a highly scalable database. The search engine ElasticSearch is used to support a highly scalable people-search.

Wire's backend and clients will be described in more detail in the following sections 3.4 and 3.5.

3.4 Wire Backend

The main part of the Wire backend and server respectively, consists of the Haskell services brig, galley, gundeck, cannon, cargohold and proxy. The reverse proxy nginx is positioned in front of those Haskell services, while several data sources i.e. the databases Cassandra and Redis, as well as the search engine Elasticsearch, are in the back. In conjunction, these services form the internal services.

Beyond those internal services, the Wire backend depends on several external services for the full feature set of Wire. These external services consist of several *Amazon Web Services (AWS)* i.e. SES, SQS, SNS, DynamoDB, S3 and CloudFront. Furthermore, the backend requires the external services Twilio, Nexmo, YouTube, SoundCloud, Spotify, Google Maps and Giphy.

The Wire backend with its components split into internal and external services can be seen in figure 3.1.

The current design of the Wire backend implies that it is meant to be deployed via AWS because of its strict dependencies on many AWS services.

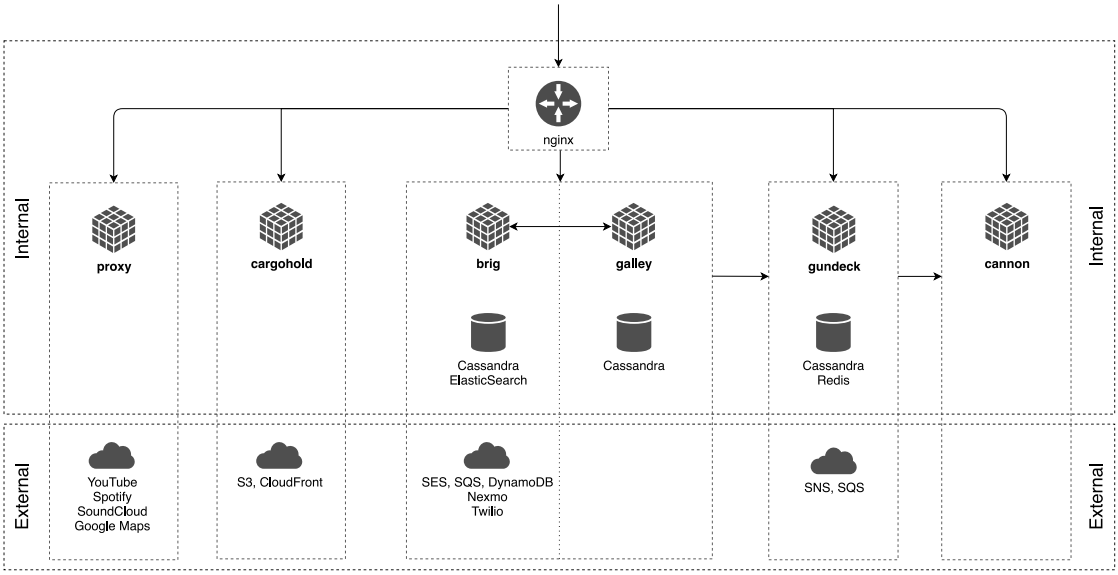

Figure 3.1: Architecture of the Wire backend [7]

The following sub section describes all components of the Wire backend.

3.4.1 Internal Services

The Wire backend consists of the following, open source, internal components (see also figure 3.1):

[7] `https://github.com/wireapp/wire-server`, Accessed: 22-February-2020

nginx. nginx [8] is used as reverse proxy for the public REST API of Wire's backend. It also handles the HTTP protocol upgrade to establish WebSocket connections. Furthermore, nginx is operated as TLS termination proxy which means it terminates the transport encryption TLS between clients and backend. The component uses three additional nginx modules: headers-more-nginx-module [9], nginx-module-vts [10] and nginx-zauth-module [11]. headers-more-nginx-module is used to set CORS and HSTS headers. The nginx virtual host traffic status module, nginx-module-vts, shows traffic and caches statistics and is used for monitoring nginx. The custom authentication module, nginx-zauth-module, authenticates all REST API calls by using Oauth 2.0 authentication with its access tokens.

brig. The Haskell service brig manages all accounts including their clients (devices) and connections. It also provides the API features to connect new users i.e. people-search and phone book upload. Brig creates the signed access token for Oauth 2.0 authentication. Persistent data are stored into brig's Cassandra database.

galley. Galley is the server component which handles all conversations (1:1 and group) and their members. It also provides the API to send messages. Galley generates notifications for incoming messages and forwards them to the push notification hub gundeck. Persistent information about conversations is stored into galley's Cassandra database.

gundeck. The push notification hub gundeck manages the distribution of all push notifications. Notifications can be either sent via WebSocket over the Haskell service cannon or via external push providers over SNS i.e. FCM and APNs. Notifications sent to external push providers are additionally encrypted to hide metadata. The keys for this encryption are stored in gundeck's Cassandra database. Notification endpoints, where notifications have to be sent, are stored in a Redis database. This enables gundeck to handle multiple instances of cannon.

cannon. Cannon is a WebSocket server implementation for sending push notifications to clients. It also provides an internal REST API where gundeck submits notifications to be pushed to the clients.

cargohold. Cargohold is used to provide an asset storage based on Amazon S3 and CloudFront. Wire clients store all encrypted assets on this service. An exception are profile pictures which are stored unencrypted.

[8] `https://nginx.org/`, Accessed: 22-February-2020

[9] `https://github.com/openresty/headers-more-nginx-module`, Accessed: 22-February-2020

[10] `https://github.com/vozlt/nginx-module-vts`, Accessed: 22-February-2020

[11] `https://github.com/wireapp/wire-server/tree/develop/services/nginz/third_party/nginx-zauth-module`, Accessed: 22-February-2020

proxy. The component proxy is used to integrate third-party services i.e. YouTube, SoundCloud, Spotify, Google Maps and Giphy for easier integration into all Wire clients. Those third-party services often require API keys, therefore, the backend service proxy manages those API keys instead of requiring them of all clients. This also mitigates metadata leakage (i.e. IP address), exposed by Wire clients, to those third-party services.

restund. Restund [12] is an open source STUN [46] and TURN [30] server used to establish peer-to-peer WebRTC connections if any of the peers is behind a firewall or a NAT.

Cassandra. Cassandra [13] is an open source, highly scalable database with no single point of failure. It is used in Wire's backend for storing data of the components brig, galley and gundeck. Each component has its own database schema and therefore could be run in completely different instances of Cassandra which improves the scalability further.

Redis. Redis [14] is an open source in-memory key-value database used by gundeck to store the destination URLs for push notifications of all connected devices i.e. which WebSocket connection from which cannon service as well as external push notification endpoints of FCM or APNs.

ElasticSearch. The Java based open source search engine Elasticsearch [15] is used by the people-search. It has its own search index which can be modified by using its RESTful API with JSON as content format. The search queries are also formatted as JSON and are sent to the REST API as well.

3.4.2 External Services

In additional to the internal services, Wire requires the following external services guarantee functionality for all features (see also figure 3.1):

Amazon Web Services (AWS). In order to run the internal services the AWS [16] services SES, SQS, SNS, DynamoDB, S3 and CloudFront are required.

SES. Simple Email Service (SES) [17] is used by brig to send emails to Wire users i.e. verification of email identities, notification about newly added devices, password resets, verification of account deletions and other important email notifications.

[12] `http://www.creytiv.com/restund.html`, Accessed: 22-February-2020
[13] `https://cassandra.apache.org/`, Accessed: 22-February-2020
[14] `https://redis.io`, Accessed: 22-February-2020
[15] `https://www.elastic.co/elasticsearch`, Accessed: 22-February-2020
[16] `https://aws.amazon.com`, Accessed: 22-February-2020
[17] `https://docs.aws.amazon.com/ses/index.html`, Accessed: 22-February-2020

SQS. Simple Queue Service (SQS) [18] is used as internal message queue to synchronize multiple instances of the services brig and gundeck. On brig, it is currently only used for user account deletion by the internal API `DELETE /i/users/:id`.

SNS. Simple Notification Service (SNS) [19] is used by gundeck to send push notifications to mobile clients on Android via FCM and on iOS via APNs.

DynamoDB. The NoSQL database DynamoDB [20] is used by brig for a best-effort optimistic locking for prekeys which is required in order to run multiple instances of brig. It is also used to store blacklisted users which are managed by brig's internal API `/i/users/blacklist`.

S3. Simple Storage Service (S3) [21] is used by cargohold to store all assets sent over Wire. It is also used for storing profile pictures.

CloudFront. The content delivery network CloudFront [22] is also used by cargohold to provide asset storage.

Nexmo. Nexmo [23] is used by brig to send SMS to verify phone identities, reset passwords and to login via phone.

Twilio. Twilio [24] is mainly used by brig to validate phone numbers. It is also used as a fallback service for sending SMS in case of a Nexmo failure.

YouTube. The third-party service YouTube [25] is integrated into Wire by the backend service proxy and is used by Wire clients to support YouTube content inside Wire conversations.

SoundCloud. SoundCloud [26] is a music streaming service which is used to share SoundCloud content inside Wire conversations.

Spotify. Spotify [27] is also a music streaming service and is used to share Spotify content inside Wire conversations.

[18] `https://docs.aws.amazon.com/sqs/index.html`, Accessed: 22-February-2020
[19] `https://docs.aws.amazon.com/sns/index.html`, Accessed: 22-February-2020
[20] `https://docs.aws.amazon.com/dynamodb/index.html`, Accessed: 22-February-2020
[21] `https://aws.amazon.com/s3/`, Accessed: 22-February-2020
[22] `https://docs.aws.amazon.com/cloudfront/index.html`, Accessed: 22-February-2020
[23] `https://www.nexmo.com`, Accessed: 22-February-2020
[24] `https://www.twilio.com`, Accessed: 22-February-2020
[25] `https://www.youtube.com`, Accessed: 22-February-2020
[26] `https://soundcloud.com`, Accessed: 22-February-2020
[27] `https://spotify.com`, Accessed: 22-February-2020

Google Maps. The Google Maps API [28] is used by Wire clients to share location data within a conversation.

Giphy. Giphy [29] is a web service for searching and sharing animated GIF pictures. It also indexes numerous GIFs from other sites. Wire clients integrate Giphy via the integration service proxy for providing the feature to search and share animated GIFs inside Wire conversations.

3.5 Wire Clients / Apps

Wire supports official open source clients for all major operating systems i.e. Android, iOS, Windows, MacOS, Linux and a universal web app for web browsers. In general, the features explained in section 3.1.1 are supported by all clients. However, there are some exceptions and certain features can differ between the clients. These differences are described in the following subsection:

3.5.1 Android App

The Android app is mostly written in Scala with some minor parts written in Java. For new features Wire recently started to use Kotlin. Further, the app depends on certain custom libraries because some particular features were able to be implemented as common shared code between the iOS and the Android app. One important and extensive library is the audio video signaling engine which is written in C++. Other shared libraries used by both apps are the end-to-end encryption libraries which are written in Rust, see also section 3.6. The Android app uses FCM for push notifications in addition to WebSocket notifications in order to improve power efficiency and thus increases the battery runtime. The app can be installed from the official Google Play store or by installing the APK file directly, which can be downloaded from Wire's website. To verify the integrity of the direct APK download the fingerprint of the app signing certificate as well as a corresponding SHA256 checksum are provided. This is especially useful for CustomROM (i.e. Lineage OS) users without GApps, who can use the Wire app without Google Play store and FCM. The Android App also supports certificate pinning which is implemented by pinning the public key of Wire's leaf certificate. Some of the recently added security features are in support of app locking, hiding the content of notifications on the lock screen, and support for hiding the screen content in the task switcher which also disables taking screen shots of the app. Another security feature that was recently added is support for encrypted backups to export the history of all conversations. Formerly, the Android app supported only unencrypted backups but now it optionally encrypts the backup with the authenticated encryption cipher XChaCha20Poly1305 and using Argon2 as password-based key derivation function [66]. A minor drawback of the Android app is that the generation of link previews cannot be disabled. Further, the multi-account

[28] `https://developers.google.com/maps/documentation/`, Accessed: 22-February-2020
[29] `https://giphy.com`, Accessed: 22-February-2020

feature of the Android app currently only allows up to 2 accounts at the same time. In contrast iOS and the desktop apps support up to 3 accounts at the same time.

3.5.2 iOS App

The iOS app was originally written in Objective-C but has been mostly rewritten in Swift. It uses the C++ audio video signaling engine, as does the Android app. In addition to WebSocket notifications, the iOS app uses APNs for push notifications to improve power efficiency and thus increases the battery runtime. Since there are no alternative options to install apps, it is only available on the official iOS app store. The app also supports encrypted backups by using XChaCha20Poly1305 for encryption and Argon2 for key derivation [66]. Another extra feature of the iOS app is app locking via Touch ID or Passcode [30]. Just like the Android app, the iOS app also does certificate pinning.

3.5.3 Web App

The web app is written in JavaScript and TypeScript and uses the React [31] JavaScript library and the NodeJS [32] ecosystem. As opposed to other clients, the web app has no support for certificate pinning. In the past, HTTP Public Key Pinning (HPKP) [11] was introduced in order to provide certificate pinning in web browsers, however its use is now discouraged and Google Chrome already removed its support for HPKP. As a replacement for HPKP, certificate transparency [25] is recommended, which is supported by Wire's certificate authority DigiCert. However, to take full advantage of certificate transparency, the HTTP header *Expect-CT* is currently missing, which would enforce certificate transparency i.e. the browser would require the certificate to contain signed certificate timestamps (SCTs) (see also the current IETF draft for Expect-CT [33]). Additionally, Wire's DNS records for the web app and backend use Certification Authority Authorization (CAA) [15] which allows only DigiCert as certificate authority to issue certificates for those domains. The web app also profits from enabled HSTS to mitigate man-in-the-middle attacks using TLS stripping.

The web app uses local storage to permanently store all cryptographic keys, device and user account data, and conversation histories. Cookies are used for session tokens. Instead of storing them client-side, assets are downloaded from server as needed. The decryption keys for assets are stored in the browser's local storage.

A minor drawback of the web app is that the generation of link previews cannot be disabled. Another drawback is that its support for multi-account is limited because a browser session can only be logged into one Wire account at the same time. As a workaround one could use different browser sessions to use more Wire accounts in parallel.

[30] https://support.wire.com/hc/en-us/articles/115001479849-How-can-I-lock-my-Wire-account-with-Touch-ID-or-Passcode-, Accessed: 22-February-2020

[31] https://reactjs.org/, Accessed: 22-February-2020

[32] https://nodejs.org/, Accessed: 22-February-2020

[33] https://tools.ietf.org/html/draft-ietf-httpbis-expect-ct-08, Accessed: 22-February-2020

An advantage of the web app is that it needs no installation and is always up to date. Additionally, it acts as a fallback for unsupported platforms i.e. Windows 10 Mobile, BSD derivatives, Haiku OS or other Unix systems as long as a supported browser is available on that platform. Currently supported browsers are Mozilla Firefox 60 or later, Google Chrome/Chromium 51 or later, Microsoft Edge 15 or later, and Opera 43 or later.

3.5.4 Desktop Apps

The desktop apps for Windows, MacOS and Linux are developed on top of the Electron [34] framework. They essentially wrap the web app inside Electron, which is running the Chromium engine, and offer some additional features. The Electron wrapper itself is specifically built for each platform. Windows and MacOS have an integrated updater for the Electron wrapper themselves. The Linux app is provided either as AppImage or as Debian package with a repository where the APT package manager handles the updates. The web app inside is self-updating on all platforms.

In contrast to the web app, the desktop apps have support for certificate pinning. They pin the public key of the Wire's leaf certificate. The desktop apps also support using multiple accounts with up to 3 accounts simultaneously. Another additional feature is support for screen sharing with up to 10 members via WebRTC video calls.

3.6 Wire Libraries

Wire has developed the platform independent libraries Proteus, HKDF and Cryptobox to implement their end-to-end encryption capabilities. Due to the fact that those libraries can be reused for platform dependent apps, the encryption implementation has to be audited only once and not for each app separately, thus serving as a major advantage compared to implementing the end-to-end encryption for each platform separately.

3.6.1 Proteus

Proteus [35] is a GPLv3+ implementation of the *Double Ratchet* algorithm and is written in Rust. For its usage of the cryptographic primitives ChaCha20, HMAC-SHA256 and Curve25519 Proteus relies on the implementation by the cryptographic library libsodium [36]. libsodium is available under the GPL compatible license ISC and was originally forked from the cryptographic library NaCl, which was introduced by Bernstein et al. [2]. Because libsodium is written in C, Proteus cannot directly use it, instead Proteus uses the Rust library sodiumoxide [37] which exports Rust bindings for libsodium. All messages, sessions and keys are serialized into the Concise Binary Object Representation (CBOR) [5] format.

[34] `https://electronjs.org/`, Accessed: 22-February-2020
[35] `https://github.com/wireapp/proteus`, Accessed: 22-February-2020
[36] `https://github.com/jedisct1/libsodium`, Accessed: 22-February-2020
[37] `https://github.com/sodiumoxide/sodiumoxide`, Accessed: 22-February-2020

3.6.2 HKDF

As key derivation function, Proteus uses the standardized *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)* [20]. The implementation of HKDF used, is also written in Rust [38].

3.6.3 Cryptobox

Cryptobox [39] is a library written in Rust which is available under GPLv3+. It offers a high-level API for more simple usage of Proteus. Additionally, Cryptobox uses a permanent storage to store miscellaneous key data i.e. session, pre- and identity keys.

3.7 Coax

Coax [40] is an experimental project designed to create a native Unix client, written in Rust, with a graphical user interface (GUI) that is meant to handle the Wire protocol. The motivation for Coax is to have a simple Wire client with a small code base, which is easily understandable and changeable. This project is also suitable for testing purposes.

Currently, Coax only supports sending and receiving of end-to-end encrypted, text-based messages and not the other features of the Wire protocol i.e. audio and video calling, transmission of data assets, ephemeral messaging and other features. Important security features, such as device verification or TLS certificate pinning are also unsupported. Therefore, the Coax client is vulnerable to *man-in-the-middle (mitm)* attacks.

Coax uses the Wire libraries Proteus 3.6.1 and Cryptobox 3.6.3 for its end-to-end encryption. It consists of the following components:

coax-net. This Rust library contains a simple HTTPS client which supports a subset of HTTP/1.1 with TLS encryption. For the TLS encryption coax-net uses OpenSSL [41] through the openssl crate [42] which provides Rust bindings for OpenSSL.

coax-ws. This component, which is also a Rust library, provides a client implementation of the WebSocket protocol [12]. The implementation makes use of coax-net to establish a WebSocket connection. This is necessary because WebSocket clients initially connect via HTTP to a web server and use the HTTP upgrade header to switch from the HTTP protocol to the WebSocket protocol. This procedure, to establish a WebSocket connection, is called WebSocket handshake.

[38] `https://github.com/wireapp/hkdf`, Accessed: 22-February-2020
[39] `https://github.com/wireapp/cryptobox`, Accessed: 22-February-2020
[40] `https://github.com/wireapp/coax`, Accessed: 22-February-2020
[41] `https://www.openssl.org/`, Accessed: 25-May-2018
[42] `https://crates.io/crates/openssl`, Accessed: 22-February-2020

coax-api-proto. This component contains message type definitions, also known as *protocol buffers*, which are generated by the official Wire protobuf definitions *generic-message-proto* [43] for the programming language Rust. *generic-message-proto* is also used by the other Wire clients to generate type definitions for the corresponding languages. These protobuf definitions are used for all messages sent between Wire clients and are end-to-end encrypted by Proton.

coax-api. This Rust library supplies Wire API data types with JSON serialization. Those data types are needed to communicate with the backend API. The backend REST API as well as the WebSocket use JSON as data exchange format.

coax-data. This Rust component provides a permanent data storage based on the SQL database SQLite [44].

coax-client. This client communicates with the Wire backend through the REST API and WebSocket connection and therefore handles all API calls, and continuously renews the required API access token when needed (Oauth 2.0 Bearer token).

coax-actor. This Rust library is higher-level client which abstracts all Coax layers below i.e. coax-client, -net, -ws, -data, -api and -api-proto. It also integrates the cryptographic libraries i.e. Cryptobox, Proteus, HKDF and libsodium.

coax-gtk. The last component of Coax contains a simple GTK user interface for interactive testing of the full Coax stack. It builds on top of coax-actor and thus uses all above-mentioned Rust libraries and combines them together into a Rust application.

3.8 Pidgin / libpurple

Pidgin [45] is a multi-protocol messenger which is written in C and available under GPLv2. As implied by the name, multi-protocol messenger Pidgin supports out of the box lots of messaging protocols i.e. XMPP, IRC, ICQ, SILC, SIMPLE and others. Originally, the project was known as Gaim and only supported the proprietary protocol AIM. Support for further protocols was added gradually. After a lawsuit by AOL about using the name Gaim, the project was renamed Pidgin, its current name.

Pidgin moved the support for its many messaging protocols to its own library libpurple. Thus, other projects i.e. Adium [46], Finch [47] or Bitlbee [48] are able to reuse the same protocol implementations.

[43] `https://github.com/wireapp/generic-message-proto`, Accessed: 22-February-2020
[44] `https://sqlite.org/`, Accessed: 22-February-2020
[45] `https://pidgin.im`, Accessed: 22-February-2020
[46] `https://adium.im`, Accessed: 22-February-2020
[47] `https://developer.pidgin.im/wiki/Using%20Finch`, Accessed: 22-February-2020
[48] `https://www.bitlbee.org`, Accessed: 28-June-2017

Both Pidgin and libpurple support a plugin system. Therefore, Pidgin and libpurple can be used for more broader applications because more features, particularly support for further messaging protocol, can be added through plugins.

Examples for such plugins are the well-known OTR plugin or the recently available OMEMO plugin for end-to-end encrypting text messages. While the OTR plugin is protocol agnostic and can be used to end-to-end encrypt text messages sent via any messaging protocol supported by libpurple, the OMEMO plugin is protocol specific and can only be used with the XMPP messaging protocol.

Additionally, there are libpurple plugins which offer support for newer messaging protocols i.e. Mattermost, Slack, Matrix.org, Discord, Rocket.Chat, Facebook Messenger, Skype and more [49].

[49] `https://pidgin.im/plugins/`, Accessed: 22-February-2020

CHAPTER 4

Implementation

This chapter contains the implementation details of PurpleWireRust with its components 4.1, the self-hosted Wire backend 4.2 and the Wire Python tools 4.3.

4.1 PurpleWireRust - PWR

PurpleWireRust (PWR) is a project which has been created during the course of this thesis to better understand the Wire protocol and for the following security analysis of Wire. The project implements a platform independent client for the Wire protocol. It extends the messenger Pidgin with support for Wire by implementing a libpurple plugin.

PWR supports the following features of the Wire protocol:

Basic account handling. PWR supports basic account handling features i.e. registering a Wire account and login/logout via email and password. Furthermore, PWR fetches the user profile and synchronizes the contact and group list. It is also supports adding new contacts and show contact information. Further, it is possible to accept incoming contact requests, or block and unblock contacts.

Text messages. As an important feature, PWR supports sending and receiving end-to-end encrypted text messages for 1:1 and group conversations. These messages are encrypted with Proteus, which implements the Double Ratched algorithm.

Ping messages. Furthermore, ping messages are available for 1:1 conversations. Unfortunately, libpurple has no built-in support for sending ping messages in group conversations. Therefore, PWR currently only exposes receiving ping messages for group conversations.

Delivery receipts. PWR automatically sends end-to-end encrypted delivery receipts for 1:1 conversations. However, it does not show incoming delivery notifications in the user interface because libpurple has no built-in support for this feature.

Device verification. Another security feature of PWR is the implementation of device verification by comparing public key-fingerprints of every device used by a contact.

TLS 1.2. As transport encryption, PWR exclusively uses TLS 1.2 and implements certificate pinning to mitigate malicious certificates issued by a trusted certificate authority.

Local aliases. By using Pidgin, PWR offers the additional feature to define local aliases which is currently not supported by any other client for Wire. This feature has already been requested by the community [1]. It can be very useful if a user has several contacts who use the same profile name, which can be confusing and can lead to security and/or privacy issues if the user shares some confidential information with the wrong contact [2]. For such cases, Wire clients usually also display a profile picture as an additional identifier. In some instances however, only the profile name is shown e.g. when the share feature is used on the Wire Android app, only a list of names of conversations and contacts appear, without additional profile pictures.

Unlimited multi account handling. Another feature that is supported by Pidgin, and thus also PWR, is to handle multiple Wire accounts simultaneously. Instead of a limited 3 accounts allowed by the official Wire clients, PWR allows unlimited multiple accounts.

PWR consists of the components *purple-wire*, *wire-rust-ffi* and *Coax* which are described in the following sections.

4.1.1 purple-wire

purple-wire is a plugin for the library libpurple and extends this library with support for the Wire protocol. It implements all functionality required of a protocol plugin. The plugin is written in C and makes use of *wire-rust-ffi* to implement the Wire protocol.

Table 4.1 defines the semantic mapping between libpurple and Coax required to connect the existing data fields of libpurple with those provided by Coax. Basically, Coax account data maps to libpurple's struct `account` struct. A contact matches libpurple's `buddy` struct and group conversations correspond to the struct `chat`. PWR uses email and password as login credentials. The profile name maps to libpurple's `alias`. The Coax' unique `user_id` (UUID) was assigned to a settings entry of libpurple to permanently store it in libpurple's profile data. It is intended that `buddy->alias` has no mapping because this field is used for the local alias feature. Each Wire contact has an implicit 1:1

[1] `https://github.com/wireapp/wire-webapp/issues/7122`, Accessed: 22-February-2020
[2] `https://github.com/wireapp/wire/issues/104`, Accessed: 22-February-2020

conversation which is identified by a unique `conv_id`. On libpurple's side it is stored into the `buddy->settings` option. The buddy name and chat name are unique identifiers in libpurple and therefore map to Coax' `user_id` and `conv_id`. The chat struct is special in that its ID needs to be a hash. Therefore, simply the hash of `conv_id` was used.

libpurple	Coax (wire-rust-ffi)
`account->username`	`email`
`account->password`	`password`
`account->alias`	`name`
`account->settings("user-id")`	`user_id`
`buddy->name`	`user_id`
`buddy->alias`	[3]
`buddy->server_alias`	`name`
`buddy->settings("conv-id")`	`conv_id`
`chat->name`	`conv_id`
`chat->alias`	`name`
`chat->id`	`g_str_hash(conv_id)`
`chat->components("conv-id")`	`conv_id`

Table 4.1: Semantic mapping between libpurple and Wire (Coax / wire-rust-ffi)

4.1.2 wire-rust-ffi

wire-rust-ffi is a library written in Rust, which makes it possible to integrate Coax into projects written in the programming language C. wire-rust-ffi exports a C interface using the *Foreign Function Interface (FFI)* feature from Rust. It contains the required functionality to abstract the Coax component coax-actor with all its underlying components into a C API.

In other words, wire-rust-ffi replaces Coax's user interface component coax-gtk and exports instead a C interface, which could then be used by other programs (i.e. purple-wire) to use the Coax stack as protocol implementation for Wire.

4.1.3 Extended Coax

During the course of implementing PWR, Coax 3.7 was extended by some new features and some existing features have been improved.

Certificate pinning. To mitigate man-in-the-middle attacks against the transport encryption, TLS certificate pinning has been added to Coax.

[3] `buddy->alias` gets defined by the user and not by the protocol plugin. As a fallback libpurple uses `buddy->server_alias` as alias if the user has not defined a local alias.

Drop TLS 1.1 support. Coax has support for TLS 1.1 and TLS 1.2 but to match what other Wire clients do at present, the support for TLS 1.1 has been removed. There is already an IETF Draft [4] in place to deprecate and disallow the usage of TLS 1.0 and TLS 1.1.

Device verification. Support for device verification (i.e. comparing public key-fingerprints) to mitigate man-in-the-middle attacks against the end-to-end encryption Proteus has also been added.

Ping messages. Another feature which was added to Coax is support for sending and receiving ping messages.

Bug fixes. Last but not least, many miscellaneous bugs have been fixed in Coax.

4.2 Self-Hosted Wire Backend

To complete the security analysis of Wire, the architecture of the backend (see also figure 3.1) was simplified slightly and its source code patched. All external cloud services and components which were not strictly needed for the analysis were removed and thus made it possible to self-host the backend (see also figure 4.1).

Disabled features. In order to run Wire's backend without any external services, several features were disabled (see also table 4.2). The feature for integrating 3rd party content was disabled, thus allowing the removal of the component *proxy* with its external services YouTube, Spotify, SoundCloud, Google Maps and Giphy. Additionally, the feature for sharing assets was also disabled in order to remove the asset server *cargohold* with its external services S3 and CloudFront. Because audio / video calling is not part of the security analysis, its support for call initiation behind firewalls or NAT was also disabled by removing the STUN/TURN service *restund*. Another feature which was disabled is the support for external push notification providers i.e. FCM or APNs. The relevant code in *gundeck* was disabled which allowed removal of the external service SNS. Furthermore, the support for phone number as identity in the component *brig* was disabled. Therefore, the external services Twilio and Nexmo were rendered useless and were removed as well. The component *brig* was further patched to disable the features responsible for user blacklisting and running multiple instances of brig which resulted in removal of the external services SQS and DynamoDB. Instead of sending email notifications to the external service SES *brig's* code was updated to write emails into the server log which is sufficient for the following security analysis. This enabled the removal of the last external service SES. Finally, the configuration of the reverse proxy *nginx* was appropriately adjusted to reflect the simplified backend architecture.

[4] `https://datatracker.ietf.org/doc/draft-ietf-tls-oldversions-deprecate/`, Accessed: 22-February-2020

Disabled feature	Obsoleted service
Integration for 3rd party content	*proxy*, YouTube, Spotify, SoundCloud, Google Maps, Giphy
Sending assets	*cargohold*, S3, Cloudfront
Audio / video calling	*restund* (STUN/TURN server)
External push notification providers	SNS
Phone number as identity	Nexmo, Twilio
User blacklisting	DynamoDB
Running multiple instances of brig	SQS, DynamoDB
Email notifications	SES

Table 4.2: Disabled features in order to remove all external services

Figure 4.1 shows the outcome of this customized Wire backend which is a simplified backend architecture consisting of only the internal services *brig*, *galley*, *gundeck*, *cannon*, the database servers Cassandra and Redis, and the search engine Elasticsearch.

Other changes. Since the Haskell services utilize the privileged user root inside the Docker container, the existing Dockerfile was edited to add an unprivileged user to execute the services. This was done to comply with the security principle of minimal privileges.

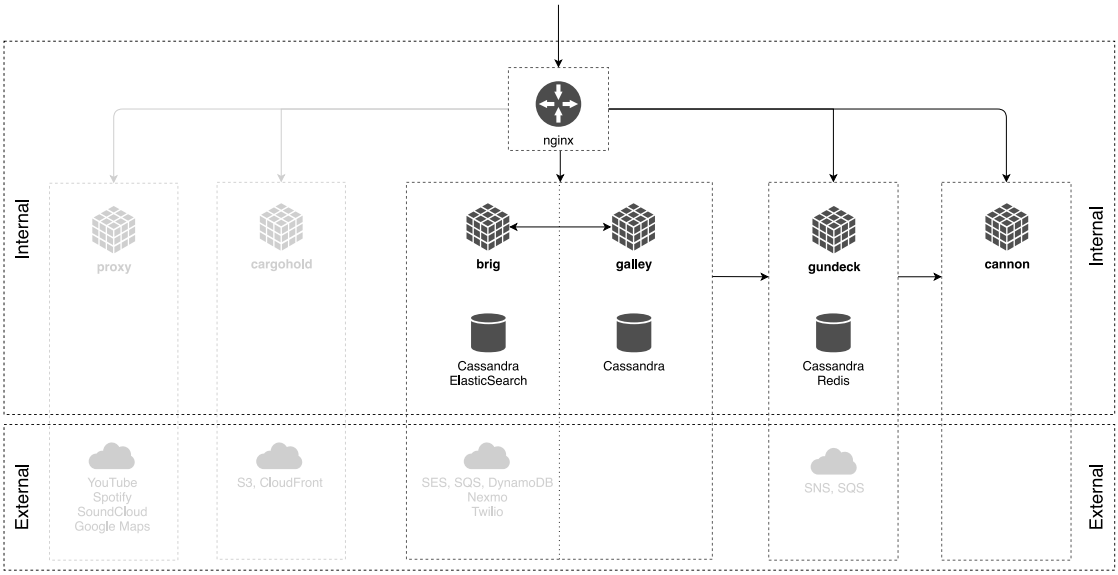

Figure 4.1: Adjusted architecture of the Wire backend

4.3 Wire Python Tools

In order to analyze Wire's REST API, a set of Python scripts have been developed. These consist of the REST client wrap.py and the scripts mitm-brig.py, mitm-gundeck.py and mitm-elastic.py for the man-in-the-middle proxy *mitmproxy* [5]. More information about mitmproxy can be found in section 5.2.

wrap.py. Wire Rest Api Python is a simple HTTP client for Wire's REST API. It wraps the Python *requests* library and enhances it with client-side handling for Wire's Oauth 2.0 authentication. wrap.py is used as a low-level interactive REST client to directly call Wire's REST API with arbitrary API calls. It is meant to be used by scripts or in an interactive Python shell i.e. ipython. It is used to demonstrate the possibilities of an adversary that has already gained access to login credentials or to the long-term authentication cookie. wrap.py allows for easy manipulation of user profile data that is stored server-side, or to retrieve the accessible metadata of a user from Wire's server.

mitm-brig.py. This Python script is for the man-in-the-middle proxy *mitmproxy* and allows to strip arbitrary client IDs of an arbitrary Wire user from the API call `GET /users/$USER_ID/clients` which is directed to the Wire service brig.

mitm-gundeck.py. This mitmproxy script allows to drop incoming `user.client-add` notifications for an arbitrary Wire user before it reaches the notification hub gundeck. It also allows to strip client IDs from the recipients of an arbitrary notification which can be used to hide a certain client and therefore prevents the delivery of this notification to that client.

mitm-elastic.py. This mitmproxy script is a short helper function which allows mitmproxy to pretty print JSON data sent to the search engine ElasticSearch. It simply injects the missing HTTP header `content-type: application/json; charset=UTF-8` to each HTTP request. Thus, enabling mitmproxy to successfully determine JSON as content type and use its built-in pretty print function to show the data in a human readable output.

[5] `https://mitmproxy.org/`, Accessed: 22-February-2020

CHAPTER 5

Security Analysis of Wire

This chapter describes all requirements and perquisites to analyze the security and privacy of Wire. Those consist of the threat model 5.1, the analysis tools 5.2 and the local test setup of the self-hosted Wire backend 5.3.

5.1 Threat Model

As prerequisite for the evaluation of Wire and subsequently also Signal, a threat model was required. It is based on Unger et al. [56] and defines the following attackers:

Local adversary is an attacker who gains access to the victim's device (i.e. physical access or remote via malware).

Local network adversary is an attacker who controls the local network (i.e. owner of the WiFi or LAN).

Global network adversary is an attacker who controls large parts of the Internet (i.e. nation states or large Internet service providers).

Service providers are the providers of the messaging service infrastructure (i.e. back-ends of Wire and Signal, third-party services) which can be malicious as well.

All these types of attackers can act as passive or active adversary. Further, it is assumed that all adversaries participate in the messaging system, allowing them to use all normal messaging features (i.e. sending messages).

5.2 Analysis Tools

The following tools were used as part of the analysis of this thesis:

mitmproxy. mitmproxy [1] is a Python man-in-the-middle proxy tool. Aside from a very useful interactive command line interface (CLI), it also features a powerful Python API which can and has been used for more sophisticated man-in-the-middle scenarios.

Wire Python Tools. These Wire Python Tools 4.3 were used to analyze Wire's REST API: REST client wrap.py and mitmproxy scripts, the latter making use of the Python API of mitmproxy.

Wireshark. Wireshark [2], a powerful network protocol analyzer, was used to analyze notifications sent over WebSocket connections. Wireshark was necessary because mitmproxy's support for WebSocket is limited: it does not show the binary WebSocket frames which Wire uses in the interactive command line interface.

redis-cli. redis-cli [3] is a command line interface for the key-value database redis. It allows the user to send queries to redis databases and can be used in interactive mode. The tool was used to inspect the key-value database stored by Wire's backend.

cqlsh. cqlsh [4] is an interactive command line interface for Cassandra databases. It is used to send CQL (Cassandra Query Language) queries to Cassandra. The tool was used to inspect all data and metadata which are stored into Cassandra by Wire's backend.

hardening-check. The command line tool hardening-check [5] allows the user to check ELF (Executable and Linkable Format) binaries, the standard format for executables on Linux, for the presence of security hardening features (i.e. Position Independent Executable (PIE), stack protected, read-only relocations, and immediate binding). PIE points out that the text section of the ELF binary can be relocated within memory, which is necessary to take advantage of Address Space Layout Randomization (ASLR) protection. ASLR makes binary exploitation of memory corruption vulnerabilities more challenging. Stack protected indicates that the program has been built with the compiler flag *-fstack-protector* which protects against stack smashing attacks. Read-only relocations (RELRO) shows that the program was built with *-Wl,-z,relro* linker flag which is also known as partial RELRO. This protection reduces the possible memory areas of an ELF binary which can be used by a memory corruption exploit. Immediate binding indicates that the program was built with *-Wl,-z,now* linker flag. If the binary was built with support for both read-only relocations and immediate binding it is called full RELRO, as

[1] `https://mitmproxy.org/`, Accessed: 22-February-2020
[2] `https://www.wireshark.org/`, Accessed: 22-February-2020
[3] `https://redis.io/topics/rediscli`, Accessed: 22-February-2020
[4] `https://cassandra.apache.org/doc/latest/tools/cqlsh.html`, Accessed: 22-February-2020
[5] `https://packages.debian.org/buster/devscripts`, Accessed: 22-February-2020

opposed to partial RELRO. Full RELRO further reduces the available memory regions which can be used by memory corruption exploits. [6]

εxodus-standalone. εxodus [7] allows the static analysis of APK (Android package) files, which is the standard package format for Android, for Android permissions and known trackers. εxodus-standalone is the standalone version of εxodus though an online service is also available [8]. The standalone tool was chosen because it allows more insight regarding tracker analysis results. It parses the Android manifest file to enumerate all permissions required by the app. In order to detect potential tracking services, it analyzes all included Java class files inside the APK file. It does not analyze the actual content of those class files instead it compares each class name, including the namespace, against a list of known tracking classes. Drawbacks of this method are that it only finds already known trackers and that εxodus does only static analyses and no dynamic analyses during runtime. Thus, it cannot detect whether the tracker classes are actually used or not, particularly if the tracking feature has been explicitly disabled i.e. via Android manifest file. Therefore, εxodus is, to a certain degree, prone to false positive results.

MobSF. The Mobile Security Framework (MobSF) [9] is a security framework for analyzing mobile apps on Android, iOS and Windows. While it can decompile APKs, it was not necessary to use this feature because the source code was readily available. MobSF also integrates the tracker analysis functionality from εxodus but does not allow such an in-depth analysis as the standalone Python tool. Further, it shows the required Android permissions and has some support for detecting known malware i.e. detecting known URLs inside the APK which are known to provide malware.

keytool. Keytool [10] is a tool that manages X.509 certificates and their cryptographic keys. It can also be used to read certificates from signed JAR files i.e. Android APK files. In addition to being able to read the certificate, it also shows all details of the certificate itself i.e. certificate owner and issuer, as well as cryptographic information i.e. certificate fingerprint, used signature algorithm and the public key algorithm with its key length.

ssltest. SSL server test (ssltest) [11] is an online service for analyzing TLS web servers. It does a deep analysis of the TLS setup consisting of TLS features, certificates, configuration and common TLS vulnerabilities. Important factors for the TLS configuration are which cipher suites and TLS protocol versions are enabled.

[6] `https://manpages.debian.org/buster/devscripts/hardening-check.1.en.html`, Accessed: 22-February-2020

[7] `https://github.com/Exodus-Privacy/exodus-standalone`, Accessed: 22-February-2020

[8] `https://reports.exodus-privacy.eu.org/`, Accessed: 22-February-2020

[9] `https://github.com/MobSF/Mobile-Security-Framework-MobSF`, Accessed: 22-February-2020

[10] `https://docs.oracle.com/en/java/javase/11/tools/keytool.html`, Accessed: 22-February-2020

[11] `https://www.ssllabs.com/ssltest/`, Accessed: 22-February-2020

securityheaders. Securityheaders [12] is a checking tool, available online, meant exclusively for HTTP security headers i.e. HSTS, CSP, Referrer-Policy and more.

webbkoll. Webbkoll [13] is another online tool used for analyzing privacy protecting measures. It checks for HTTPS, HTTP security headers, cookies (first-party and third-party), cookie attributes, third-party requests and server location.

dig, host. These two tools [14] [15] can be used to perform DNS lookups, with dig being the most powerful tool of the two. They allow interaction with DNS servers in order to query information about domain names and IP addresses.

5.3 Local Test Setup of the Self-Hosted Wire Backend

For the analysis of Wire's protocol, backend communication, clients and also for analyzing the metadata, the self-hosted Wire backend, as already described in section 4.2, has been used. To better analyze the backend communication, multiple man-in-the-middle proxies were added to the backend setup i.e. between client and server, as well as between the internal services of the backend (see also figure 5.1). For this, the Python tool *mitmproxy* [16] was used as man-in-the-middle proxy tool.

Each service of the backend was deployed inside its own Docker container and all containers were connected to each other within an internal Docker network. By default Docker networks always have access to the external world through the Docker host. However, internal Docker networks restrict external access in a way that containers cannot connect to the outside of the internal network. The following Docker command was used to create the internal Docker network for the Wire backend:

Listing 5.1: Docker command to create internal network

```
1 docker network create --driver bridge \
2     --subnet 172.20.0.0/24 \
3     --internal \
4     wire-backend
```

By using this internal network feature of Docker, it was ensured that the internal services cannot make any requests to any external service i.e. some AWS services. This network isolation feature was therefore used to verify that all external dependencies of the self-hosted Wire backend were properly removed.

[12] https://securityheaders.com/, Accessed: 22-February-2020
[13] https://webbkoll.dataskydd.net/en/, Accessed: 22-February-2020
[14] https://packages.debian.org/buster/dnsutils, Accessed: 22-February-2020
[15] https://packages.debian.org/buster/bind9-host, Accessed: 22-February-2020
[16] https://mitmproxy.org/, Accessed: 22-February-2020

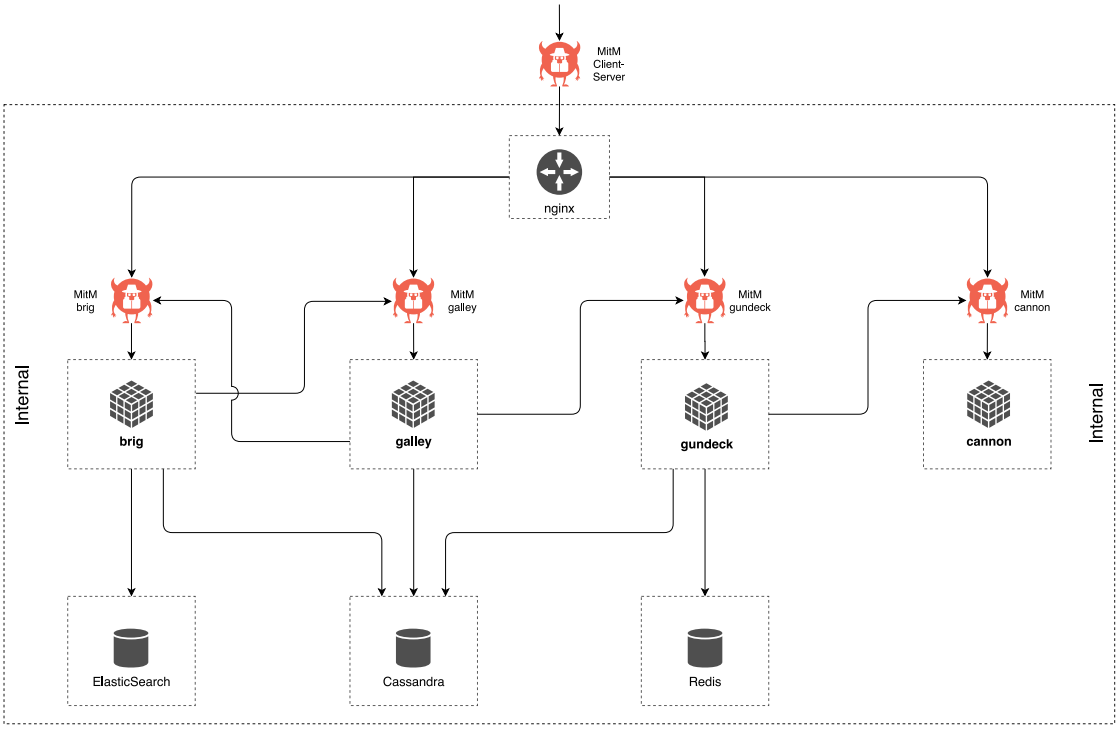

Figure 5.1: Setup of the Wire backend for the security analysis

For transport encryption the reverse proxy nginx was configured with a self-signed certificate. Further, the zauth keys for the nginx-zauth-module, which are needed for the OAuth 2.0 authentication, had to be generated.

The clients: web app, Coax and PWR had to be configured to use the self-hosted backend instead of the official Wire backend. Accordingly, the URL of the REST API and the URL of the WebSocket were changed to the self-hosted backend. To establish the transport encrypted HTTPS connection it was required to import the self-signed certificate into the certificate store of the browser. Because the extended Coax, and thus also PWR, implements certificate pinning it was required to add a corresponding pinning for this self-signed certificate.

The web app, Coax and PWR was used to successfully verify that the self-hosted backend with the reduced feature set was still working properly.

CHAPTER 6

Results

This chapter describes the results of the security and privacy analysis of Wire. To better understand and interpret the results, Signal too has been analyzed where needed. The results are split into the analysis of the messaging protocol itself 6.1, the production environments of the messaging platform 6.2, the app analysis 6.3 and metadata analysis 6.4.

6.1 Analysis of Signal's and Wire's Messaging Protocols

The analysis of Signal's and Wire's messaging protocols was performed by using the systematization of Unger et al. [56]. Therefore, the analysis is divided into the following three problem areas: *trust establishment*, *conversation security* and *transport privacy*.

Trust establishment. The process of users exchanging and verifying each other's cryptographic long-term keys is called *trust establishment*. The performance results of the trust establishment analysis can be seen in table 6.1. Both Wire and Signal use a layered approach for trust establishment: Signal uses key directory, TOFU and optional key fingerprint verification. Wire uses key directory and optional key fingerprint verification.

Signal has partial support for multiple keys. A Signal profile can only be linked to one primary device (either Android or iOS app) but offers the option for multiple secondary devices (desktop apps). The advantage of this primary-secondary device approach is that users simply compare the master key for fingerprint verification. Secondary devices are transitively trusted.

On the other side, Wire has full support for multiple key support. A Wire profile can have up to eight devices (i.e. Android, iOS, desktop or Web app) attached to it. However, a drawback is that for each additional device the manual key verification task has be to performed again with all other users. This lowers the usability and thus lowers the

Scheme	Protocol	Security Features						Usability										Adoption					
		Network MitM Prevented	Operator MitM Prevented	Operator MitM Detected	Operator Accountability	Key Revocation Possible	Privacy Preserving	Automatic Key Initialization	Low Key Maintenance	Easy Key Discovery	Easy Key Recovery	In-Band	No Shared Secrets	Alert-less Key Renewal	Immediate Enrollment	Inattentive User Resistant	Multiple Key Support	No Service Provider	No Auditing Required	No Name Squatting	Asynchronous	Scalable
Key Directory+TOFU+Optional Verification	Signal	◐	◐	◐	◐	●	-	●	●	●	●	●	●	-	●	-	◐	-	●	●	●	●
Key Directory+Optional Verification	Wire	◐	◐	◐	◐	●	-	●	●	●	●	●	●	◐	●	-	◐	-	●	●	●	●

● = provides property; ◐ = partially provides property; - = does not provide property

Table 6.1: Trust establishment of Signal & Wire, based on Unger et al. [56]

probability that users perform the manual key fingerprint verification task and further lowers strong authentication. Compared with Signal, Wire lacks support for fingerprint verification based on QR-codes. Thus, Wire users have to use the hexadecimal based representation of key fingerprint which is not very user friendly. Another downside of Wire in terms of trust establishment security is that the TOFU warning message (whenever a contact's key changes) only appears after an initial key fingerprint verification of that contact. On the other hand, this improves the usability because fewer warnings are displayed if users have not managed to verify their fingerprints.

Conversation Security. In order to protect the security and privacy of exchanged messages, a secure messaging solution has to provide *conversation security*. This requires the use of cryptographic protocols to encrypt messages. The results of how Signal and Wire perform in conversation security can be seen in table 6.2. The features differentiate between 1:1 and group messaging and therefore Signal and Wire are reported in separate rows.

Scheme	Protocol	Security and Privacy													Adoption					Group Chat				
		Confidentiality	Integrity	Authentication	Participant Consistency	Destination Validation	Forward Secrecy	Backward Secrecy	Anonymity Preserving	Speaker Consistency	Causality Preserving	Global Transcript	Message Unlinkability	Message Repudiation	Particip. Repudiation	Out-of-Order Resilient	Dropped Message Resilient	Asynchronicity	Multi-Devices Support	No Additional Service	Computational Equality	Trust Equality	Subgroup Messaging	Contractable	Expandable
Authenticated DH +Double Ratchet+3DH AKE+Prekeys	Signal	●	●	●	●	●	●	●	-	◐	●	-	●	●	●	◐	◐	●	◐	-					
+Double Ratchet+3DH AKE+Prekeys	Wire	●	●	●	●	●	●	●	-	◐	●	-	●	●	●	◐	◐	●	●	-					
OTR Network+Star Topology +Pairwise Axolotl+Multicast Encryption	Signal	●	●	●	-	●	●	●	-	●	●	-	●	●	●	●	●	●	◐	-	●	●	●	●	●
+Pairwise Axolotl+Multicast Encryption	Wire	●	●	●	-	●	●	●	-	●	●	-	●	●	●	●	●	●	●	-	●	●	-	●	●

● = provides property; ◐ = partially provides property; - = does not provide property

Table 6.2: Conversation security of Signal & Wire, based on Unger et al. [56]

For 1:1 message encryption Signal uses the double ratchet algorithm and the triple Diffie-Hellman authenticated key exchange (3-DH AKE) as well as prekeys for asynchronous messages. Incidentally, Signal has added multi-device support since the publication of Unger et al. [56]. Wire uses the same message security protocol as Signal for 1:1 messages.

For group messaging Signal uses the scheme OTR network, star topology, pairwise axolotl (also known as double ratchet) and multicast encryption. Wire does not use the multicast encryption for simple text messages in group messaging. Instead it uses the same message encryption as in 1:1 messages. However, for media files (internally also known as assets) Wire use the multicast encryption feature due to performance reasons as messages are usually much larger than just simple text messages.

Transport privacy. Transport privacy defines how much communication metadata (i.e. sender, receiver, which conversation) is hidden during the exchange of messages between the participants. The results of the transport privacy analysis can be seen in table 6.3. Both Signal and Wire use the store-and-forward scheme.

Scheme	Protocol	Privacy					Usability						Adoption							
		Sender Anonymity	Recipient Anonymity	Particip. Anonymity	Unlinkability	Global Adv. Resistant	Contact Discovery	No Message Delays	No Message Drops	Easy Initialization	No Fees Required	Topology Independent	No Additional Service	Spam/Flood Resistant	Low Storage	Low Bandwidth	Low Computation	Asynchronous	Scalable
Store-and-Forward	Signal	◐	-	◐	◐	-	●	◐	●	●	●	●	-	-	●	●	●	●	●
Store-and-Forward	Wire	-	-	-	-	-	●	◐	●	●	●	●	-	◐	●	●	●	●	●

● = provides property; ◐ = partially provides property; - = does not provide property

Table 6.3: Transport privacy of Signal & Wire, based on Unger et al. [56]

Since the publication of Unger et al. [56], Signal introduced sealed sender [29] which improves sender anonymity. Additionally, Signal has better participation anonymity and unlinkability properties than Wire because it hides its group ID inside end-to-end encrypted 1:1 messages, thus the server does not see which messages belong to which conversation.

On the other side, Wire has better spam control, because users need to accept 1:1 conversations before a participant can send messages. However, this means that user connections are stored in the backend, thus storing additional non-temporary metadata.

6.2 Analysis of Signal's and Wire's Production Environments

The production environments of Signal's and Wire's backend and websites have been analyzed. For this analysis the following tools have been used: the online services ssltest 5.2, securityheaders 5.2 and webbkoll 5.2 as well as the command line tools dig and host 5.2. The results of this analysis can be seen in table 6.4 for the backend analysis and table 6.5 for the website analysis.

	Signal Backend	Signal CDN	Wire Backend	Wire WebSocket
URL	textsecure-service.whispersystems.org	cdn.signal.org	prod-nginz-https.wire.com	prod-nginz-ssl.wire.com
Server hosted by	Amazon	Amazon	Amazon	Amazon
Server location	US	US	EU	EU
SSL Report	T (A)	T (A)	A+	A+
Certificate issuer	TextSecure (untrusted CA)	TextSecure (untrusted CA)	DigiCert	DigiCert
Cert valid until	12-Mar-2029	12-Mar-2029	11-Mar-2020	11-Mar-2020
Key	RSA 2048 bit	RSA 2048 bit	RSA 2048 bit	RSA 4096 bit
Signature algorithm	*SHA256withRSA*	*SHA256withRSA*	*SHA256withRSA*	*SHA256withRSA*
Certificate transparency	✗	✗	✓	✓
Revocation information	None	None	CRL, OCSP	CRL, OCSP
DNS CAA	✗	✓	✗	✓(wire.com)
TLS protocols	TLS 1.0, 1.1, 1.2	TLS 1.0, 1.1, 1.2	TLS 1.2	TLS 1.2
FS only cipher suites	✗	✗	✓	✓

Table 6.4: Environment of Signal's and Wire's messaging backend

Backend services analysis. Signal and Wire are both hosted by Amazon AWS with the difference that Signal's servers are located in the US and Wire's servers in the EU. Signal uses an untrusted root certificate for its backend services which means that Signal does not build on trusted root certificate authorities but instead uses its own self-signed root certificate. Obviously, all Signal clients need to trust this root certificate. On the other hand, Wire uses a certificate signed from the publicly trusted certificate authority DigiCert. Further, it uses DNS Certification Authority Authorization (CAA) [15] and certificate transparency to mitigate certificates created from a malicious CA. Wire allows only DigiCert as exclusive certificate authorization to issue certificates for Wire's backend domains (see listing 6.1 for analysis of the DNS CAA records).

Listing 6.1: Inspecting Wire's DNS record for CAA support

```
1 $ host -t caa app.wire.com
2 app.wire.com has CAA record 0 issue "digicert.com"
3
4 $ host -t caa prod-nginz-https.wire.com
5 prod-nginz-https.wire.com has CAA record 0 issuewild "digicert.com"
6
7 $ host -t caa prod-nginz-ssl.wire.com
8 prod-nginz-ssl.wire.com has CAA record 0 issuewild "digicert.com"
```

For all Wire clients except the web app Wire additionally uses certificate pinning. As transport protocol Wire uses TLS 1.2 exclusively. In addition to TLS 1.2, Signal also supports the deprecated TLS protocols 1.0 and 1.1 on its backend services which should be removed in the near future. The leaf certificates for Signal's backend are valid until March 12th 2029. An expiration date 9 years in the future is much too long for a leaf certificate in combination with no support for certificate revocation. Therefore, Signal would need to update all its clients to explicitly untrust this valid certificate in case it was stolen. In this scenario it would unnecessarily expose all devices without the update to this potential threat.

Website analysis. Despite Wire stating that their servers are located in the EU, some parts of the websites are hosted in the US. For instance, *wire.com* uses third-party

	Signal	Wire	Signal	Wire
URL	signal.org	wire.com	support.signal.org	support.wire.com
Mail hosted by	Google	Google	Amazon	Amazon
Website hosted by	Amazon	Amazon	Zendesk	Zendesk
Server location	US	EU	US	US
SSL Report	A+	A+	A	A
Certificate issuer	Amazon	DigiCert	Let's Encrypt	Let's Encrypt
Key	RSA 2048 bit	RSA 2048 bit	RSA 4096 bit	RSA 4096 bit
Signature algorithm	*SHA256withRSA*	*SHA256withRSA*	*SHA256withRSA*	*SHA256withRSA*
Certificate transparency	✓	✓	✓	✓
Revocation information	CRL, OCSP	CRL, OCSP	OCSP	OCSP
DNS CAA	✗	✓	✗	✓(wire.com)
TLS protocols	TLS 1.1, 1.2	TLS 1.2	TLS 1.2	TLS 1.2
FS only cipher suites	✗	✓	✗	✗
3rd party requests	9 to 5 unique hosts	22 to 2 unique host	15 to 7 unique hosts	19 to 5 unique hosts
3rd party server location	US	US, DE	US	US
1st party cookies	2	4	7	4
3rd party cookies	1	0	3	3
Secure cookies only	✗	✗	✗	✗
No Referrers leaked	✗	✓	✗	✗
Security Headers Report	D	A	C	C
HSTS	✓	✓	✓(too short)	✓(too short)
HSTS include subdomain	✗	✓	✗	✗
HSTS preload	✓	✓	-	-
Content-Security-Policy	✗	✓	✗	✗
Referrer-Policy	✗	✓	✗	✗
X-Frame-Options	✗	✓	✓	✓
X-XSS-Protection	✗	✓	✓	✓
X-Content-Type-Options	✗	✓	✓	✓

Table 6.5: Analysis results of Signal's and Wire's websites

requests to *images.ctfassets.net*, a site hosted in the US, to store images of the Wire website. Another important website is Wire's support website which is also referenced from within Wire's apps. This support site is hosted by Zendesk which is also located in the US. Incidentally, Signal hosts its support site on the same provider. Actually, both support subdomains resolve to the same public IPv4 addresses as can be seen in listings 6.2 and 6.3.

Listing 6.2: Inspecting Wire's DNS record for its support site

```
1  $ dig support.wire.com
2  ...
3  ;; ANSWER SECTION:
4  support.wire.com.          300     IN      CNAME   wearezeta.ssl.zendesk.com.
5  wearezeta.ssl.zendesk.com. 900     IN      CNAME   wearezeta.zendesk.com.
6  wearezeta.zendesk.com.     900     IN      A       104.16.51.111
7  wearezeta.zendesk.com.     900     IN      A       104.16.52.111
8  wearezeta.zendesk.com.     900     IN      A       104.16.53.111
9  wearezeta.zendesk.com.     900     IN      A       104.16.54.111
10 wearezeta.zendesk.com.     900     IN      A       104.16.55.111
```

Listing 6.3: Inspecting Signal's DNS record for its support site

```
$ dig support.signal.org
...
;; ANSWER SECTION:
support.signal.org.      300     IN      CNAME   owssupport.zendesk.com.
owssupport.zendesk.com.  900     IN      A       104.16.51.111
owssupport.zendesk.com.  900     IN      A       104.16.52.111
owssupport.zendesk.com.  900     IN      A       104.16.53.111
owssupport.zendesk.com.  900     IN      A       104.16.54.111
owssupport.zendesk.com.  900     IN      A       104.16.55.111
```

The TLS configuration of both support sites can be improved as well as Signals main site by dropping all non forward secrecy cipher suites and dropping TLS 1.1 from Signals main site. The HTTP security headers of Wire's main site are appropriately used but should be improved on Wire's support site, particularly the HSTS header, which is poorly configured. The *max-age* parameter is set to 3 days, but should be set to a minimum of 30 days, if not to the recommended 6 months, to be useful in mitigating attacks i.e. SSL-stripping. Cookie flags should be improved too by always setting the Secure and SameSite flags and set HTTPOnly where applicable. Signal on the other side has even more room for improvement regarding HTTP security headers and cookie flags on any of its websites.

6.3 Security Analysis of Wire's Apps

The following subsections describe the results of the security analysis of Wire's apps. To better understand and interpret the results, they have been either compared with each other or compared to the Signal equivalent.

6.3.1 Differences of Security and Privacy Between Wire Apps

The following Wire apps have been analyzed regarding privacy and security features: Wire Android app, Wire web app, Wire desktop app on Linux (which is based on Electron), and the experimental clients Coax and PurpleWireRust (PWR). For this comparison the apps have been analyzed by manual code inspection and manual testing. The results can be found in table 6.6.

Generally, all clients use TLS 1.2, only Coax additionally supports TLS 1.1. Certificate pinning is used to harden the TLS connection on Android, the Electron desktop app and PWR. The web app does not implement certificate pinning but it profits from some alternative technologies which improve the trust in the certificate: DNS CAA and certificate transparency. The original Coax client does not support certificate pinning either.

All apps, except Coax, support device verification and show a warning message if a new device is used after prior verification. Only PWR shows an additional warning message

Feature	Android	Web app	Desktop (Linux)	Coax	PWR
TLS	1.2	1.2	1.2	1.1, 1.2	1.2
Certificate pinning	✓	✗	✓	✗	✓
Device verification	✓	✓	✓	✗	✓
Alert new devices without prior verification	✗	✗	✗	✗	✓
Alert new devices with prior verification	✓	✓	✓	✗	✓
Number of prekeys	100	10	10	200	200
Generates new keys	✓	✓	✓	✗	✗
Encrypted backup	✓	✗	✗	N/A	N/A
Disable link preview	✗	✗	✓	N/A	N/A

Table 6.6: Wire apps - where they differentiate regarding privacy & security features

if the device has not been verified before. Therefore, PWR is the only Wire client which properly implements the TOFU technique.

The management of prekeys differs significantly between all Wire apps. Only the Android app has a good implementation. It supports up to 100 prekeys and generates new keys as needed if the count of available prekeys falls under the threshold of 50 prekeys. The prekeys management of the web app and the Electron desktop app supports only a maximum number of 10 prekeys. Although there is no threshold, like on Android, the available pool of prekeys will always be filled up to 10. This means that only 9 prekeys can be used with full perfect forward secrecy because the last key is used as fallback key for each new session as long as there are no further available prekeys.

The Android app is the only one that supports encrypted backups. Wire's web app and the desktop client only support unencrypted backups. Coax and PWR do not have a backup feature at all.

Coax and PWR do not feature a link preview. The desktop app supports the optional disabling of generating link previews, which leaks the IP address of the client to the service provider of the provided link. Although the service provider most likely has the IP address already, if the sender visited the website with a web browser before. A benefit for privacy is that the preview is only generated on sender-side which does not automatically leak the IP address of all receiving clients. The user's IP address is only revealed if the recipient opens the link with a web browser.

6.3.2 Security and Privacy Analysis of Signal's and Wire's Android Apps

The tools εxodus, MobSF and keytool have been used to analyze the Android apps of Signal and Wire. The results of this analysis can be found in table 6.7.

In contrast to Wire, where the phone number is optional and an email address can be used, Signal cannot be used without a phone number. If the permission to access

Feature	Signal	Wire
Analyzed app version	4.52.4	3.44.877
Usable without phone number	✗	✓
Usable without address book upload	barely	✓
Usable without Google Play Store	✓	✓
Usable without FCM	✓	✓
Provides direct APK download	✓	✓
Provides SHA-256 sum of file	✗	✓
Provides fingerprint of app signing certificate	✓	✓
App signing certificate key	RSA 1024 bit	RSA 2048 bit
App signing certificate valid until	16-May-2045	30-June-2040
App signing algorithm	*SHA1withRSA*	*SHA1withRSA*
Certificate pinning	✓	✓
App locking	✓	✓
Screenshot protection	✓	✓
Key fingerprint verification	✓	✓
Verification via QR-Code	✓	✗
Sealed sender	✓	✗
3rd party trackers	0	0
(all) Android permissions	65	19
(dangerous) Android permissions	17	7

Table 6.7: Privacy & security features of Signal and Wire on Android

the phone's address book is not given, Signal is barely usable since any contact will be listed as phone number only. Especially group conversations can become confusing easily. Signal profiles are only a partial solution to this problem because they are only shown if the profile owner shared them.

Wire and Signal work well in an Android environment if the environment is without GApps and thus without FCM i.e. Custom Android ROMs. Both, Wire and Signal, support an alternative APK download via their websites in the absence of Google play store. However, there is no direct link to the APK download on Signal's website. Instead, users have to know the exact URL for this page [1] or search the page via an internet search engine. Further, the page requires a third-party JavaScript from Google to actually show the download link.

The app signing certificate key of Signal is only 1024 bit long which is too short for a certificate that is valid until May 16th 2045. Wire has a 2048 bit key which should also be longer as it is valid until June 30th 2040. See also listings 6.4 and 6.4. Both Android

[1] `https://signal.org/android/apk/`, Accessed: 22-February-2020

apps use the insecure app signing algorithm SHA1withRSA. SHA1 must not be used anymore.

Listing 6.4: keytool output from Signal APK

```
$ keytool -printcert -jarfile Signal-website-universal-release-4.52.4.apk
Signer #1:

Signature:
Owner: CN=Whisper Systems, OU=Research and Development, O=Whisper Systems, L=Pittsburgh, ST=PA, C=US
Issuer: CN=Whisper Systems, OU=Research and Development, O=Whisper Systems, L=Pittsburgh, ST=PA, C=US
Serial number: 4bfbebba
Valid from: Tue May 25 17:24:42 CEST 2010 until: Tue May 16 17:24:42 CEST 2045
Certificate fingerprints:
  SHA256: 29:F3:4E:5F:27:F2:11:B4:24:BC:5B:F9:D6:71:62:C0:EA:FB:A2:DA:35:AF:35:C1:64:16:FC:44:62:76:BA:26
Signature algorithm name: SHA1withRSA
Subject Public Key Algorithm: 1024-bit RSA key
Version: 3
```

Listing 6.5: keytool output from Wire APK

```
$ keytool -printcert -jarfile wire-prod-release-3.44.877.apk
Signer #1:

Signature:
Owner: CN=Unknown, OU=Unknown, O=Zeta, L=Unknown, ST=Unknown, C=Unknown
Issuer: CN=Unknown, OU=Unknown, O=Zeta, L=Unknown, ST=Unknown, C=Unknown
Serial number: 511a2d3b
Valid from: Tue Feb 12 12:53:31 CET 2013 until: Sat Jun 30 13:53:31 CEST 2040
Certificate fingerprints:
  SHA256: 16:26:E3:F8:5D:FD:84:34:F7:86:66:44:48:61:F4:E5:C8:FB:37:7A:28:4C:1C:30:4C:B9:D5:85:28:8F:A3:52
Signature algorithm name: SHA1withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3
```

Both apps have important security features i.e. certificate pinning, app locking, screenshot protection, key fingerprint verification. One difference is that Signal has support for verification via QR-Code which greatly improves the usability of key fingerprint verification. Further, Signal has support for sealed sender which improves sender anonymity.

Fortunately, both Signal and Wire come without any third-party trackers. In Wire, the last tracker *Google Firebase Analytics* was disabled in Wire's Android manifest 6.6 and some last remaining traces of the firebase analytics library were removed in the Gradle build file 6.7 to silence a false-positive in εxodus. As can be seen on table 6.7, Signal requires many more permissions from Android than Wire. Additionally, the number of dangerous permissions is also higher. Some of those permissions are related to Signal's

additional support for sending and receiving SMS and MMS. Some permissions are unused i.e. read and write calendar and should be removed. See also table 6.8 for the full list of dangerous Android permissions required by both Signal and Wire.

Listing 6.6: Disabled FCM tracking in Wire's Android manifest

```
1 <!--Disable tracking in the FCM core-->
2 <meta-data android:name="firebase_analytics_collection_deactivated"
      android:value="true" />
3 <meta-data android:name="google_analytics_adid_collection_enabled"
      android:value="false" />
```

Listing 6.7: Remove last traces of firebase analytics library in app/build.gradle

```
1 dependencies {
2   ...
3   implementation ('com.google.firebase:firebase-messaging:17.3.0') {
4     exclude group: 'com.google.firebase', module: 'firebase-analytics'
5     exclude group: 'com.google.firebase', module: 'firebase-measurement-
          connector'
6   }
7   ...
8 }
```

Permission	Signal	Wire
ACCESS_COARSE_LOCATION	✗	
ACCESS_FINE_LOCATION	✗	✗
CALL_PHONE	✗	
CAMERA	✗	✗
GET_ACCOUNTS	✗	
READ_CALENDAR	✗	
READ_CONTACTS	✗	✗
READ_EXTERNAL_STORAGE	✗	✗
READ_PHONE_STATE	✗	✗
READ_SMS	✗	
RECEIVE_MMS	✗	
RECEIVE_SMS	✗	
RECORD_AUDIO	✗	✗
SEND_SMS	✗	
WRITE_CALENDAR	✗	
WRITE_CONTACTS	✗	
WRITE_EXTERNAL_STORAGE	✗	✗
Dangerous permissions count	17	7

Table 6.8: Dangerous Android permissions of Signal and Wire

6.3.3 Analysis of Desktop Apps for Linux Security Hardening Features

To compare the Linux security hardening features (i.e. Position Independent Executable (PIE), stack protected, read-only relocations, and immediate binding) of Signal's and Wire's desktop apps, the tool *hardening-check* 5.2 has been used in addition to manual code inspection. Further, the Rust client coax and Pidgin with the Wire plugin PWR have been analyzed as well. The results can be seen in table 6.9. In a nutshell, Signal and Wire's desktop electron apps and pidgin with the PWR plugin make use of all four hardening features on Linux. Only the experimental Wire client Coax has not enabled stack protection. The desktop apps of Wire and Signal only recently enabled support for PIE, read-only relocation and immediate binding. This was caused by the upstream Electron project.

Feature	Signal Desktop	Wire Desktop	Coax	Pidgin + PWR
Analyzed app version	1.29.3	3.12.2916	Git	2.12.0 + Git
PIE (ASLR)	✓	✓	✓	✓
Stack protected	✓	✓	✗	✓
Read-only relocations	✓	✓	✓	✓
Immediate binding	✓	✓	✓	✓

Table 6.9: Desktop apps - security hardening features on Linux

6.4 Metadata Analysis of Wire

This section enumerates all metadata which get accumulated in Wire. This metadata can be differentiated into two distinct types of metadata: First, temporary metadata which is sent within the network traffic or stored in the database but gets deleted after a strictly defined period of time. The second type is non-temporary metadata i.e. data which is stored in the database without intended deletion.

Temporary metadata (i.e. data sent within network traffic). There is some amount of metadata which is not end-to-end encrypted with Proteus between clients, but only encrypted via TLS between client and server. Additionally, inside the private network of Wire's backend, this metadata is sent in plain text because the TLS encryption is terminated at the reverse proxy nginx as described in section 3.4. This metadata is needed by the backend to operate i.e. deliver messages to the right clients. An adversary who successfully breaks the transport encryption TLS or gains access to the internal private network of Wire's backend (i.e. service providers) could read this metadata. This section enumerates this metadata.

Typical metadata for messaging services contains information on who sends messages to whom and when. To deliver the end-to-end encrypted messages to the right recipients,

Wire's messaging protocol needs some metadata to operate i.e a unique message ID, UUIDs of the sending user and all recipient users as well as all involved devices from the sender and the recipients.

Wire has no support for sealed sender like Signal which would improve sender anonymity of messages. The end-to-end encrypted messages, together with the unencrypted metadata, are temporarily stored on the server until the message is forwarded to all devices but not longer than 4 weeks. Further, each notification 3.2.4 contains metadata too.

Another example for temporary metadata in Wire is, who searches whom and when by using Wire's people search feature. Or when is a user online and when do they write messages. Furthermore, the underlying protocol (HTTPS) also produces metadata i.e. HTTP user agent, operating system, accepted languages and the IP address of the device or devices if the user uses multiple devices.

Non-temporary metadata (i.e. data stored in server database). Wire permanently stores some metadata in plain text in its backend databases. Therefore, any adversary (i.e. malicious service providers or rogue administrators) who gains access to this database can read this non-temporary metadata.

For each user Wire stores: a unique user ID, an email address and/or phone number, the profile name, a user handle (also known as username), an optional profile picture, a profile color (mapped as accent_id) and the locale. In addition to the email address or phone number SHA256 hashes of those are also stored to support the address book upload feature.

For each user device the following metadata is stored: a device id, the device class (i.e. desktop or phone), the device label (OS for web app: Linux and the phone model for phones), and model (i.e. mobile phone vendor and phone model, or the browser name for the web app). Further, the time and geolocation data (latitude and longitude) at the time of device creation is stored as well. The geolocation data originates from a MaxMind's GeoIP database [2] which is derived from the IP address of the device at the time of creation.

In contrast to the mobile apps, the web app additionally stores some metadata on server-side via the API endpoint `/properties/webapp` which contains web app settings i.e. privacy settings: improve wire (yes/no), report errors (yes/no) and send link previews (yes/no).

Further, Wire also stores metadata about user connections i.e. who sends messages to whom. This means, once a connection between two users has been established it cannot be deleted and will be stored forever. Since there is no API for DELETE, a user can only be blocked via PUT for connection updates. These metadata encompass the following fields: connection status (accepted, blocked, pending, ignored, sent or cancelled), timestamp of last update, and invite message (automatically generated by the apps with

[2] `https://hackage.haskell.org/package/geoip2`, Accessed: 22-February-2020

the names of both profiles i.e. Hi Alice, Let's connect on Wire. Bob). Although, this invite message is currently not in use and only gets sent in the background.

As part of conversation metadata Wire stores the following fields: conversation creator, the boolean field deleted, conversation name, the type of conversation (i.e. 1:1 or group conversation), conversation members and the conversation role of each member. Once again, all this metadata is stored in plain text, which could leak important information about the conversation. And to make matters worse, once created, a conversation cannot be deleted because there is no API endpoint for deleting a conversation and thus leaving all the metadata on the server indefinitely.

Another type of metadata which is stored on server-side are the symmetric encryption keys for encrypted notification delivery. To this metadata also includes the user ID, client ID, encryption key, and MAC key (message authentication code). These keys are used for encrypting the temporary metadata of notifications. However, this additional encryption is only used for push notification services. On Wire's REST API and WebSocket connection such notifications are only transport encrypted with TLS.

Metadata exfiltration with stolen credentials. An attacker can gain access to metadata by getting hold of a user's credentials or the long-lived user token which is stored as a browser cookie in case of the web app. This can result in exfiltration of a user's entire non-temporary metadata as well as some temporary metadata from the server via the REST API. Possible attack scenarios would be i.e the victim user just closed the browser tab instead of logging out and the attacker has access to the browser afterwards. Or another scenario would be in a corporate network where the network operator does TLS deep packet inspection by using mitm attacks. Because the metadata is not end-to-end encrypted, the attacker does not need access to the actual private key material of the devices.

User enumeration and partial metadata exfiltration. The people search feature 3.2.5 can be abused to enumerate most Wire users. Combined with the user API `GET /users` some of the above non-temporary metadata could be accessed by a malicious user revealing i.e. profile name, profile picture and profile color for any given user ID.

6.4.1 Unused API Feature: Opt-Out From People-Search

During the course of the security analysis of Wire I found some API features which are currently not in use by Wire clients.

One of these unused API features, or rather backend features, allows users to opt out of being included in search results of the people-search. By using this opt-out a user cannot be found by neither profile name nor partial username. This feature was accessed by the following API call:

Listing 6.8: API call: Opt-out from people-search

```
1 PUT /self/searchable {'searchable': false}
```

This API call updates the Boolean field *searchable* of the corresponding user in the brig table *user* of the Cassandra database to the value *false*. Furthermore, the search index of the search engine Elasticsearch is updated by removing the source data (username, profile name and normalized name) of the corresponding user from the index.

This ensures that the user cannot be found by the search API `GET /search/contacts`, which obtains its answers by accessing Elasticsearch. However, the user handle API `GET /users/handles/` is not affected by this opt-out feature, as it returns the corresponding user ID if the handle (username) exists. This user ID can subsequently be used to retrieve information about the user by accessing the user API `GET /users` which returns the profile name, profile picture and profile color for any given user ID. Wire clients use both APIs to search for people as described in section 3.2.5.

To add people who have chosen to opt-out of people-search, it is necessary to search with the exact username, otherwise the user cannot be found.

This opt-out feature can increase the privacy of users because they cannot be easily found via people-search and thus the public profile information (non-temporary metadata) i.e. username, profile name, profile picture and profile color, can only be accessed by people who know or guess the correct username. Additionally, an automated enumeration of all Wire users by using the people-search function is therefore less straightforward.

CHAPTER 7

Discussion

In this chapter I discuss the results of the security and privacy analysis of Wire. It is split into the sections: Trust establishment 7.1, production environment 7.2, app security 7.3 and metadata 7.4.

7.1 Trust Establishment

It can be cumbersome or nearly impossible to establish trust in Wire. Users have to verify each single device of all their contacts. Which can be hundreds or thousands of devices.

Multiple devices of an account (i.e. own devices) should always be verified to reduce the risk of man-in-the-middle attacks as seen in the previous chapter. This should be straight forward because first, the user should have easy access to the other devices and second there can only be up to 8 devices which should not be too time consuming to verify.

Nevertheless, it is highly recommended to verify as many conversations as possible. Although, verifying other people's devices does not scale for big groups with many members, especially since Wire bumped the limit of maximum people per group from 128 to 300 members [1] and later bumped the limit again to 500 members [2]. In large groups like this, it is highly unlikely that all members verify each other's devices. Therefore, Wire should invest in the improvement of trust establishment i.e. introduction of cross-signing where a user cross-signs all their own devices. Then Wire users would need to verify only one device and thus trust transitively all other devices of this user and won't need

[1] https://medium.com/wire-news/wire-for-web-2018-08-06-46fcaaf131f, Accessed: 22-February-2020

[2] https://medium.com/wire-news/wire-for-web-2019-07-47a13a06ea7c, Accessed: 22-February-2020

to verify each device individually. This would significantly reduce the large number of comparisons.

Additionally, comparing multiple strings of hexadecimal digits is not an easy task for the human eye and brain. Thus, Wire should improve the usability of device verification to make the verification easier and less error-prone e.g. by introducing QR-codes as an additional option to compare key-fingerprints. That way, users could simply scan the QR-code of the other device and the software compares if both key-fingerprints match.

Another improvement would be to replace the hexadecimal representation of the key-fingerprint with a better textual representation (e.g. numeric) as proposed by Dechand et al. [9]. They have performed a user study with different textual representations of key-fingerprints to benchmark the comparison speed and accuracy. They concluded that hexadecimal representation is not the best textual representation for key-fingerprint comparisons.

As a general advice users should remove old or unused devices from their account to reduce the amount of possible key-fingerprint comparisons. As a side effect this also reduces the network traffic and computing power because fewer messages have to be encrypted and sent to the server. This also implies less energy consumption and thus could potentially increase the battery run-time on mobile devices.

A further downside of Wire in regard to trust establishment security is that Wire shows a TOFU warning message (whenever a contacts' key changes) only after a prior key verification. But if the user has not managed to compare the key yet, it does not show any notification that the cryptographic key of the communication partner has changed. Therefore, Wire is prone to man-in-the-middle attacks by i.e. malicious service providers. It would be advantageous to have at least a setting for advanced users to enable TOFU for the devices on contact establishment. Ideally this should be enabled by default, but first Wire needs to support cross-signing to significantly decrease the number of new device notifications.

Since the user friendliness of trust establishment is far from perfect, users likely won't verify the devices of all contacts. Thus, all users who have not verified their device are potentially vulnerable to man-in-the-middle attacks by malicious service providers. Wire should improve this to properly protect users against man-in-the-middle attacks.

Signal performs much better in trust establishment than Wire. It shows a TOFU warning for changed master devices without prior key verification as suggested for Wire and provides a QR-code for key verifications for a much improved usability. Additional used devices are transitively trusted, thus no additional verification is needed for new devices. Furthermore, Signal uses decimal numbers instead of hexadecimal numbers as representation for the key fingerprints which is also more user friendly. So, with a much better usability of Signal's trust establishment the probability that users compare their key fingerprints is much higher compared to Wire. Which in turn further improves the trust establishment of Signal.

A related problem to trust establishment is that Signal uses phone numbers as identifiers. This comes with a great drawback: if the phone number changes, a Signal user has to create a new account. Which implicates further tasks i.e. ask each group conversation to be invited again or in case the old phone number can still be used, add the new phone number to each group conversation manually. Further, caused by the new account, all verified contacts have to be verified again to establish trust. On Wire this is a non-issue. A Wire user simply changes the phone number in the profile settings or uses an email instead of a phone number for the account. The reason for this is that Wire does not use the phone number or email as identifier, instead it uses an UUID as identifier and stores the established contact connections on the server. In addition usernames are used for contact lookup on Wire.

7.2 Production Environment

Wire promises to host everything within the EU but this is not the case for its support website as it is hosted by Zendesk with a server location in the US. Also, the main site *wire.com* contains third-party requests to a website hosted in US. Wire should move the hosting of its site to the EU to comply with its own promises. Additionally, the security headers could be improved for the support site. Especially the HSTS header which currently lasts 3 days only. This is much too short to be useful for mitigating SSL-stripping attacks. Its recommended use is at least 6 months.

Overall Wire has the potential for improvements regarding their production environments. Nevertheless, compared with Signal, Wire performs better in its production environment.

7.3 App Security

The numbers of prekeys should be increased on the Web app and especially on the desktop app to ensure good forward secrecy support. Particularly if the clients are offline for longer periods of time, the pool of available prekeys should be large enough.

Further, Wire should add the missing encryption support for backups on the web app. Thus, the desktop app would inherit the encryption support too.

Wire does well, requiring only a minimal amount of Android permissions compared to Signal which requires many.

The Android app signing algorithm *SHA1withRSA* which is used by Wire and Signal must be replaced with a secure signing algorithm i.e. *SHA256withRSA* which makes use of cryptographic hash function *SHA256* instead of *SHA1*. The hash function *SHA1* is not secure any longer and must not be used for cryptographic operations i.e. digital signatures since there exist practical attacks on this hash function [54] [27].

Wire should also provide the Android app via F-Droid, an app store that exclusively provides open source apps.

Nowadays, Wire uses all state-of-the-art security hardening features on Linux for its desktop app which has improved enormously in the past.

7.4 Metadata

Several unnecessary metadata get stored on the server i.e. geolocation data of device registration, conversation creator and more. An example from the official support page:

> "I've accidentally connected with someone I don't know. What should I do?" [3]

Once a connection between two users has been established, it cannot be deleted and will be stored forever on Wire's backend. In such a case a user can only be blocked. There is no API endpoint for deleting a connection. Wire should add such an API endpoint to allow users to delete their unneeded metadata. All invite messages of connections should be removed as they are completely unused and additionally store the profile names at the time of invitation.

The same drawback for connections applies also to conversations. Once created, a conversation cannot be deleted anymore and there is no API endpoint for deleting a conversation. A possible workaround for users to reduce this metadata on the server is to rename the group name into something meaningless and all members leave the group. This way, only the group creator and the meaningless name remain on the server. There is also no reason to store the conversation creator as this person does not have any special permissions.

Geolocation data is also considered needless metadata. It was originally shown on the "own devices" page of the apps, but this function has since been removed. Thus, the geolocation metadata is unused and should be removed from the database.

Profile pictures are stored unencrypted on Wire's servers, while Signal saves profile pictures and profile names on client-side which only get sent to other clients end-to-end encrypted via opt-in.

Some metadata would be difficult to remove i.e. user connections and conversation members, because of product design decisions as stated in [60]. Federation would help to split the metadata between different stakeholders and jurisdictions.

Either way, Wire should learn from Signal and reduce the amount of temporary and non-temporary metadata. They should start with the low hanging fruits i.e. delete unused metadata and provide deletion APIs for connections and conversations.

[3] https://support.wire.com/hc/en-us/articles/203122450-I-ve-accidentally-connected-with-someone-I-don-t-know-What-should-I-do-, Accessed: 22-February-2020

CHAPTER 8

Conclusion

In general, secure messaging has greatly improved since Signal introduced the Double Ratcheting algorithm for end-to-end encryption. Still, many messaging services have issues concerning security and privacy. This thesis shows how to evaluate secure messaging services regarding trust establishment, conversation security and transport privacy. For this evaluation, the secure messaging service Wire was chosen.

Overall, Wire has a good security level, thanks to the many security audits, but it has room for several improvements. Proper trust establishment is one of the issue most lacking in regard to security. It is nearly impossible to verify each single device of all conversation partners. There are also several usability issues when it comes to trust establishment which should be improved. Further, Wire does not perform very well in transport privacy in general and compared to Signal in particular. Certain metadata could be removed without losing features and other metadata could be mitigated by using the same techniques as Signal uses. Consequently, Wire should reduce metadata where possible and support removing connections and conversations for more privacy. More control over metadata can be achieved by self-hosting a Wire server in a separate AWS account which is already possible as of now. In this thesis it was shown that it is possible to self-host a Wire server with a reduced feature set, even without AWS dependencies which were a prerequisite for the security evaluation of the Wire protocol. With some further developmental effort, more features could be supported in a self-hosted, non AWS environment. Currently, a major downside of self-hosting is that a federation with other Wire servers, especially the official Wire server, is not supported. It will become more feasible to host own Wire servers once the federation is fully supported. For better understanding of how a Wire client works and what is essentially required for end-to-end encrypted messaging, a Pidgin plugin, which implements the most essential features of the Wire protocol, has been developed. Finally, it was also shown in this thesis where Wire could improve security for its apps and production environment.

8.1 Future Work

Future work should include, among other things, analyses of features not touched upon in this thesis such as (1) the Wire Pro features i.e. the guest feature also known as ephemeral users, or the bot / integration API (2) the security of the iOS client and (3) the two services *Proxy* and *Cargohold*. Additionally, the experimental Wire client Coax should be updated and extended with more features. The same can be done with the Pidgin plugin PWR. There are also several interesting security features for Wire which are currently in development and could be analyzed in future: Messaging Layer Security (MLS) [67], post-quantum resistance [63] and federation.

Bibliography

[1] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol (SRTP). RFC 3711 (Proposed Standard), March 2004. Updated by RFCs 5506, 6904.

[2] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. *The Security Impact of a New Cryptographic Library*, pages 159–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[3] J. Bian, R. Seker, and U. Topaloglu. Off-the-record instant messaging for group conversation. In *2007 IEEE International Conference on Information Reuse and Integration*, pages 79–84, Aug 2007.

[4] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, pages 77–84, New York, NY, USA, 2004. ACM.

[5] C. Bormann and P. Hoffman. Concise Binary Object Representation (CBOR). RFC 7049 (Proposed Standard), October 2013.

[6] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880 (Proposed Standard), November 2007. Updated by RFC 5581.

[7] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. OpenPGP Message Format. RFC 2440 (Proposed Standard), November 1998. Obsoleted by RFC 4880.

[8] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. Cryptology ePrint Archive, Report 2016/1013, 2016. `http://eprint.iacr.org/2016/1013`.

[9] Sergej Dechand, Dominik Schürmann, Karoline Busse, Yasemin Acar, Sascha Fahl, and Matthew Smith. An empirical study of textual key-fingerprint representations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 193–208, Austin, TX, 2016. USENIX Association.

[10] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919.

[11] C. Evans, C. Palmer, and R. Sleevi. Public Key Pinning Extension for HTTP. RFC 7469 (Proposed Standard), April 2015.

[12] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), December 2011. Updated by RFC 7936.

[13] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Joerg Schwenk, and Thorsten Holz. How Secure is TextSecure? Cryptology ePrint Archive, Report 2014/904, 2014. http://eprint.iacr.org/2014/904.

[14] Ian Goldberg, Berkant Ustaoğlu, Matthew D Van Gundy, and Hao Chen. Multi-party off-the-record messaging. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 358–368. ACM, 2009.

[15] P. Hallam-Baker and R. Stradling. DNS Certification Authority Authorization (CAA) Resource Record. RFC 6844 (Proposed Standard), January 2013.

[16] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard), October 2012.

[17] J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). RFC 6797 (Proposed Standard), November 2012.

[18] P. Hoffman. SMTP Service Extension for Secure SMTP over Transport Layer Security. RFC 3207 (Proposed Standard), February 2002. Updated by RFC 7817.

[19] M. Jones and D. Hardt. The OAuth 2.0 Authorization Framework: Bearer Token Usage. RFC 6750 (Proposed Standard), October 2012.

[20] H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869 (Informational), May 2010.

[21] Kudelski and X41 D-Sec. Security Review – Phase 1 for Wire Swiss GmbH. https://wire-docs.wire.com/download/Wire+Audit+Report.pdf, February 2017.

[22] Kudelski and X41 D-Sec. Security Review – Phase 2 – Android Client for Wire Swiss GmbH. https://www.x41-dsec.de/reports/X41-Kudelski-Wire-Security-Review-Android.pdf, March 2018.

[23] Kudelski and X41 D-Sec. Security Review – Phase 2 – iOS Client for Wire Swiss GmbH. https://www.x41-dsec.de/reports/X41-Kudelski-Wire-Security-Review-iOS.pdf, March 2018.

[24] Kudelski and X41 D-Sec. Security Review – Phase 2 – Web, Calling for Wire Swiss GmbH. https://www.x41-dsec.de/reports/X41-Kudelski-Wire-Security-Review-Web-Calling.pdf, March 2018.

[25] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962 (Experimental), June 2013.

[26] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), July 2005.

[27] Gaëtan Leurent and Thomas Peyrin. SHA-1 is a Shambles - First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust. Cryptology ePrint Archive, Report 2020/014, 2020. https://eprint.iacr.org/2020/014.

[28] Hong Liu, Eugene Y Vasserman, and Nicholas Hopper. Improved group off-the-record messaging. In *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, pages 249–254. ACM, 2013.

[29] Joshua Lund. Technology preview: Sealed sender for signal. https://signal.org/blog/sealed-sender/, October 2018. [Online; accessed: 22-February-2020].

[30] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766 (Proposed Standard), April 2010. Updated by RFC 8155.

[31] Moxie Marlinspike. Advanced cryptographic ratcheting. https://signal.org/blog/advanced-ratcheting/, November 2013. [Online; accessed: 22-February-2020].

[32] Moxie Marlinspike. Forward secrecy for asynchronous messages. https://signal.org/blog/asynchronous-security/, August 2013. [Online; accessed: 22-February-2020].

[33] Moxie Marlinspike. Private group messaging. https://signal.org/blog/private-groups/, May 2014. [Online; accessed: 22-February-2020].

[34] Moxie Marlinspike. Facebook messenger deploys signal protocol for end to end encryption. https://signal.org/blog/facebook-messenger/, July 2016. [Online; accessed: 22-February-2020].

[35] Moxie Marlinspike. Open whisper systems partners with google on end-to-end encryption for allo. https://signal.org/blog/allo/, May 2016. [Online; accessed: 22-February-2020].

[36] Moxie Marlinspike. Reflections: The ecosystem is moving. https://signal.org/blog/the-ecosystem-is-moving/, May 2016. [Online; accessed: 22-February-2020].

[37] Moxie Marlinspike. Whatsapp's signal protocol integration is now complete. https://signal.org/blog/whatsapp-complete/, April 2016. [Online; accessed: 22-February-2020].

[38] Moxie Marlinspike. Signal partners with microsoft to bring end-to-end encryption to skype. https://signal.org/blog/skype-partnership/, January 2017. [Online; accessed: 22-February-2020].

[39] D. McGrew and E. Rescorla. Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP). RFC 5764 (Proposed Standard), May 2010. Updated by RFC 7983.

[40] C. Newman. Using TLS with IMAP, POP3 and ACAP. RFC 2595 (Proposed Standard), June 1999. Updated by RFCs 4616, 7817.

[41] C. Percival and S. Josefsson. The scrypt Password-Based Key Derivation Function. RFC 7914 (Informational), August 2016.

[42] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/, November 2016. [Online; accessed: 22-February-2020].

[43] Damian Poddebniak, Jens Müller, Christian Dresen, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. USENIX Association.

[44] B. Ramsdell. S/MIME Version 3 Message Specification. RFC 2633 (Proposed Standard), June 1999. Obsoleted by RFC 3851.

[45] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347 (Proposed Standard), April 2006. Obsoleted by RFC 6347, updated by RFCs 5746, 7507.

[46] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389 (Proposed Standard), October 2008. Updated by RFC 7350.

[47] Christoph Rottermanner, Peter Kieseberg, Markus Huber, Martin Schmiedecker, and Sebastian Schrittwieser. Privacy and data protection in smartphone messengers. In *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '15, pages 83:1–83:10, New York, NY, USA, 2015. ACM.

[48] Paul Rösler, Christian Mainka, and Jörg Schwenk. More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema. Cryptology ePrint Archive, Report 2017/713, 2017. https://eprint.iacr.org/2017/713.

[49] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 3920 (Proposed Standard), October 2004. Obsoleted by RFC 6120, updated by RFC 6122.

[50] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. RFC 3921 (Proposed Standard), October 2004. Obsoleted by RFC 6121.

[51] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard), March 2011. Updated by RFC 7590.

[52] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. RFC 6121 (Proposed Standard), March 2011.

[53] Peter Saint-Andre and Kevin Smith. XEP-0163: Personal Eventing Protocol. `https://xmpp.org/extensions/xep-0163.html`, 2010. [Online; accessed: 22-February-2020].

[54] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 570–596, Cham, 2017. Springer International Publishing.

[55] Andreas Straub. XEP-0384: OMEMO Multi-End Message and Object Encryption. `https://xmpp.org/extensions/xep-0384.html`, 2016. [Online; accessed: 22-February-2020].

[56] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith. Sok: Secure messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249, May 2015.

[57] Wire Swiss GmbH. You can now build your own Wire client. `https://wire.com/en/blog/open-source-code/`, July 2016. [Online; accessed: 22-February-2020].

[58] Wire Swiss GmbH. Call security – constant bit rate encoding and improving WebRTC. `https://wire.com/en/blog/constant-bit-rate-calls-improving-webrtc/`, March 2017. [Online; accessed: 22-February-2020].

[59] Wire Swiss GmbH. Open sourcing Wire server code. `https://wire.com/en/blog/server-code-open-source-2017/`, April 2017. [Online; accessed: 22-February-2020].

[60] Wire Swiss GmbH. Product design decisions for secure messengers. `https://wire.com/en/blog/secure-messenger-product-design-decisions/`, May 2017. [Online; accessed: 22-February-2020].

[61] Wire Swiss GmbH. Wire server code now 100% open source – the journey continues. `https://medium.com/@wireapp/wire-server-code-now-100-open-source-the-journey-continues-88e24164309c`, September 2017. [Online; accessed: 22-February-2020].

[62] Wire Swiss GmbH. Wire's independent security review. https://wire.com/en/blog/independent-security-audit-2017/, February 2017. [Online; accessed: 22-February-2020].

[63] Wire Swiss GmbH. Wire and post-quantum resistance. https://wire.com/en/blog/post-quantum-resistance-wire/, July 2018. [Online; accessed: 22-February-2020].

[64] Wire Swiss GmbH. Wire application-level security audits. https://wire.com/en/blog/application-level-security-audits/, March 2018. [Online; accessed: 22-February-2020].

[65] Wire Swiss GmbH. Wire privacy whitepaper. https://wire-docs.wire.com/download/Wire+Privacy+Whitepaper.pdf, August 2018.

[66] Wire Swiss GmbH. Wire security whitepaper. https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf, August 2018.

[67] Wire Swiss GmbH. MLS - The future of collaboration? https://wire.com/en/blog/mls-future-of-collaboration/, December 2019. [Online; accessed: 22-February-2020].