Masters Thesis

Towards Secure Key Verification Mechanisms in End-to-End Encrypted Multi-Device Instant Messaging

Paul Moritz Schaub

Matr. 406218

Supervisor: Prof. Dr. Sergei Gorlatch
Second Supervisor: Prof. Dr.-Ing. Thomas Hupperich

Institute of Computer Science, University of Münster, Germany

May 17, 2021

Abstract

In times of surveillance capitalism proper implemented encryption is an essential tool for digital self-defense and self-determination. Encryption guarantees that messages can only be read by a targeted group of recipients while unauthorized outsiders cannot access their content.

The most effective class of encryption mechanisms is called End-to-End Encryption, short *E2EE*. End-to-end encrypted messages are being enciphered with the recipients key before leaving the senders device and can only be decrypted by the intended recipients. This is guaranteed by the fact that decryption keys of the involved parties are kept only on the users devices. Intermediate parties like servers that are used for message delivery cannot gain access to, and/or tamper with, the messages content.

However, security of the communication relies heavily on authentication of identities and keys. In other words; Confidentiality can only be guaranteed if the sender is certain about which keys are being used by the recipient of the message.

To illustrate this problem, the use case of end-to-end encrypted *instant messaging* or simply *chat* will hereafter be considered. In the field of encrypted instant messengers, *Signal*[1] stands out as a prominent contender. Its developers successfully made end-to-end encryption usable and transparent enough to be suitable for the mass. However, since the Signal ecosystem is centralized and isolated and excludes developers of third-party clients[2], this work will instead focus on the open, decentralized eXtensible Messaging and Presence Protocol (XMPP) network.

Today, users often use multiple devices and switch between them, expecting a seamless experience. This is especially true for instant messaging. As a result, each user potentially ends up managing multiple encryption keys. This presents them with the challenge to keep track of which keys are in use not only for themselves but also for their contacts. The complexity of this problem becomes clear when end-to-end encrypted group chats are considered. In a group chat with m users with an average of n keys per user, the total number of involved keys amounts to $m \times n$. The total number of verifications that traditionally have to be carried out by hand amounts to $n \times m - 1$ per user. If there is a total of x devices involved in a conversations, the number of manual fingerprint verifications required is $x(x - 1)$. It should be apparent that this system does not scale in the long run. Many users are quickly scared off by the huge effort they are required to make and neglect key verification completely. This leads to a situation where it is easy for an attacker to convince their victim of the legitimacy of a rogue key, thereby breaking the security guarantees of the encryption. This is known as a Man in the Middle Attack. For this reason it is worthwhile coming up with a simplified mechanic for key verification in scenarios where each user owns multiple keys.

This work will present a short summary of how cryptography works and will explain their functionality by example.

[1] see `https://signal.org`

[2] see `https://github.com/LibreSignal/LibreSignal/issues/37#issuecomment-217211165`

The different security guarantees and properties of cryptographic methods will be presented and surveyed. At the same course the use of a cryptographic fingerprint will be explained and its relevancy to encryption protocols will be illustrated.

Furthermore the properties of different network topologies (centralized/federated messaging service, P2P messaging) will be discussed and some of the consequences that arise from them when it comes to attack scenarios will be identified.

Last but no least this work will shine some light onto existing methods for simplified management of key material, discuss their advantages and shortcomings and will investigate how those systems can be improved. For this purpose a method for authenticating device keys using account-wide identity keys will be formalized and a prototype will be implemented and integrated into an existing messaging application (Mercury-IM).

To model the use case of Instant Messaging, this work will resort to the XMPP protocol (*eXtensible Messaging and Presence Protocol*) as it is an open and extensible protocol that can be used for chat. Furthermore the *XMPP* protocol can be used to depict both centralized as well as decentralized and federated systems.

Contents

List of Figures

List of Tables

Listings

Acronyms

AES Advanced Encryption Standard. xi, 2

ASCII American Standard Code for Information Interchange. xi, 33

ATM Automatic Trust Messages. xi, 20, 21

BTBV Blind Trust Before Verification. xi, 19

C2S Client-to-Server communication. xi, 10

CLI Command Line Interface. xi, 50

DH Diffie-Hellman. xi, xiii, 15

DOS Denial-of-Service. xi, 18

DSA Digital Signature Algorithm. xi, xiii, 15

ECDH Elliptic Curve Diffie-Hellman. xi, 15

ECDSA Elliptic Curve Digital Signature Algorithm. xi, 15

EdDSA Edwards-curve Digital Signature Algorithm. xi, 15

IKey Protocol Identity Key Protocol. xi, 29

JID Jabber-ID / XMPP address. xi, 42

MitM Man-in-the-Middle. xi, 1

MSC Matric Spec Change. xi, 21

MVVM Model-View-ViewModel. xi, 41, 50

OMEMO OMEMO Multi End Message and Object encryption. xi, 14, 15, 30

ORM Object Relational Mapping. xi, 40

OX OpenPGP for XMPP. xi, xiii, 14, 15, 39, 41

OX-IM OpenPGP for XMPP - Instant Messaging. xi, 15

P2P Peer-to-Peer communication. xi, 9, 11

S2S Server-to-Server communication. xi, 10

SSH Secure SHell. xi, 4

SSL Secure Socket Layer - Encryption protocol. xi, 6

TLS Transport Layer Security - Encryption protocol. xi, 6

TOFU Trust On First Use. xi, 19

XEP XMPP Extension Protocol. xi, 14

XMPP eXtensible Messaging and Presence Protocol. iii, xi, xiii, 14, 41, 42

Glossary

Asymmetric Encryption Also known as Public-Key-Encryption. Cryptographic system where the key that is used for decryption of a message is different from the key that is used to encrypt the message in the first place. The key used to encrypt messages and verify signatures is called public key, while the key to decrypt messages and create signatures is called private key. xi, xvi, 2

Blind Trust before Verification Trust model where keys of a contact are blindly trusted until the user decides to upgrade the security by manually verifying at least one key. xi, xiii, 19

Break-in Recovery Also called *future secrecy* or *self-healing*. Property of some encryption schemes that prevents an attacker that compromised the encryption key of a message from being able to decrypt future messages. The Signal Protocol (formerly known as Axolotl Protocol) possessed both forward- and future secrecy. xi

Certificate Authority (CA) A certificate authority is an entity which is capable of issuing certifications for a certain domain. For example letsencrypt[3] is a certificate authority which is capable of issuing TLS certifications for website owners. xi, 28

Denial-of-Service-Attack Attack method that has the goal to disturb the availability of a service for one or multiple users. Usually executed by flooding a service with requests that consume most of the services resources so that it cannot respond to proper requests anymore. xi, 9

Downgrade Attack Attack on a cryprographic protocol, where the attacker forces the victims to fall back to weaker encryption than would otherwise be possible. This can be done eg. by making a party believe that their peer only supports weak algorithms. xi, 6

End-to-End Encryption Encryption technique where messages are encrypted on the senders device and can only be decrypted on the device of the intended recipient. iii, xi

Fingerprint Short letter- and number sequence which is used as identificator of a cryptographic public key. Usually the fingerprint is calculated by applying a cryptographic hash function on the public key. xi, xvi, 19

Forward Secrecy Property of some encryption schemes that prevents an attacker that compromises the encryption key of a message from also decrypting previously sent messages. xi, 18

Full Disk Encryption Encryption of the whole disk or partition of a device. xi, 17

Hash Function Deterministic mathematical function which calculates a short digest (Hash Value) from a message. Cryptographic hash functions are trap door functions in the

[3] see `https://letsencrypt.org/`

sense that it is very easy to calculate the hash of a message, but it is mostly impossible to reverse the function i.e. draw conclusions about the message from the hash value alone. xi, xvi, 4

Hash Value Pseudo-random sequence of letters and numbers of fixed length. Is calculated by a hash function. xi, xv

Hybrid Encryption Mix of symmetric encryption and asymmetric encryption. Typically the message content is encrypted using a random symmetric key, which is afterwards encrypted for each recipient asymmetrically. Most encryption programs that support asymmetric encryption for file/message protection are utilizing hybrid encryption methods to improve performance. xi, 3

Man in the Middle Attack Attack where the attacker impersonates entities that take part in a usually encrypted communication protocol, secretly relaying and re-encrypting their messages. iii, xi, xiii, 1, 6

OMEMO End-to-end encryption specification (XEP-0384 [35]) for XMPP based on the double ratchet algorithm. xi, xiii, 14

Preimage Attack Attack where the attacker calculates a key that shares the fingerprint of the victims key. A *partial* preimage attack can be carried out if the attacker found a key whose fingerprint does not fully match that of the victims key, but shares enough similarity to be mistaken during hastily manual comparison by eye. xi, 4, 5, 19

Private Key Part of an asymmetric key pair. The private key has to be kept secret by its owner. It is used to decrypt messages and create cryptographic signatures. xi, xv

Public Key Part of an asymmetric key pair. The public key is meant to be published and will be used for encryption and verification of signatures. xi, xv

RSA Asymmetric cryptosystem named after its inventors Rivest, Shamir and Adelman. xi, 3, 4, 6

Shadow IT IT-infrastructure that is used by a company but which is not under direct control of it. Examples are cloud systems or other externally hosted services. xi, 8

Stanza Top-level XMPP stream element; A stanza is basically an XMPP network package and can either be a *message*, a *presence* or an *iq* element. xi, 39

Symmetric Encryption Cryptographic system where the same secret key is used for both encryption and decryption of a message. The use of a passphrase for file encryption is an example of symmetric encryption. xi, xvi, 2

TOFU Abbrevation for "Trust on First Use"; Also referred to as *Leap of Faith*[34]. A trust model where an entity is considered trusted upon first contact despite missing authentication. xi, xiii, 19

XEP XMPP Extension Protocol. Describes an addition to the base XMPP protocol mostly adding new functionality. xi, xiii, 20

XMPP eXtensible Messaging and Presence Protocol (RFC6120 [26]). Open standard for decentralized, federated communication. Mostly used for instant messaging applications. iii, xi, xiii, xvi, 14

CHAPTER 1

Introduction

End-to-end encryption has become a very important feature of chat applications. Especially users who are interested in data protection and privacy often see the availability of e2ee as a vital criterion when choosing which messenger to use. However, the manual verification of fingerprints, which is necessary to guarantee the effectiveness of the encryption is a burden and presents many users with a challenge. While encryption without authentication may protect the user against passive attacks where the attacker simply observes network packages, for example on one of the participants servers, more sophisticated active attacks however present a threat that can only be countered by actively authenticating keys by fingerprint verification [10, p. 2]. The fact that encrypted communication without authentication is vulnerable to Man-in-the-Middle (MitM) attacks and is therefore prone to the complete compromise of promised security properties requires the user to have some basic understanding of the meaning of fingerprints.

In their seminal paper *Why Johnny can't encrypt* [37, p. 4] Whitten and Tygar are describing this as the *unmotivated user property* (quote):

> **The unmotivated user property**
> Security is usually a secondary goal. People do not generally sit down at their computers wanting to manage their security; rather, they want to send email, browse web pages, or download software, and they want security in place to protect them while they do those things. It is easy for people to put off learning about security, or to optimistically assume that their security is working, while they focus on their primary goals. Designers of user interfaces for security should not assume that users will be motivated to read manuals or go looking for security controls that are designed to be unobtrusive. Furthermore, if security is too difficult or annoying, users may give up on it altogether.

As one can easily see, end-to-end encryption is heavily affected by this, as it requires key verification by the user. Dechand et al. showed, that on average a single human centric fingerprint verification takes between 8 and 13 seconds, depending on the used representation of the fingerprint [7]. We can therefore assume that for an unmotivated user, the chances that they give up on proper fingerprint comparison is higher, the more fingerprints need to be compared. A conversation with n participating devices potentially requires a total of $n(n-1)$ fingerprint comparisons (see fig. 5.1). Therefore it is important to design systems in a way which reduces potential for user mistakes and that make the use of end-to-end encryption as easy as possible.

1.1. Basics of cryptography

The primary goal of encryption is to alter a secret message (plain text) with the help of a key in such a way that only the intended recipient who is in possession of the key can recover the original message. For a user without knowledge of the key, the encrypted message is indistinguishable from randomness. At the same time, it is assumed that an attacker is able to observe and intercept any exchanged messages, modify the order of messages and their content and even craft new messages under false persona. This assumption is known as the Dolev-Yao attacker model [9].

When only intended recipients are able to access the secret message, we speak of *confidentiality*. Further goals of cryptography are *authenticity* (the recipient can be sure that the message was created by the sender) and *integrity* (the message has not been modified along the way).

1.1.1. Kerckhoffs Principle

In modern cryptography the security of an encryption system does not rely on the encryption algorithm being kept a secret. Instead the encryption algorithms need to be secure even though they have been subject of research for a long time. The security of the system solely relies on the used encryption keys being kept secret by the users. This is known as Kerckhoffs' principle [15]. If on the contrary a system is only secure as long as its inner functioning is not revealed, we disparagingly speak of *security by obscurity*. Systems that rely on this principle are to be avoided.

1.1.2. Symmetric and Asymmetric Encryption

In cryptography there are two major categories of encryption methods; *symmetric* and *asymmetric* encryption. In symmetric encryption systems the same key is used both for encryption as well as decryption of a message while in asymmetric encryption systems a key pair consisting of a private and public key are in use.

Symmetric encryption is best explained with the example of file encryption. To protect some data (eg. an important file) against unauthorized access, a symmetric key (eg. a passphrase) is used. With the help of a symmetric encryption algorithm the file is being encrypted, so that it can only be decrypted by someone who has knowledge of the passphrase. The important detail: The exact same key is used both for encryption as well as decryption.

One of the most prominent algorithms for symmetric encryption is the *Rijndael Algorithm* [6], better known under the name Advanced Encryption Standard (AES).

The principle of asymmetric encryption is less intuitive and demands an illustrative example. In their work *Why King George III can encrypt* [36] Tong et al. avoid overloading the term *key* for both parts of a key pair and instead explain asymmetric encryption with the help of keys and corresponding locks; The public key is therefore described as a public lock, which can only be opened with the help of the private key.

To encrypt a message the sender merely uses the public lock of the recipient to "seal" the message. Even though the sender is in possession of the recipients lock, they are now no

longer able to reopen the lock to regain access to the message. This operation requires the private key, which only the recipient of the message has access to.

Bai et al. took up on this terminology in their work *An Inconvenient Trust* [2] and applied it to investigate different mechanisms for key distribution.

The most important asymmetric encryption algorithm is RSA [25], named after its inventors Rivest, Shamir, and Adleman. RSA utilizes the problem of integer factorization of very large numbers.

Both symmetric and asymmetric encryption mechanisms are applied in practice. Oftentimes they are combined to form so called hybrid encryption methods. The reason is that symmetric algorithms are usually much faster than asymmetric algorithms.

Hybrid encryption is exemplified by OpenPGP [11], which uses symmetric keys to encrypt (potentially large) messages after which the symmetric key is encrypted to all recipients of the message using their asymmetric public keys. The result is an asymmetric encryption scheme which is adapted to encrypt huge amounts of data for multiple recipients without noteworthy negative impact on performance of ciphertext size.

For the course of this work we will focus on asymmetric and hybrid cryptography, respectively.

1.1.3. Point-to-Point, End-to-End Encryption

Historically, it was common practice for a long time to encrypt messages only during transportation from the device of the user to the server. This so called transport encryption or point-to-point encryption only protects messages from third parties (see fig. 1.1). However, the service operator is still able to access the message contents, as messages are being decrypted on the server. For a long time services such as Skype[1] only offered transport encryption. This type of encryption is easy to implement and has some advantages in respect to usability. For one thing, messages are accessible to the server in plaintext, so that functionality such as searchable message archives can be realized without much effort. For another thing the service operator can further process the content of messages and provide the user with targeted ads based on the topic of conversations (see GMail[2] as an example). Furthermore algorithms for transport encryption such as SSL/TLS have been subject of research for a long time and their application is well understood and well tested. Lastly the user does not have to think about key management, as the application solely has to trust the server certificate (in case of SSL/TLS). Therefore the encryption happens fully transparent to the user and does not require any kind of user-interaction. Additionally, users do not lose access to past messages should they lose or fully replace their devices.

Nevertheless, transport encryption alone does not offer much security with regard to privacy, as the service operator has full access to the users message archive and is able to modify messages during delivery.

Today, many messaging services offer end-to-end encryption, which is combined with transport encryption to guarantee that the message contents can only be accessed by the sender

[1] see `https://www.skype.com/`

[2] see `https://www.theguardian.com/technology/2014/apr/15/gmail-scans-all-emails-new-google-terms-clarify`

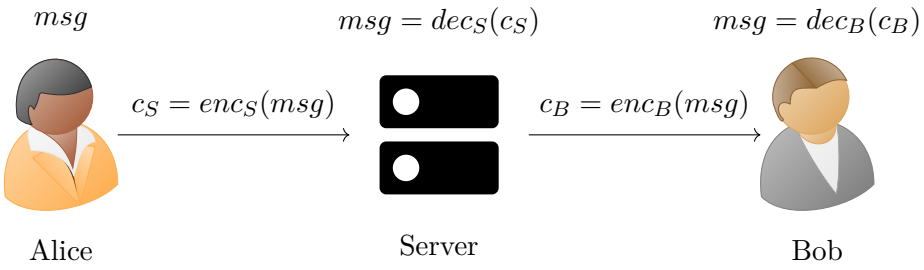

Figure 1.1.: Point-to-point encryption; while the message is always encrypted in transport, each hop - including the server - has access to the plain text. Here c_X denotes a message encrypted for recipient X, while enc_X, dec_X denote encryption and decryption operations using key material of entity X

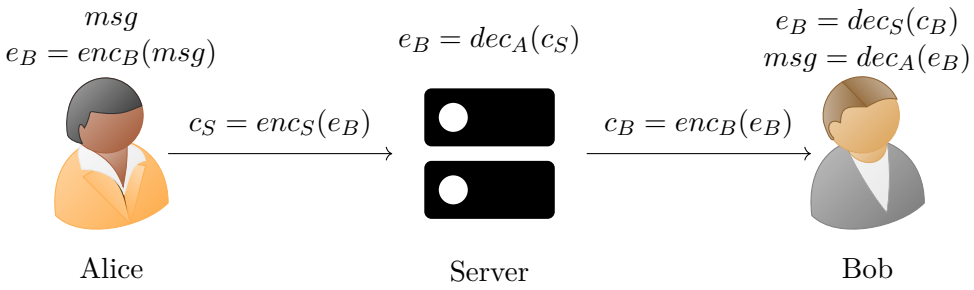

Figure 1.2.: End-to-End Encryption; Alice encrypts the message for Bob, so the server can no longer decrypt it. In this example, the message is additionally protected by transport encryption. Additionally to terminology from the diagram above, e_X denotes a message end-to-end encrypted for recipient X

and receivers (see fig. 1.2). For end-to-end encryption to be effective, users have to verify the fingerprint (see section 1.1.4) of their contacts encryption keys to prevent certain attacks (see section 1.2).

1.1.4. Fingerprints

Since in modern cryptography, keys itself can grow relatively long (at the time of writing the recommended length of RSA keys is around 4096 bits), keys are instead being represented by their fingerprint. Oftentimes a hash function is used to calculate a shorter representation of the key. From the perspective of mathematics a hash function is a function $f : X \to Y$ which maps from the set of keys X to the set Y of fingerprints. Generally the cardinality of X is greater than that of Y, so the function f is an injection but not a surjection. This means that theoretically it can happen that more than one different key share a common fingerprint. In that case one speaks of a hash collision which leads to a preimage attack [7].

In most systems such as *OpenPGP* [11] fingerprints are being used as identifiers of keys.

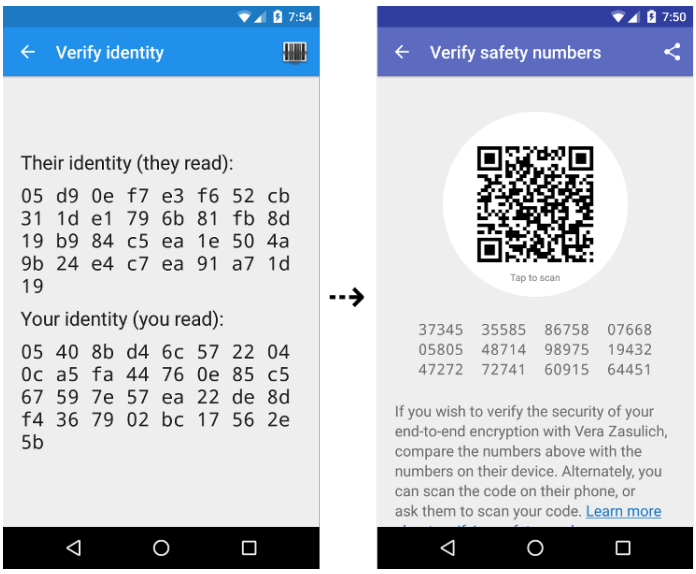

Figure 1.3.: Signal - Comparison of Fingerprints (old, left side), Comparison of Safety Numbers (new, right side). Source: `https://signal.org/blog/safety-number-updates/`

```
1  sec   rsa4096 2016-10-04 [SC]
2  7F9116FEA90A5983936C7CFAA027DB2F3E1E118A
3  uid         [ultimate] Paul Schaub <vanitasvitae@mailbox.org>
```

Listing 1.1: Example of an OpenPGP key fingerprint

During initial connection to a remote host, the system service SSH prompts the user to verify the validity of the remotes SSH key by presenting the user with its fingerprint.

Furthermore the system alerts the user, should this fingerprint change over time.

```
1  The authenticity of host 'AAA.BBB.CCC.DDD (AAA.BBB.CCC.DDD)' can't be
       established.
2  RSA key fingerprint is 01:23:56:67:89:ab:cd:ef:fe:dc:ba:98:76:54:32:10.
3  Are you sure you want to continue connecting (yes/no)?
```

Listing 1.2: Fingerprint verification in SSH

The concept of fingerprints is also known in instant messaging applications such as *Signal* and *WhatsApp*, even though they make use of the more user friendly term *Safety Number* and a slightly modified algorithm (see fig. 1.3). Instead of presenting the user with their own fingerprint and that of their contact, a single safety number if being generated for the pair of both devices. This combination of numbers and letters is the same on both ends, so that the comparison is simplified. The use of safety numbers particularly decreases the risk of mixing up both fingerprints during comparison. Unfortunately this current procedure is limited to systems with one key per user, so it is not applicable to chat systems where the devices of a user have multiple different keys.

For the efficiency of a fingerprint multiple factors come into play.

On one hand a fingerprint must describe the underlying key well enough to create a mapping which is as unique as possible. An attacker must not be able to generate another key which shares the same fingerprint as the first key (preimage attack). Otherwise the attacker could exploit this by pretending to be a contact to the user and presenting them with the malicious key. The victim would compare the fingerprint of the malicious key and possibly recognize it as the benign fingerprint of the valid contact, therefore erroneously trusting the attackers key. If this attack succeeds, the attacker would not even have to break the encryption, as they would possess a valid key for decryption.

On the other hand a fingerprint must be easily comparable, meaning differences between two similar fingerprints must quickly be apparent to the user and the risk of confusion must be minimized. As already mentioned, in addition to different fingerprint comparison methods, there are also different ways to represent fingerprints. While the hexadecimal system is often used as alphabet for fingerprints, other alphabets are conceivable as well. The messaging application Telegram[3] for example makes use of emoji to enable the user to quickly verify the session keys used during encrypted telephone calls (see fig. 1.4).

1.2. Attacks on End-to-End Encryption

There is a number of different kinds of attacks on end-to-end encryption. The goal of the attacker is always to compromise one or more principles of encryption - confidentiality, integrity and authenticity. The most rewarding goal of course is the unauthorized access to the secured message content.

Generally, we have to distinguish between two classes of attacks. With an active attack the attacker is actively interfering with the communication between users. This may happen through modification, injection or suppression of messages. Contrary to this, a passive attack is carried out without any intervention. Hereby the attacker solely records all encrypted traffic without further influencing the communication. Usually, passive attacks require a direct attack on the underlying cryptography of the used encryption, respectively its concrete implementation.

It is worth mentioning that encryption without verification of used keys can only provide protection against passive attacks [10, p. 2]. As soon as the attacker is assumed to also carry out active attacks (eg. by presenting communication peers with malicious encryption keys) the user needs to verify keys in order to detect and repel these attacks.

While a successful attack on the used cryptographic system itself would certainly be sensational, the bulk of attacks executed in the wild is certainly of different nature. The mathematical problems that form the foundation of modern encryption protocols are well understood and usually subject of research for a long time. To break a system like RSA which is based on the problem of integer factorization [25], might present most attackers with too much of a challenge to make a worthwhile target.

A more attractive attack is one that does not break the encryption, but circumvents it. An example for this are so called downgrade attacks. Hereby, an encryption method where participants have to negotiate used algorithms (eg. hash algorithms for signatures, sym-

[3] see `https://telegram.org/`

Figure 1.4.: Telegram: Fingerprint of the session key of the call represented as emoji (top right)

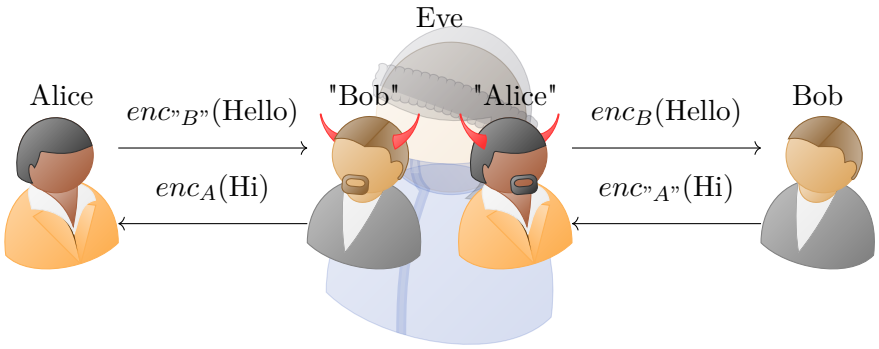

Figure 1.5.: Man-in-the-Middle Attack; Eve is impersonated Alice and Bob and is listening in on the communication

metric algorithms for encryption with a negotiated session key) is deliberately weakened or disabled completely by pretending missing support for strong algorithms. This can happen through manipulation of network packages on their way from the sender to the recipient. A prominent example is the *POODLE* [22] attack on SSL/TLS. Since downgrade attacks are mostly carried out in systems without human interaction, those are out of scope for this work.

Another class of attack techniques are Man-in-the-Middle attacks. Instead of attacking the encryption algorithms directly, the attacker merely makes the victim to encrypt messages for them. To accomplish this goal, they pretend to be the victims communication partner, thus positioning themselves as a middleman between the two. The victim, believing they would communication directly with their communication partner, encrypts the secret messages with the key of the attacker. The attacker on the other hand can simply decrypt the message and access its contents without having to attack the cryptography directly. If an attacker carries out this attack in both directions, relaying the messages, they can eavesdrop on the conversation of two participants without being noticed (see fig. 1.5).

Solely a comparison of fingerprints over an external channel (eg. meeting in person) will give away the attack. Both victims of the attack would notice that the fingerprint of the other party does not match the one the attacker presented them with. In scenarios with multiple keys per user the quantity of involved keys aggravates this considerably.

1.3. Network Topologies

Communication systems differ in their network topology. In most cases end-user devices (clients) log into one or more servers, which are responsible for message routing and delivery. Most solutions are *centralized*, meaning there is only a single service which forms the center of communication. Examples for this are WhatsApp[4] and Signal. However, there are also alternatives with more than just one single server. An example for such a service is *Mattermost*[5]. Contrary to centralized systems, so called *decentralized* systems have the

[4] see `https://www.whatsapp.com/`
[5] see `https://mattermost.com/`

advantage that users do not have to rely on the availability of a single server. In the event of an outage they can fall back to another server instance. Moreover companies have the ability to host their own, locked down servers for their staff, preventing information leaving the own infrastructure (keyword *Shadow IT*). In order to communicate with another, all communication peers have to be logged into the same server.

A third category consists of so called *federated* systems, where multiple server instances build a shared network. Users of federated services do not need to be logged into the same instance, they do not even need to be registered with the same provider, yet still they can communicate with another. The most prominent example of a federated system is e-mail. There is a huge number of independent providers, yet sending messages across servers is customary. Self-hosting an e-mail server is conceivable, although it means too much effort for most users.

Network topology has an impact on the security requirements of key management systems in the context of end-to-end encryption. The servers used play a special role here, as they may form the heart of the communication network.

Should a server be compromised, it may for example execute a so called Denial-of-Service Attack, simply no longer delivering messages. Since the server could serve each user with different information, it is in principle able to deliver fake contact encryption keys to the user. Furthermore it could suppress protocol messages like revocation of old, and introduction of new encryption keys.

This is not the case with server-less P2P networks (see section 1.3.3). Differences between network topologies and the resulting security aspects will be examined in more detail further below.

1.3.1. Centralized Systems

In centralized systems all users log into the same service. There is only one instance of this service and users rely on the availability of this instance. As any and all communication is routed through a single server, basic trust in its correct functioning is a requirement. Topologically, the system architecture therefore forms a star-network with the server the as central node and clients as its leafs (see fig. 1.6). Most commercial messaging solutions are centralized.

Advantages of centralized systems are that the entity behind the service has full control over it. If can see all communication happening and can also often control and enforce which software is being used to access the network. Furthermore development of the system is agile, cheap and flexible[6]. Potential security flaws can be addressed quick and easy without the need to coordinate changes with multiple parties. Furthermore there is no need to maintain backwards compatibility to legacy systems.

Disadvantages are that users are reliant on the single entity that controls the entirety of the system. If this entity decides to act against the interest of its user-base, users have very little choice other than to stop using the service. Centralized systems also create the effect of vendor lock-in. Users are required to register an account with the service to be able to

[6] `https://signal.org/blog/the-ecosystem-is-moving/`

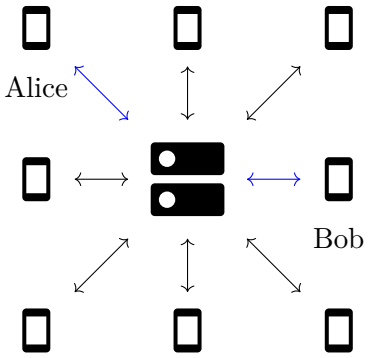

Figure 1.6.: Topology of a centralized network; Communication between Alice and Bob is routed by the server

contact someone who uses said service. The result are isolated messaging silos that grow because of the network effect and peer-pressure. Lastly, centralized systems are prone to censorship through internet service providers. Signal and Telegram for example are being blocked in some countries of the world.

1.3.2. Decentralized and Federated Systems

In decentralized environments there is more than one service. Users can oftentimes choose which instance they want to use which gives them sovereignty.

Decentralized systems can be divided into federating and non-federating networks. Users of a service that does not federate are restricted to communicate with users on the same instance and all of their communication is routed by, and over their server. Topologically these systems therefore form a forest of star networks (see fig. 1.7). An example for a decentralized, yet non-federating service would be Mattermost which allows users and institutions to run their own server. The advantage of decentralized services is that users remain in control over their communication. Should the service provider decide to act unfavorable, users can move to another instance (at the cost of leaving their contacts behind, unlike in federated systems). A company can host their own decentralized chat service to prevent information from leaving the companies network. Unlike centralized systems, decentralized networks are resistant against full network outages, as it is highly unlikely that all instances of the service go down at the same time.

In a federated network users can communicate with another no matter which service provider either of them chose. This is because different instances of a federated service can talk to each other and forward communication from a user on one instance to another users server which then delivers it. Contrary to Client-to-Server communication (C2S), communication between servers is called Server-to-Server communication (S2S). Therefore the network topology of such a system would be a fully connected graph of servers with clients as leafs (see fig. 1.8). Examples for federated systems are telephone and SMS, E-mailand instant messaging networks such as XMPP[7] and Matrix[8]. These systems grant

[7] see `https://xmpp.org`
[8] see `https://matrix.org/`

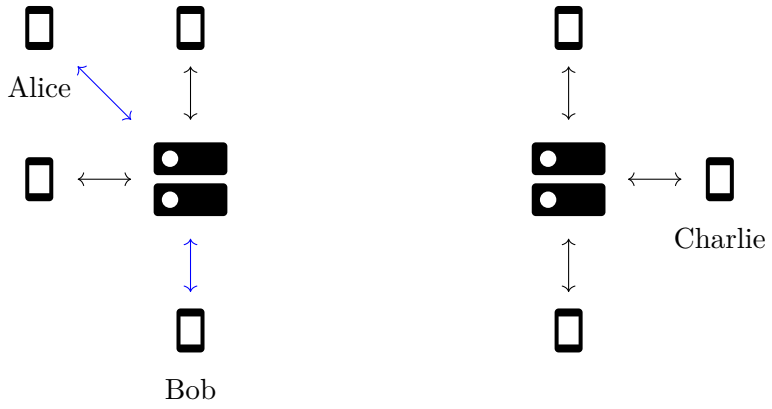

Figure 1.7.: Decentralized, non-federating systems form a forest of star networks; Alice can communicate with users from the same server such as Bob, but not with Charlie which is on another server

the user most sovereignty over their communication, as it is easy to switch the network provider without losing access to contacts. Another analogue for a federated system in the analog world would be the global postal system. In federated systems communication is potentially routed across multiple servers. On one hand this increases the attack surface, as there is more than one point at which an attacker might eavesdrop on network packets. On the other hand, a successful attack on a server only impacts a small fraction of users that communicate within the network. The decentralized nature of federated systems means that outages only affect fractions of the network.

Unlike their non-federating pendants federated systems often suffer from a slow development process as backwards compatibility to older software deployments has to be maintained to prevent network segmentation[6].

1.3.3. P2P Systems

The last category are Peer-to-Peer communication (P2P) systems. Here, clients communicate directly with another without the need for a server. P2P systems are most resistant against censorship and denial of service attacks, simply for the reason that there is no server to censor or attack. Furthermore depending on the used communication method, P2P networks can keep functioning when there is a network infrastructure outage as messages can be exchanged via local WiFi hotspots or via Bluetooth. Gossiping between clients can be used to transfer messages via multiple hops over longer distances (see fig. 1.9). That makes P2P solutions particularly interesting for application in conflict areas or for disaster relief workers.

The downside of P2P networks is that oftentimes they drain more battery than their centralized or decentralized competitors [24]. This might present a significant hurdle for broad adoption on mobile devices.

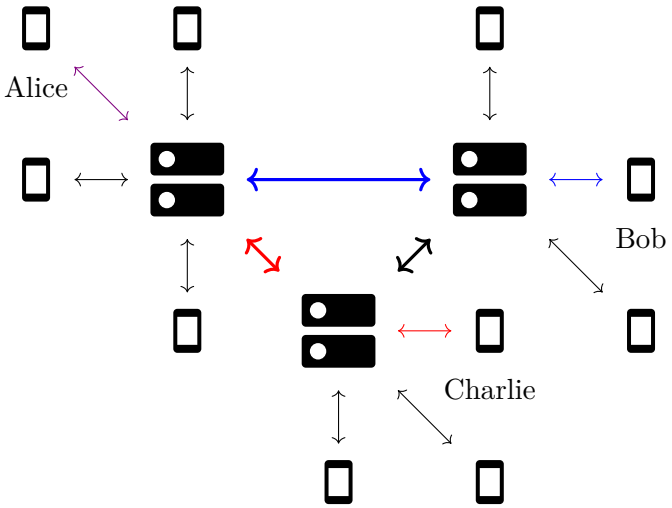

Figure 1.8.: Topology of federated systems; Although not on the same server, Alice can communicate with Bob (blue) and Charlie (red) because of the servers communicating through server-to-server connections (thick)

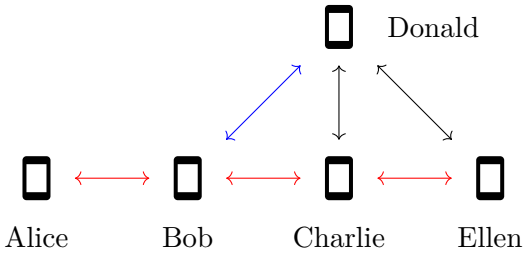

Figure 1.9.: Topology of a meshing P2P system: Client can communicate with another without the need for a server. Alice can indirectly communicate with Ellen via gossiping (red), while Bob can talk to Donald directly (blue).

Albeit being a conceivable candidate for future research, and are also able to be represented using XMPP via *XEP-0174: Serverless Messaging* [27], P2P networks are out of scope for the course of our investigations. The reason for this is that P2P networks cannot rely on a server for distribution of key material, so the solutions proposed in this work are not guaranteed to be applicable to P2P systems.

CHAPTER 2

On Key Management

The problem of key management is as old as cryptography itself. The most significant improvement to key management was probably the invention of asymmetric encryption. It reduced the number of keys that users would need to keep track of drastically.

However, the challenge of key management is still far from being solved, even though there is some interesting research going on as will be described in section 2.4. The reason why key management is such a hard problem is, because it resides right at the interface between the blackbox that is cryptography, and the human that is relying on it. As such, it often cannot be solved through abstraction, and therefore represents a greater challenge than the encryption of a message itself.

Chapter 2 highlights methodology and techniques that are used throughout this work and discusses some approaches related to the research question of key management.

2.1. The Challenge of Key Management

Management of encryption keys is a non-trivial problem in information security. Users usually are not familiar with concepts of encryption, so incomprehension or ignorance quickly lead to mistakes that potentially compromise the security completely. It is therefore important that key management schemes are as simple and error-resistant as possible. Key management in the following describes the task of keeping track of which keys are trustworthy, belong to a contact, and which are not (because they might be superseded, no longer used, compromised or belonging to an adversary).

In their work *Why Johnny Can't Encrypt* [37] Whitten and Tygar formulate the following properties of *usable* security software (citation [37, p. 4]):

> Definition: Security software is usable if the people who are expected to use it:
>
> - Are reliably made aware of the security tasks they need to perform
> - Are able to figure out how to successfully perform those tasks
> - Don't make dangerous errors
> - Are sufficiently comfortable with the interface to continue using it

It is a non-trivial task to design user-interfaces that meet these requirements. In section 1.1.4 we saw some examples of how some popular messaging solutions present fingerprints to the user. Messengers like Signal already do a pretty good job of simplifying the presentation of fingerprints. Unfortunately, however, Signals approach cannot directly be applied to other messaging architectures. Among other things this is due to each signal contact only having

one key. As soon as a contact can have more than one key (which is the case for some encryption schemes like OMEMO [35]) things get more complicated.

In a perfect world the user would do a perfect job at keeping track of which keys they and their contacts use. Presenting the user with a list of keys and switches to "activate" and "deactivate" those keys for encryption is the most naive and least comfortable way of implementing trust management. However it is also the most straight forward way and works great if each user only has a single key. As described in chapter 1 the complexity increases when users have multiple devices.

We will investigate possible approaches to this problem in section 3.1.

2.2. XMPP

The eXtensible Messaging and Presence Protocol (XMPP) is a network protocol which can be used for instant messaging. It is based around asynchronous XML streams which carry information in the form of `<message/>`, `<presence/>` and `<iq/>` packets. A noteworthy property is XMPPs end-to-end principle which states that the server mostly relays XML packets (called *stanzas*) between clients. Important to note is that the server does not have to understand the content of these packets, as long as they obey the specification and are valid XML. This makes it possible to quickly implement protocol extensions on the client level without the need to add support for them on the server side. As the identity key protocol is a client-side feature without the need for server-side support, this makes XMPP an ideal fit for our proof of concept implementation.

Furthermore XMPP already comes with a bunch of XMPP Extension Protocols (XEP) which specify additional features built on top of XMPP. Relevant to our proof of concept implementation is XEP-0060: Publish-Subscribe [21]. This specification describes a way for clients to publish pieces of information (items) to a so called PubSub node on the server, which will then notify all clients that are subscribed to that node about the item. XEP-0163: Personal Eventing Protocol [30] (short *PEP*) is a simplified subset of the features of PubSub which simplifies the usage. Examples of how both protocols can be utilized are microblogging [28] and user avatars [29]. PEP and PubSub can also be used to distribute encryption key material (see OMEMO Multi End Message and Object encryption (OMEMO) [35, § 5.3.2] and OpenPGP for XMPP (OX) [32, § 4]) which makes it especially interesting for our approach.

2.2.1. Related Extension Protocols

Since later on we are basing our experiments on the XMPP protocol, this section will take a look at existing XMPP extension protocols that are tangent to the problem of end-to-end encryption and key management.

The primary focus of the XMPP protocol lays on extensibility. The main specification (RFC 6120 [26]) only contains the very bare minimum that is required to send and receive messages. For this reason there is a pool of already far more than 400 different extension

protocols[1], called XEPs. Proposals for such extensions (called ProtoXEPs) can be submitted to the XMPP Standards Foundation by anyone with an interest in XMPP. Proposals then go through a standardization process which itself is described in XEP-0001. In case the proposal is accepted, it results in an official XEP and is assigned a number.

End-to-end encryption for XMPP is also specified via XEPs. In 2015 Andreas Straub proposed *XEP-0384: OMEMO Encryption* [35], which specifies how to apply the Double Ratchet Algorithm [23] known from the Signal Protocol to XMPP. OMEMO was first implemented in the Android chat client *Conversations*, though at the time of writing it is available in a number of different clients[2].

Furthermore *XEP-0373: OpenPGP for XMPP* [32] (short OX) and *XEP-0374: OpenPGP for XMPP: Instant Messaging* [33] (short OX-IM) describe how OpenPGP [12] can be used for encryption of XMPP messages. Contrary to OMEMO, support for OX is not yet very widespread in XMPP clients.

Both OX and OMEMO allow the use of multiple devices at the same time. However, OX does not support forward secrecy yet, so compromise of the users key results in the attacker gaining access to the entire message history of the user. Furthermore OMEMO also comes with break-in recovery, meaning if an attacker gains access to the users key, confidentiality of the communication is restored after a few messages were exchanged.

OX supports encryption of arbitrary message extension elements out of the box. OMEMO on the other hand only added support for arbitrary content encryption relatively recently with the integration of *XEP-0420: Stanza Content Encryption* [31]. Support for this in clients is not yet widely available.

While OX explicitly allows the user to back up the password protected identity key on the user's server to share it with their other devices, OMEMO is designed in a way that every device generates its own identity key. Thus, one can easily see the problem of mapping keys to users described in the abstract of this work.

If OX was the only encryption protocol that is used within XMPP it wouldn't be that hard to implement a mechanism for simple device key management, as the subkey mechanism of OpenPGP could simply be utilized without even changing the existing XEP. As OX merely specifies the distribution of OpenPGP keys and serialization of encrypted messages, the usage of key rings with subkeys is already possible and in some cases even necessary. An example for this would be the use of keys which are based on elliptic curves. Such key rings consist of a primary key pair of an algorithm which is suitable for creating certifications (eg. Elliptic Curve Digital Signature Algorithm (ECDSA), Edwards-curve Digital Signature Algorithm (EdDSA)), and a subkey which is used for encryption and decryption of messages (Elliptic Curve Diffie-Hellman (ECDH), XDH). The public parts of both keys then form a key ring which is published as the users public key. This key ring is identifiable by the ID of its primary key.

While as we saw there are a bunch of protocol extensions that deal with end-to-end encryption, there are not many that tackle the problem of key management. One approach is written down in XEP-0450: Automatic Trust Management [16]. This specification aims at simplifying key management by synchronizing the graph of trust relations via so called

[1] see `https://xmpp.org/extensions/`

[2] see `https://omemo.top`

trust messages [17] (see section 2.4.2).

2.3. OpenPGP

In this section we will take a closer look at the capabilities of the OpenPGP [11] specification. OpenPGP is a cryptographic standard which is widely used for encryption of sensitive information and signature creation.

Since OpenPGP is used in the experiments later on, this section will give a brief overview of notable features and properties of the specification.

2.3.1. Signatures

OpenPGP specifies a way to create signatures using a secret key. Cryptographic signatures can be used for a number of purposes. Primarily they function as a proof of control. To create a signature, control over the secret part of a key pair is necessary. Therefore a signature testifies that the creator of the signature has control over the secret key.

Furthermore signatures can be used to tie information to an identity. A signature over a document can be interpreted as a confirmation that the signature creator owns or created the document. A typical use for signatures is to prove that a document has not been modified by a third party.

What makes OpenPGP stand out from other signature creation mechanisms like signify[3] is the fact that OpenPGP keys are bound to identities, eg. email addresses. This is done by having a special User-ID packet with a binding signature on the key. Furthermore, entities can place certification signatures on the User-ID packets of their own and foreign keys (after verifying the authenticity of those keys out of band). That way the authenticity of a key/identity can be verified by checking if a trusted entity has certified the key in question.

2.3.2. Primary and Subkeys

OpenPGP defines the concept of primary and *subkeys*[4]. *Subkeys* are full-fledged key pairs that are subordinate to a *primary key*. This is being done by creating a special signature by the primary key on the subkey (see fig. 2.1).

This signature is a statement of the primary key that testifies that the subkey belongs to the primary key. It can be used to proof the relationship cryptographically, so users that trust the primary key of a contact can trust its subkeys as well. Particularly the user now no longer has to decide for each subkey whether or not to trust it. It suffices to check whether or not a binding to the primary key is present and whether the primary key is trusted or not.

Subkeys can be revoked independently from the primary key, which make them very interesting in the context of multi-device messaging. If a subkey gets compromised the primary

[3] see `https://www.openbsd.org/papers/bsdcan-signify.html`
[4] see `https://tools.ietf.org/html/rfc4880#section-12.1`

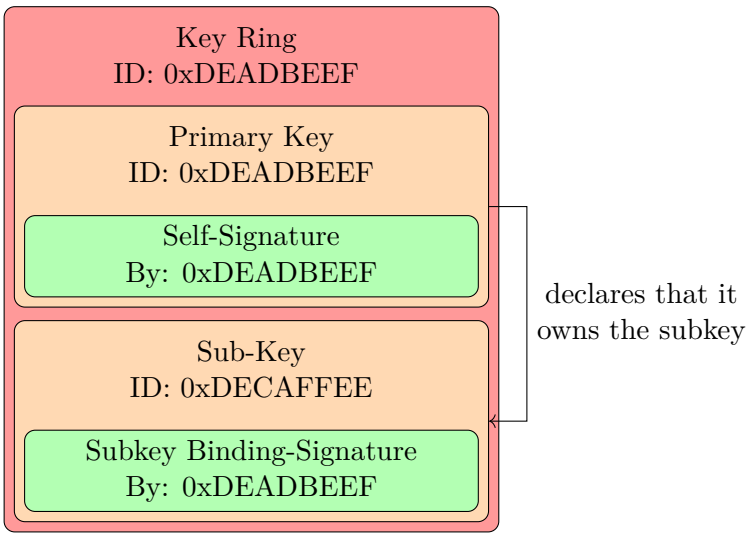

Figure 2.1.: Schematic representation of an OpenPGP key consisting of a primary key and a sub key

key can publish a *subkey revocation signature* which states that the subkey is no longer to be used. If instead the primary key gets revoked it and all of its subkeys get invalidated.

2.3.3. Key Rings/Certificates

There is some ambiguity to what exactly a key ring is. Some implementations like GnuPG primarily use the term key ring for the set of OpenPGP keys of the user and their contacts. On the other hand, a key ring might refer to a primary key and its subkeys, eg. Bouncycastle uses this terminology while in the OpenPGP specification the terms *Transferable Public/Secret Keys* are used. The *Stateless OpenPGP Command Line Interface* specification [13] proposes to use the term *certificate* in place of *Transferable Public Key* and *key* as a synonym for a *Transferable Secret Key*.

OpenPGP allows for "incomplete" certificates/keys. A key ring might consist of the primary public key, as well as secret + public subkeys (see fig. 2.2). As a consequence, the user can verify that the subkeys indeed belong to the primary key, but they cannot create certifications/signatures that require access to the secret primary key. This can be utilized to have a primary identity key that is kept offline in a safe somewhere, but which has encryption/signing subkeys. On their devices the user would have a key which consists of the secret subkeys and only the public primary key. In case of a compromise, the user can simply revoke their subkeys and create new ones with the help of their primary secret key.

2.3.4. Revocation

As with all asymmetric key encryption schemes, revocation of compromised or superseded keys plays an important role. There is always the risk that the user somehow loses their device. They could leave it on a table in a café or an attacker could steal it from their pockets. If the device is not secured properly (eg. through full disk encryption), an at-

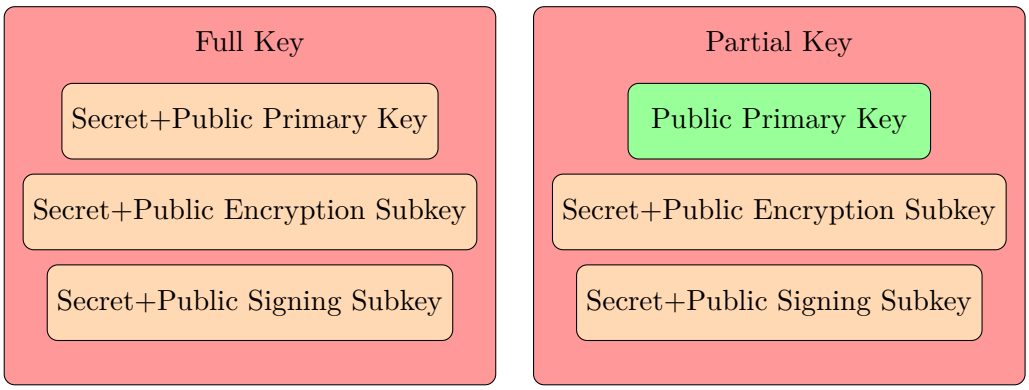

Figure 2.2.: Full/partial OpenPGP key. Compromise of the partial key (right) would not result in loss of secret primary key

tacker can easily get hold of the device's encryption keys. In any of such cases the key must be considered *compromised*. In systems without forward secrecy, the attacker can use the compromised key to decrypt past messages. To limit the attack window to only past messages, compromised keys must no longer be used for encryption, as otherwise the attacker could continue to decrypt incoming messages with them. A simplified key management system should deploy some mechanism to prevent compromised keys from being a continuous security risk.

OpenPGP offers some useful mechanisms that can be utilized to reduce these risks. First of all, signatures can carry an expiration date. This means that after a certain amount of time has passed, a signature is no longer accepted as valid. By placing an expiration date of device key certifications, the attack window in which a lost device key can be exploited to access future communication is limited. After the signature on the device key certification expires, sending devices must no longer encrypt messages to the devices listed in the certification. This approach requires the user to regularly re-certify their current devices though, so that legitimate devices always have a valid certification.

Signatures can also be manually invalidated by publishing a revocation signature. That is a special piece of signature packet that applies to a previously made signature and invalidates it. To utilize this, a client could publish a revocation signature that revokes the previous device key certification for a lost or stolen device, along with a new certification of all devices that are still in use. This would close the window of attack immediately, as from that point in time on sending device must no longer encrypt messages for the revoked device.

Of course, for revocation signatures to work the server must distribute them properly. If revocation signatures are being held back, sending devices would not be made aware of the device being compromised and continue encrypting messages for it. This would potentially result in message contents leaking to an attacker that got hold of a secret device key. Detection and prevention of such Denial-of-Service (DOS) attacks, however, is out of scope for this work.

2.4. Related Work

The developers of the messaging app *Signal* published a number of blog posts[5][6] reporting of their endeavors of making fingerprint verification easier for the end-user. As an example, they replaced the ambiguous terminology fingerprint with the easier to understand term of *safety number* (although **security** number would probably be a more appropriate term). The intention is to prevent users misunderstanding a fingerprint as a piece of sensitive information, while at the same time creating a logical association between the security number and the security of their communication.

In their paper *An empirical Study of Textual Key-Fingerprint Representations* [7] Dechand et al. analyzed which methods of visual representation are best suited for intuitive fingerprint verification. Therein the authors compared different representations of fingerprints and conducted a user study to figure out which fingerprint format is preferred by users and leads to the best results. The paper concludes that the traditional representation of fingerprints as hexadecimal characters is susceptible to partial preimage-attacks. As an improvement they propose the use of sentence-based representations where a fingerprint has the form of a sentence which is syntactically, but not necessary semantically correct. Additionally they show that even a simple representation based on numeric numbers is better suited for manual fingerprint comparison than hexadecimal numbers are. Lastly their work showed that on average, a single fingerprint comparison takes between 8 and 13 seconds, depending on the used fingerprint representation.

In RFC 7435: *Opportunistic Security* [10] Dukhovni is describing the opportunistic use of encryption. The work describes a shift in paradigm for encrypted communication. Instead of pessimistically refusing to encrypt communication at all in cases where it cannot be guaranteed that the encryption will guarantee security, the paper follows a philosophy of *a little is better than nothing* and proposes to fallback to possibly flawed encryption instead of no encryption at all.

In their master's thesis titled *Complementing the OpenPGP Web of Trust with Linked Identities* [3] Breitmoser is investigating, how different online identities can be linked to a single OpenPGP key by mutual proof of control. OpenPGP keys assert control over an identity by adding a special user attribute packet to a self-signature, which points to the identity. Meanwhile the identity (e.g. a website or social media profile) is pointing back to the key via a URI which follows a newly introduced, dedicated format.

Keyoxide[7] is a tool for verification of linked identities in a decentralized way using OpenPGP keys. The idea is quite similar to what Breitmoser proposed, only the concrete implementation is different. Identities are linked by adding a special type of annotation[8] to the key which points to the identity which shall be linked to. Again, the proof is mutual, so there must be a pointer back to the key in a fixed location on the linked identity.

In the very recently published work *SoK: Multi-Device Secure Instant Messaging* Dimeo et al. are giving an overview of the current situation in multi-device messaging. The researchers document methods of how devices are cryptographically linked and unlinked, and inspect

[5] see `https://signal.org/blog/safety-number-updates/`
[6] see `https://signal.org/blog/verified-safety-number-updates/`
[7] see `https://keyoxide.org/`
[8] see `https://metacode.biz/openpgp/proofs`

different implementations with regard to security, usability and privacy goals. Their work goes even further and discusses the problem of accessing old chats on newly linked devices. Unfortunately, the paper was published too late (April 17, 2021) to be deeper incorporated into this work, although it would have been an excellent resource to extend and build upon.

2.4.1. Blind-Trust-Before-Verification

Daniel Gultsch is following Dukhovni's example and proposes the trust model Blind Trust Before Verification (BTBV)[9] which is implemented in the messaging application *Conversations*, as well as in a growing number of other apps. Contrary to Trust On First Use (TOFU), BTBV is suitable for encryption systems where each party may have multiple keys. In this trust model, all keys of a contact are considered *blindly trusted* until the user actively verifies one of these fingerprints, eg. by scanning a QR-code. From that point on only manually verified keys of that particular contact are considered trusted. This trust model follows the idea of opportunistic security. The baseline is encrypted, but unauthorized encryption that can be upgraded to authenticated encryption by the user taking initiative. That way the user is protected against passive attacks at all times, while active attacks are still possible until the user decides to verify the fingerprints of their communication partner. Blind Trust before Verification therefore provides mostly transparent protection against passive attacks while at the same time providing the opportunity to upgrade for the expense of user effort.

2.4.2. Automatic Trust Management

In the XMPP extension protocols *XEP-0434: Trust Messages* [17] and *XEP-0450: Automatic Trust Management* [16] (ATM) Melvin Keskin is describing a mechanism to synchronize the graph of trust relations between the user and contact keys with the help of dedicated *trust messages*. Those are encrypted, signed messages that contain records stating which keys are to be trusted or distrusted. Since a sending device must only send those commands to verified devices and those devices must only follow these directions when they verified the senders key, there exists a complete chain of trust between the recipient of the trust message and the devices that are subject of the trust message. The proposal of Keskin reduces the required number of manual trust decisions, as decisions from existing devices can be reused. Let m be the total number of keys used by contacts and n the number of devices of the user. The total number of trust decisions in this situation can be modeled by a complete directed graph with

$$(n+m) \times (n+m-1)$$

edges where the mn nodes represent devices and each of the edges is a unidirectional trust decision (see fig. 5.1 for a visual representation). In the traditional model, if now the user starts using a new $n+1$th device, they would have to make a decision for all their own devices from the new one, as well as all contact devices. Additionally they would have to decide to trust the new device from all their existing devices, bringing the total number of trust decisions to $2n+m$. Keskins model can reduce the number of trust decisions to

[9] see `https://gultsch.de/trust.html`

only 2. When the user starts using a new device, they only have to mutually verify the fingerprints of that and one of their other, already established devices in order to copy over the trust graph to the new device and embed it in it. Now each of their devices trusts the new one and vice versa. Also the new device now trusts all already trusted contact devices. The user does no longer have to re-decide which devices are trustworthy and which aren't.

In a scenario with n devices, the traditional manual verification model would cost a total of

$$n \times (n-1) \in \mathcal{O}(n^2)$$

manual fingerprint comparisons. ATM reduces this number to

$$2(n-1) \in \mathcal{O}(n)$$

comparisons.

The idea is similar to the Web-of-Trust but with every already trusted key being an introducer. At the same time the fact that *any* already trusted device can act as an introducer presents the greatest weak spot of this approach, as there is no clear path to recover from one device being compromised, introducing further malicious devices.

```
1  Alice has devices A1, A2.
2  Alice mutually verifies keys of devices A1 and A2.
3  A1 trusts Bobs B1.
4  B1 trusts A1.
5  All of Alice's devices now trust [A1, A2, B1].
6  All of Bobs devices now trust [B1, A1, A2].
7  --- Attack
8  Attacker compromizes A2.
9  Attacker introduces malicious A'3 by mutually verifying A2 and A'3.
10 ---
11 All of Alice's devices now trust [A1, A2, B1, A'3].
12 All of Bobs devices now trust [B1, A1, A2, A'3].
13 ---
14 Attacker untrusts A1 and A2 on A'3.
15 All of Bobs devices now trust [B1, A'3].
16 There is no way for Alice to recover.
```

Listing 2.1: A non-recoverable attack scenario with Automatic Trust Management

While it is also possible to *distrust* devices with XEP-0450, this does not present a bullet proof recovery mechanism. Any trusted device is capable of distrusting other devices. However, once the attacker either compromises a trusted device or manages to inject a malicious device into the set of trusted devices by some other means, their malicious device is also assumed trusted. This means that now the attacker is able to act quick and distrust all devices that the victim still has control over. As there are no trusted devices left in the hands of the victim, they cannot recover from the attack (see listing 2.1).

Moreover, the validity of trust messages is based on the order in which they are delivered. If the attacker controls the server, they can try to suppress or reorder trust messages which can lead to a scenario where Alice's trust graph is different from Bob's trust graph. Of course, trust messages are encrypted, so the attacker would have to guess, which messages contained trust messages, but this is not a reliable way to prevent reordering and dropped

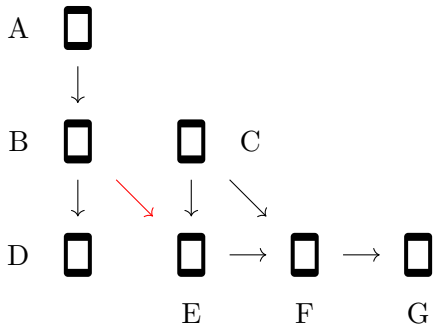

Figure 2.3.: MSC-1680: Trust graph of Alice' devices; if device B stops trusting device E (red) the graph is split in two. C will continue trusting E, while A, B will not.

messages. That way there isn't even a fail-proof way for Alice to detect successful attacks reliably. Therefore one can argue that Automatic Trust Management only provides a marginal upgrade in security compared to BTBV.

2.4.3. Matrix Device Cross-Signing

Hubert Chathi authored two Matric Spec Change (MSC) proposals for the matrix protocol related to simplified device key management. *MSC-1680: Cross-signing devices* [4] proposes a scheme where the devices would cross-sign other trusted devices with their device keys. The resulting trust graph could then be traversed to derive transitive trust. The proposal was met with some criticism as similar to ATM there is no clear way of recovering from a compromised device. Furthermore the trust graph could get arbitrarily complex and difficult to parse. Since the graph could contain cycles, it would not be trivial to interpret it properly. Lastly revocations of certain devices could result in a disconnected graph where, depending on which contact devices a user trusted, different sets of device keys would appear as trustworthy (see fig. 2.3).

Therefore this MSC was rejected in favor of *MSC-1756: Cross-signing devices with device signing keys* [5]. This MSC introduces 3 new keys; a master signing-key which acts as the users identity key. This key has two subordinate key pairs; a self-signing key which is used to sign device keys of the users own devices, and a user-signing key which is used to sign other contacts master signing keys. When a user starts using a new device, they use their self-signing key to sign the key of the new device. Since their existing devices all trust their master key and this key signed the self-signing key, all their existing devices will now trust their new device. Furthermore all contacts that signed the users master key with their user-signing keys will now trust the new device as well, as they can traverse the graph to that key (see fig. 2.4).

The reason why there are 3 different keys used in this proposal is to provide greater flexibility when it comes to revocations. The self-signing and user-signing key pairs can be replaced with new key pairs pretty easily without breaking the graph of trust. Furthermore it is possible to keep the master key locked away most of the time while the user is still able to

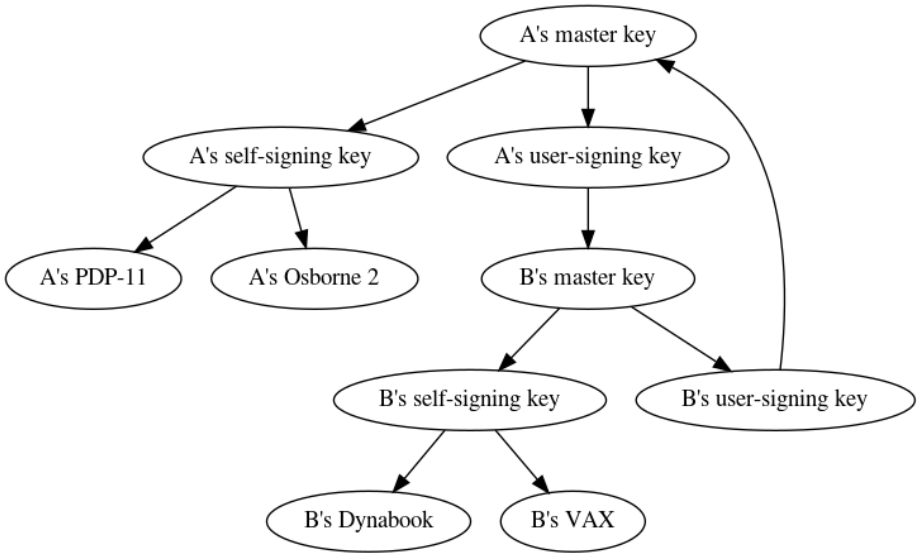

Figure 2.4.: MSC-1756: Directed graph of trust. A device A trusts another device B if and only if the graph contains path from A's master key to B.

Source: MSC-1756: Cross-signing devices with device signing keys [5]

publish new signatures.

CHAPTER 3

Design of a Simplified Key Management System

This chapter describes the attempt of designing a system which simplifies key management in XMPP-based instant messaging by binding device encryption keys to an identity, e.g. a user.

3.1. Protocol Architectures

In order to simplify key management for the user, it is desirable to reduce the number of fingerprints that need to be verified through fingerprint comparison (see fig. 3.5).

One possible approach to this problem would be to reduce the number of keys that are used by participants. This solution is chosen by eg. WhatsApp where each user only has one key. This model is also called **End-to-End Encryption Per Recipient** [8, p. 23] In case of WhatsApp this degrades the multi-device user experience somewhat, as the user has to have their phone available whenever they want to chat through a second device (eg. Desktop). It is also not possible to use the same WhatsApp account on two phones simultaneously. The web client merely connects to the users phone and sends and receives messages through the main WhatsApp application. So the web client is not a full fledged WhatsApp application, but a web interface to the phone app.

Another solution would be to share the same key across multiple devices. This model is also known as **End-to-End Encryption Per Device Per (Client Fanout)** [8, p. 23]. This solution requires some kind of synchronization or onboarding mechanism to transport an existing key to a new device. Conceivable mechanisms would be the user scanning a QR code on their old device which contains its key or the user typing in a password which restores the key from an encrypted server-side backup. The trust decision process of this approach is outlined in fig. 3.3 and will be discussed in section 3.1.2.

Lastly, a third solution to this problem is to introduce an account-wide identity key which acts as the primary key to per-device subkeys. This is also known as **Cross-Signing** [8, p. 25]. Instead of deciding for each device key of a contact whether or not to trust it, the user would only make this decision for the contacts identity key and then transitively trust each of its subkeys (device keys). The advantage of this approach is that the user would not have to decide whether or not to trust new devices that a contact may onboard, as their device keys would also be declared subkeys of the identity key. Furthermore the device keys of lost devices could be revoked as subkeys by the contact, marking them as distrusted for the user without requiring further user interaction. The process of making trust decisions is illustrated in fig. 3.1.

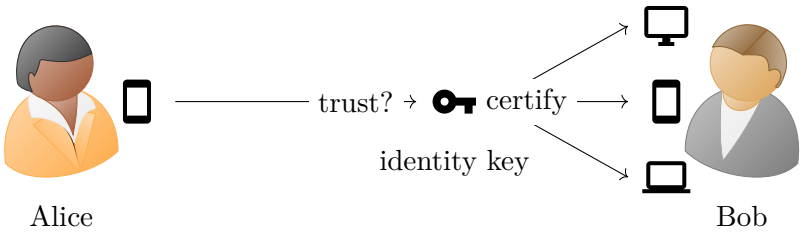

Figure 3.1.: Identity Key; Alice decides whether to trust Bobs identity key which tells her which of Bobs devices to trust

In the following we will further investigate solution 2 and 3, dissecting requirements and properties of key synchronization and then proceed to design a protocol for simplified key management based on approach number 3.

3.1.1. Secret Key Distribution

Public key encryption requires the use of key pairs. Those need to be generated at some point. In end-to-end encryption schemes only the end-points of communication (the devices of users) are allowed to have knowledge about the users secret key. Therefore key material has to be generated on the users device. This makes transferring the secret key material from one user device to another a challenge as we will see in the following sections.

3.1.2. Device Key Sharing

In a multi device messaging environment where the user has n devices each device potentially has its own device key with a unique fingerprint. While encryption mechanisms like OMEMO don't make suggestions about how to implement trust decisions [35, §2], most implementations simply present all those n fingerprints to the contact and prompt them to decide which keys should be trusted before messages can be exchanged (see fig. 3.5). The approach of device key sharing aims to reduce the number of presented fingerprints from n to 1 by reusing the same device key on all devices (see fig. 3.3).

Note that some encryption systems build stateful sessions between devices. As a result, a message might need to be encrypted for each device separately, despite all sharing the same device key. In such cases, Alice's client still has to keep track of all the different sessions, but that can simply be done by assigning each device a unique identifier (device id).

Sharing a device key between devices requires some kind of synchronization mechanism. The key needs to be copied over from an existing device to the one that gets onboarded. This transfer must be secure against eavesdropping to prevent an attacker from stealing the device key. Therefore it is a good idea to encrypt the device key before transferring it. Clients should also implement some kind of authentication (e.g. entering a password) that is required to trigger the transfer to prevent an attacker from copying the device key in an unwary situation (e.g., if the victim carelessly leaves their unlocked phone unsupervised for

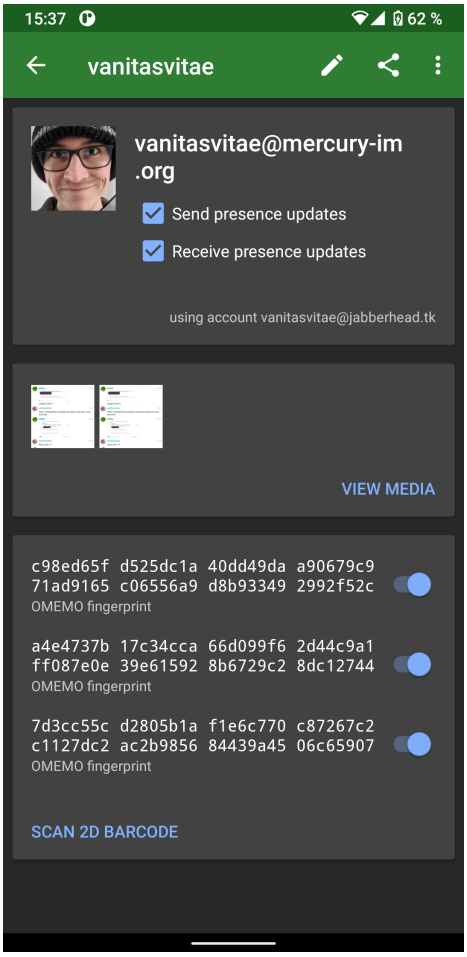

Figure 3.2.: Conversations is displaying all available contact key fingerprints to the user, letting them decide which to trust

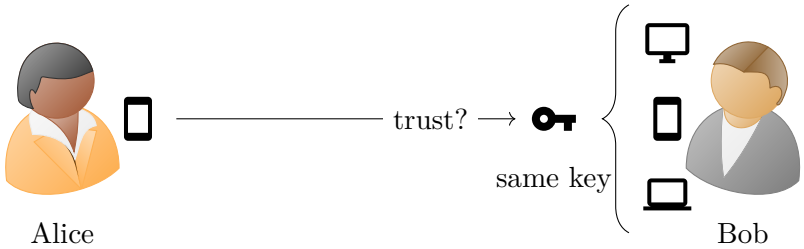

Figure 3.3.: Device Key Sharing; If Bob shares the same unique key across all his devices, Alice only has to decide whether to trust that one key

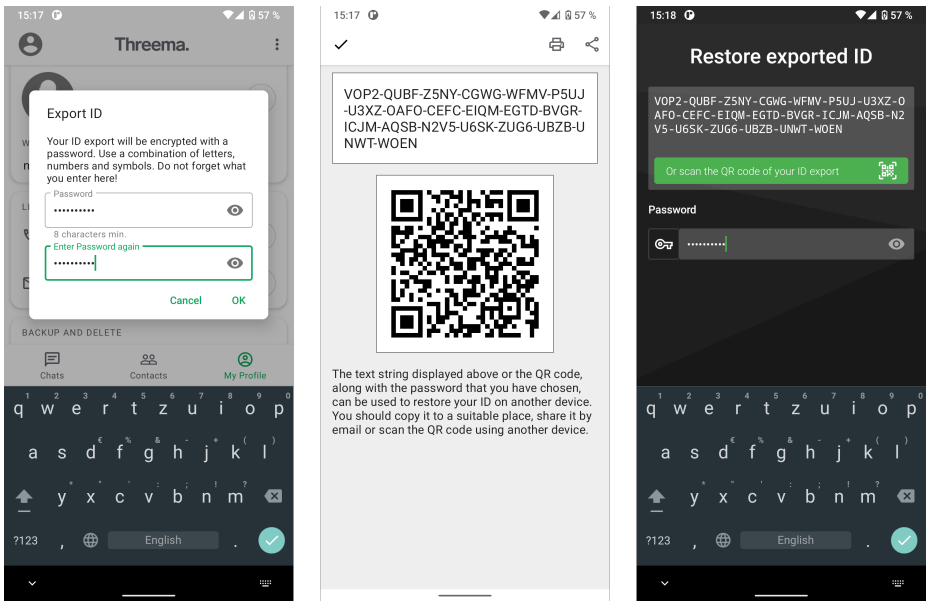

Figure 3.4.: On Threema the user can backup their ID offline and restore it on another device

a moment).

The transfer itself could be done via a number of ways which we will roughly divide into the categories of "online" and "offline" transfers. An *online* transfer entails the key being shared via the communication protocol and typically includes the server as participant in the communication. An example would be sending the key to the new device via direct message or by publishing it to a private PubSub node on the server. In an online transfer end-to-end encryption is a MUST as the server must not learn about the key, otherwise the end-to-end encryption principles are lost.

Offline transfers do not involve the server and are typically done out of band e.g. by restoring a backup of the device key from a file on the new device. The messaging application Threema for example relies on this technique (see fig. 3.4). Another example would be to encode the encrypted device key as a QR code which is scanned by the new device. That way the key is transferred to the new client without the help of the network.

In both cases the device key should be encrypted using a secret that is chosen on one of the established devices. This may be a password for the backup, an ephemeral random key, or even both.

3.1.3. Identity Key

Another approach at reducing the number of fingerprints that need to be manually authenticated is to introduce an account-wide identity key. That is a signing-capable asymmetric key which is used to attest valid device keys of the user. This can be done by signing a list of public device keys for example. In *The PGP Trust Model* [1] Abdul-Rahman calls the role that the identity key plays *introducer*. In other lectures the identity key would be called a decentralized Certificate Authority (CA) which, once trusted, certifies device keys.

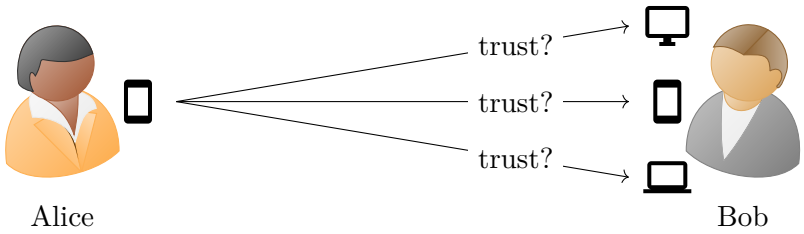

Figure 3.5.: The "classic" model; Alice decides one after the other which of Bobs 3 devices to trust. An example for this model can be seen in fig. 3.2.

If Alice is trusting the identity key of Bob, she also trusts those of Bob's device keys that are being attested by it (see fig. 3.1).

Attestations for device keys can be published in different ways. In the context of the XMPP protocol, it would be conceivable to publish a list of trusted device key fingerprints along with a signature covering the list to a dedicated PubSub node. That way the XMPP server would push the device list and signature to all clients that are interested in it.

Instead of signing a device list it would also be possible to sign individual device keys directly. The signature could either be done as a wrapping, or a detached signature. Detached signatures can be distributed independently of the signed information. In case of device keys this would come with the benefit of leaving the device key structure intact, so it could easily be deployed along-side the encryption mechanism without requiring any changes to the specification of the encryption mechanism.

One real-world example of a protocol that's based around the idea of a primary identity key which attests devices is `MSC-1756: Cross-signing devices with device signing keys` [5] (see section 2.4.3). This *Matrix Spec Change* proposal describes a scheme where a master cross-signing key is used as identity key. This key has two subordinate keys; a self-signing key used to sign other devices of the user, and a user-signing key used to sign the master keys of contacts. The overall idea of the protocol follows the identity key scheme described above. MSC-1756 splits the signing key into three key pairs to provide flexibility and to allow individual keys to be revoked/replaced when compromised though.

3.2. The Ikey Protocol

In this section we present the IKey Protocol which was developed as part of this work. The primary goal was to come up with a protocol which can be used to authenticate device keys.

The resulting protocol reduces the number of verifications per user in a group chat with n participants and an average of m devices per user to

$$2m - 2 + m(n - 1)$$

or, with further improvements (*trust sync*), to

$$2m - 2 + n - 1$$

. More on that in table 5.1.

One observation that was made is that the lifetime of device keys as used in protocols like OMEMO is often much shorter than the lifetime of the identity (the XMPP account). On the other hand OpenPGP (and respectively OX) keys are long-lived. Therefore it is a sensible decision to bind OpenPGP keys to the users identity and use them to authenticate device encryption keys.

The Ikey protocol follows the idea of the identity key (see section 3.1.3) approach, adopted to the XMPP protocol. It was explored with OpenPGP identity keys and OpenPGP, as well as OMEMO device keys. However, it should be possible to use other key types as identity keys (such as X.509). Moreover, the protocol supports further device key types as well.

3.2.1. Protocol Glossary

The Ikey protocol defines some terms which shall now be explained.

3.2.2. Assembly of Device List

The general idea of the protocol is simple. First, a list of device keys that are assumed to be trusted is assembled (e.g. listing 3.1).

```
1  <subordinates xmlns='urn:xmpp:ikey:0'
2  jid='juliet@capulet.lit'
3  stamp='2020-09-25T18:10:26.296+00:00'>
4  <sub uri='xmpp:juliet@capulet.lit?;node=urn:xmpp:openpgp:0:public-
       keys:1357B01865B2503C...;item=2020-01-21T10:46:21Z'
5  fpr='1357B01865B2503C...'/>
6  <sub uri='xmpp:juliet@capulet.lit?;node=urn:xmpp:omemo:1:bundles;item
       =31415'
7  fpr='e64dc9166dd34db6...'/>
8  <sub uri='xmpp:juliet@capulet.lit?;node=eu.siacs.conversations.
       axolotl.bundles:31415'
9  fpr='e64dc9166dd34db6...'/>
10 </subordinates>
```

Listing 3.1: Subordinates element which contains 3 device keys

Each `<sub/>` element contains the keys URI (which might be an XMPP URI to a PubSub item or a URL to a file on a web server. That way the protocol is not only agnostic to the used key distribution mechanism, but it also allows a receiving client to fetch keys that it has not yet been aware of.

Furthermore the element has a `fpr` attribute which contains the keys cryptographic fingerprint. The value of this field has to be compared to the fingerprint of the key found at the URI to prevent an attacker from replacing that key with a malicious one.

The `<subordinates/>` element must have a `jid` attribute containing the user's XMPP address as well as a `stamp` attribute with the timestamp of signature creation time. These attributes are used to prevent further attacks; Recipients must verify that the value of `jid`

Term	Abbr.	Description
Identity Key	IK	Signing-capable superordinate OpenPGP key which is used to certify subordinate keys
Device Key	SK	Subordinate key which is used for encryption and which is certified by the superordinate key
`<ikey/>`	IE	XMPP extension element with the namespace `urn:xmpp:ikey:0`. This element contains one `<superordinate/>` SO, one `<signed/>` S and one `<proof/>` child element P.
`<superordinate/>`	SO	child element of IE that contains the users public identity key IK
`<signed/>`	S	child element of IE that contains the UTF-8 encoded string representation E' of the base64 encoded XML representation of a `<subordinates/>` element E.
`<subordinates/>`	E	XML element that contains a `jid` and `timestamp` attribute as well as a list of `<sub/>` elements SUB. This element can be seen as the users device list.
`<sub/>`	SUB	child element that describes a device key SK. It holds a `type` attribute that denotes the type of SK by its protocol namespace (eg. `urn:xmpp:openpgp:1`), an `uri` attribute that points to a resource where the public key can be obtained from, as well as a `fpr` attribute which contains the keys cryptographic fingerprint.
`<proof/>`	P	child element that holds the base64 encoded detached signature DS which was made by the identity key IK over the utf8 encoded string representation E' of the `<subordinates/>` element E. This element can be seen as the signature over the users device list.

Table 3.1.: Terminology of the Ikey Protocol

is equal to the XMPP address of the owner of the identity key to prevent an attacker from presenting a recipient with the `<ikey/>` elements of another contact. The timestamp is used to prevent replay attacks. The recipient must reject the `<ikey/>` element if the timestamp is either in the future or if they have already seen another element from the same sender with a later timestamp. That way the server cannot execute attacks on the protocol by reordering `<ikey/>` elements. Since these attributes are being signed by the identity key, we can be assured that an attacker without access to the secret identity key will not be able to craft valid `<signed/>`-`<proof/>` pairs for the user.

To create the `<proof/>` element, we sign the `<subordinates/>` element with the secret identity key. In a naive approach we would simply sign the XML representation of the `<subordinates/>` element and be done. However, this would pretty quickly lead to signature verification failing on the receiving end, as the XML might get slightly modified in transit. An intermediate hop might, for example, deserialize and re-serialize the XML, resulting in different representations of the elements. So could `<element/>` be serialized to `<element></element>` or `<foo bar="baz"/>` would be changed to `<foo bar='baz'/>`. This would lead to invalid signatures as the hash sum over the plain text would differ. Therefore the XML needs to be *canonicalized*. That is to bring the XML X into a form so that for a slightly different representation X' it would hold that $\mathit{canonicalize}(X) = \mathit{canonicalize}(X')$. Then simply the output of $\mathit{canonicalize}(X)$ could be signed and send over the wire while the receiving end would be able to reconstruct $\mathit{canonicalize}(X)$ reliably, no matter how intermediate hops (non-maliciously) mangled the XML.

There is a XEP that discusses the challenges that signing XML presents[1]. Along with this document the author Zeilenga proposed two different strategies to deal with the problem[2] [3]. Both approaches have different caveats and none of them stands out as the ideal solution.

To sign XML elements the approach from XEP-0285 was briefly explored with the help of the canonicalization library *xmlsec* and the C14N11[4] standard. Unfortunately though, this would not yield sufficient results in some cases, as it is unclear if XMPP servers are required to preserve the order of elements in XMPP stanzas[5]. So instead an alternative, simpler approach was chosen. Base64 [14] encoding was found to produce representations that were stable enough to protect against reformatting. Therefore the XML from listing 3.1 was serialized into an XML string which was then base64 encoded. The result was a stable base64 string which we can denote as E' and wrap inside a `<signed/>` element S. Furthermore the base64 string E' could be signed using the OpenPGP identity key, resulting in a detached signature DS. This signature would then be base64 encoded and wrapped inside a `<proof/>` element P. Lastly, the OpenPGP public identity key would be serialized (without ASCII armor), base64 encoded and placed inside a `<superordinate/>` SO.

A downside of base64 is the increased encoding size, as base64 adds an overhead of around 33%. However for our purposes this is acceptable as we don't expect the user to have hundreds of device keys. Therefore the size overhead is neglectable.

The elements SO, S and P are then place inside an `<ikey/>` element which is then ready to be published.

```
<ikey xmlns='urn:xmpp:ikey:0' type='OX'>
<superordinate>
mFIEX24ykhMIKoZIzj0DAQcCAwTBJJKEyyU1bu+kf2Oq0fdY26S0nRELpPgm1pp4Uq9SKR4KDUP4
yykT+LDBLZwijbmPpSEEJ0qllLa7G3Mz7OIDtBd4bXBwOmp1bGlldEBjYXB1bGV0LmxpdIh1BBMT
CgAdBQJfbjKSAhsDBRYCAwEABAsJCAcFFQoJCAsCHgEACgkQEs9jFvIGSKss0wD+L7CTu/L3iwrK
MwcHkaNM5tKs/5SaCBDZFwCoFtR7ECIA/2HGcEZ6SollVaUEt6za2B8D8a02n18jHCSA07uDdSnc
</superordinate>
<signed>
PHN1Ym9yZGluYXRlcyB4bWxucz0ndXJuOnhtcHA6aWtleTowJyBqaWQ9J2p1bGlldEBjYXB1bGV0
LmxpdCcgc3RhbXA9JzIwMjAtMDktMjVUMTg6MTA6MjYuMjk2KzAwOjAwJz48c3ViIHVyaT0neG1w
cDpqdWxpZXRAY2FwdWxldC5saXQ/O25vZGU9dXJuOnhtcHA6b3BlbnBncDow0nB1YmxpYy1rZXlz
OjEzNTdCMDE4NjVCMjUwM0MxODQ1M0QyMDhDQUMyQTk2Nzg1NDhFMzU7aXRlbT0yMDIwLTAxLTIx
VDEwOjQ2OjIxWicgZnByPScxMzU3QjAxODY1QjI1MDNDMTg0NTNEMjA4Q0FDMkE5Njc4NTQ4RTM1
Jy8+PHN1YiB1cmk9J3htcHA6anVsaWV0QGNhcHVsZXQubGl0Pztub2RlPXVybjp4bXBwOm9tZW1v
OjE6YnVuZGxlcztpdGVtPTMxNDE1JyBmcHI9J2U2NGRjOTE2NmRkMzRkYjY0YzkyNDdiZDUwMmM1
OTY5ZTM2NWE5OGYzYWE0MWM4NzI0N2QxMjA0ODdmZGQzMmYnLz48c3ViIHVyaT0neG1wcDpqdWxp
ZXRAY2FwdWxldC5saXQ/O25vZGU9ZXUuc2lhY3MuY29udmVyc2F0aW9ucy5heG9sb3RsLmJ1bmRs
ZXM6MzE0MTUnIGZwcj0nZTY0ZGM5MTY2ZGQzNGRiNjRjOTI0N2JkNTAyYzU5NjllMzY1YTk4ZjNh
YTQxYzg3MjQ3ZDEyMDQ4N2ZkZDMyZicvPjwvc3Vib3JkaW5hdGVzPg==
</signed>
<proof>
iF4EABMIAAYFAl9uMpIACgkQEs9jFvIGSKubgAD+Kf2spFV0LXIqsK75lY9xxmXI2Y+zug4QSliE
ShtvPHQA/0c5agEuaixJBI5iuFs1HHbpvb3UvMjrzVsosd0Z/Y+z
</proof>
</ikey>
```

Listing 3.2: Assembled Ikey element

[1] Design Considerations for Digital Signatures in XMPP:
`https://xmpp.org/extensions/xep-0274.html`

[2] Encapsulating Digital Signatures in XMPP:
`https://xmpp.org/extensions/xep-0285.html`

[3] Encapsulated Digital Signatures in XMPP:
`https://xmpp.org/extensions/xep-0290.html`

[4] see `https://www.w3.org/TR/xml-c14n11/`

[5] see `https://mail.jabber.org/pipermail/standards/2021-February/038189.html`

3.2.3. Publication of Device List

For distribution of `<ikey/>` elements to recipients the XMPP Extension Protocol PubSub-/PEP was chosen. As an implementation of the Publish-Subscribe pattern for XMPP, it makes for the perfect candidate for the job as the sender would simply publish the element to a PubSub node and the fan-out to all recipients would be handled by the server.

For the distribution of `<ikey/>` items, a dedicated PubSub/PEP node with the name `urn:xmpp:ikey:0:subordinates` is used. A publishing client first has to check if that node already exists on the server and if not, create it. Then it publishes the new `<ikey/>`, which will trigger the server to send a notification to all clients that subscribed to the node. This incorporates all devices of the user themselves, as well as all clients of the users contacts that support the IKEY protocol.

3.2.4. Receiving a Device List Update

Clients that support the IKEY protocol have to register for updates on the `urn:xmpp:ikey:0` PEP node. Once registered, they will receive a notification with the latest item in the node of each contact, as well as subsequent notifications any time the content of the node changes. That way clients will always have a copy of the latest item available.

If a client receives a notification, it first has to check if the key in the `<superordinate/>` element carries an OpenPGP user-id that matches the contacts XMPP address (eg. `xmpp:juliet@capulet.lit`). If it does not match the contacts address the item has to be rejected. Then it decodes the contents of the `<signed/>` element to reconstruct the `<subordinates/>` element. Now it checks that the value of the `jid` attribute also matches the contacts XMPP address.

Next, the client verifies that the signature inside the `<proof/>` element was created using the public key inside the `<superordinate/>` element. It also checks if the signature is valid and that the signature creation date roughly (+-5 seconds) matches the timestamp of the `<subordinates/>` element. If the timestamp lays in the future, the element is rejected. If the client has already seen an item with a timestamp younger than that of this item, the element gets rejected as well. This is done to prevent replay attacks (see fig. 5.3).

If all those checks are successful, the item is accepted. If the client already knew that exact identity key for the contact and the user already verified this identity key, it processes all the `<sub/>` elements and marks these device keys as trusted in its database (see fig. 3.6). Other device keys of the contact that are not included in the items `<subordinates/>` list are marked as no longer trusted.

If the user did not yet verify this particular identity key (either because it has not yet seen any identity key from the contact or another identity key instead) the client has to prompt the user to verify the new fingerprint before processing the `<sub/>` elements. It may also be conceivable to mimic Signals warning about changed safety numbers here.

If the user decides to reject the fingerprint during verification, the item is rejected as well.

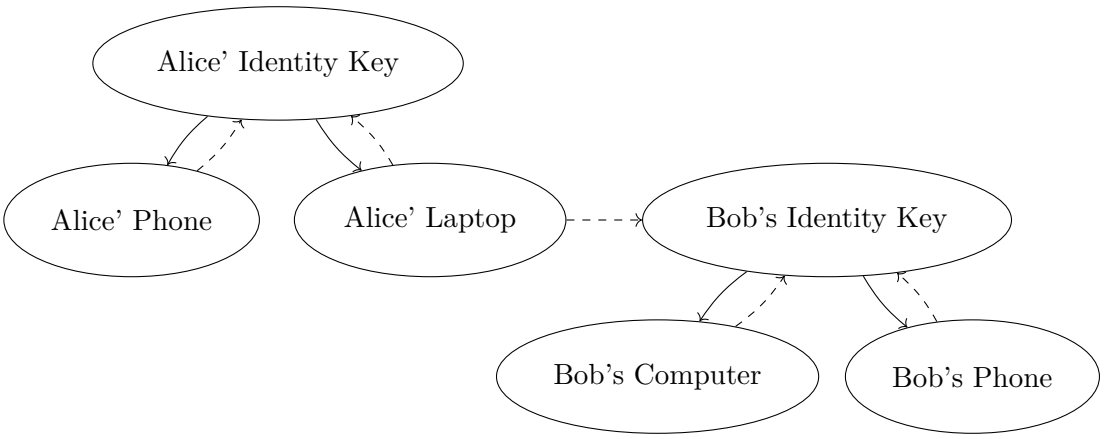

Figure 3.6.: Example trust graph of the IKEY protocol. Alice' laptop trusts (dashed) Bob's identity key, so it trusts her own as well as all of Bob's devices. Her phone does not yet trust Bob's identity key, hence it does not trust his devices.

3.2.5. Backup of Secret Identity Key

While verifying the validity of received `<ikey/>` elements only requires the public identity key of the sender, publishing new `<ikey/>` items is only possible if the user has access to their secret identity key on the device they are currently using. It might be a perfectly valid choice to keep the secret identity key only on their desktop computer and make key management decisions only on that device. However if the user wants to also make those decisions on another device, they need to copy over their secret identity key somehow.

The IKEY protocol provides a way to create encrypted backups of the secret identity key on the server. For this purpose the secret key is encrypted using a backup passphrase picked by the client. This passphrase must follow the secret key backup code format as specified by Schmaus, Schürmann, and Breitmoser in XEP-0373 [32, § 5.4] (24 upper case letters from the Latin alphabet and numbers without 'O' and '0', grouped into chunks of 4 characters divided by '-'). This passphrase is considered resistant against offline attacks and is then used to encrypt the secret key inside an OpenPGP Symmetric-Key Encrypted Session Key Packet using one of the standard OpenPGP symmetric algorithms. The result is base64 encoded and published to the secret key node `urn:xmpp:ikey:0:superordinate`. This procedure is very similar to the secret key synchronization method specified in XEP-0373 [32, §5].

To restore the identity key on another device, the client fetches the content of the secret key node and prompts the user to provide the decryption passphrase. To enhance the user experience, the client can offer to retrieve the passphrase by scanning a QR code which would be displayed by a client that already has access to the secret key.

To keep the user-experience consistent, each client with access to the secret key would also store the backup passphrase to be able to periodically fetch the secret key node and get updated about new signatures on the OpenPGP key and to provide new clients with a scannable QR code. This section of the IKEY specification might, however, leave room for improvements and would make a great candidate for further research (see chapter 6).

One approach to allow a client to regularly fetch the identity key backup node without the

need to store the passphrase would be to not only encrypt the backup with the backup code. Additionally, the backup can be encrypted with one of the encryption subkeys of the identity key itself. That way the backup code can change whenever a new backup is made, but clients with access to the identity key can still retrieve and decrypt the backup to keep it in sync. For this, a dedicated backup subkey could be used.

3.2.6. Security Considerations

The security of the IKEY protocol relies on the secrecy of the identity key. If the user loses physical access to the secret identity key, they can no longer publish device list updates to introduce new, and revoke potentially compromised devices. Should an attacker get hold of the identity key they can publish new device list updates introducing rogue devices and removing legitimate ones. To recover from such a situation, the user has to revoke the lost/compromised identity key and generate a new one. Since generating revocation signatures requires access to the secret identity key it is advisable to generate a revocation certificate for the identity key in advance and prompt the user to back it up somewhere safe.

Revocations could be published along with new identity keys by adding another element (eg. `<revocation/>`) containing a revocation signature to the `<ikey/>` element. Revoking identity keys is, however, out of scope for this work and has not yet been implemented.

To be able to exploit a compromised identity key, an attacker needs to have access to the user's XMPP account. Otherwise they would not be able to impersonate the user to publish device list updates for them. Therefore, an attack on the IKEY protocol would require a preceding successful compromise of either one of the user's devices or the users server. If the attacked user's device has access to the secret identity key, it likely also has a copy of the backup code, which presents a weak link in the protocol.

Published IKEY device list updates are being delivered to all devices of the user as well as contact devices. Therefore third parties might gain knowledge about the user's habits in respect to how often they swap out devices. This information, however, is already known to contacts, as onboarding new devices will result in new device encryption keys being exchanged. The only added information might be what devices are currently in use, however encryption protocols like OMEMO come with measures to clean up the device list to remove unused devices eventually. Therefore, the IKEY protocol does not leak further information in this regard.

3.2.7. Further Improvements

The IKEY protocol as described above can be used to quickly identify trusted devices of a user/contact. However, in its current form the protocol still requires the user of a newly onboarded device which has n contacts to do either $n+1$ or $n+2$ trust decisions, depending on the onboarding process. If the user uses identity key synchronization by restoring a backup of the identity key, the device will add itself to the users device list after the user has verified that the identity key fingerprint in the backup matches their expectations. Other devices that already trust the identity key will now trust the new device. If, on the other hand, the user decides not to transfer the identity key over to the new device an

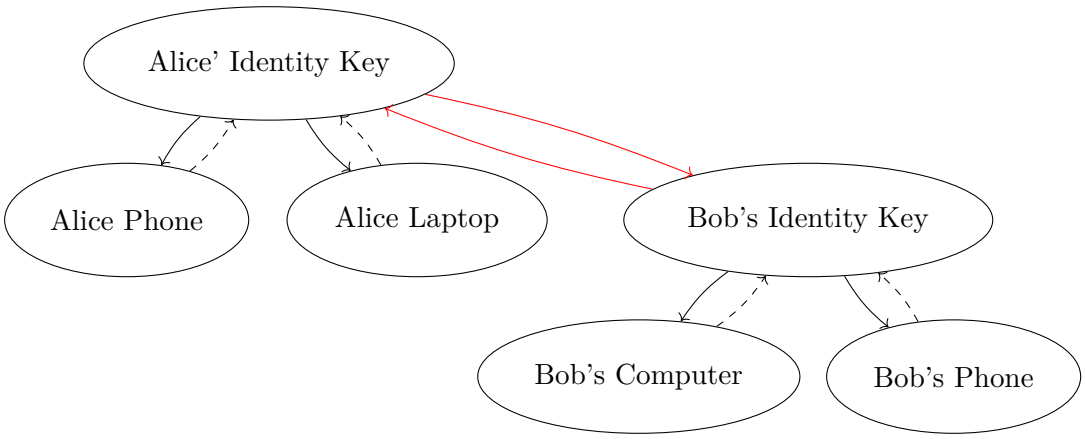

Figure 3.7.: IKEY protocol extended with synchronization of contact identity key attestations (*trust sync*) (red). Any device trusts any other device if there is a path to it in the graph.

instead manage their device list from an existing device, they'll have to manually add it to the device list by mutual fingerprint verification. First, an existing device with access to the secret identity key has to verify the fingerprint of the new device to add it to the device list. After that the new device has to trust the identity key fingerprint of the user to trust all their other devices.

Trust Sync

The n additional verifications stem from the user having to re-decide whether or not to trust their contacts identity keys with their new device. These decisions have already been made by the user previously. It is therefore desirable come up with some means of optimizing the protocol to reuse previous decisions here. One possible approach is to keep a record of trusted contact identity keys and synchronize it as attestations across devices (see fig. 3.7). That can be done by not only publishing a list of trusted *user*-devices, but also an additional list of trusted *contact*-identity keys.

The OpenPGP specification contains a section about trust signatures. Those are signatures made on keys that specify the level of trust that the owner of the signing key places in the signed key. Trust signatures are mostly used in the Web-of-Trust.

Contrary to the users device list, however, a public list of keys that are trusted by the user would leak their social graph. One could argue that the graph is already known to a server administrator, as they can see what contacts a user has in their contact list. A cryptographically signed statement, however, might be more sensible, as it cannot be forged without access to the secret identity key and might therefore encumber the user.

To prevent an attacker from using the list of trusted identity keys against the user, this list therefore has to kept secret. That can be done by encrypting it so that the secret identity key of the user is required to decrypt it. Furthermore, the list should also be published to a private PubSub node, so that it does not get distributed to third parties.

With these improvements the total number of necessary trust decisions after a new de-

vice has been onboarded would be reduced to two or one, depending on the onboarding mechanism.

Dedicated Signing Keys

Another very sensible optimization is to remove the necessity to have access to the primary secret identity key in order to be able to create device list updates. The fact that the secret identity key is required to sign new device lists imposes a security risk, as the device might get stolen or lost, resulting in compromise of the identity key. In that scenario the user would have to replace their secret key, resulting in a new identity fingerprint, which would have to be re-verified by all their contacts. To deal with this issue, the identity key could be split up into a primary identity key and attestation subkeys, similar to how MSC-1756 (see section 2.4.3) approaches cross signing. This can be accomplished by designating one or more signature subkeys to the identity key ring (see section 2.3.3). The user can then chose to upload a secret key ring without the primary secret key when creating a backup. This modification would change the responsibility of the identity key to solely be used for identification of the user, while the creation of device lists would be offloaded to one or more attestation subkeys. This enables the user to store their secret identity key offline, while synchronizing only their secret attestation keys on all their devices. If an attacker manages to compromise an attestation key, the user can simply recover from this situation by revoking the attestation keys and publishing a new one, recreating the device lists. Since the attestation key would be a subkey of the identity key, contacts would not even need to be made aware of the changed attestation keys, as the fingerprint of the identity key would still be unchanged.

CHAPTER 4

Implementation

The concept for simplified key management was implemented in an instant messaging application called *Mercury-IM*. Mercury-IM is an Android app for instant messaging over the XMPP protocol which supported encrypted chat using OX. Within the scope of this work, the app was extended to support key management using the identity key protocol as described in section 3.1.

4.1. Libraries

The implementation was based around a set of open-source libraries which will herein be presented briefly.

Smack[1] is an XMPP client library which is written in Java and supports Android as a first class citizen. It is mostly licensed under the Apache 2.0 license with the exception for some modules related to its OMEMO implementation which are licensed as GPLv3[2]. Due to Smacks modular architecture, it is easy to extend the library with new functionality. It was therefore possible to implement the identity key protocol with relative ease, which is the primary reason why Smack was chosen in this work. In the following we will briefly take a closer look on Smack's architecture.

The most important classes that are required to implement the bare XMPP specification [26] can be found in the `smack-core` module. At the heart of Smack is a `XMPPConnection` class which implements a transport method (TCP, WebSocket, BOSH...) and which is responsible for managing the stream of XMPP stanzas. Additional functionality is implemented by classes that model XMPP extension elements (DTOs), as well as provider classes that are responsible for parsing them from XML. The respective interfaces are called `ExtensionElement` and `ExtensionElementProvider`. The high-level API that is exposed to the user is implemented in a `Manager` class. Here the user can execute tasks or add listeners for events. Most of these classes are located in the `smack-extensions` or `smack-experimental` modules, mostly based on the state of the specification.

At the time of writing, Smack implements around 90 different XMPP Extension Protocols[3], which gives evidence of its extensibility.

In the course of this work, Smack was extended by creating a custom module (`smack-ikey`) which implements the identity key protocol.

[1] `https://github.com/igniterealtime/Smack`
[2] `https://github.com/igniterealtime/Smack/wiki/OMEMO-libsignal-Licensing-Situation`
[3] `https://github.com/igniterealtime/Smack/blob/master/documentation/extensions/index.md`

PGPainless[4] is Java library which strives to simplify the use of OpenPGP. It mostly acts as a wrapper around the cryptographic library Bouncycastle[5]. It was written by the author of this work in 2018, along with a module for Smack which implements OX[6]. PGPainless came in handy for this work as it could be used to implement the key generation and signing parts of the IKEY protocol.

The ORM-framework *requery*[7] was used as database back-end. The reason why this was chosen over the otherwise recommended database framework *room* is that while requery comes with support for reactive queries, it is platform independent compared to room which (at the time of writing) is Android exclusive.

To simplify injection of dependencies into class instances, a dependency injection framework can be used. Among the most popular tools for this purpose is *dagger*[8]. Dagger uses Java annotations and interfaces to construct the dependency graph at compile time. This results in very efficient generated code, which resembles what the user would have written for manual dependency injection. Furthermore missing or cyclic dependencies are being detected at compile time.

4.2. Architecture

Mercury-IM aims to follow the architectural rules laid out by Martin in Clean Architecture [19]. Most of the business logic is encapsulated in the `core` module. This module does not contain any references to Android classes, so it is possible to reuse all of the `core` code in a non-Android application. This is being achieved by following the `SOLID`-principles [20]. Many functionalities whose concrete implementation would depend on external dependencies (eg. database access) have been generalized into interfaces. Instead of implementing those in the `core` module, the implementations can instead now be located in separate domain specific modules (eg. `data` which implements the database back-end using requery). So instead of letting `core` depend on the database module, the database module now depends on `core` (Dependency Inversion Principle [20, p. 12]).

Applying the Dependency Inversion Principle leads to loosely coupled code where external dependencies can be swapped out relatively easy. Instead of letting a service instantiate and setup its database connection, the database is passed in as constructor parameter of the service to avoid a hard dependency. However, for the application to function correctly, all those decoupled classes and modules need to be assembled during application startup. This is where dependency injection (DI) comes into play. Each distribution of the application (Android app, terminal client, ...) has its own dependency injection container class, whose responsibility it is to instantiate classes and inject dependencies. This task can be tedious so instead a dependency injection framework (e.g. dagger) can be used to automate the task. This leads to very clean code.

Mercury-IM implements the Model-View-ViewModel (MVVM) design pattern first pro-

[4] see `https://pgpainless.org`
[5] see `https://www.bouncycastle.org/java.html`
[6] see `https://github.com/igniterealtime/Smack/tree/master/smack-openpgp`
[7] see `https://github.com/requery/requery`
[8] see `https://dagger.dev`

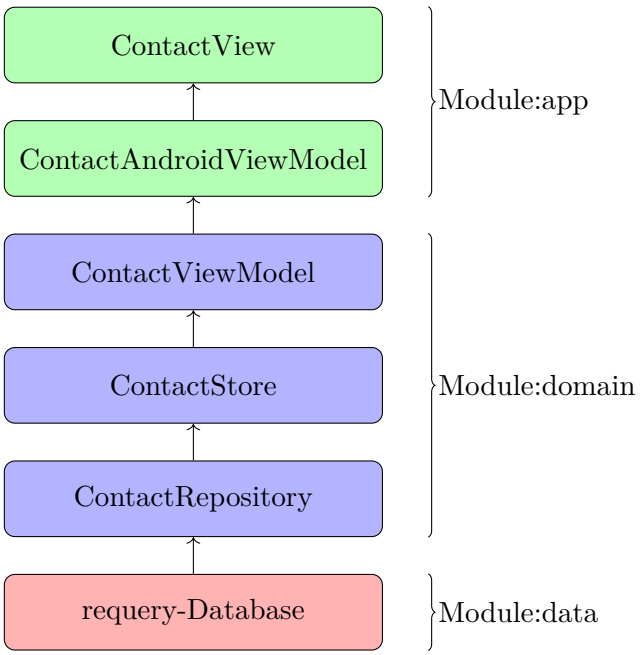

Figure 4.1.: Data flow from the model to the view

posed by Gossman[9]. As a consequence method the view doesn't directly access the model, but instead subscribes to changes in the model through the ViewModel. The ViewModel serves information via reactive means. On Android it is recommended to use *LiveData*[10] from the Jetpack architecture components for this purpose as these are life-cycle aware. This prevents leakage of references as the LiveData components automatically observe the life-cycle of the activities and fragments they are referenced in and act accordingly.

At the time of writing, however, LiveData is an Android exclusive library. Since Mercury-IM aims to eventually become platform independent, LiveData cannot be used in the core module. Therefore it was decided to use RxJava in for platform independent core ViewModels. In the Android `app` module which is a very thin wrapper around the core module, special AndroidViewModels are used to wrap the core ViewModels and adapt any RxJava Observables with LiveData. That way we have the best of both worlds: Platform independence and a rich mapping API from RxJava, and life-cycle-awareness from LiveData. By following this approach we can move much of the business logic into the platform independent core module and rather quickly add user-interfaces (Android, CLI) on top.

A downside of this design decision is another layer of added abstraction which adds complexity to the code (see fig. 4.1).

[9] see `https://docs.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps`

[10] see `https://developer.android.com/topic/libraries/architecture/livedata`

4.3. The outcome: Mercury-IM

The IKEY protocol has been implemented in the XMPP messenger *Mercury-IM*. Mercury-IM is a chat application capable of end-to-end encryption using XEP-0373/XEP-0374 OpenPGP for XMPP (OX). Originally, the app supported synchronization of the secret OpenPGP key across devices (basically implementing device key sharing, see section 3.1.2), so there wouldn't really be a need for an identity key protocol to be added on top. Therefore it was modified to disable secret key synchronization and generate a separate OpenPGP key for each device instead in order to create a per-device key scenario similar to the OMEMO ecosystem.

4.3.1. Account Setup

The account setup in Mercury-IM resembles that of a typical XMPP client. When a new account is added, the user is prompted to enter their Jabber-ID / XMPP address (JID) and password. The app then connects to the server and tries to log in. After successful authentication the app generates and publishes an OX device key. It then checks if the IKEY backup PubSub node exists and if it has an item in it. Meanwhile, the user is asked whether or not they want to enable the IKEY feature (see fig. 4.2 a). If they decline the app continues to function as a normal (end-to-end encrypted) XMPP messenger. However, if the user enables the IKEY protocol and an encrypted backup has been found on the server the user is prompted to enter the backup passphrase (fig. 4.2 b). They can now either enter the code manually or by scanning the QR code of an existing installation. Afterwards the backup will be restored and the identity key will be available on the new device.

If the backup node didn't exist or did not contain a backup, a fresh identity key will be generated instead. In any case the app will then display the identity keys fingerprint to the user, offering to create a backup on the server (fig. 4.2 c). It was chosen to divide the fingerprint into blocks of 4 characters which are then being colored with the help of *XEP-392: Consistent Color Generation*. Fortunately XEP-0392 had already been implemented in Smack, so all that was left was to convert the RGB values into a single integer color code and to use that with Androids `Spannable` class to color in the different blocks of the fingerprint. While not specified anywhere, this way of coloring fingerprints to improve comparability was introduced by the XMPP client Dino[11].

4.3.2. Device Management

Mercury-IM implements basic device management using the IKEY protocol. The users account detail screen was extended by their identity key fingerprint which can be seen in fig. 4.4 a) on the top. Below, the app displays the devices encryption key fingerprint. Lastly, there is a list displaying the fingerprints of other devices the user owns. While previously the switches next to each key could be toggled to change whether or not the user's device would encrypt to them, this has been changed in the course of this work. Now, when the user toggles a device key on it will be included in the next IKEY device list update as a

[11] see `https://dino.im`

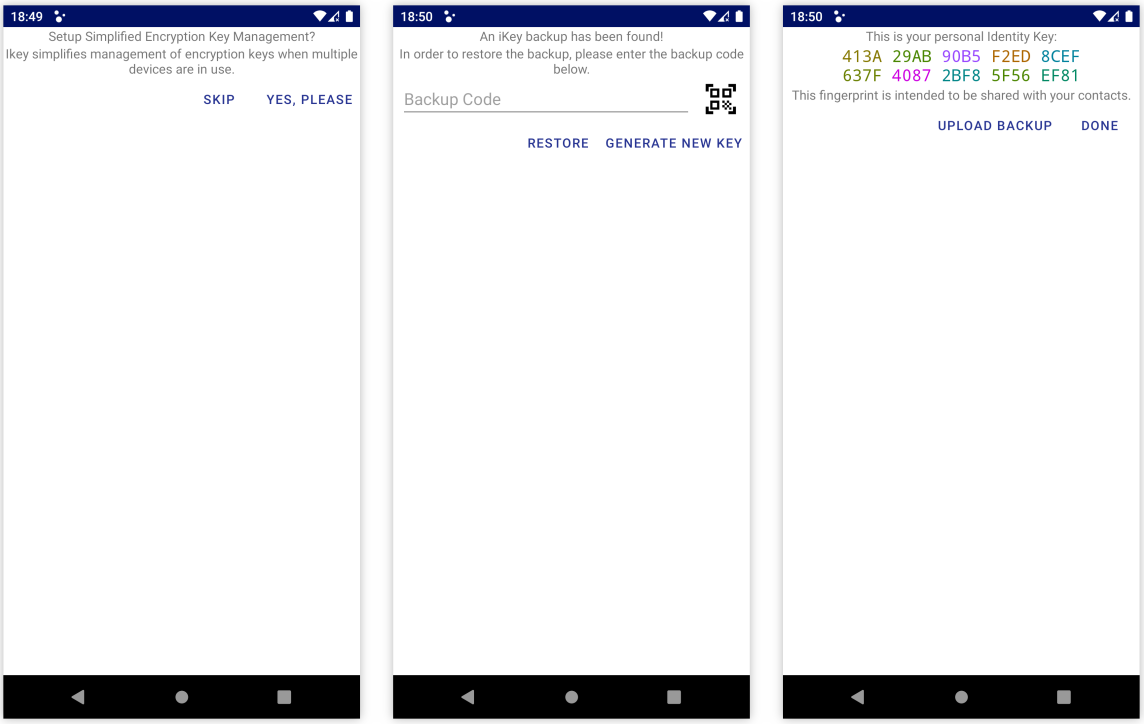

Figure 4.2.: Ikey account setup prompt; a) user is asked whether or not to enabled Ikey feature; b) an existing identity key backup was found; c) a fingerprint of identity key is being displayed

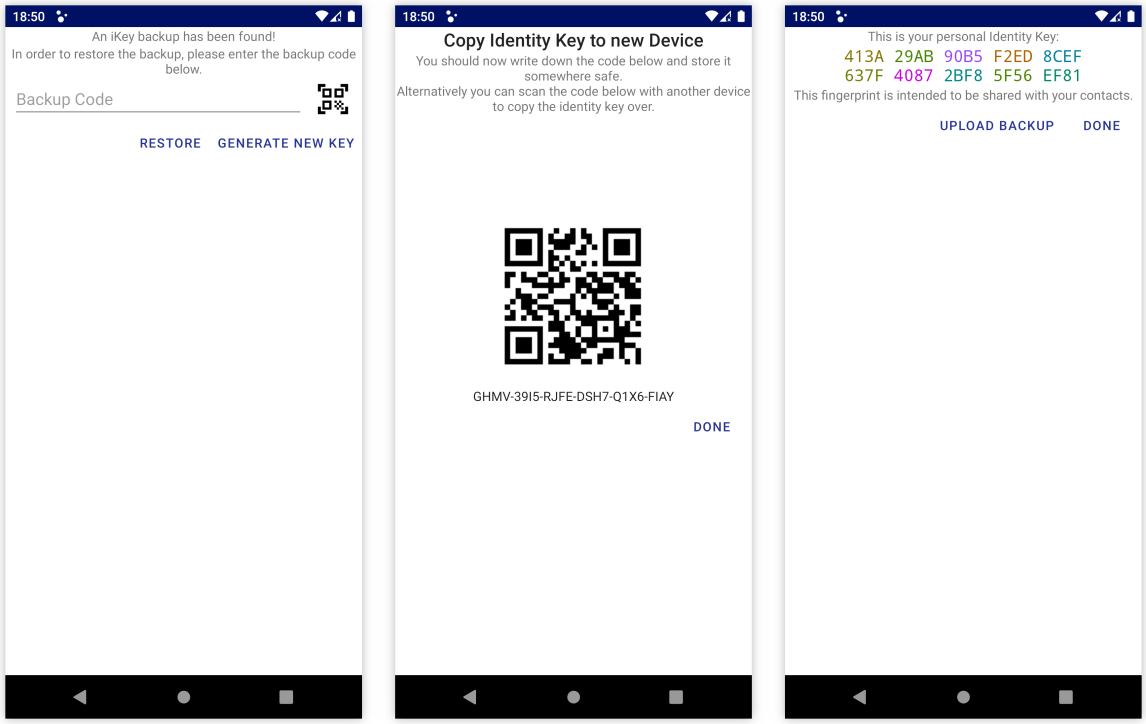

Figure 4.3.: Ikey Backup Mechanism; a) user is prompted to enter backup passphrase; b) established device displays QR code containing backup passphrase; c) after restoration of backup the identity key fingerprint is being displayed

trusted device. Such updates currently have to be published manually by pressing a button at the bottom of the device list. If a key is included in an IKEY device list update of a trusted identity key a black lock is being displayed on the switch. Examples for this can be seen in fig. 4.4 b). In fig. 4.4 c) a contact details screen is shown as it would be displayed to contacts. This screen features the contact's identity key, which can be toggled on and off depending on whether the user trusts it. As soon as the identity is being marked as trusted, the list of device keys below will show which devices keys are being authenticated by it. These device keys are then also considered trusted by the messenger. If the user now sends messages, those will only be encrypted to trusted keys.

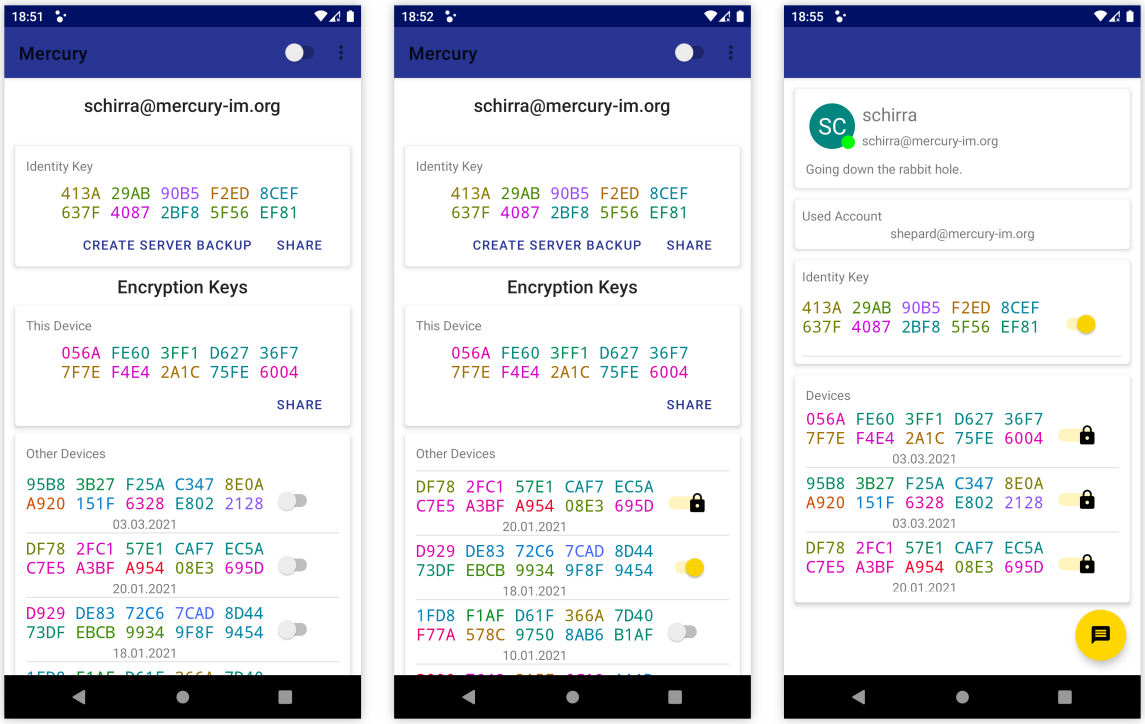

Figure 4.4.: Ikey device management; a) list of initially undecided devices; b) user marked some devices as trusted; c) trusted devices as displayed to a contact

CHAPTER 5

Discussion

In this work, the problem of increasing complexity of key management in end-to-end encrypted multi-device instant messaging has been explained. Different examples to how this problem is being approached by popular messengers have been broken down into 3 different architecture models. These models were examined in detail and one model was chosen as the basis for a protocol tailored to the use-case of instant messaging via XMPP. The resulting IKEY protocol was explained and an experimental implementation of it was integrated into the messaging application *Mercury-IM*.

5.1. Protocol

The IKEY protocol is a good first step towards a solution for simplified key management. While a full security analysis of the protocol specification is out of scope for this work, particular care was taken to prevent obvious security risks and popular attacks. If properly applied by the user, the protocol enhances the user experience by reducing the number of required fingerprint verifications.

Relations between devices can be modeled as a complete graph and the number of edges in a complete graph with n nodes is calculated as

$$\frac{n(n-1)}{2}.$$

Since trust relations are bidirectional, we can deduce that the total number of manual trust decisions that are necessary to establish a fully connected bidirectional trust graph with the traditional model would sum up to

$$2\frac{n(n-1)}{2} = n(n-1).$$

See fig. 5.1 as a visualization of this formula.

Dechand et al. showed that a single fingerprint comparison takes an average of 8 to 13 seconds, depending on the used fingerprint representation [7]. This further illustrates that the status quo simply does not scale at all, given the *unmotivated user property* [37, p. 4].

The IKEY protocol reduces the number of fingerprint verifications needed to establish a complete trust graph of devices of a single user to

$$2n - 2$$

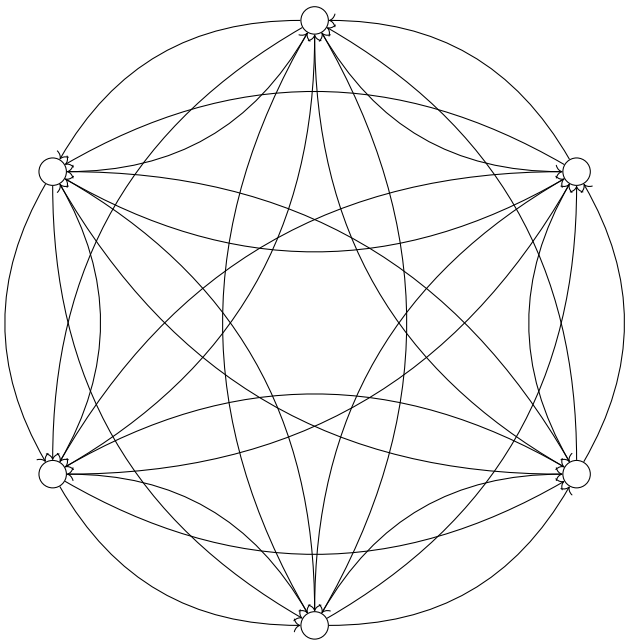

Figure 5.1.: Fully connected, directed graph with n nodes and $n(n-1)$ edges

since the very first device does not need to compare any fingerprints and for each new device only 2 additional comparisons are needed.

If trust in contact identity keys is also synchronized (IKEY + *trust sync*, see section 3.2.7), one mutual fingerprint comparison is sufficient to establish trust between all devices of the user and those of the contact. In total, if there are n users with an average of m devices, a total of

$$n(2m-2)+n(n-1)$$

fingerprint comparisons would be necessary to establish mutual trust between each and every device. Per user that would be a workload of $2m-2$ comparisons of their own device key fingerprints plus an additional $(n-1)$ comparisons of contact identity key fingerprints, compared to traditionally required $m(mn-1)$ comparisons (see fig. 5.2, as well as table 5.1). Therefore synchronizing contact device trust is a worthwhile extension to the IKEY protocol.

The protocol should be safe against man-in-the-middle attacks iff the user compares key fingerprints properly. In that case it inherits most security guarantees from OpenPGP. Furthermore, the protocol is resistant against replay attacks, due to the use of timestamps in the signed domain of exchanged messages (see fig. 5.3).

It is possible though for a malicious server to suppress individual device list updates. That way the server can keep devices in the belief that an outdated set of devices is to be trusted. Unfortunately it is not trivial to detect such attacks without further countermeasures.

Another possible weak point is the migration to new identity keys. If the user loses access to the secret part of their current identity key they have to migrate to a new one. This migration has to be communicated to their contacts out of band, as there is no way to

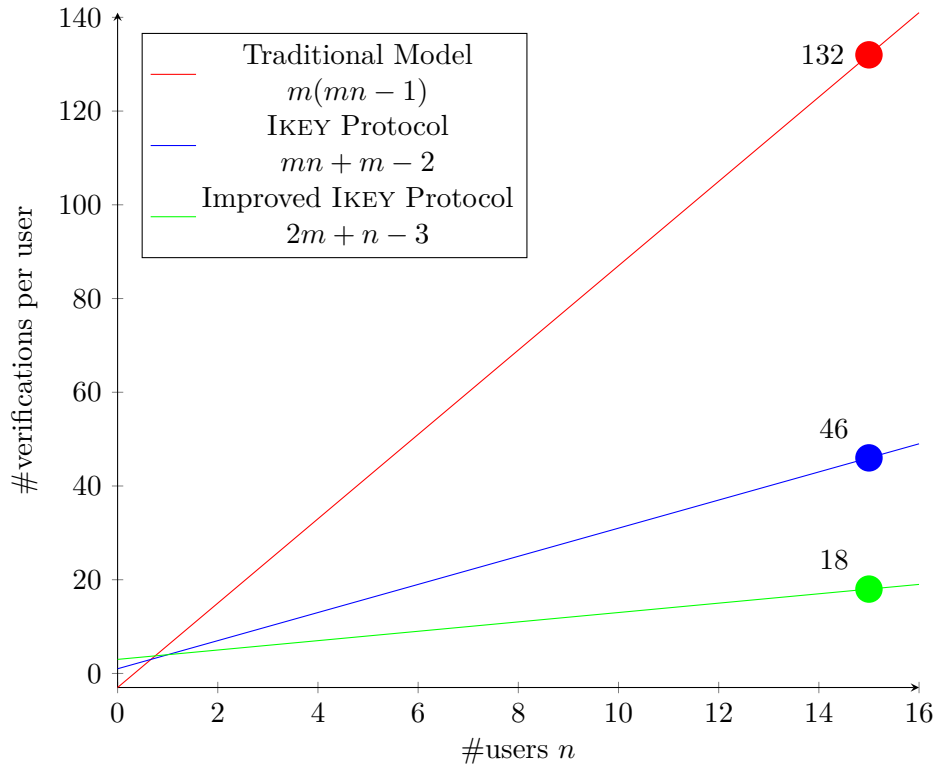

Figure 5.2.: Comparison of required fingerprint verifications per user. Traditional model (red), Vanilla IKEY protocol (blue), IKEY protocol with synchronization of trusted contact identity keys (green). In each case an average of $m = 3$ devices per user is assumed. In this case, up to $86,364\%$ of comparisons can be eliminated

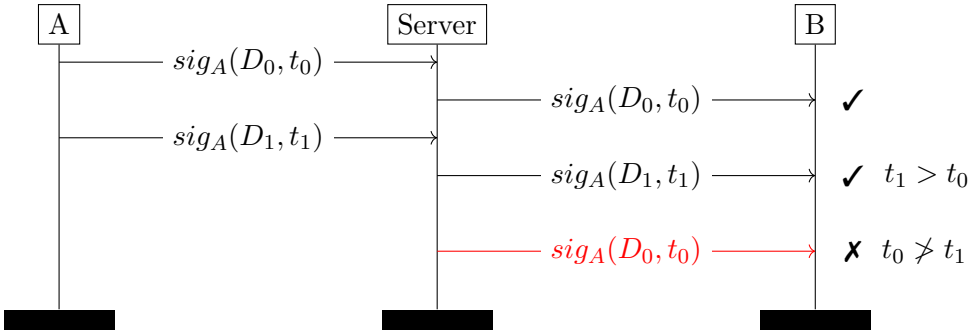

Figure 5.3.: The IKEY protocol is resistant against certain replay attacks of device list updates (red). D_i denotes the set of attested devices at time t_i.

Mechanism	Total Comparisons	Comparisons per user
Traditional Model	$(mn)(mn-1)$	$m(mn-1)$
IKEY protocol	$n(mn+m-2)$	$mn+m-2$
IKEY + trust sync	$n(2m+n-3)$	$2m+n-3$

Table 5.1.: Comparison of required fingerprint verifications to establish a complete trust graph for n users with an avg. of m devices each

revoke the old key and cryptographically proof that the new key indeed belongs to the user. Therefore, at this point, the new identity key has to be manually verified by each contact once again.

Lastly, it might be hard for a user to detect compromises of their secret identity key. If an attacker manages to get hold on a user's secret key, they could simply introduce rogue devices for the user. To mitigate those from being undetected, already established devices could notify the user whenever a new user-device is being introduced.

5.2. Prototype Implementation

The messaging application *Mercury-IM* has been extended to support the Ikey protocol. The app is still in a very early state of development and is not considered ready for everyday use under any circumstances. However the implementation suffices to demonstrate that the complexity of key management in multi-device messaging can indeed be reduced in a user-friendly way.

Reliability

There are still bugs left in the prototype implementation. Those can result in device list updates not being processed properly, or changes not showing up on the users screen. With enough dedication, these problems can eventually be ironed out.

Maintainability

The implementation of the Ikey protocol follows the ideas of Clean Code. Therefore the code is kept modular and allows for easy extension.

Testability

There is a small set of JUnit tests that were used to verify functionality of some components during development. Creating new tests is relatively easy, as the MVVM pattern allows for the UI to be swapped out for testing code quite easily.

Portability

The source code of the prototype is available[1] under the GPLv3[2] license. The fact that one eventual design goal of the app is platform independence introduces some complexity, since use-cases have to be implemented on multiple abstraction layers. As an improvement, the code could be popularized by use-case instead of by layer.

This modular design allows most of the code of the Android application to be reused in a desktop or Command Line Interface (CLI) application.

[1] see `https://zivgitlab.uni-muenster.de/ag-gorlatch/abschlussarbeiten/paul-schaub/mercury-im`

[2] see `https://www.gnu.org/licenses/gpl-3.0.en.html`

Reusability

During development, close attention was payed to increase reusability as much as possible. Very rarely are there duplicated code segments in different classes. Most of those occurrences have been moved into delegate classes which can be called from multiple spots.

5.3. Limitations

This work focused primarily on the technical specification and its prototype implementation. The aspect of user interface design was mostly neglected.

As a result the current implementation of the user interface is far from ideal. While the description texts promise simplified key management, the user is presented with even more different keys and fingerprints that they have to take care of. This shows that a good user-interface is just as important as the underlying protocol. This observation is in line with the observations that Whitten and Tygar made about the usability of PGP implementations.

Further, there has not been any kind of user-study which evaluates the usability of the proposed mechanism.

While the messenger will only encrypt messages to trusted devices, it will decrypt messages from any sending device (given it has access to the sender's public key). This is a problem, as the user will not receive any feedback about the origin of the message. Messages from distrusted devices will show up just as messages from trusted devices would. While this problem is rather easy to solve, a solution has not been implemented in the prototype application. It makes for an excellent candidate for further improvement of the prototype though.

This work did not conduct a formal analysis of the proposed Ikey protocol. Therefore it cannot make any definitive claims regarding its security properties.

CHAPTER 6

Further Research

Simplifying key management in multi-device messaging is an exciting field of research. While this work hopefully succeeded in providing an elevated overview of the problem and also came up with some strategies to tackle it, there is still room for improvement and future research.

6.1. X.509 Certificates

While the Ikey protocol has been explored with OpenPGP and OMEMO device keys and OpenPGP identity keys, it might be desirable for implementations in corporate environments to make use of X.509 certificates as identity keys. In fact, X.509 might provide better tooling on most platforms and might therefore be the better choice. A centralized certificate authority could be used to certify devices and create `<ikey/>` device list updates in corporate environments.

6.2. Signing XML

Signing XML reliably is not as straight forward as one might think. Since intermediate hops could alter the representation of the XML while maintaining its semantic meaning, the use of some sort of canonicalization mechanism is a requirement. The method of creating signatures over base64 encoded XML is not ideal and could be improved by specifying a more elaborate encoding method or data structure. Googles protobuf[1] library might be a good candidate to look at for this purpose as it allows for very compact and most importantly unambiguous representations of the data compared to XML.

6.3. Improving Identity Key Synchronization

OpenPGP keys might change over time. New signatures might be added to it to change its algorithm properties or extend its expiration date. When this happens, the key needs to be re-synced across devices. It is unreasonable to expect the user to manually re-enter the backup passphrase on already established devices only to re-sync the identity key. In the current implementation of the Ikey protocol, each client with access to the secret identity key also stores the backup code that was used to create the backup. Furthermore each

[1] see `https://developers.google.com/protocol-buffers`

backup of the same identity key uses the same exact backup passphrase which is also stored by clients. That way clients can use the code to decrypt future backups made by other clients.

A conceivable improvement would be to add a dedicated backup-encryption subkey to the identity key ring. As OpenPGP allows to decrypt messages and files to multiple recipients and passphrases, the identity key backup can be encrypted for the backup passphrase and the dedicated backup encryption key at the same time. This would relieve clients from needing to store the backup passphrase and would allow them to use a new backup code each time they create a backup. At the same time, clients with access to the backup encryption key can use it to decrypt future backups and keep the identity key in sync with others. This could be accomplished by equipping the backup encryption key with a custom annotation or a key flag of `ENCRYPT_STORAGE` to denote its use.

6.4. Fingerprint Formats

In *An empirical study of textual key-fingerprint representations* [7] Dechand et al. explored the effectiveness of different fingerprint formats for verification. They concluded that number-based representations are best suited for manual fingerprint comparison. It would be interesting to utilize these representations for identity key fingerprints. It may also be interesting to adopt Signals *Safety Numbers* scheme [18] in the Ikey protocol. This is possible as there is only one identity key per user, so two key fingerprints can be combined into one safety number.

6.5. User-Acceptance of Fingerpring Comparison

It would be interesting to know, how acceptable users are when it comes to comparing multiple fingerprints in a row. After how many comparisons do users give up on proper fingerprint verification or start to become sloppy? How do users react if they are presented with non-matching or additional, foreign fingerprints?

CHAPTER 7

Conclusion

In this work the challenges of key management in end-to-end encrypted multi-device instant messaging were discussed. The complexity of tracking one's own and contacts' active fingerprints while preventing the use of rogue keys were outlined in detail. This work identifies the necessity to come up with solutions for this problem since authentication of keys in end-to-end encryption is vital for security. Different models for key management solutions were highlighted and one model was chosen as the basis for a prototypical XMPP extension. As defined in this work the Ikey protocol utilizes an OpenPGP key as identity key, which is used to create certifications for per-device encryption keys. That way the number of keys that need to be manually authenticated is drastically reduced. The protocol supports synchronization of secret and public keys via the PubSub specification.

The prototype implementation in the messaging app *Mercury-IM* demonstrates that a deployment of the described protocol in a federated environment is viable and worth to be investigated further. It also unveils that it is by no means trivial to design a user-friendly interface for cryptographic applications.

While a complete analysis of the Ikey protocol is out of scope, the experiments suggest that it has the potential to massively reduce (from $\mathcal{O}(n^2)$ down to $\mathcal{O}(n)$) the number of necessary manual fingerprint verifications (see table 5.1 and fig. 5.2) without degrading security. This indicates that simplification of the fingerprint managing process will have a positive effect on user experience which raises the hope that, as a result, users will take fingerprint verification more seriously, thereby improving the effectiveness of end-to-end encryption solutions.

Bibliography

[1] Alfarez Abdul-Rahman. "The PGP Trust Model". In: *EDI-Forum: the Journal of Electronic Commerce*. Vol. 10. 3. 1997, pp. 27–31.

[2] Wei Bai, Moses Namara, Yichen Qian, Patrick Gage Kelley, Michelle L Mazurek, and Doowon Kim. "An inconvenient trust: User attitudes toward security and usability tradeoffs for key-directory encryption systems". In: *Twelfth Symposium on Usable Privacy and Security ({SOUPS} 2016)*. 2016, pp. 113–130.

[3] Vincent Breitmoser. "Complementing the OpenPGP Web of Trust with Linked Identities". MA thesis. Technische Universität Braunschweig, 2015.

[4] Hubert Chathi. *MSC-1680: Cross-signing devices*. 2018. URL: `https://github.com/uhoreg/matrix-doc/blob/cross-signing/proposals/1680-cross-signing.md` (visited on 03/07/2021).

[5] Hubert Chathi. *MSC-1756: Cross-signing devices with device signing keys*. 2018. URL: `https://github.com/matrix-org/matrix-doc/blob/master/proposals/1756-cross-signing.md` (visited on 03/07/2021).

[6] Joan Daemen and Vincent Rijmen. "AES proposal: Rijndael". In: (1999).

[7] 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)*. 2016, pp. 193–208.

[8] Antonio Dimeo, Felix Gohla, Daniel Goßen, and Niko Lockenvitz. "SoK: Multi-Device Secure Instant Messaging". In: (2021).

[9] Danny Dolev and Andrew Yao. "On the security of public key protocols". In: *IEEE Transactions on information theory* 29.2 (1983), pp. 198–208.

[10] Viktor Dukhovni. *Opportunistic Security: Some Protection Most of the Time*. RFC 7435. Dec. 2014. DOI: `10.17487/RFC7435`. URL: `https://rfc-editor.org/rfc/rfc7435.txt`.

[11] Hal Finney, Lutz Donnerhacke, Jon Callas, Rodney L. Thayer, and David Shaw. *OpenPGP Message Format*. RFC 4880. Nov. 2007. DOI: `10.17487/RFC4880`. URL: `https://rfc-editor.org/rfc/rfc4880.txt`.

[12] Hal Finney, Rodney L. Thayer, Lutz Donnerhacke, and Jon Callas. *OpenPGP Message Format*. RFC 2440. Nov. 1998. DOI: `10.17487/RFC2440`. URL: `https://rfc-editor.org/rfc/rfc2440.txt`.

[13] Daniel Kahn Gillmor. *Stateless OpenPGP Command Line Interface*. URL: `https://tools.ietf.org/html/draft-dkg-openpgp-stateless-cli-02` (visited on 03/11/2021).

[14] Simon Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. Oct. 2006. DOI: `10.17487/RFC4648`. URL: `https://rfc-editor.org/rfc/rfc4648.txt`.

[15] Auguste Kerckhoffs. *La cryptographie militaire, ou, Des chiffres usités en temps de guerre: avec un nouveau procédé de déchiffrement applicable aux systèmes à double clef*. Librairie militaire de L. Baudoin, 1883.

[16] Melvin Keskin. *Automatic Trust Management (ATM)*. XEP 0450. Version 0.1.0. XMPP Standards Foundation, Nov. 5–Dec. 15, 2020. URL: https://xmpp.org/extensions/xep-0450.html (visited on 03/01/2021).

[17] Melvin Keskin. *Trust Messages*. XEP 0434. Version 0.3.0. XMPP Standards Foundation, Feb. 15–Dec. 19, 2020. URL: https://xmpp.org/extensions/xep-0434.html (visited on 03/01/2021).

[18] Moxie Marlinspike. *Signal Blog: Safety number updates*. URL: https://signal.org/blog/safety-number-updates/ (visited on 05/31/2020).

[19] Robert C Martin. *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall, 2018.

[20] Robert C Martin. "Design principles and design patterns". In: *Object Mentor* 1.34 (2000), p. 597.

[21] Peter Millard, Peter Saint-Andre, and Ralph Meijer. *Publish-Subscribe*. XEP 0060. Version 1.19.1. XMPP Standards Foundation, Nov. 19, 2002–Mar. 4, 2021. URL: https://xmpp.org/extensions/xep-0060.html (visited on 04/07/2021).

[22] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. "This POODLE bites: exploiting the SSL 3.0 fallback". In: *Security Advisory* 21 (2014), pp. 34–58.

[23] Trevor Perrin and Moxie Marlinspike. "The double ratchet algorithm". In: *GitHub wiki* (2016).

[24] Mayank Raj, Krishna Kant, and Sajal K Das. "Energy adaptive mechanism for P2P file sharing protocols". In: *European Conference on Parallel Processing*. Springer. 2012, pp. 89–99.

[25] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342.

[26] Peter Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Core*. RFC 6120. Mar. 2011. DOI: 10.17487/RFC6120. URL: https://rfc-editor.org/rfc/rfc6120.txt.

[27] Peter Saint-Andre. *Serverless Messaging*. XEP 0174. Version 2.0.1. XMPP Standards Foundation, Feb. 7, 2006–Feb. 8, 2018. URL: https://xmpp.org/extensions/xep-0174.html (visited on 03/01/2021).

[28] Peter Saint-Andre, Joe Hildebrand, Sergey Dobrov, and Valerian Saliou. *Microblogging over XMPP*. XEP 0277. Version 0.6.4. XMPP Standards Foundation, May 7, 2008–Oct. 6, 2020. URL: https://xmpp.org/extensions/xep-0277.html (visited on 03/01/2021).

[29] Peter Saint-Andre, Peter Millard, Thomas Muldowney, and Julian Missig. *User Avatar*. XEP 0084. Version 1.1.4. XMPP Standards Foundation, May 7, 2003–Sept. 20, 2019. URL: https://xmpp.org/extensions/xep-0084.html (visited on 03/01/2021).

[30] Peter Saint-Andre and Kevin Smith. *Personal Eventing Protocol*. XEP 0163. Version 1.2.1. XMPP Standards Foundation, Oct. 24, 2005–Mar. 18, 2018. URL: https://xmpp.org/extensions/xep-0163.html (visited on 03/01/2021).

[31] Paul Schaub. *Stanza Content Encryption*. XEP 0420. Version 0.3.2. XMPP Standards Foundation, June 3, 2019–Mar. 4, 2021. URL: https://xmpp.org/extensions/xep-0420.html (visited on 04/07/2021).

[32] Florian Schmaus, Dominik Schürmann, and Vincent Breitmoser. *OpenPGP for XMPP*. XEP 0373. Version 0.6.0. XMPP Standards Foundation, Mar. 25, 2016–Nov. 22, 2020. URL: https://xmpp.org/extensions/xep-0373.html (visited on 03/01/2021).

[33] Florian Schmaus, Dominik Schürmann, and Vincent Breitmoser. *OpenPGP for XMPP: Instant Messaging*. XEP 0374. Version 0.2.0. XMPP Standards Foundation, Mar. 25, 2016–Jan. 25, 2018. URL: https://xmpp.org/extensions/xep-0374.html (visited on 03/01/2021).

[34] Robert W. Shirey. *Internet Security Glossary, Version 2*. RFC 4949. Aug. 2007. DOI: 10.17487/RFC4949. URL: https://rfc-editor.org/rfc/rfc4949.txt.

[35] Andreas Straub, Daniel Gultsch, Tim Henkes, Klaus Herberth, Paul Schaub, and Marvin Wißfeld. *OMEMO Encryption*. XEP 0384. Version 0.7.0. XMPP Standards Foundation, Oct. 25, 2015–Sept. 5, 2020. URL: https://xmpp.org/extensions/xep-0384.html (visited on 03/01/2021).

[36] Wenley Tong, Sebastian Gold, Samuel Gichohi, Mihai Roman, and Jonathan Frankle. "Why king george iii can encrypt". In: *Freedom to Tinker* (2014). URL: http://randomwalker.info/teaching/spring-2014-privacy-technologies/king-george-iii-encrypt.pdf.

[37] Alma Whitten and J Doug Tygar. "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0." In: *USENIX Security Symposium*. Vol. 348. 1999, pp. 169–184.

APPENDIX A

User Guide

Mercury-IM as presented in this work is a proof of concept XMPP client. There are some bugs and rough edges which encumber the user experience. For that reason this appendix shall act as a step-by-step user-guide to the application which lets the user experience the working of the app without having to deal with buggy behavior.

A.1. Building and Installing the App

Mercury-IM is built using gradle[1] and makes use of *composite builds*[2]. Unfortunately there is an issue[3] in Android Studio, preventing composite builds from properly functioning in later versions of the IDE. For that reason the development of Mercury-IM needs to be conducted using Android Studio 3.5.3 which can be retrieved from the Android Studio download archive[4]. Additionally, the Project Lombok[5] plugin is needed.

If the source code of the app is being retrieved via git, the following commands should be sufficient to clone the repository:

```
1 ~$ git clone git@zivgitlab.uni-muenster.de:ag-gorlatch/abschlussarbeiten/
     paul-schaub/mercury-im.git
2 ~$ cd mercury-im/
3 ~$ git submodule update --init --recursive
```

Listing A.1: Cloning the repository

The final commit made to the application repository before submitting the thesis is `2c97e00187bddb4bc2057cd40ebc68e85b7162ea`.

It is recommended to compile the app from Android Studio. Now the application can be installed on a device running Android 4.4 or newer.

[1] see `https://gradle.org/`
[2] see `https://docs.gradle.org/current/userguide/composite_builds.html`
[3] see `https://stackoverflow.com/q/61410231`
[4] see `https://developer.android.com/studio/archive`
[5] see `https://projectlombok.org/setup/android`

A.2. Initial Account Setup

In order to use the app, an account on any XMPP server is required. During development the app has mostly been tested on a private ejabberd[6] installation on a vServer.

Mercury-IM does not support registering of new accounts from within the app itself (In-Band-Registration), so the account needs to be registered though other means.

After the app has been started, click on the right-most icon in the bottom bar, labeled *Accounts*. This brings up the (now empty) account list. Mercury-IM is multi-account capable, so new accounts can be added by pressing the floating action button labeled with a plus sign. In the next screen the users XMPP Address (e.g. `juliet@capulet.lit`) and password need to be entered. The account is saved and logged into once the user confirms by pressing the *Login* button. If an error occurs (e.g. the user entered wrong credentials) an error message will be displayed, so that the user can re-enter their credentials.

Next the app asks the user whether they want to enable simplified key management with help of the IKEY protocol (see fig. 4.2 a)). If the user declines, the app will function as a normal XMPP client which uses OX for message encryption.

In our case we want to enable the IKEY feature, so the user is expected to press *Yes, Please*.

On a fresh account there should not be an existing identity key backup, so the app will generate a fresh key and display the fingerprint to the user (see fig. 4.2 c)). The user can now create an encrypted server-side backup of the secret key, which enables the use of that key on multiple devices. The server-side backup will be created once the user presses *Upload Backup*.

Now the app displays the backup passphrase, along with a QR code (see fig. 4.3 b)). This code can be used on a second device to restore the encrypted identity key. It should now be written down by the user (long pressing the backup passphrase will copy it into the devices clipboard). If the screen is closed, the passphrase and QR code can be shown again by going into the accounts details by clicking on the account in the account-list and clicking *Create Server Backup* below the identity key fingerprint.

A.3. Setup on a Second Device

Mercury-IM can be used with the same account on multiple devices. For best user-experience, the user should restore the secret identity key on the second device.

Once the account credentials are entered on the second device like above, the user will be asked to enter the backup password to restore the secret identity key on the new device (see fig. 4.3 a)). It is also possible to scan a QR code by clicking the QR-code button next to the text field. See above for how to display the backup code/QR code on the first device. After the identity key has been restored successfully, its fingerprint will be displayed to the user.

[6] see `https://www.ejabberd.im/`

A.4. Managing Devices

Other devices encryption keys are listed in the account details screen (see fig. 4.4 a)). This screen can be reached by clicking on an account from the account list screen.

Fingerprints of other devices are listed under the section labeled *Other Devices*. Here the user can decide whether or not they want to trust individual devices by flipping the switch next to the fingerprint (*on* means *trusted*, *off* means *distrusted*). Once the user made their decisions, they can publish a signed device list update by pressing the *Publish Decisions* button on the bottom of the screen.

A.5. Adding Contacts

New contacts can be added by navigating to the contact list screen by pressing the *Contact List* button in the bottom navigation bar and then pressing the floating action button with the *plus* label.

If more than one account are present on the device, the user is prompted to select to which account's contact list they want to add the contact. In any case they have to enter the contacts XMPP address and the press *Add*.

Note that for the IKEY feature to work, both accounts need to add another mutually.

The added contact should now show up in the contact list. The contact details screen (see fig. 4.4 c)) can be opened by clicking on any contact. This screen shows fingerprints and other information (if not, see appendix A.7). It is also possible to change the contacts nick name by clicking on it. A conversation with the contact can be started by pressing the floating action button with the speech bubble icon. Once a conversation has been started, it can also be accessed by going to the chat list by pressing the *Chats* button in the bottom navigation bar and selecting the contact.

A.6. Sending Messages and Trusting Contacts

Once a conversation is opened, the user can enter a message just like in any other messenger. Note though, that in order to send a message, the user first has to trust the contacts identity key. That can be accomplished by navigating to the contact details screen (e.g. from the contact list screen or by clicking the contacts name in the chat view) and flipping on the switch next to the contacts identity key fingerprint (see fig. 4.4 c)). Once the identity key is toggled on, navigate out and back into this view to trigger an update of the device keys trust state. The app will now display yellow locks next to fingerprints which are trusted via the IKEY feature. If no fingerprints are shown, see appendix A.7.

A.7. Bugs

There is a bug with device keys showing up on other devices. In such a case it might be worth to try the following steps in order to fix this issue:

1. Navigate out of and back into the contact/account details screen

   The list of fingerprints does not update in real time, so causing it to be loaded again can make the new fingerprint show up.

2. Logging out of the accounts on both devices and back in

   For some reason, PubSub items are sometimes not being delivered to other devices. Logging out and back in may solve this. Logging out and back in can be done either gracefully via the switch next to the account in the accounts list, or forcefully by killing the app.

3. Make sure that both accounts added another as contacts

4. If still no fingerprints show up, go to account details and click *Publish Decisions* or select *Publish Ikey Element* from the overflow menu in account details.

   Combine this with logging out/back in on both devices and maybe force-killing the app until fingerprints show up.

It might happen that a sent message shows up multiple times on the receiving device. If that happens, try force-killing and restarting the app on the receiving device.

It might also happen that a device does not receive messages at all. The cause might either be that the device is not trusted by the sender (see appendix A.6), or a bug. In the latter case, try force-killing the app.

Eigenständigkeitserklärung

Hiermit versichere ich, dass die vorliegende Arbeit mit dem Titel *Towards Secure Key Verification Mechanisms in End-to-End Encrypted Multi-Device Instant Messaging* selbstständig von mir und ohne fremde Hilfe verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind. Mir ist bekannt, dass es sich bei einem Plagiat um eine Täuschung handelt, die gemäß der Prüfungsordnung sanktioniert werden kann.

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in einer Datenbank einverstanden.

Ich versichere, dass ich die vorliegende Arbeit oder Teile daraus nicht anderweitig als Prüfungsarbeit eingereicht habe.

Münster, July 9, 2021

(Paul Moritz Schaub)