

# **SoK: Multi-Device Secure Instant Messaging**

Antonio Dimeo, Felix Gohla, Daniel Goßen, Niko Lockenvitz  
{antonio.dimeo, felix.gohla, daniel.gossen, niko.lockenvitz}@student.hpi.de

Hasso Plattner Institute, University of Potsdam

April 17, 2021

## **Abstract**

The secure multi-device instant messaging ecosystem is diverse, varied, and under-represented in academia. We create a systematization of knowledge which focuses on the challenges of multi-device messaging in a secure context and give an overview of the current situation in the multi-device setting. For that, we analyze messenger documentation, white papers, and research that deals with multi-device messaging. This includes a detailed description of different patterns for data transfer between devices as well as device management, i.e. how clients are cryptographically linked or unlinked to or from an account and how the initial setup can be implemented. We then evaluate different instant messengers with regard to relevant criteria, e.g. whether they achieve specific security, usability, and privacy goals. In the end, we outline interesting areas for future research.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Group Messaging vs. Multi-Device Messaging . . . . .	4
1.2	Methodology . . . . .	4
<b>2</b>	<b>Multi-Device Messaging</b>	<b>7</b>
2.1	Context . . . . .	7
2.2	Transferring Data Between Different Devices of One User . . . . .	7
2.2.1	Storing Data on a Server . . . . .	8
2.2.2	Using Messages to Exchange Data . . . . .	9
2.3	Transferring Data to a Different User . . . . .	11
2.3.1	Without End-to-end Encryption . . . . .	11
2.3.2	End-to-end Encryption With Shared Group Key . . . . .	13
2.3.3	End-to-end Encryption Per Recipient . . . . .	13
2.3.4	End-to-end Encryption Per Device Pair (Client Fanout) . . . . .	14
2.3.5	End-to-end Encryption Per Sending Device (Sender Keys) . . . . .	14
2.4	Messenger Overview . . . . .	15
<b>3</b>	<b>User Device Management</b>	<b>20</b>
3.1	User and Device Identity . . . . .	20
3.2	Adding More Devices . . . . .	21
3.2.1	Concepts per Messaging Pattern . . . . .	21
3.2.2	Cryptographically Linking a New Device to an Account . . . . .	24
3.2.3	Methods to Exchange Required Information . . . . .	28
3.2.4	Synchronizing Old Messages . . . . .	31
3.3	Removing Devices . . . . .	31
3.4	Recovering Accounts, Messages, and Verifications . . . . .	32
3.5	Security Considerations . . . . .	33
3.6	Messenger Overview . . . . .	36
<b>4</b>	<b>Future Work</b>	<b>40</b>
<b>5</b>	<b>Conclusion</b>	<b>41</b>
	<b>References</b>	<b>42</b>

# 1 Introduction

Since the early 2000s, the ways that people communicate have changed dramatically. One of those changes has come with the advent of encrypted Instant Messenger (IM) applications, which make end-to-end encrypted communication ubiquitous, cheap, and easy. Anecdotally, some people want to be able to have the same conversation on multiple devices, which we define as a messaging client (device) the user can use to interact with the messaging system (e.g. App, Browser, CLI, etc.). This additional requirement has led to the creation of “Secure Multi-Device Messaging”, which we define as programs that send end-to-end encrypted messages between users, who may interact with those messages from more than one device each.

These multi-device messengers differ in how they achieve their corresponding definition of security. We consider these differences significant enough to warrant additional attention. Many of the implemented solutions were created as part of commercial products and the number of academic publications on the topic is fairly limited.

Therefore, our work also incorporates white papers of IMs explaining their security models, evaluates the security properties of the corresponding protocols as well as giving an overview of the approaches that are used. This includes the used techniques for exchanging necessary cryptographic keys, synchronizing messages, and device management.

Nevertheless, we identified three academic papers of interest. Unger et al. [43] created a systematization of knowledge (SoK) on secure messaging in 2015. The messenger market being very fast-paced, a lot of new technologies and IMs have evolved since then, the Double Ratchet protocol by Marlinspike [26] making end-to-end encryption (E2EE) readily available to the public being just one of them. Therefore, the SoK by Unger et al. is not up-to-date in all aspects anymore. Although the topic of multi-device is addressed, it is no longer appropriate in the context of the increasing number of devices that users own today. Atwater and Hengartner [3] describe a way of distributing keys among multiple devices for cryptographically secure operations by using threshold cryptography, which tackles the problem but also introduces complexity for the user. Champion et al. [7] propose a different mechanism for using secure instant messaging on multiple devices built on top of already existing protocols, trying to scale better than existing protocols like the Sesame protocol [27]. With little current literature being available, it is hard to quickly gain an overview over the current state of the art.

We hope this systematization of knowledge can remediate the situation somewhat and simplify understanding the secure multi-device messaging ecosystem in early 2021.

## 1.1 Group Messaging vs. Multi-Device Messaging

Thinking about multiple devices of one user, the scenario does not seem to be different from group messaging at the first glance: There are multiple devices participating in one chat, they need to exchange messages and could send them from one to another, except that more than one device belongs to the same user. But on closer inspection, problems stand out that do not occur in the group messaging setting. First, devices that belong to the same user should also be perceived as belonging to the same user. This problem is not solved by simply using group messaging in all cases of the protocols and techniques that will be presented. Second, the usage of group IM protocols could disclose metadata in terms of the number of devices that are present for each user. Third, group messaging does not solve the authorization for adding or removing devices. Another protocol is needed in order to be able to add or remove devices to or from all conversations of a particular user who decides, which devices may manage other devices of the same user. It also needs to ensure that—opposed to standard group messaging—the corresponding operations to devices are applied to all conversations of the user, as users probably want to be able to read and write messages on all devices. Fourth, the administration operations addressed of authenticating devices of one user to other devices of the same user has a different security model than in group messaging, as “one can easily assume the devices will be physically close at some point” [7]. And last, whereas users have a high probability for coming online regularly, the same does not necessarily apply to all devices, as they can be offline for a very long time. This has implications on the security properties of the protocols, like the time an (encrypted) message must be stored for a device being offline.

## 1.2 Methodology

We first identified different IMs that might be relevant to our report. Since many similar messengers exist, we limited our analysis to messengers of some notability. Therefore, we only considered IMs listed on the Wikipedia list “Comparison of cross-platform instant messaging clients” [12]. We then pruned the resulting collection of IMs, removing all messengers that do not implement E2EE and multi-device functionality, as well as those we believed to be discontinued. Furthermore, we excluded IMs that do not provide a documentation of their E2EE and multi-device implementation. For the remaining messengers we tried to gain an understanding of how they implemented their multi-device functionality by reviewing academic and non-academic literature and media, such as relevant papers, white papers, talks, and protocol definitions. For some IMs we also examined the functionality and implementation ourselves. During this stage, we also reviewed additional protocols that we were aware of that had been proposed, but not implemented yet. We then tried to classify the protocols according to common properties and evaluated them under a number of security goals.

We had a closer look at the following IMs and their respective protocols and communication methods:

**WhatsApp** First released in 2009, WhatsApp is one of the most used messengers in the world with more than 2 billion registered users as of February 2020 [46]. The company was bought by Facebook in 2014 [13].

**Facebook Messenger** Facebook Messenger is the chat client used for communication between members of the Facebook Platform. In January 2021, there were 1.3 billion users using the Facebook Messenger App actively on a monthly basis [36].

**Signal** Signal was developed by a development group called Open Whisper Systems since 2013 [25]. As an open source project, the source code is hosted on GitHub<sup>1</sup>

**Element** Element (known as Riot before July 2020) is an open source messenger<sup>2</sup> that implements the Matrix protocol and was first released in 2016. Because of its federated architecture, users can host their own servers.

**Threema** The Swiss company Threema created the homonymous messenger in 2012 [40]. Since then, it has advertised end-to-end encryption and a high level of data protection and complete anonymity while using the app. The (web) apps are open source software<sup>3</sup> the server components however are currently proprietary and cannot be audited by the public.

**Telegram** Telegram was first released in 2013 [38]. The messenger, originally developed in Russia, also offers an open platform for creating bots and other clients in addition to a custom encryption protocol. The source code for some of the apps is publicly available<sup>4</sup> but the server code has not been published. As of January 2021, Telegram has around 500 million active users per month [36].

**iMessage** Integrated into the Messages app of most of Apples operating systems, iMessage is a proprietary IM only available on Apple devices. No source code is publicly available.

**Keybase** Mostly known for being a public key directory, Keybase also provides an end-to-end encrypted messaging platform, with clients available for most platforms. In 2019, Keybase reported about 400,000 users. Keybase was acquired by Zoom in 2020 [19].

**Skype** First published in 2003, then acquired by eBay in 2005 and by Microsoft in 2011, Skype has supported optional end-to-end encrypted chats since 2018. Skype originally featured a peer-to-peer architecture, but this was later removed [34].

---

<sup>1</sup>Signal source code: <https://github.com/signalapp>

<sup>2</sup>Element source code: <https://github.com/vector-im>

<sup>3</sup>Source code for the Threema apps: <https://github.com/threema-ch>

<sup>4</sup>Source code for the Telegram apps and libraries: <https://github.com/TelegramMessenger>

**Viber** With more than 260,000 reported active users, Viber is mostly popular in Eastern Europe, Russia, the Middle East, and some Eastern Markets [44].

**Wickr** Wickr was founded in 2012, and has focused on security and privacy since inception. Wickr published a white paper explaining its encryption in 2015 and published one of its crypto protocols in 2017 [50].

**Wire** Wire was founded in 2014, with many of its founding members coming from Skype. Unlike many competitors, Wire focuses mainly on the enterprise market. Since 2016 communication via Wire is end-to-end encrypted per default and source code for its clients has been released [51].

As we focused on IMs with good notability, we did not consider all messengers that claim to have multi-device support in some way. For the messengers in focus, we first want to evaluate criteria regarding messaging in a multi-device context (see [chapter 2](#)) and after that criteria regarding managing multiple devices, i.e. clients, of one user (see [chapter 3](#)). The criteria being evaluated can be classified into three different categories:

### **Security**

- whether a method provides Forward Secrecy and Post-Compromise Security
- how linking of devices is achieved

### **Usability**

- the time complexity for encrypting messages
- whether the devices can independently send messages (without the help of other devices)
- how information is exchanged during device linking
- whether old messages are decryptable on new devices
- whether accounts, messages, or user verifications can be recovered

### **Privacy**

- whether the devices need to be known by the server
- whether the devices are known to other users

## 2 Multi-Device Messaging

### 2.1 Context

In this chapter we describe the protocols and procedures that are necessary to actually transfer data between different devices in a multi-device messaging context. Data in this context can mean any application-specific data and is not solely restricted to messages being exchanged actively by a user (we call them user messages). In addition to user messages, other data, such as read receipts, status messages, and metadata, is also part of the data transmitted between devices. We call those meta messages.

We differentiate between communication between devices that belong to one user (*same-owned devices*) and devices that belong to different users. For the latter case, we describe the communication between two users with multiple devices only since the communication between more users is not substantially different from a protocol perspective for the protocols we have examined.

At this point, we assume that both users who want to communicate have a set of trusted devices. How these devices gain and lose their trusted status is described in [chapter 3](#). Furthermore, we assume that there is a central service (that is possibly aware of all devices) and all devices know how to interact with that central service. We define central service as a service that can be provided by central infrastructure of the messaging service provider or by a group of decentralized servers. The role of this service depends on the underlying security architecture. The devices are also aware of each other if required by the corresponding protocol.

### 2.2 Transferring Data Between Different Devices of One User

Some protocols used for multi-device messaging transfer data between same-owned devices. It should be noted that not all protocols deal with data transfer between same-owned devices, *OpenPGP* being a well known example for a protocol that does not [6]. Lesser known examples for protocols that do not synchronize data between same-owned devices include the *Surespot* [35] and *ADAMANT* messengers [1].

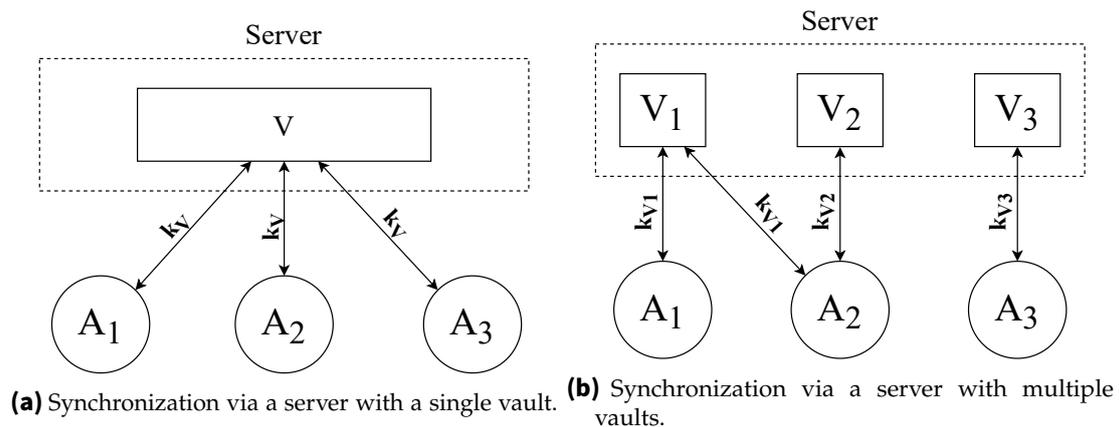
In *OpenPGP*'s case, synchronization between devices is considered out of scope, and features such as synchronized read receipts are only available because the surrounding environment, in this case *IMAP*, provides them. The downside is that these additional features may be designed insecurely. If the surrounding environment does not provide those features, they simply do not exist. This is the case with *ADAMANT*, which does not track which messages a user has seen beyond a single client and thus lacks a feature that is quite common for other messaging services.

For those protocols that do transfer data between same-owned devices, a common use case is the synchronization of metadata, such as contact books, read receipts, or the transfer of keys used for decrypting incoming data. Protocols that follow the reflection pattern (see subsection 2.2.2) also transfer received messages and, depending on the specific implementation, messages that should be sent between devices that belong to one user.

We do not believe that the different patterns for transferring data between trusted devices, when E2EE is not necessary, require in-depth explanation. For protocols that leverage E2EE and data transfer between same-owned devices, two patterns exist: The first pattern is to encrypt the data and store it on a server while the decryption keys are known to authorized devices only. These devices can then access or update the data. The second pattern is to encrypt and transfer the data directly between devices.

### 2.2.1 Storing Data on a Server

For the simplest form of the first pattern, only one key exists, which is known to all devices and can decrypt all data that should be shared. We believe this is how *ProtonMail* synchronizes contacts between devices [30]. *XMPP-OX*, an extension to the *XMPP* protocol that uses OpenPGP to achieve E2EE, also achieves multi-device functionality by encrypting the OpenPGP key of the user and storing it on a server that only allows authorized clients to access it.



**Figure 2.1:** Sharing data between same-owned devices via a server.

While this approach allows for the synchronization of arbitrary data without restrictions and limits the number of encryptions that need to occur to share a piece of data, it also allows an attacker access to all data of all devices if only a single device is compromised. Additionally, to guarantee that a device you once trusted cannot continue to decrypt the shared data, you need to either guarantee that the relevant key was deleted from the device or re-encrypt all data with a different key that needs to be shared among all other devices.

The *Matrix* protocol proposes to use multiple keys instead, with each key granting access to a subset of the data [9]. These subsets may be overlapping, so that different

keys can access different instances of the same data. If a device wants access to a piece of data it does not yet have access to, it queries all other devices for a device that is willing to share the corresponding secret. That device then encrypts the data for the requesting device and transfers it directly. Since keys do not need to be known to all devices of a user, this approach allows for more selective sharing. If a key is breached or needs to be deprecated, the blast radius can be much smaller if the data it has access to was appropriately selected.

### 2.2.2 Using Messages to Exchange Data

As already mentioned, the second pattern is the direct transfer of data between trusted devices of one user. “Direct” does not imply that the communication is not routed through a server, but it does imply that the main task of the server is to organize data delivery, as opposed to storing the data. A popular approach, used by the messengers *Wickr*, *Wire*, *Viber*, *Signal* and presumably other messengers that implement the Signal protocol, and the XMPP extension OMEMO, is to not differentiate between devices of one user and those of other users, but to just treat data transfer exactly the same for both cases.

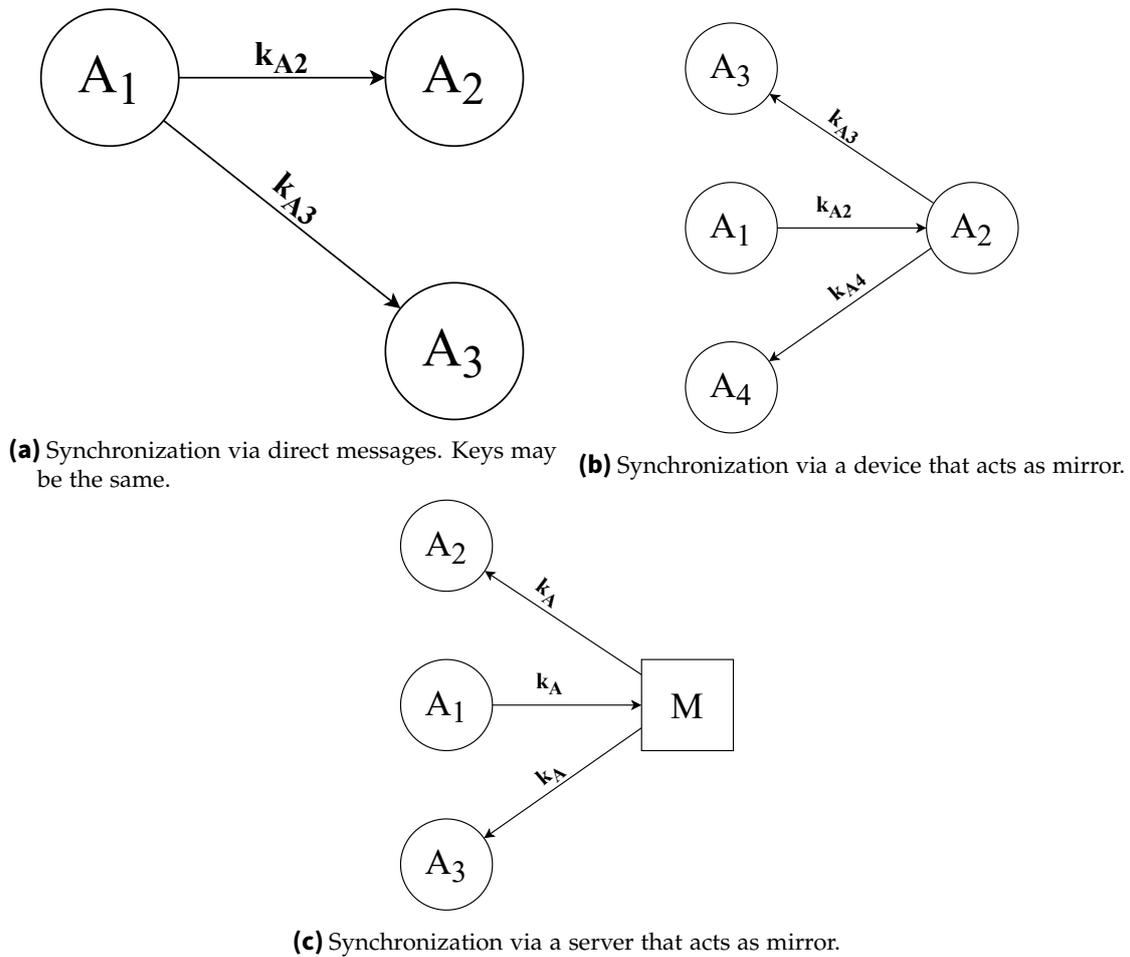
The outgoing data is encrypted for the target devices. If the protocol dictates unique keys per device, the data is encrypted once per device but it is also possible to use unique keys per user instead and to encrypt the data once per user. The originating device then sends the data to all devices it believes should have access to the data. Thus, the same data may be sent to devices of the same user as well as devices of other users in the same transaction. It should be noted that this does not imply that the data exchange between devices of different users and devices of one user looks the same, since additional types of data may be transferred between devices of one user.

The main upsides of this implementation are that data sharing between same-owned devices can sit on top of the default message sending infrastructure and that the mapping of one key per device without data stored on a server in the long-term makes key deprecation as easy as stopping to encrypt data with that specific key. However, the number of necessary encryptions scales linearly with the number of devices. This might pose a problem for some workloads, especially if data needs to be shared with same-owned devices and a large number of devices that belong to different users at the same time.

A different option is the so-called reflection pattern. With the reflection pattern, not all devices are treated equally. A single device acts as a coordinator and is responsible for distributing or mirroring the data to all other devices. This coordinator is often static, in the sense that the same device is the coordinator every time. The coordinator builds a channel to all non-coordinator devices. This channel, which is usually either a WebSocket or WebRTC connection, may be a direct peer-to-peer connection or via a server that knows how to reach all devices. If a non-coordinator device wants to transfer data to other same-owned devices, it sends that data to the coordinator device via the established channel. The coordinator device then sends the data to all same-owned devices. If the coordinator itself wants to transfer data to other same-owned devices the first step is skipped. This option is used by both Threema and WhatsApp

for their web clients. The biggest usability challenge with this approach is that the coordinator needs to be online when other devices want to share data. This, combined with the static nature of the coordinator, limits the usability of non-coordinator devices somewhat.

A variation of this pattern, that as far as we know has not been implemented yet, was proposed by Threema [41]. For this variation, the coordinator is not a device but a server and all devices share the same group key, which is used to encrypt and decrypt messages from and to other devices. Since the coordinator is a server and not a user device, this variation eliminates the original weakness of one device needing to be online when synchronizing data between other devices. Additionally, as the coordinator is not a device owned by the user, it does not have access to this group key and thus does not have access to the shared data.



**Figure 2.2:** Different message sending patterns between different devices of one user.

## **2.3 Transferring Data to a Different User**

This section deals with sending data from one user's device to one or more devices of a different user. There are many variations in how a message can be sent to another user depending on the underlying security architecture. Without E2EE, users have to trust all intermediate services of the messenger, which cannot be assumed with the use of publicly available messenger services for end consumers. The needed trust in the server depends on the communication. A large public group possibly could not require encryption at all, as the content is available to the public. Only message authenticity is necessary in that case. Using E2EE, messages are encrypted with a key unknown to the server, which is why they cannot be read by the server without first breaking the encryption. After encrypting a message with some kind of E2EE key, there are multiple ways for distribution. Nevertheless, often an intermediate (central) service is used to buffer the encrypted user messages until at least one device of the recipient has fetched the message or a specific timeout is reached. In a peer-to-peer implementation, messages can also be buffered by a device of the sending user (instead of by a server) until the message is delivered.

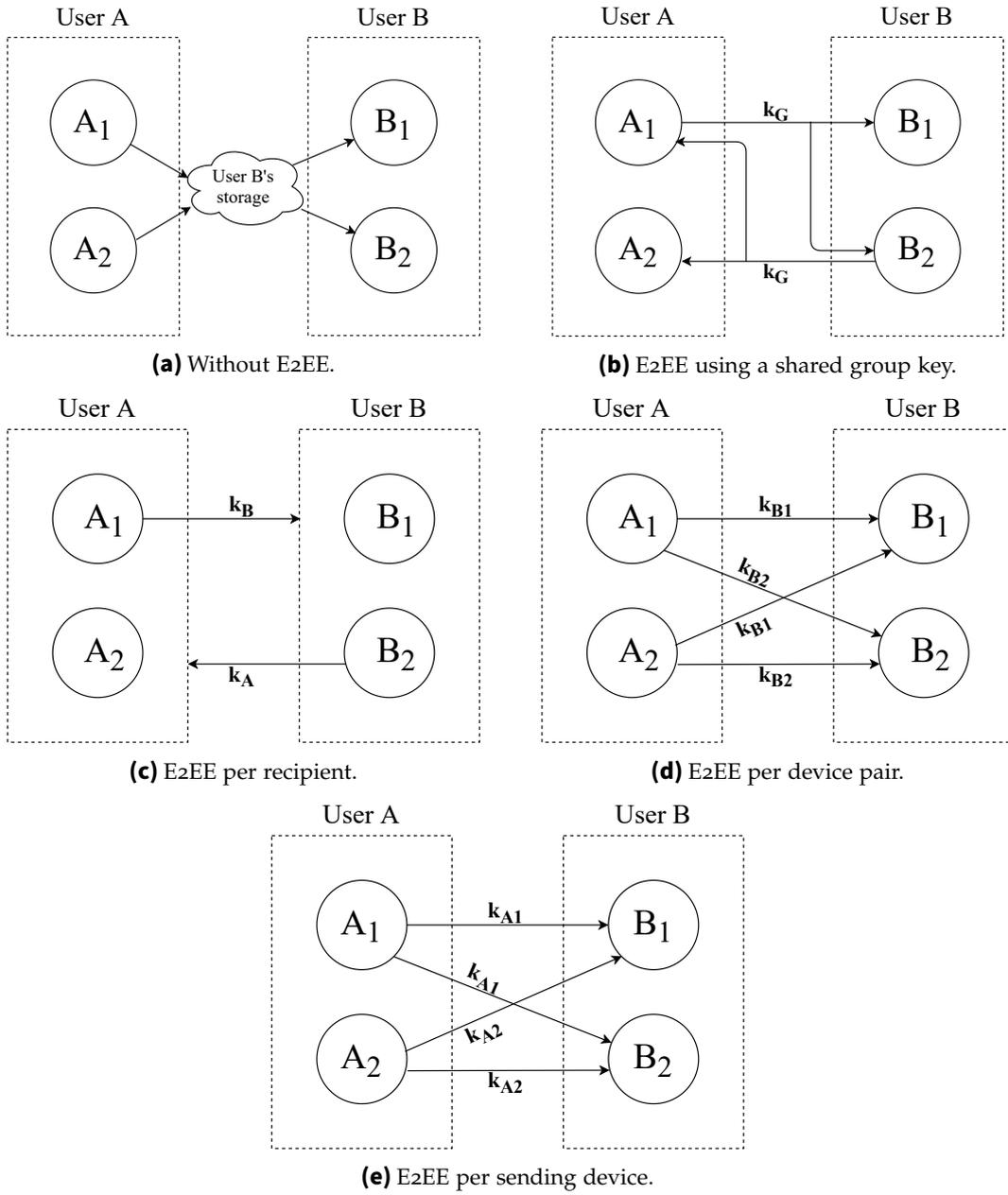
As the distribution of messages without E2EE does not fulfill our definition of secure, because all intermediate services would have to be trusted ones, the focus will be on methods that leverage E2EE. The first option is a shared key that is known to all participating devices and can be used for E2EE. For the second option, messages are encrypted per user with the corresponding receiving user's key. For the third option, a secure channel with separate keys exists between each device pair of the communicating users. Fourth and last, the sending device can encrypt the message with its own key and all receiving devices know that key for decryption. The different patterns we identified are shown in Figure 2.3.

### **2.3.1 Without End-to-end Encryption**

Not using E2EE allows very simple ways of distributing messages. After creating a message with the corresponding metadata, there are two possibilities for sending it.

Often, a central service is the heart of the messenger's infrastructure. It can be seen as a trusted central server. These central servers have the task of receiving sent user messages and storing them in (often persistent) memory for being fetched later on. Some meta messages like read receipts can also be stored on the server, whereas live meta messages only reflect a current status like the other user being online or typing. Any of the recipient's devices can fetch these messages from the server after authenticating.

Alternatively, they can be sent to the receiving devices via a server-push architecture. As a result the messages not being end-to-end encrypted, the user messages can be accessed as plaintext and the server is able to read the content. Nevertheless, saving the messages on the server in plaintext helps in achieving good usability as messages can be easily synchronized between same-owned devices.



**Figure 2.3:** Different message sending patterns between two users with two devices each.

The *Telegram* messenger service uses this pattern for *Cloud Chats* and *Public Chats* [39], the same applies to Facebook Messenger's and Skype's chats that do not utilize E2EE (also see subsection 3.2.4).

### 2.3.2 End-to-end Encryption With Shared Group Key

Assuming that E2EE is a desired feature, the process of sending a message is slightly different as messages need to be encrypted for specific recipients. One way of encrypting messages is using a shared key that is known to all devices partaking in the communication. Therefore, a method for agreeing on and exchanging that particular key or key material is necessary. A sending device can then encrypt messages with a shared key once. If required by the protocol, the message can also be authenticated either by a key that is associated to the sending user's account or via a key that is owned by a specific device and is also associated with the account. The distribution can be the responsibility of the server, so that a sending device can send the message once along with an identifier of the receiving user (server-side fanout via user identity). Note that this is optional, if it is required for the server not to know metadata like user-device associations. In that case, targeting a user's account as identity instead of the user's devices directly, without the server knowing any metadata, requires the receiver to have a device that receives messages and synchronizes them among the other same-owned devices, e.g. via the reflection pattern (also see subsection 2.2.2 and subsection 2.3.3). Otherwise, the sender can provide a list of desired receiver device identifiers.

Shared group keys do not provide Forward Secrecy (FS) or Post-Compromise Security (PCS) per se. Breaking the key that is used for encryption does not only enable to decrypt messages of one user but the messages of all users in that conversation. Thus, a method for creating and exchanging new entropy is required in order to change keys on a regular basis in order to achieve FS and on demand to achieve PCS. This exchange can happen in different ways, including pairwise channels (see subsection 2.3.4) or targeting subgroups. As an example, the *Messaging Layer Security (MLS)* protocol allows for an efficient key rotation algorithm that scales logarithmically for a large number of partaking devices [5, 24].

### 2.3.3 End-to-end Encryption Per Recipient

In the following method, the sender is given a key for encrypting the message for the recipient. We distinguish different methods for a device to decrypt the messages. On the one hand, same-owned devices can simply share a key among each other. In this case, each of the same-owned devices can decrypt messages with that key. In its simplest form, this requires a long-term valid key. Therefore, no FS can be achieved in this case, which is why this approach is rarely used in practice. One example that uses a long-term valid key is *XMPP OX* (XEP-0374) [32].

A more elaborate protocol was proposed by *Campion et al.* [7] and is based on the *Signal* protocol. It takes care of synchronizing the internal state of the necessary ratchets for key derivation via a secure multicast channel between same-owned devices.

Thus, each device can encrypt and decrypt using the same key material. The receiving party needs to synchronize the per-user-ratchet too. As a result, the receiver is unable to distinguish which device sent the message, which can be a desired privacy feature. FS is achieved by the properties of the Signal channel and the Double Ratchet mechanism that creates new entropy [26].

On the other hand, sharing a long-term key is a potential security risk, e.g. when a device is compromised. This can be solved by declaring one of the recipient's devices as a *primary device*. The primary device will receive the messages sent by the sender first. It is able to decrypt the messages and has the task of synchronizing the other devices, as being described in section 2.2. Some protocols use a primary device for both sending and receiving messages. Thus, in that case all messages from secondary devices need to be transferred to the primary in a secure way first, before being end-to-end encrypted for the receiver. This approach is currently being used by *WhatsApp* and *WhatsApp Web* so that exactly one secondary device will be able to receive or send messages [16].

### 2.3.4 End-to-end Encryption Per Device Pair (Client Fanout)

Instead of having one key per user, it is also possible to use one key (pair) per device. Every message must be encrypted and sent once per recipient's linked device. Therefore, this approach sometimes is also called *Client Fanout* [31] because instead of server-side distribution, the sending client is responsible for encrypting and sending data to every recipient's device individually. The concept of Client Fanout can be used for encrypting group chats but it can also be applied to the multi-device context as messages are sent to all recipient's devices like it is the case in a group chat.

This can still happen via a (central) service that stores the messages in a queue for being received later on. But in comparison to other techniques, the server is not responsible for finding the right recipient's devices directly. The sending device has to explicitly name the receiving devices. As a result, the sending device needs to know all encryption keys of the receiving devices and therefore the number of existing devices. The encryption effort and the amount of data being sent over the network scales linearly with the number of recipient devices [11].

One widespread protocol using that technique is Signal's *Sesame* protocol which was introduced in 2017 [27, 7]. It is in turn implemented into many other messengers like Facebook Messenger for optional secret chats [14] and Wire [7]. Here, a Signal channel is established between every device to ensure E2EE and all the security properties of the Signal protocol itself, i.e. mainly FS and PCS.

### 2.3.5 End-to-end Encryption Per Sending Device (Sender Keys)

The following method describes E2EE using a shared channel between multiple (receiving) devices. In comparison to the first approach mentioned in subsection 2.3.3, each sending device has its own encryption key for that particular conversation. In the group-messaging context, this is also called *Sender Keys*. When establishing the conversation, the key of participating device is distributed to all other receiving devices, e.g. via a secured pairwise channel as outlined in subsection 2.3.4 which can be

temporary. After the initial exchange of keys, the sending device needs to encrypt each message only *once* with its own key. Thereafter, the ciphertext can be sent to a server for distribution (server-side fanout) or to the receiving devices directly. These devices in turn can decrypt the message with the key shared beforehand.

The shared key in this case could also be derived from a shared ratchet-state in order to achieve FS by ratcheting the state each time a message from a particular device is received.

A shared secured channel as such is used in the Matrix Megolm protocol specification. Megolm<sup>5</sup> relies on the Olm “peer-to-peer encryption” protocol in order to exchange a ratchet state that is used to derive symmetric encryption keys per sending device [4].

## 2.4 Messenger Overview

The following section shall give a comparison of the evaluated messengers regarding the criteria related to multi-device messaging. The results were primarily taken from our experiments and official security white papers of the corresponding messengers [14, 18, 26, 27, 28, 29, 42, 45, 47].

**End-to-end Encryption (E2EE)** Like stated in the introduction, only messengers that support E2EE in some way were evaluated like shown in Table 2.1. There are three messengers that do not provide E2EE by default: Facebook Messenger, Telegram, and Skype. Whereas Facebook Messenger uses the Signal protocol for Secret Conversations [14] and synchronizes the messages sent to all devices of the sender and receiver, Telegram’s Secret Chats are not synchronized across devices. Skype uses the Signal protocol too but it does not synchronize the messages of Private Conversations across multiple devices.

**Forward Secrecy (FS)** Forward Secrecy in our model describes the property of a protocol to be resistant against *passive* attackers decrypting past messages despite having gathered keys at a particular point in time. In our definition, this comprised all keys of one device, e.g. an identity private key or message encryption keys. Passive in this case means that no impersonation attacks are executed.

Messengers that use the Signal protocol utilize the Double Ratchet Mechanism [26] to derive a new encryption key *for each message* with a one-way function. After a new key is derived via the key derivation function, the old key cannot be recovered easily and efficient attackers cannot recover the old keys for previous messages. This is done across all pairwise channels being used. Viber uses an approach very similar to Signal’s but the implementation is entirely separate [45].

The Matrix protocol and therefore the Element messenger uses the Megolm protocol to exchange messages with a Sender Keys configuration as described in subsection 2.3.5. This key for a specific message – like in the Signal protocol – is also derived from the

---

<sup>5</sup>An overview of the algorithm can be found in the Matrix project’s GitLab repository: <https://gitlab.matrix.org/matrix-org/olm/-/blob/4bae4134eb115859038de4e8eab11b22baf80b0c/docs/megolm.md>

previous key via a key derivation function (on all sending and receiving devices instead of every device pair). But unlike in the Signal protocol, Matrix proposes to store the “earliest value of the ratchet” [17] in order to be able to decrypt past messages by sharing the ratchet state with another or new device. Therefore, an attacker having compromised that state is able to decrypt all following messages by deriving the following keys. The Megolm standard proposes that applications should provide a way to delete the old ratchet states, which is why we evaluate this as partial FS.

iMessage does not achieve FS, because it uses a per-message symmetric encryption key that “is encrypted using RSA-OAEP to the public key of the receiving device” [2]. We assume that this key pair is static, as it is generated on the activation of iMessage on a new device. If the private key is compromised, the per-message key of captured messages and thus the messages can be decrypted by a passive attacker.

The protocols used by Threema and Telegram do not leverage a ratcheting algorithm or a comparable mechanism, so the same key is used for encrypting multiple messages. Threema only achieves FS via transport encryption [42], thus FS is not achieved against an attacker who has access to the servers used by Threema. We rate this as no FS being achieved.

Telegram’s protocol allows for, and official Telegram clients require, regular rotation of the communication keys used for secret chats, either after 100 messages or after one week. The previously used keys are then deleted [37], which achieves FS according to our model. However, compared to Signal, the number of messages that an attacker, who has access to one key, can decrypt is much greater.

Keybase stores a symmetric encryption / decryption key that is shared with new devices in order to decrypt old messages [20], so it does not provide FS according to our definition.

**Post-Compromise Security (PCS)** Post-Compromise Security in our model is defined as an attacker not being able to decrypt future messages after having compromised keys. This is often achieved by introducing new entropy and creating new encryption keys as a “healing event”.

The Signal protocol and messengers using it do this by creating a new ratchet state essentially every message roundtrip, i.e. each time both sides of a conversation send a message, new keys are generated, a new key exchange is performed, and a new secret is used for the double ratchet mechanism [26]. A passive attacker who has compromised a set of message encryption keys can therefore only decrypt the messages up to the end of the current roundtrip. Viber uses the same approach and PCS is achieved here too.

For iMessage, we did not find a way to rotate the keypair used for encrypting messages, therefore there is no PCS according to our definition if a passive attacker gathered the private key.

Keybase introduces a healing event by rotating the symmetric group encryption keys for all conversations channels when a device is removed from an account. This is done by generating new symmetric encryption keys and sending them to all remaining same-owned devices and corresponding devices of other users for each communication

channel. Hence, a passive attacker cannot decrypt messages in the future, if any device is removed. As this does not happen automatically, and may not happen at all, we do not consider PCS as fully achieved, even though this removal of a device can be viewed as a “healing event”.

Threema does not have such an algorithm, as messages are only encrypted with a static key exchanged once via a Diffie-Hellmann key exchange when the communication is established [42].

Because the Telegram messenger rotates the keys periodically in order to achieve FS it also attains PCS.

**Sending Complexity** The time complexity of sending messages depends on the underlying protocol and messaging pattern. Messengers that share one identity for encrypting messages across all devices (see subsection 2.3.3), like Threema and WhatsApp, only need to encrypt the messages to be sent once. Those messengers that use encryption per device pair (see subsection 2.3.4) however have to encrypt each message for every receiving device and therefore have time complexity of  $\mathcal{O}(n)$  for sending a message. This does not apply to messengers that use the Signal protocol but do not support E2EE for multiple devices like Skype. Element as an example for the Sender Keys pattern (see subsection 2.3.5) or Keybase with a shared group key (see subsection 2.3.2) only need to encrypt the message once for all recipients – with the sender key or group key respectively.

**Independent Sending** Independent sending in our context means that devices can send messages to other users (or same-owned devices) without the help of other devices that need to be online. With some approaches like the primary device and reflection pattern, this is not possible as one device plays a major role in representing the user to servers or other users. As already described in subsection 2.2.2, all non-primary devices send their messages to the primary first. If this primary device is not available, no message can be sent. The current WhatsApp and Threema web clients, for example, cannot be used without the primary (phone) being online. To tackle that problem, Threema describes a new approach with a mediator server and changing primary devices [41]. In that case, every device will be able to send messages if the mediator server is available.

Other architectures are not dependent on a single primary as they utilize encryption identities per device. They therefore do not need to exchange messages with a primary but can encrypt messages to other users on their own. This applies to Facebook Messenger, iMessage, Signal, Viber, Wickr, and Wire.

As Telegram only supports E2EE between specific device pairs and otherwise does not encrypt messages to other users via E2EE, Telegram can also be seen as being able to send messages independently.

In Element, and the corresponding Megolm protocol, every device in a conversation has its own sender key and therefore can encrypt and sign messages without other devices. The same applies to Keybase, with the difference that here the group-wide key comes into play, despite devices signing messages with their own key [20].

**Device Opacity** From a privacy standpoint, multi-device messaging could unintentionally leak information about other users. This can be the case if the messenger (publicly) shows, which devices another user owns. Depending on the level of information that is shown to other users, that could lead to surveillance, but the information leakage could be a necessary trade-off if required by the underlying architecture.

Those messengers that use a user level identity (also see section 3.1), like Threema and WhatsApp, the receiver cannot necessarily distinguish which devices the other user has (as long as this is not built into the protocol).

In Signal and many other messengers using the protocol with device level identities, the protocol requires the sender to encrypt the message once for every receiving device with that devices key. Thus, the senders needs to be explicitly aware of all devices that belong to a communication partner. Facebook Messenger and Wire furthermore provide a graphical user interface for inspecting the participating devices' public keys and device types (Desktop, Tablet, Phone) and make this an explicit feature. The Element messenger shows a similar amount of information. Keybase even shows a detailed timeline of added and removed devices and keys publicly based on the username.

Skype and Telegram do not display information on other user's devices. Furthermore, Secret Chats always exchange new random keys when the communication channel is established. Therefore, it is not clear which device sent which message.

**Table 2.1:** Comparison of multi-device messaging related criteria.

Pattern/Messenger	Security			Usability		Privacy
	E2EE	FS	PCS	Sending Complexity	Independent Sending	Device Opacity
<b>E2EE with shared group key</b>						
Keybase	●	○	◐	$\mathcal{O}(1)$	●	○
<b>E2EE per recipient</b>						
Telegram	◐	● <sup>†</sup>	● <sup>†</sup>	$\mathcal{O}(1)$ <sup>‡</sup>	●	●
Threema	●	○	○	$\mathcal{O}(1)$	◐	●
WhatsApp	●	●	●	$\mathcal{O}(1)$	○	●
<b>E2EE per device pair</b>						
Facebook Messenger	◐	● <sup>†</sup>	● <sup>†</sup>	$\mathcal{O}(n)$ <sup>†</sup>	●	○
iMessage	●	○	○	$\mathcal{O}(n)$	●	◐
Signal	●	●	●	$\mathcal{O}(n)$	●	◐
Skype	◐	● <sup>†</sup>	● <sup>†</sup>	$\mathcal{O}(1)$ <sup>‡</sup>	●	◐
Viber	●	●	●	$\mathcal{O}(n)$	●	◐
Wickr	●	●	●	$\mathcal{O}(n)$	●	◐
Wire	●	●	●	$\mathcal{O}(n)$	●	○
<b>E2EE per sending device</b>						
Element (Matrix)	●	◐	●	$\mathcal{O}(1)$	●	○

- Provides property.
- ◐ Partially provides property.
- Does not provide property.
- <sup>†</sup> Only supported in secret chats.
- <sup>‡</sup> E2EE only supported between two devices and not in the multi-device setting.

## 3 User Device Management

After the previous chapter investigated the different ways how devices can exchange data in a multi-device setting, this chapter builds up on this and considers how users can manage their devices.

We introduce the general distinction between identities on the user and device level. Next, adding new devices, an essential aspect of multi-device messaging, is examined. Depending on the specific messaging pattern that is used different keys need to be shared, created, and linked to the user. Therefore, we evaluate different patterns how devices are linked to users' accounts cryptographically. The way the required initial information is distributed varies and influences security and usability, which is why we consider this next. This also includes access to old messages on a new device.

If it is possible to add new devices, the next question is how to remove devices, in particular on the cryptographic level. Ideally, removed devices can no longer access old messages, receive new ones, and perform actions on behalf of the user. We then consider recovering instant messaging accounts in case users loose access to all of their devices. For recovery, the account with all its related information such as message history and also verification of chat partners needs to be considered. Moreover, we discuss security considerations specifically occurring in the context of multi-device management.

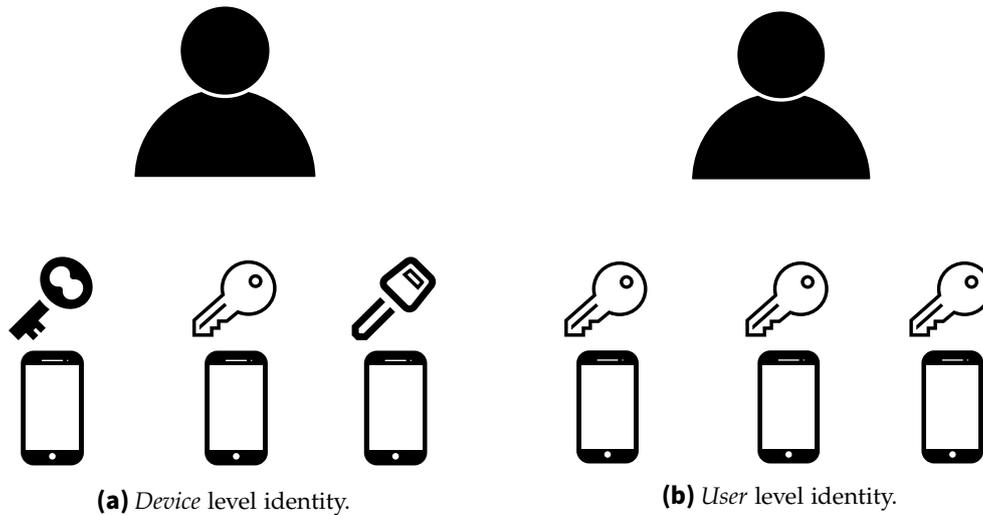
Finally, we evaluate the device management related criteria of section 1.2 for the different messengers. This results in an overview of differences between messengers as well as their underlying messaging protocols.

### 3.1 User and Device Identity

In messaging protocols without multi-device support, messages are sent to a single receiving device. Therefore, the terms *device* and *user* can be used interchangeably. But as soon as users are allowed to add more devices, one needs to differentiate between identities on the user level and on the device level.

The main reason why the concept of identities needs to be considered is that users want to make sure they are communicating with the expected communication partner. Otherwise, users who encrypt a message cannot know whether the keys they are using for encryption belong to the assumed communication partner or an attacker and thus whether secure encryption was actually achieved. This means that users need to exchange or verify keys in the real world to ensure that the encrypted connection is really between themselves and not with any other person. Therefore, in the context of messaging, identity is all about the (public) keys of a user. The relation to the real-world

identity has to be made outside of the messenger, e.g. by comparing a fingerprint of the public keys in person.



**Figure 3.1:** Different identity levels depending on key distribution.

Consequently, the concept of identity is closely linked to how keys are managed. Figure 3.1 shows an abstract example. Keys can belong to exactly one device of a user, thus they form an identity on the *device level*. Each device has its distinct key. Alternatively, keys can be shared across a few or all devices of a user. In this case, they are part of the *user level* identity.

User level keys and device level keys are often not used exclusively but instead used at the same time for different purposes and keys. The simplest way of associating a key with the user level identity is to hold a distinct copy of the key on each device and to copy it onto new devices as they are added. Such a key can then be used by all devices to represent a single user to the outside.

Both user level identities and device level identities have different advantages and disadvantages that make them more suitable for different use cases. These differences are outlined in the following sections.

## 3.2 Adding More Devices

### 3.2.1 Concepts per Messaging Pattern

How adding devices works in the first instance depends on the used messaging pattern, i.e., how data is transferred and encrypted between devices of different users. For each messaging pattern presented in section 2.3, this section explains what data needs to be shared and presents implementation approaches. While this provides an overview based on the messaging patterns, the utilized methods are presented in more detail afterwards.

**Without End-to-end Encryption** The technically simplest solution is a messenger without end-to-end encryption, as each new device only needs to be authenticated to the server. This can be achieved e.g. by a password-based login or by having an existing device share an authentication token.

On the one hand, this simplicity allows for the best possible usability. On the other hand, per definition this setup does not implement end-to-end encryption, and thus does not fall under our definition of secure. However, it is therefore an interesting baseline for assessing the usability drawbacks that improved security introduces.

**End-to-end Encryption With Shared Group Key** When a shared group key is used, the new device needs to get that key. While a group key might be updated if a new user joins a group of multiple users, this is not really necessary if only a new device of an already participating user is added. Nevertheless, it might be done anyway depending on the specific protocol.

Furthermore, it needs to be distinguished whether identities are managed on a device or user level, as introduced in section 3.1. In case of device level identities, not only the server will need to know of the new device, but also the communication partners. On the one hand, this is necessary so that the new device can be notified about group key updates without another device of the user being online. On the other hand, communication partners should be able to verify the authenticity of messages. For that, they must know e.g. the public key of the new device.

With user level identities, the new device needs to get access to user level private keys and synchronize with the other devices of the user. Depending on the protocol, the server might need to know of the new device to distribute messages to it.

Keybase, a messenger that uses a group key to encrypt all chat messages, uses device identities. The new device receives the group keys and is added to the user's sigchain, a public cryptographically linked list of events affecting the user's account [22], so that communication partners are aware of the new device [20, 21]. The corresponding entry includes a signing and an encryption key. With the first one, communication partners can verify the authenticity of messages, and with the latter one they can send new group keys also to that device, e.g. when the group key is rotated or a new conversation is initiated.

With MLS [5], there is a protocol in development that provides a method to efficiently update a group key. Thus, it may not only be used for group communication but also for multi-device messaging (in 1:1 chats and in groups). Similar to the aforementioned differentiation, this can be implemented either on a user or on a device level. This corresponds to the question whether leafs in the MLS tree represent devices or users. In the device level approach, an existing device can add the new device to all conversations and groups. The new device may also be linked to the user account by one of the methods explained in subsection 3.2.2. For the user level approach, devices of a user would need to share keys and synchronize. For that, different approaches are possible, e.g. a mediator server as proposed by Threema [41], synchronizing internal states as presented by Campion et al. [7], or the devices of a user could maintain their own MLS tree to derive user level key material [11].

**End-to-end Encryption Per Recipient** Messengers like WhatsApp or Threema<sup>6</sup> use a pattern we describe as *primary device and reflection pattern* [48, 42]. As explained in subsection 2.3.3, secondary devices do not connect to the server, but send and receive messages via the primary device. Therefore, a secure communication channel between the primary and the secondary device needs to be established. Information required to set up this channel and mutually authenticate is often exchanged by having the secondary device display a QR code that the primary device can scan. The end-to-end encrypted communication channel is then not only used to transfer messages but also for synchronizing metadata. This includes contact lists, chat history, and everything else that should be displayed on the secondary device. As the primary device and reflection pattern requires the primary device to be online for the secondary device to send and receive messages anyway, only a minimum of data might be exchanged directly, while most of it can be shared later when needed. If a central service is used in the messaging protocol, it does not need to be informed about the new device and may only be involved in the setup of the communication channel.

Another approach, as outlined in subsection 2.3.3, is to share a long-term key among the devices. In that case, this long-term key needs to be shared securely with the new device, e.g. via a service that stores encrypted keys [32]. The new device may be notified about public keys of communication partners.

In the concept proposed by Campion et al. [7], a new ratchet is created with every communication partner so that new devices cannot read old messages. One of the existing devices sends the corresponding update and also shares the new ratchets with the new device. Other already existing devices of the user are notified about the new device so that they from now on synchronize their internal states also with the new device.

In the new concept of Threema [41] where a mediator server is used to distribute messages, the so-called *Threema ID's private key* (similar to a main or identity key) has to be shared with new devices. The new device derives further keys from that and can authenticate to the mediator server by that. Other devices of the user need to be made aware of the new device, so that messages will also be reflected to the new device. As further data such as contacts, group chats, or settings is synchronized among the devices by using the mediator server, the new device will also receive that information. While the published concept does not provide all details yet, it can be assumed that a secure way of sharing the private key will be implemented, e.g. via a QR code similar to the current procedure in Threema's web client for pairing devices. Notifying other devices of the user about the new device will probably be done by sending messages to the other devices, similar to the synchronization mentioned before. As the proposed concept aims to conceal the devices from communication participants and even the chat server, those are not involved in the process of adding new devices.

**End-to-end Encryption Per Device Pair (Client Fanout)** This pattern is used within the Signal protocol, so many messengers building on top of this protocol use it. There,

---

<sup>6</sup>Threema's version in production as of writing (web client), thus without their planned mediator server. We cover their new approach later in this paragraph.

messages are encrypted for each receiving device. The server is notified about the new device, so that communication partners and same-owned device also encrypt for the new device. Therefore, the new device creates own prekeys which are uploaded to the server. Additionally, in messengers that use the Signal protocol and link devices, an existing device shares the identity key which the new device needs to perform the Extended Triple Diffie-Hellman [27, 28]. For this initial setup, a secure connection between the mobile device and a new one is set up via QR codes. Groups and contacts are imported from the mobile device.

Messengers that use the Signal protocol but maintain device identities and do not link the devices of a user do not share the identity key, but the new device creates an own one. Communication partners might get a warning that a new unverified device has been added to the chat and that they should verify the new device's identity (key).

Those two different approaches correspond to the two variants outlined in the *Sesame Algorithm* [27]. Either devices have their own identity key or all devices of a user share one identity key so that the devices are linked together, as we will explain in subsection 3.2.2.

**End-to-end Encryption Per Sending Device (Sender Keys)** In the Matrix protocol where device level Sender Keys are used [17], the new device authenticates to the server based on the password associated with the user's account. It creates a device identity key and prekeys similar to the Signal protocol. These are uploaded to the server and for every current conversation the new device exchanges Sender Keys with its communication partners via secure peer-to-peer channels. To verify the new device, a procedure called cross-signing is used, which is explained in section 3.2.2.

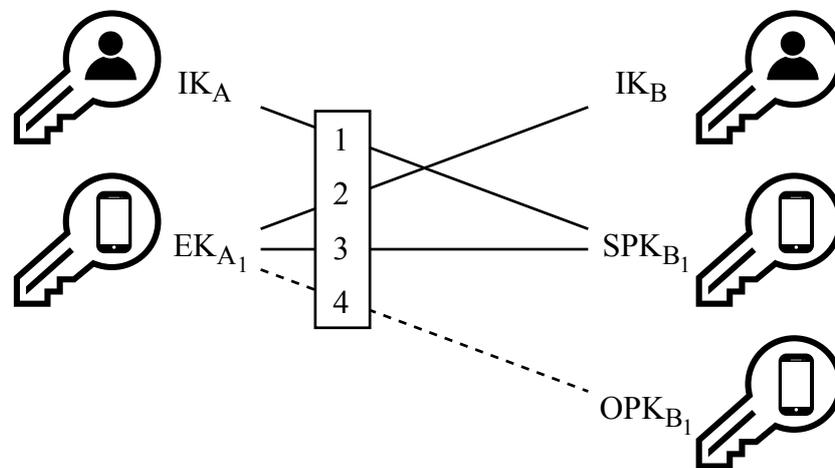
#### 3.2.2 Cryptographically Linking a New Device to an Account

This section mainly deals with the question of how communication partners trust new devices. While a central server might check that only authorized devices send and receive messages, users will probably want to check this on their own, as they may mistrust the server. Otherwise, if they do not verify the identity of the communication partner, they could easily become victims of a man-in-the-middle attack. In the multi-device context, this slightly modifies the question "Do I really talk to Alice?" to "Do I really talk to a device of Alice?". Therefore, this sections presents ways how devices can be cryptographically linked to a user account, so that other users can validate this. In this context, a user account is everything identity related that belongs to the same user. This comprises all devices that appear under the same user identifier, e.g. username, phone number, or email address.

For example, if a messenger allows comparing key fingerprints to verify the identity of communication partners, cryptographically linking devices helps that this key fingerprint comparison does not have to be performed again with the new device. Instead, trust transfers to the new device. Some messengers however do not implement a linking mechanism. While users can still verify that a device belongs to a user, this requires more effort as the verification needs to be done with each device separately.

**User Level Encryption** The simplest way is concealing details about own devices from communication partners. Thus, the question whether to trust a certain device, never arises. This applies to all patterns that implement user level encryption, i.e. sharing a long-term key among all devices, the multi-device concept for Signal as proposed by Campion et al. [7] where internal states are synchronized between the devices, and all patterns that use the reflection pattern. In addition, it provides the benefit that communication partners do not know by which device a message was sent.

**Sharing the Identity Key** All patterns using device identities need to actually link devices cryptographically. Signal uses the per-user model of the *Sesame Algorithm* [27] to implement multi-device support. There, the identity key is shared across all devices of a user. This is the key that is used for authenticating the communication and which should be verified e.g. by fingerprint comparison. The messaging protocol can still use device specific keys, but by including the identity key in the key agreement protocol, every device is authenticated as belonging to the same user.

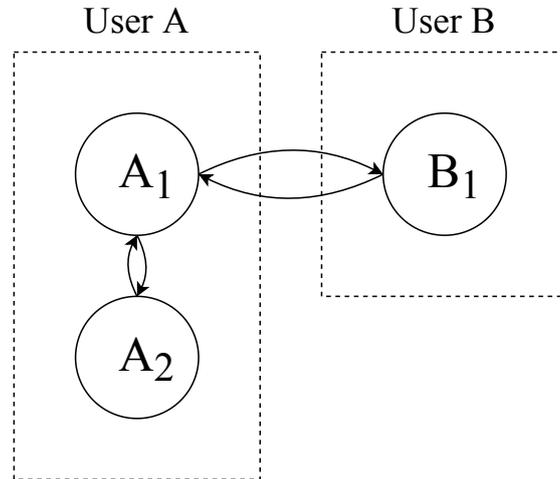


**Figure 3.2:** Extended Triple Diffie-Hellman (X3DH) key agreement with user level identity keys and device level ephemeral and prekeys, based on [28].

As depicted in Figure 3.2, the key agreement protocol uses multiple keys. User  $A$  and  $B$  each have an identity key  $IK_A$  and  $IK_B$  respectively. This key is shared among all devices of a user and therefore on the user level. The ephemeral keys  $EK$  and the prekeys  $SPK$  and  $OPK$  are device level keys, that means, every device has its own. If another device  $A_2$  of user  $A$  would like to send messages to user  $B$ , it would use the same shared identity key  $IK_A$  and an ephemeral key  $EK_{A_2}$  and perform the key agreement protocol with every device of user  $B$ . Since the new device  $A_2$  had to know the identity key  $IK_A$  to successfully execute the key agreement protocol, the devices of  $B$  know that it belongs to user  $A$  and trust it.

**Cross-signing** However, not all messengers share an identity key to ensure trust. In the Matrix protocol, every device maintains its own key and there is no encryption

or decryption key on a user level. Hence, other users cannot trust a claimed new device without further proof. In the early days of Matrix, there was no concept for this and users needed to verify the identity of each new device manually. Obviously, that approach lacks usability and that is why something called *cross-signing* was evaluated. The first draft for cross-signing [8] worked on the device level, while the current approach is based on the user level.



**Figure 3.3:** Device level cross-signing.

In device level cross-signing, depicted in Figure 3.3, devices of the same user and of different users sign each other with the device's keys, building up a web of trust. This web of trust can be described as a directed graph, where an edge from device  $A_1$  of user  $A$  to device  $B_1$  of user  $B$  corresponds to a signature of  $B_1$ 's device key made by  $A_1$ 's device key, meaning that  $A_1$  attests that it verified  $B_1$ . If there is no direct path from device  $A_2$  (of user  $A$ ) to device  $B_1$  (of user  $B$ ), they will only trust each other if:

- there is a path from  $A_2$  to  $B_1$  only consisting of devices from  $A$  and  $B$ ,
- and there is a path from  $B_1$  to  $A_2$  only consisting of devices from  $A$  and  $B$ ,
- and if there are no revocations of that attestations.

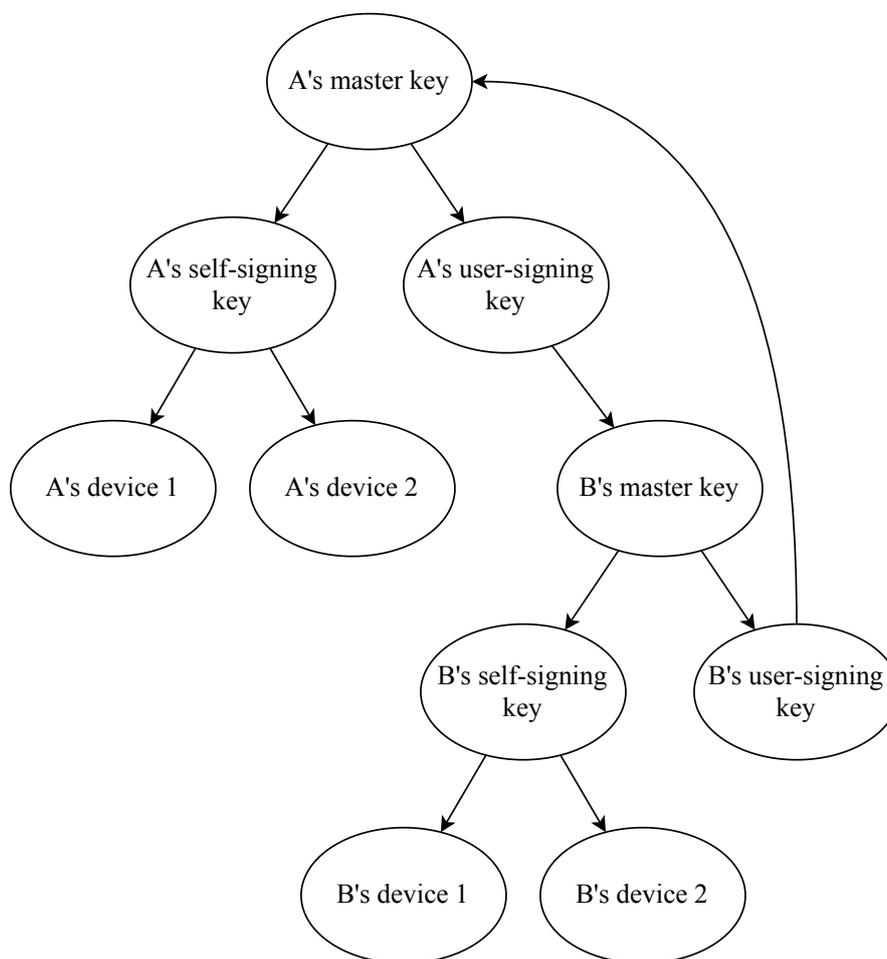
By that, users do not need to manually verify each other's devices. Instead, they can rely on the attestations of their communication partners about their devices, which reduces the manual effort and thus increases usability.

However, this concept includes several problems [10]. First, revocations of devices or rather the corresponding signatures need to be handled carefully. An attacker who compromised one device could revoke all other devices of that user and take over the account. The second problem about revocations is that they do not indicate why a device was revoked. A revocation could mean the device has been lost, stolen, or the user just does not want to use it any longer. The third problem is that revocations can partition the graph. Thereby, devices not involved in the revocation, i.e. which are

neither the revoking nor the revoked device, could lose the trust between them. This, plus the fact that the graph can become very complex, entails considerable difficulties for the users. Additionally, there is no way to centrally manage attestations as every device issues its own.

That is why a new cross-signing concept was developed [10]. While Matrix previously handled everything on device level, this new concept introduces some keys on the user level. Instead of devices signing each other, all devices are signed by the self-signing key of a user to express that they are associated with the user's account.

The self-signing key can be either shared with all devices or only with a subset of devices. As a result, only devices that know the self-signing key can add new devices. Through this, it is theoretically possible that a user arranges own devices by trust so that only a few devices are able to authorize new devices. For example, a temporary, short-term web session might not require this privilege.



**Figure 3.4:** User level cross-signing in Matrix, based on [10].

As depicted in Figure 3.4, the self-signing key of a user is used to sign each device of that user. Furthermore, there are two other user level keys in Matrix. The user-signing

key is utilized for verifications with other users and the master key signs all these other user level keys of the user, i.e. the self-signing key and the user-signing key. When users compare the key fingerprints and verify each other, they sign the master key of the other user with their user-signing key. As this master signing key signed the self-signing key which then signed the device keys, users can trust all devices of their communication partner after such a verification. If a new device is added, this is accordingly signed by the self-signing key so that trust is automatically established by the previously explained mechanism.

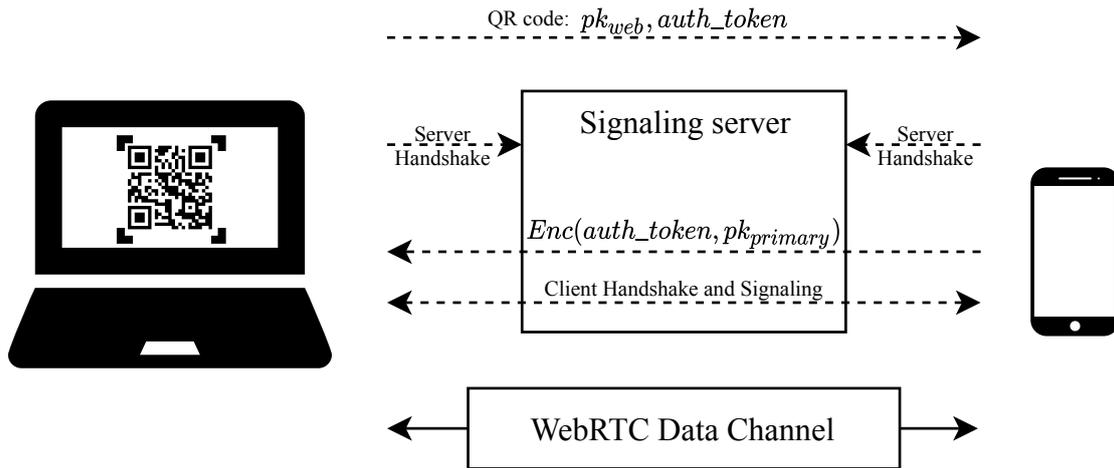
The concept of user level cross-signing includes similarities to sharing the identity key. The major difference is, that in cross-signing the user level keys do not necessarily need to be shared with other devices as they are not used in the actual messaging. While a shared identity key mixes trust and encryption, those two parts are separated in user level cross-signing. As adding and removing devices as well as verification with other users happens significantly less frequent than message encryption and decryption, the user level cross-signing keys need to be used only rarely and can therefore be managed and stored more securely. The identity key in the Signal protocol, however, is used continuously and therefore at a higher risk of being compromised. In terms of security and removing devices, cross-signing provides some further advantages, as will be presented in section 3.3.

Two other messengers that also use cross-signing are Wickr and Keybase. In Wickr, every user has an identity key that signs all the devices [18]. Therefore, this is user level cross-signing. Keybase does not have user level keys for the purpose of cross-signing and instead uses device level cross-signing. Any device can add a new device by signing it. This is then added to the user's public sigchain, a cryptographically linked list of events that affect the user's account [22]. By that, communication partners can see all devices of a user and verify that each has been authorized and really belongs to the user.

#### 3.2.3 Methods to Exchange Required Information

As the previous sections showed, certain information needs to be exchanged initially. This can include user level private keys and public keys of the new device, but also old messages and metadata. While the kind of data that is exchanged differs and is related to the pattern that is used for linking devices, the ways how to exchange that information are independent of that. However, these approaches differ in terms of usability.

**Secure Connection Establishment via QR Codes** A common way to exchange the required data is by setting up an end-to-end encrypted connection between an existing and the new device. Often, this is done via QR codes. For example, Threema uses the *SaltyRTC* protocol [15] for establishing a connection between both devices securely, as shown in Figure 3.5. In the end, these devices can communicate end-to-end encrypted either peer-to-peer or via a relay server [42]. Therefore, the new device, a web client, creates a so-called permanent key pair and an authentication token.



**Figure 3.5:** Threema web client connection establishment.

The permanent public key and the authentication token are shared with the primary device via a QR code the primary device can scan. The primary device also creates a new permanent key pair and both devices will connect to a signaling server so that they can exchange handshake messages. The primary device will send its permanent public key to the new device, encrypted with the authentication token that was provided within the QR code. Assuming information exchange via QR code is secure, the primary device can be sure to know the correct public key of the new device, and the new device can be sure to know the correct public key of the primary device, because it has been encrypted with the authentication token only the primary device knows. To conclude this handshake, both devices create session key pairs, exchange the public session keys, and agree on how to communicate afterwards, either peer-to-peer or via a relay server.

If the communication between both devices is not only needed for an initial setup, but also used e.g. for ongoing exchange of messages, both devices can re-establish the connection with the same permanent keys. They can start directly with the handshake where they agree on new session keys and how to communicate afterwards.

While using QR codes for information exchange can improve the usability as users do not need to enter data manually, it is only possible if one device is able to display a QR code and the other device is able to scan this QR code. Keybase also uses a QR code based approach to exchange required information. However, out of the explained reason, Keybase additionally offers the alternative that users type the data by themselves. For that, information is encoded as random words [21].

**Verify New Device and Utilize Messaging Protocol** While the previous approach is used to set up a new channel independent of the actual messaging (e.g. WebRTC or Web-Socket), it is also possible to exchange data directly via the messaging protocol if encryption happens on a device level. Therefore, an old and the new device need to verify their device identity keys so that they can communicate securely.

After the new device successfully logged in at the server and uploaded its device key, current devices of the user are notified and asked to verify the new device. This verification is similar to fingerprint comparison when verifying other users. After both devices acknowledged each other, they can exchange messages end-to-end encrypted and, because they verified each other's public keys, be sure to actually communicate with each other.

In contrast to establishing a secure connection via QR codes, this approach uses the actual messaging protocol and not another channel. It only works for protocols where encryption and decryption is implemented on a device level. Since users only need to verify the new device and do not need to exchange any data manually, it may be perceived as a more usable alternative. Especially for devices that are not able to display or scan QR codes for information exchange, this is a much better way. Still, this approach can probably only be used if the new device can at least somehow prove that it belongs to the same user account. In Matrix this is done by a username/password login on the server. Otherwise, this functionality could be used to spam users with verification requests.

**Encrypted Secret Storage** Both methods explained previously have the downside that at least one existing device needs to be online. Even in cases where devices can send and receive messages independently of each other, this is required for the initial setup where e.g. private keys are shared. This can be a problem if users just want to log in on a temporary device and do not have another device at hand. To remove that restriction, necessary private keys can be stored on a server, similarly to what is described in subsection 2.2.1. This concept is implemented in Matrix [9] in addition to the previously explained one. After authenticating with the account password, the new device needs to be cryptographically linked to the user account as explained in section 3.2.2 on cross-signing. The new device can retrieve the encrypted self-signing key from the server, decrypt it with another passphrase that has been defined especially for encrypting and storing the shared secrets, and afterwards perform all the required steps to link itself to the user account.

**Password Based Key Derivation** Additionally, it is possible to derive user level private keys from passwords, e.g. using a key derivation function, instead of creating them randomly. While this is possible in theory, no relevant messenger we are aware of does this. The problem is that from the security perspective, only truly random chosen passwords and keys with enough entropy guarantee security. Unfortunately, it is very difficult if not unfeasible for users to remind such random or high entropy passwords. Additionally, due to the input constraints mobile devices bring, it is very inconvenient for users to enter long and secure passwords on those. Because of that, messengers create keys instead of deriving them from passwords. Related to this, some messengers use passwords as additional protection to access private keys that are stored in a server side backup. While this is a bit more secure than deriving keys directly from passwords because the server can prevent attackers from gaining access to an encrypted backup, strong passwords are still required, as otherwise the encrypted keys in the backup

could be easily decrypted (e.g. by the server or an attacker who is able to circumvent the server-side protection).

### **3.2.4 Synchronizing Old Messages**

A feature required from users is to have a consistent chat history across all their devices. This includes the ability to read old messages on a recently added device.

But obviously it is not possible to decrypt old messages if forward secrecy is implemented. This has led to some messengers abandoning forward secrecy, and hence security, in favor of better usability. For example, in the Megolm protocol of Matrix [17], it is possible to derive the current ratchet state from any previous one. Thus, an existing device can share an early ratchet state and the new device can decrypt all messages sent later. Because of the introduced drawbacks, Matrix speaks of “Partial Forward Secrecy”. They point out that users should have the possibility to discard previous key material so that forward secrecy for corresponding previously sent messages is ensured.

Alternatively, it is always possible that an existing device shares messages by re-encrypting and re-sending them. This can be observed in web clients where messages are shared on demand by the primary device [42].

For messengers where devices send and receive messages independently, this is not possible because it is not guaranteed that another device is online to share the old messages. For those it would be possible to share the old messages during the initial setup because it is often required that an old device is online to perform the initial setup. However, this would mean encrypting and sending relatively huge amounts of data and can therefore not be observed in practice for the major messengers.

While the ability to read old messages on a new device of course improves usability, there are other messengers like Wire that explicitly do not support it, pointing out privacy concerns [52]. For example, an attacker who gains access to the password can log in to the account but is not able to retrieve the chat history. As opposed to this approach, other messengers such as Telegram explicitly advertise high usability and provide cloud message access [38]. This is easily possible as Telegram does not use end-to-end encryption.

## **3.3 Removing Devices**

After we considered adding devices in the previous section, we now take a look at how those devices can be removed. Users might want to remove a certain device because they do not longer want to use it, it got lost, or stolen. How devices can be removed securely depends on how keys are managed. Therefore, this is also related to the linking and identity management in general.

Removing devices becomes a problem when the device to be removed has access to shared user level keys. If the device is still controlled by the user, those keys can safely be removed from the device. But this kind of cooperation can not be assumed if the device is lost or has been stolen. Then, a key rotation of the particular compromised

keys is required. Depending on the purpose of the keys, a new verification with other users might be required as a rotation of identity keys can not be differentiated from a man-in-the-middle attack. This applies to two of the device linking concepts of subsection 3.2.2, the one where the identity key is shared and the one that uses shared long-term keys as part of user level encryption.

Device level keys do not need to be rotated, because they are no longer used. Therefore, they just need to be invalidated, e.g. by revoking a cross-signature. While for device level cross-signing it is only required to revoke the signature, user level cross-signing can also require to rotate the cross-signing key if this was shared with the device that should be revoked.

For web clients where no key is shared, it is even easier as the primary device just needs to terminate the connection with the device to be removed.

Summarizing, this shows that how keys are managed between the devices directly correlates to the simplicity of removing devices. Messengers like Signal, where the user level identity key needs to be shared with all devices, require a key rotation once any of the devices is compromised. In practice, this implies an account reset including major efforts for the user. To make this step easier, *Campion et al.* [7] proposed to allow an identity key rotation directly inside the messenger, so that users do not need to create their account from scratch but keep their devices, settings, and old chats. In contrast, messengers like Matrix where user level keys do not need to be shared with all devices only require a key rotation if a device that had access to that particular user level key is compromised. By that, it is theoretically possible to share those keys only with trustworthy devices, making the process of removing devices easier in most cases.

This directly correlates to the handling of temporary devices like public computers. If the messenger allows to not share any key at all with that device, this increases security and usability.

However, while Matrix's cross-signing concept offers the mentioned differentiation of more and less trustworthy devices in theory, to our knowledge this has not been implemented in practice. In the widely used Matrix client Element for example, every new device gets access to all user level keys. Probably, this kind of differentiation would be too difficult to understand for users.

## 3.4 Recovering Accounts, Messages, and Verifications

In case users lose access to all their devices, they expect to be able to recover their account and ideally even the chat history. For such kind of recovery, we differentiate three types. First, the account recovery, i.e. the ability to get access to the messenger account, e.g. the username. Second, message recovery, i.e. the ability to restore message history. And third, verification recovery, i.e. that the verification with other users persists after successful recovery and does not need to be done again.

To recover the account, further authentication factors are required, e.g. phone number or email. They may either be set up when registering the account or added later optionally. In messengers like Threema where the account is tightly coupled to the cryptographic identity, i.e. the identity key, a backup of that key material is required

to prove ownership of the account. Messengers like Signal and WhatsApp even offer two-factor authentication for re-registration, as we present in the next section.

Recovering messages only works by creating and importing backups. While most messengers rely on message backups, it is also possible to just create a backup of the keys in Matrix as the encrypted messages are stored on the server forever. Due to the missing forward secrecy, the amount of keys that need to be stored is rather small.

Verification recovery requires restoring all key material that is used for verification with other users. For example in Matrix this comprises the master, the self-signing, and the user-signing key and in Signal this comprises the identity key of the user. If that key material is not derived from something but has been generated completely randomly, a backup is required. Without a backup, verification with other users is lost and needs to be done again. As many messengers use key fingerprint verification, this is associated with major efforts as it needs to be done with each communication partner individually. However, other mechanisms for trust establishment exist that allow an easy key rotation [43]. In those cases, verification recovery simply works without any backup. One example for that is Keybase where trust is established by linking the messenger identity to social media accounts. The effort to recover verifications does not scale with the number of communication partners but remains constant and does not even require major efforts.

### 3.5 Security Considerations

To conclude the section on user device management, we consider some security aspects and how an attacker might exploit the device management mechanisms.

**Two-Factor Authentication for Re-registration and Registration Locks** As mentioned previously, messengers like Signal or WhatsApp offer two-factor authentication if a user wants to re-register [33, 49]. If this is activated, an additional user-defined PIN is required besides proving ownership over the phone number to regain access to the account. In case the user forgot the PIN, it is still possible to perform a re-registration.

For Signal, there is a registration lock which means that the PIN is no longer required only if no device of the user interacted with the central server for more than seven days. In WhatsApp, the user can provide an optional email address while initially activating the two-factor authentication in order to perform PIN resets via email. If no email was used, one can reset the PIN after seven days. Compared to Signal’s approach, this cooldown does not require inactivity of all devices during the cooldown period.

**Primary Device for Device Management** We observed that some messengers such as Signal only allow a primary device to manage the devices of a user, e.g. only the mobile device that was used in the registration and linked with the phone number can be used to add or remove devices. This is a simple measure to prevent some attacks because the attack surface of attacks against the device management does not scale with the number of linked devices but remains constant. Consequently, an attacker needs to compromise the primary device to add own devices or remove other devices of the user.

When attackers compromises any other device, they can still read and write messages but at least they cannot add or remove devices. The user can remove the compromised device if it is noticed, and do not risk being kicked out by the attacker.

While this reduces the general attack surface, it is less comfortable for the user to only have a single device that allows adding new devices. Therefore, such a primary device is a special problem if it gets lost or stolen.

However, it needs to be differentiated whether this limitation is enforced by cryptography or just by implementing it within the central server. For example, in Signal, on the one hand, only the primary device can add other devices, but on the other hand, each device has the necessary information to cryptographically link a new device to the account. A device would just need to share the identity private key. The only reason why it is not possible for other devices to add new ones, is that new devices need to be known by the server and need to upload their prekeys to it.

**Notifications for Device Changes** To make it possible for users to detect if an attacker adds a new device, some messengers send notifications when a device is added. For example, Wire sends an email but in general such notifications can be sent via any channel or also be displayed inside the messenger application. Telegram does the latter, as it sends a message informing about the new login and used IP address to each logged in device.

**Reducing Number of Shared Keys** As shown in section 3.3 on removing devices, it can become a problem if the device to be revoked has access to shared keys as they need to be re-issued if they cannot be deleted securely from the device. Since those key rotations decrease the usability, they should only be necessary as rarely as possible. To still keep security, the best way to reduce the need for key rotations is by sharing as few as possible keys between devices. However, this can decrease the usability, e.g. in user level cross-signing only devices that have access to shared user-level keys are able to add new devices. Therefore, it would be the best if users could decide the devices to share certain keys to. As the considerations behind such a decision are probably too complicated for most users, it is very difficult to implement in practice, as a good and understandable abstraction would need to be found.

**Threshold Cryptography** Another threat in the multi-device setting is the loss of a device, where a possible attacker can obtain access to its data. Especially shared cryptographic keys such as identity keys should not get compromised.

To deal with this problem, Atwater and Hengartner [3] suggest threshold cryptography. The main idea is to split a cryptographic key and distribute the different parts across several devices. Then, cryptographic operations using that key can only be performed by a number of devices together. This limit is called threshold of devices and can be adjusted as needed. Depending on the desired threshold, an attacker needs access to at least this certain number of devices in order to compromise the key and use it to perform malicious operations.

In multi-device messaging, threshold cryptography could be used for the keys responsible for message encryption and decryption. As proof of concept, Atwater and Hengartner integrated their framework Shatter into the messenger ChatSecure [3]. Then, in case a device is lost, the user is able to instruct the remaining devices to not trust the stolen device anymore. Therefore, the stolen device cannot perform cryptographic operations with the distributed private key anymore. However, as long as the remaining devices reach the threshold, they have enough information to re-establish the full original private key and distribute it in new shares. Thus, no key rotation is necessary, while the stolen device is successfully revoked. Unfortunately, the huge drawback is that each cryptographic operation needs  $t$  devices to be online, where  $t$  is the specified threshold. The certain number of always online devices and accompanying performance losses are disadvantages for message exchange. Furthermore, we do not see the need for threshold cryptography in message encryption and decryption but rather in other multi-device related problems like device management.

One example is to hinder attackers from adding or removing devices from one messaging account by requiring an authorization signature of more than one device. Specifically for the cross-signing concept from subsection 3.2.2, the user level self-signing key could be distributed by using threshold cryptography. Then, several distinct devices would need to agree to sign new devices such that they become trusted.

But for that, also other threshold cryptosystems could be thought of, e.g. multisignature schemes where no private key exists at all. This would reduce the risk in disclosing private key material. In their concept, the original private key is constructed in the initial setup and reconstructed whenever devices are added or removed.

Overall, threshold cryptography introduces complexity for the user. In the concept of Atwater and Hengartner, not only several devices need to be online, but users need to understand the underlying concept. Otherwise, they may experience behavior they do not expect, e.g. they cannot delete or add new devices because they lost access to the majority of clients they once added that are now necessary for the threshold.

**Intercepting Verification Codes** Telegram sends verification codes to add a new device. While this might improve usability, it bears the risk that an attacker intercepts those verification codes to add an own device. To conduct such an attack, an attacker needs to initiate the process of adding a new device and, once the corresponding verification code is sent to the existing devices of the users, intercept it, e.g. by social engineering or shoulder surfing.

A similar attack is possible by performing an account re-registration. Many phone number based messengers send a verification code via SMS to verify that the user really owns the claimed phone number. By intercepting such an SMS verification code, an attacker can successfully register an own device for a different phone number and therefore impersonate the user who actually owns that phone number. If communication partners do not recognize the warning that the identity of the user changed, they continue to communicate with the attacker and not with the user they assume to communicate with. To successfully intercept the SMS verification codes, an attacker can either employ the techniques mentioned to obtain device enrollment verification

codes or try to eavesdrop SMS messages itself. Compromising one account can improve the success probability of further attacks on other users because the attacker can impersonate already compromised users and conduct a social engineering attack on their contacts by asking for verification codes. To prevent such re-registration attacks, another authentication factor can be requested like explained in the previous section on two-factor authentication and registration locks. There, the faked proof of controlling the phone number is not enough to perform a re-registration. Also, users should lock their phone and ensure that notifications on the lock screen do not reveal sensitive information. In both attack scenarios, vigilance of users can help to detect and prevent them.

### 3.6 Messenger Overview

Similar to the evaluation of messengers with respect to messaging related criteria in section 2.4, we now take a look at device management related criteria in this section. Therefore, we investigated the messengers' whitepapers and documentation, while some information was collected by examining the apps.

**Linking Devices** In subsection 3.2.2, we presented the different concepts how devices can be linked. Now, we evaluate the different messengers based on the concepts they use for linking a device to an account. Threema and WhatsApp both offer web clients [42, 48]. This corresponds to the user level encryption pattern as encryption and decryption takes place on the primary device. Secondary devices can send and receive messages by using the reflection pattern. Even with the new concept that Threema plans to introduce [41], they will still use user level encryption since all devices share the required user level key material. Signal, and therefore also Viber with their own implementation of the Signal protocol, in combination with the user level variation of the Sesame Algorithm, share the identity key to link a device to a user account [27, 45]. Some other messengers that use the Signal protocol in combination with the device level variation of the Sesame Algorithm like Facebook Messenger, Skype, or Wire do not use Signals concept for multi-device support as they do not link devices at all but use device identities [14, 29, 53]. There, the key verification needs to be done manually for each device. Element, as a Matrix client, uses user level cross-signing to link the devices to an account [10]. Wickr, while using a different protocol, also has an identity key on the user level that signs each device of the user [18]. Keybase uses device level cross-signing as any device can sign a new device and add it to the user's sigchain [21].

**Information Exchange** For the initial information exchange we only consider messengers that actually link devices. Most of them use QR codes for that and setup a connection which is used to exchange all the data. Wickr stores the encrypted identity private key on the server. For its encryption, a random recovery bundle key is used. This is then encrypted with a key that is derived from the user's password and, by default, also stored on the server. However, the user can optionally choose to only store the encrypted backup of the identity private key on the server [18]. Element also offers

such a secret storage where user level keys are backed up. However, Element has this as an optional feature and uses a separate passphrase for encrypting the keys and not the account password. Therefore, one could argue that this is more secure than the secret storage of Wickr. By default, Element asks the current devices whether the new device, that recently logged in, belongs to the user. After comparing the key fingerprint with the new device, all the information is exchanged by using the actual messaging protocol.

**Old Chats** In this criterion, we analyze the extent messengers allow accessing old chats. As soon as users add a new device, the chat history made by the already existing devices is considered as old chats. For best usability, users expect to see their old chats on the new device.

Threema and WhatsApp mirror old messages to the web client and therefore support accessing old messages on the new device [42, 48]. How this will look like in the new Threema concept [41] is not specified. Element, Keybase, and Viber also allow reading old messages on a new device. Element achieves this by not implementing forward secrecy so that old messages that stay on the server forever can still be decrypted [17]. For Keybase, also no forward secrecy is implemented and a current device can just share the symmetric encryption keys that were used for all messages in a chat [20]. Viber asks during device enrollment whether old messages should be shared with the new device and therefore probably just re-encrypts and resends them. Skype and Telegram only support this feature in unencrypted chats where the messages are stored in the cloud in plaintext. In contrast, other messengers such as Facebook Messenger, iMessage, Signal, Wickr, and Wire do not synchronize old chats with a new device. This may be due to the necessary effort or messengers like Wire directly point out privacy concerns [52].

**Recover** As explained before, we consider three different types of recovery. The account recovery means that the user can still get access to the account, e.g. the username, even if currently no device is added. We only regard this as fulfilled if it is very unlikely that the user fails to setup all prerequisites. This is true for a mandatory second authentication factor that is hard to lose, e.g. a phone number or an email address, and applies to most messengers we evaluated. Messengers like Element or Wickr that use passwords allow to recover the account easily, but only if the user still remembers the password. As this can easily be forgotten, we see the property of account recovery only as partially fulfilled. Element, like Keybase, also allows to add a second authentication factor like an email address. While users can gain access to their account again with that, it is not guaranteed that users set this up because it is only optional. Threema requires a key backup because the account identifier is tightly coupled to the cryptographic identity and therefore the key [42]. This needs to be set up manually and also bears the risk to lose it. That is why we also see this as only partially providing account recoverability.

Regaining access to old messages after all devices were removed from the account is only possible through backups. While Element only needs to backup the necessary

keys, most of the other messengers supporting that feature create a backup of all the messages. Skype and Telegram can just offer access to the unencrypted messages in the cloud again after the account itself has been recovered. Facebook Messenger, Keybase, and Wickr do not allow creating message backups in their app.

However, Keybase still allows users to recover messages. Communication partners see a warning that the account of the user they want to communicate with has been reset. They should either verify in person or check that a valid proof is added to the recently reset account, and can then decide whether they want to accept this and continue the communication. If users confirm that this was a legitimate account reset chat history is shared with the other user. By this method it is possible to recover old chats through the communication partner. This process needs to be completed with each communication partner separately.

Also recovering the verification with other users is only supported by a few messengers. Element, Signal, Threema, and Wickr allow to recover the previous identity key so that the key fingerprints that can be compared do not change. Keybase does not offer to create a backup of the keys but also achieves the recovery of verifications. This is because of the different authentication mechanism that is provided by Keybase where users link their messenger identity to other identities, e.g. to social media accounts or their website [23]. Other messengers do not allow to create a backup of the identity key and require users to repeat a key fingerprint comparison with each other again. For iMessage and Viber, we did not even find a way to compare key fingerprints at all.

**Table 3.1:** Comparison of user device management related criteria.

Messenger	Security			Usability						
	Linking Devices			Information Exchange			Old Chats	Recover		
	User Level Encryption	Sharing Identity Key	Cross-signing	QR Code	Verify Device Key	Secret Storage	Access	Account	Messages	Verifications
Element (Matrix)	○	○	●	○	●	●	●	◐	k	●
Facebook Messenger	○	○	○	N/A	N/A	N/A	○	●	○	○
iMessage	○	○	○	N/A	N/A	N/A	○	●	m	N/A
Keybase	○	○	●	●	○	○	●	◐	●	●
Signal	○	●	○	●	○	○	○	●	m	●
Skype	○	○	○	N/A	N/A	N/A	◐ <sup>§</sup>	●	◐ <sup>§</sup>	○
Telegram	○	○	○	N/A	N/A	N/A	◐ <sup>§</sup>	●	◐ <sup>§</sup>	○
Threema <sup>¶</sup>	●	○	○	●	○	○	●	◐	m	●
Viber	○	●	○	●	○	○	●	●	m	N/A
WhatsApp <sup>¶</sup>	●	○	○	●	○	○	●	●	m	○
Wickr	○	○	●	○	○	●	○	◐	○	●
Wire	○	○	○	N/A	N/A	N/A	○	●	m	○

- Provides property / Uses this pattern.
- ◐ Partially provides property.
- Does not provide property.
- ¶ Multi-device only supported by web client.
- § Not supported in secret chats.
- m Via message backup.
- k Via key backup.

## 4 Future Work

As a pointer for those interested in further research, we suggest prioritizing the examination on Signal, Matrix, and Threema. Evaluating these three covers a lot of ground, because they are different from another. The encryption schema pioneered by Signal has been widely adopted by numerous other messengers, up to the point where it can be described as the industry standard.

What this systematization of knowledge does not cover is both the historical and future perspective. While the ecosystem has settled somewhat, we still see interesting new developments, especially considering the multi-device messenger niche. The MLS protocol [5] is currently being standardized and innovations in multi-device messaging have been announced by Threema [41], with rumors hinting at possible steps forward by WhatsApp. We also did not investigate whether or not fields adjacent to messaging, where devices communicate in different contexts, have something to offer to the instant messaging ecosystem, or if ideas from instant messaging could be applied to make other contexts more secure.

While MLS improves the scalability of group messaging, it does not directly solve the multi-device problem according to the differentiation of group and multi-device messaging we gave in the introduction. Therefore, future work should also research how the MLS protocol can be utilized for multi-device. Some things we see here is whether devices or users form the leafs of the MLS tree and, assuming leafs represent the users, how the devices of a user synchronize each other. Also related to group messaging, it needs to be assessed how MLS can be employed in a decentralized environment.

Furthermore, the usage of threshold cryptography in the messaging context can be further evaluated. For that, we see the need for user research whether this is actually a desired feature and whether users are able to understand this.

From a non-academic perspective, we believe that the differences we highlighted also present significant challenges to consumers. The label “secure messenger” is very broad, and it is not trivial to differentiate between various definitions of security. We have yet to identify a good communication strategy that allows potentially uninformed users to quickly identify which messengers correspond to their specific security needs.

## 5 Conclusion

The secure messaging ecosystem is diverse and varied. A lot of different messengers claim to achieve security, but for definitions of security that differ in sometimes significant, and often non-obvious ways. For messengers that support multi-device messaging the situation is no different. With this systematization of knowledge, we tried to simplify the process of gaining an overview over the current situation of the secure multi-device messaging ecosystem. We identified and described different methodologies for transferring data securely between devices, and evaluated and compared messengers in regard to different security, usability, and privacy objectives, such as forward secrecy, post-compromise security, sending complexity, and device opacity.

Next, we presented device management which covers the addition and removal of devices as well as recovery. In the beginning, we explained the concept of user and device level identities. We identified and explained the different patterns that exist for linking devices to an account, namely user level encryption, sharing an identity key, and cross-signing. Further, we considered the initial setup and outlined different approaches to share required information. After that, we presented trade-offs with regard to synchronizing chat history with new devices. For the removing of devices, we showed differences depending on user and device level keys. We presented different ways how recovery of accounts, messages, and verifications can be achieved. Lastly, we outlined some security considerations and evaluated messengers regarding security and usability criteria.

With this, we believe we have covered some of the most important points for understanding the differences between these messengers and gave a good starting point for understanding this specific aspect of the messaging ecosystem in particular.

We do not recommend any messenger over another, since even with our very limited criteria there is no messenger that fulfills all criteria better than all other messengers. There are still trade-offs to be made, which also means that there is still room for improvements. Whether these improvements mean an actual perceivable security, privacy, or usability increase that consumers want is not something we investigated. A systematization of knowledge that focuses more on user needs and perceptions could help to further classify the different approaches that we identified.

## Bibliography

- [1] ADAMANT Tech Labs. *ADAMANT: THE FUTURE OF MESSAGING*. Whitepaper v. 1.5.2. URL: <https://web.archive.org/web/20201109011600/https://adamant.im/whitepaper/adamant-whitepaper-en.pdf> (visited on Mar. 6, 2021).
- [2] Apple Inc. *Apple Platform Security*. Feb. 2021. URL: [https://web.archive.org/web/20210316093022/https://manuals.info.apple.com/MANUALS/1000/MA1902/de\\_DE/apple-platform-security-guide-d.pdf](https://web.archive.org/web/20210316093022/https://manuals.info.apple.com/MANUALS/1000/MA1902/de_DE/apple-platform-security-guide-d.pdf) (visited on Mar. 1, 2021).
- [3] Erinn Atwater and Urs Hengartner. “Shatter: Using Threshold Cryptography to Protect Single Users with Multiple Devices”. In: *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks - WiSec '16*. The 9th ACM Conference. Darmstadt, Germany: ACM Press, July 2016, pages 91–102. DOI: 10.1145/2939918.2939932.
- [4] Alex Balducci and Jake Meredith. *Olm Cryptographic Review*. Version 2.0. 2016. URL: [https://www.nccgroup.com/globalassets/our-research/us/public-reports/2016/november/ncc\\_group\\_olm\\_cryptographic\\_review\\_2016\\_11\\_01.pdf](https://www.nccgroup.com/globalassets/our-research/us/public-reports/2016/november/ncc_group_olm_cryptographic_review_2016_11_01.pdf) (visited on Jan. 5, 2021).
- [5] R. Barnes, B. Beurdouche, J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. *The Messaging Layer Security (MLS) Protocol*. Internet-Draft. Work in Progress. Internet Engineering Task Force, Dec. 22, 2020. URL: <https://tools.ietf.org/pdf/draft-ietf-mls-protocol-11.pdf> (visited on Mar. 7, 2021).
- [6] Jon Callas, Lutz Donnerhacke, Hal Finney, David Shaw, and Rodney Thayer. *OpenPGP Message Format*. RFC 4880. Nov. 2007. DOI: 10.17487/RFC4880. URL: <https://tools.ietf.org/html/rfc4880> (visited on Dec. 9, 2020).
- [7] Sébastien Campion, Julien Devigne, Céline Duguey, and Pierre-Alain Fouque. “Multi-Device for Signal”. In: *Applied Cryptography and Network Security*. Edited by Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi. Volume 12147. Rome, Italy: Springer International Publishing, 2020, pages 167–187. DOI: 10.1007/978-3-030-57878-7\_9.
- [8] Hubert Chathi. *MSC1680: cross-signing of devices to simplify key verification*. GitHub. Nov. 26, 2018. URL: <https://github.com/matrix-org/matrix-doc/blob/d5df3947dc60dd26ebb35a7b34299ab68b8a09db/proposals/1680-cross-signing.md> (visited on Mar. 5, 2021).

## Bibliography

- [9] Hubert Chathi, David Baker, Travis Ralston, noteness, Matthew Hodgson, Erik Johnston, and Andrew Morgan. *MSC1946: Secure Secret Storage and Sharing*. May 21, 2020. URL: [https://github.com/matrix-org/matrix-doc/blob/1f7cba9bdad12fde7b716079d6caf0dc9e9b5850/proposals/1946-secure\\_server-side\\_storage.md](https://github.com/matrix-org/matrix-doc/blob/1f7cba9bdad12fde7b716079d6caf0dc9e9b5850/proposals/1946-secure_server-side_storage.md) (visited on Dec. 16, 2020).
- [10] Hubert Chathi, J. Ryan Stinnett, Matthew Hodgson, David Baker, and Richard van der Hoff. *MSC1756: Cross-signing devices with device signing keys*. Jan. 8, 2021. URL: <https://github.com/matrix-org/matrix-doc/blob/76788843761491a2f6d31befa88775997db676fe/proposals/1756-cross-signing.md> (visited on Mar. 5, 2021).
- [11] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. "On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Toronto Canada: ACM, Oct. 2018, pages 1802–1819. DOI: 10.1145/3243734.3243747.
- [12] *Comparison of cross-platform instant messaging clients*. In: *Wikipedia*. Page Version ID: 997597707. Jan. 1, 2021. URL: [https://en.wikipedia.org/w/index.php?title=Comparison\\_of\\_cross-platform\\_instant\\_messaging\\_clients&oldid=997597707](https://en.wikipedia.org/w/index.php?title=Comparison_of_cross-platform_instant_messaging_clients&oldid=997597707) (visited on Jan. 2, 2021).
- [13] Facebook Inc. *Facebook to Acquire WhatsApp*. Feb. 19, 2014. URL: <https://web.archive.org/web/20210325210748/https://about.fb.com/news/2014/02/facebook-to-acquire-whatsapp/> (visited on Jan. 25, 2021).
- [14] Facebook Inc. *Messenger Secret Conversations: Technical Whitepaper*. May 18, 2017. URL: <https://about.fb.com/wp-content/uploads/2016/07/messenger-secret-conversations-technical-whitepaper.pdf> (visited on Nov. 6, 2020).
- [15] Lennart Grahl and Danilo Bargen. *SaltyRTC – End-to-End-Encrypted Signalling*. 2019. URL: <https://github.com/saltyrtc/saltyrtc-meta/blob/e3e86b67427002c2e56051d20471b566c84fdb2d/Protocol.md> (visited on Mar. 6, 2021).
- [16] Amir Herzberg and Hemi Leibowitz. "Can Johnny finally encrypt?: evaluating E2E-encryption in popular IM applications". In: *Proceedings of the 6th Workshop on Socio-Technical Aspects in Security and Trust - STAST '16*. Los Angeles, California: ACM Press, Dec. 2016, pages 17–28. DOI: 10.1145/3046055.3046059.
- [17] Richard van der Hoff, Matthew Hodgson, Mark Haines, Aaron Raimist, and Hubert Chathi. *Megolm group ratchet*. GitLab. Sept. 17, 2020. URL: <https://gitlab.matrix.org/matrix-org/olm/-/blob/4bae4134eb115859038de4e8eab11b22baf80b0c/docs/megolm.md> (visited on Jan. 5, 2021).
- [18] Chris Howell, Tom Leavy, and Joël Alwen. *Wickr Messaging Protocol: Technical Paper*. 2017. URL: [https://wickr.com/wp-content/uploads/2019/12/WhitePaper\\_WickrMessagingProtocol.pdf](https://wickr.com/wp-content/uploads/2019/12/WhitePaper_WickrMessagingProtocol.pdf) (visited on Mar. 6, 2021).

## Bibliography

- [19] Keybase. In: *Wikipedia*. Page Version ID: 1002158043. Jan. 23, 2021. URL: <https://en.wikipedia.org/w/index.php?title=Keybase&oldid=1002158043> (visited on Mar. 5, 2021).
- [20] Keybase. *Keybase Book - Chat - Crypto - High Level Overview*. 2020. URL: <https://web.archive.org/web/20210121143454/https://book.keybase.io/docs/chat/crypto> (visited on Mar. 3, 2021).
- [21] Keybase. *Keybase Book - Crypto - Keybase Key Exchange (KEX) Protocol*. 2020. URL: <https://web.archive.org/web/20210117001349/https://book.keybase.io/docs/crypto/key-exchange> (visited on Mar. 6, 2021).
- [22] Keybase. *Keybase Book - Server: Meet your sigchain (and everyone else's)*. 2020. URL: <https://web.archive.org/web/20210205195623/https://book.keybase.io/docs/server> (visited on Mar. 6, 2021).
- [23] Keybase. *Keybase Book - Your Keybase Account: Proofs*. 2020. URL: <https://web.archive.org/web/20210308150800/https://book.keybase.io/account#proofs> (visited on Mar. 6, 2021).
- [24] Silas Lenz. "Evaluation of the Messaging Layer Security Protocol - A Performance and Usability Study". Master Thesis. Linköping University, 2020. URL: <https://www.diva-portal.org/smash/get/diva2:1388449/FULLTEXT01.pdf> (visited on Nov. 30, 2020).
- [25] Moxie Marlinspike. *A New Home*. Jan. 21, 2013. URL: <https://web.archive.org/web/20210307040529/https://signal.org/blog/welcome/> (visited on Jan. 25, 2021).
- [26] Moxie Marlinspike. *The Double Ratchet Algorithm*. Edited by Trevor Perrin. Nov. 20, 2016. URL: <https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf> (visited on Nov. 8, 2020).
- [27] Moxie Marlinspike. *The Sesame Algorithm: Session Management for Asynchronous Message Encryption*. Edited by Trevor Perrin. Apr. 14, 2017. URL: <https://signal.org/docs/specifications/sesame/sesame.pdf> (visited on Nov. 4, 2020).
- [28] Moxie Marlinspike. *The X3DH Key Agreement Protocol*. Edited by Trevor Perrin. Nov. 4, 2016. URL: <https://signal.org/docs/specifications/x3dh/x3dh.pdf> (visited on Jan. 10, 2021).
- [29] Microsoft Corporation. *Skype Private Conversation: Technical white paper*. June 20, 2018. URL: <https://web.archive.org/web/20201112015447/https://az705183.vo.msecnd.net/onlinesupportmedia/onlinesupport/media/skype/documents/skype-private-conversation-white-paper.pdf> (visited on Nov. 20, 2020).
- [30] Proton Team. *Introducing ProtonMail Contacts - the world's first encrypted contacts manager*. ProtonMail Blog. Nov. 21, 2017. URL: <https://web.archive.org/web/20210128124045/https://protonmail.com/blog/encrypted-contacts-manager/> (visited on Dec. 19, 2020).

## Bibliography

- [31] Raphael Robert. *Messaging Layer Security: Towards a new era of secure messaging...* BlackHat USA, Las Vegas, CA, 2019. URL: <https://i.blackhat.com/USA-19/Wednesday/us-19-Robert-Messaging-Layer-Security-Towards-A-New-Era-Of-Secure-Group-Messaging.pdf> (visited on Dec. 8, 2020).
- [32] Florian Schmaus, Dominik Schürmann, and Vincent Breitmoser. *OpenPGP for XMPP Instant Messaging*. XMPP Extension Protocol. Publisher: XMPP Standards Foundation. Jan. 2018. URL: <https://web.archive.org/web/20210317203840/https://xmpp.org/extensions/xep-0374.html> (visited on Jan. 8, 2021).
- [33] Signal Support. *Signal PIN*. 2021. URL: <https://web.archive.org/web/20210215104456/https://support.signal.org/hc/en-us/articles/360007059792-Signal-PIN> (visited on Mar. 6, 2021).
- [34] Skype. In: *Wikipedia*. Page Version ID: 1009529819. Mar. 1, 2021. URL: <https://en.wikipedia.org/w/index.php?title=Skype&oldid=1009529819> (visited on Mar. 5, 2021).
- [35] surespot. *faq and support*. URL: <https://web.archive.org/web/20200919160030/https://www.surespot.me/support.html> (visited on Dec. 18, 2020).
- [36] H. Tankovska. *Most popular global mobile messenger apps as of January 2021, based on number of monthly active users*. Feb. 10, 2021. URL: <https://web.archive.org/web/20210331203031/https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/> (visited on Mar. 7, 2021).
- [37] Telegram Messenger Inc. *Perfect Forward Secrecy*. URL: <https://web.archive.org/web/20210119184017/https://core.telegram.org/api/end-to-end/pfs> (visited on Feb. 28, 2021).
- [38] Telegram Messenger Inc. *Telegram FAQ*. URL: <https://web.archive.org/web/20210412224142/https://telegram.org/faq> (visited on Feb. 28, 2021).
- [39] Telegram Messenger Inc. *Telegram Privacy Policy*. 2018. URL: <https://web.archive.org/web/20201207171726/https://telegram.org/privacy> (visited on Dec. 5, 2020).
- [40] Threema GmbH. *About*. 2021. URL: <https://web.archive.org/web/20210406152638/https://threema.ch/en/about> (visited on Jan. 25, 2020).
- [41] Threema GmbH. *Threema Multi-Device: An Architectural Overview*. Nov. 3, 2020. URL: <https://web.archive.org/web/20210317082506/https://threema.ch/en/blog/posts/md-architectural-overview> (visited on Mar. 6, 2021).
- [42] Threema GmbH. *Threema. Cryptography Whitepaper*. June 9, 2020. URL: [https://web.archive.org/web/20210202063215/https://threema.ch/press-files/2\\_documentation/cryptography\\_whitepaper.pdf](https://web.archive.org/web/20210202063215/https://threema.ch/press-files/2_documentation/cryptography_whitepaper.pdf) (visited on Nov. 9, 2020).
- [43] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. "SoK: Secure Messaging". In: *2015 IEEE Symposium on Security and Privacy*. San Jose, CA, USA, May 2015, pages 232–249. DOI: 10.1109/SP.2015.22.

## Bibliography

- [44] *Viber*. In: *Wikipedia*. Page Version ID: 1001166315. Jan. 18, 2021. URL: <https://en.wikipedia.org/w/index.php?title=Viber&oldid=1001166315> (visited on Mar. 5, 2021).
- [45] Viber Media S.à r.l. *Viber Encryption Overview*. 2018. URL: <https://www.viber.com/app/uploads/viber-encryption-overview.pdf> (visited on Mar. 1, 2021).
- [46] WhatsApp LLC. *Two Billion Users – Connecting the World Privately*. Feb. 12, 2020. URL: <https://web.archive.org/web/20210309160821/https://blog.whatsapp.com/two-billion-users-connecting-the-world-privately/> (visited on Jan. 25, 2021).
- [47] WhatsApp LLC. *WhatsApp Encryption Overview: Technical white paper*. Oct. 22, 2020. URL: <https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf> (visited on Mar. 6, 2021).
- [48] WhatsApp LLC. *WhatsApp Help Center - About WhatsApp Web and Desktop*. 2021. URL: <https://web.archive.org/web/20210406152706/https://faq.whatsapp.com/web/download-and-installation/about-whatsapp-web-and-desktop/?lang=en> (visited on Mar. 6, 2021).
- [49] WhatsApp LLC. *WhatsApp Help Center - How to reset your two-step verification PIN*. 2021. URL: [https://web.archive.org/web/20210108215131if\\_/https://faq.whatsapp.com/general/verification/how-to-reset-your-two-step-verification-pin/?lang=fb](https://web.archive.org/web/20210108215131if_/https://faq.whatsapp.com/general/verification/how-to-reset-your-two-step-verification-pin/?lang=fb) (visited on Mar. 6, 2021).
- [50] *Wickr*. In: *Wikipedia*. Page Version ID: 1000507737. Jan. 15, 2021. URL: <https://en.wikipedia.org/w/index.php?title=Wickr&oldid=1000507737> (visited on Mar. 5, 2021).
- [51] *Wire (software)*. In: *Wikipedia*. Page Version ID: 1010308615. Mar. 4, 2021. URL: [https://en.wikipedia.org/w/index.php?title=Wire\\_\(software\)&oldid=1010308615](https://en.wikipedia.org/w/index.php?title=Wire_(software)&oldid=1010308615) (visited on Mar. 5, 2021).
- [52] Wire Swiss GmbH. *Why can't I see my conversation history?* 2021. URL: <https://web.archive.org/web/20201129082459/https://support.wire.com/hc/en-us/articles/207834645> (visited on Mar. 6, 2021).
- [53] Wire Swiss GmbH. *Wire Security Whitepaper*. Oct. 22, 2020. URL: <https://web.archive.org/web/20210406152649/https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf> (visited on Mar. 6, 2021).