HACKMANIT

Interoperability between Messaging Services Secure Implementation of Encryption

Study for the Federal Network Agency

Version: FINAL VERSION
28.04.2023

Prof. Dr. Paul Rösler, Prof. Dr. Jörg Schwenk
Phone: 0234/54459996 | E-Mail: joerg.schwenk@hackmanit.de

Project Information

Customer:	Bundesnetzagentur Bonn Referat 121 Tulpenfeld 4, 53113 Bonn
Contact:	121-Postfach@BNetzA.de
Commissioned to:	Hackmanit GmbH Universitätsstraße 60 (Exzenterhaus) 44789 Bochum, Germany
Project executive:	Prof. Dr. Jörg Schwenk Phone: +49 (0)234 / 54459996 Fax: +49 (0)234 / 54427593 E-Mail: joerg.schwenk@hackmanit.de
Project members:	Prof. Dr. Paul Rösler, +49 (0) 9115302-96641, paul.roesler@fau.de Prof. Dr. Jörg Schwenk, +49 (0)234 / 54459996, joerg.schwenk@hackmanit.de
Project period:	2022-10-18 – 2023-04-10
Version of the report:	Final Version

This report was technically verified by project members.
This report was linguistically verified by project members.

Hackmanit GmbH
Represented by: Prof. Dr. Jörg Schwenk, Prof. Dr. Juraj Somorovsky,
Dr. Christian Mainka, Prof. Dr. Marcus Niemietz
Register court: Amtsgericht Bochum, HRB 14896, Germany

Contents

List of Figures

List of Tables

Glossary

AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard; a block cipher
AKE	Authenticated Key Exchange
CA	Certification Authority
CCA	Chosen Ciphertext Attack
ChaCha20	stream cipher
CMA	Chosen message attack
CO	Ciphertext only
Competitor	(IM) company with small market share
CPA	Chosen plaintext attack
DES	Data Encryption Standard
DH	Diffie-Hellman
DHKE	Diffie-Hellman Key Exchange
DMA	Digital Markets Act
DSA	Digital Signature Algorithm
DSS	Digital Signature Standard
EC	elliptic curve; a mathematical structure
ECDH	elliptic curve DHKE
ElGamal encryption	named after Taher ElGamal
ElGamal signature	named after Taher ElGamal
Encryption	mechanism to protect the confidentiality of data
EUF	Existential Unforgeability
E2EE	End-to-end encryption
FS	Forward Secrecy
Gatekeeper	(IM) company with dominant market share
GF	Galois field
HMAC	Hashed MAC
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IKE	Internet Key Exchange
IM	Instant Messaging
IND	Indistinguishability
IP	Internet Protocol
IPsec	IP Security
JSON	JavaScript Object Notation
KDF	Key Derivation Function
KEM	Key Encapsulation Mechanism
KPA	Known plaintext attack
MAC	Message Authentication Code
MD5	Message Digest 5; a hash function
Megolm	Group messaging protocol of Matrix, based on Signal's Sender Key
MIME	Multipurpose Internet Mail Extensions
MTProto	Two-party messaging and transport protocol of Telegram
Olm	Two-party messaging protocol of Matrix, based on Signal's Double Ratchet
OpenPGP	Open PGP, the standard for PGP

OTR	Off-The-Record protocol
OW	One-Way
PCS	Post-Compromise Security
PGP	Pretty Good Privacy
PKCS	Public Key Cryptography Standard
Proteus	Two-party messaging protocol of Wire, based on Signal's Double Ratchet
Ratcheting	a technique to continuously update keys
RC4	Rivest Cipher 4; a stream cipher
RFC	Request for Comments; name of IETF standards
RSA	Rivest-Shamir-Adleman algorithm; a public key encryption and signature algorithm
Sender Key	Group messaging protocol desgined by Signal and used, e.g., in WhatsApp
SAML	Security Assertion Markup Language
SHA	Secure Hash Algorithm; two families of hash algorithms
S/MIME	Secure MIME
SMS	Short Message Service
SMTP	Simple Mail Transfer Protocol
SRTP	Secure Real-Time Protocol
SSH	Secure SHell
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TOFU	Trust On First Use
TLS	Transport Layer Security
UDP	User Datagram Protocol
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WebRTC	a framework for real-time communication in the web
X3DH	The X3DH Key Agreement protocol
X.509	standard format for digital certificates
XMPP	Extensible Messaging and Presence Protocol

List of Symbols

$\mathbb{Z}$	the set of integers
$\mathbb{Z}_n$	the set of integers modulo n
$\mathbb{Z}_n^*$	the set of integers modulo n which are coprime to n

Executive Summary

In essence this study contains the following contributions.

Part 1: Real-World Crypto (Section 2) Here, the building blocks and constructions from cryptographic applications deployed on a large scale are introduced. *Encryption* is the mechanism to protect the confidentiality of a message; with the help of *keys*, encryption algorithms make the plaintext of a message unreadable, except for the recipient who can use another key to restore the plaintext. The *Diffie-Hellman Key Exchange* (DHKE) helps to establish the secret keys needed for encryption at both the sender and recipient. *Hash functions* compress the content of large messages into small *hash values*, in a secure way. *Message Authentication Codes* (MAC) and *digital signatures* both protect the integrity and authenticity of a message; they ensure that the message cannot be altered during transmission. In real-world applications, these building blocks are combined to achieve more sophisticated security goals (e.g., Perfect Forward Secrecy). Important real-world applications are *Authenticated Key Exchange* (AKE) protocols: *Transport Layer Security* (TLS) is used to secure web shops and online banking, and to protect the user-to-provider communication in instant messaging (IM) systems. The Noise framework is also used in IM systems; it implements a secure communication channel that is more geared towards high efficiency and offers less flexibility compared with TLS. The X3DH key exchange protocol is used in many IM applications to establish an initial shared secret when users start a conversation. Finally, the psychological and legal concept of *Trust* is considered in the context of cryptography; different techniques have been developed to securely extend trust from small groups to larger ones using mainly digital signatures.

Part 2: Instant Messaging (IM, Section 3) IM apps provide several basic functionalities which may be subject to interoperability regulations under the *Digital Martkets Act* (DMA). In this second part, we explain how building blocks and construction from Part 1 are used to secure these basic functionalities. After installation of an IM app at a client, this new IM client must be securely integrated into the IM network (authentication). It must be able to *discover the identity* of other IM clients, to *exchange initial keys* with these clients, and to *establish trust* between any pair of IM clients. The basic functionality of IM apps upon which all others are based is the secure exchange of short text messages. Most IM apps secure these message with end-to-end encryption (E2EE)—no network operator can read the contents of these messages, not even the IM provider itself. To establish the keys for E2EE, a wide variety of mechanisms are used, ranging from simple AKE protocols (e.g., in Telegram) to sophisticated key update mechanisms called *Ratcheting* which provide novel security guarantees (e.g., in the Signal, WhatsApp messengers). *File attachments*, *Group messaging* and *real-time communications* are based on the E2EE exchange of text messages. Attached files are stored on a server in encrypted form, and the location where to retrieve the file and the key to decrypt it are sent to the recipient via an E2EE text message. For group messaging, either group keys are exchanged via E2EE text messages, or, instead of encrypting group messages with the same group key to all other members, the same message is duplicated and sent to each member of the group in a separate E2EE message channel. For audio and video calls, typically protocols like the *Secure Real-Time Protocol* (SRTP) are used, and again the keys to encrypt the audio/video fragments are exchanged over E2EE text messages. We conclude this part by giving an overview on the implementation of these basic services in different IM applications like Signal, WhatsApp, Facebook Messenger, Wire, Matrix, iMessage, and Telegram.

Part 3: Interoperable Instant Messaging (Section 4) Since the basic functionalities described in Part 2 are typically implemented differently in different IM applications, interoperability between these applications cannot be achieved by simply connecting their architectures. Instead, some kind of "translation" between the different transmission and encryption protocols and their underlying data formats must be added. We first systematically describe options where an *interface* for this translation could be located between the communicating users of the different providers. Importantly, we take into account that the larger provider, called *gatekeeper* in the DMA and in this report, is responsible for specifying how interoperability is implemented. To maintain E2EE, we see two feasible options and describe them in more detail: (1) provision of cryptographic interfaces, which is possibly supported by libraries of the gatekeeper and (2) standardization of a new interoperability protocol. We sketch interoperability-enabling solutions for both approaches, for the basic functionalities described in Part 2. We summarize our elaboration by describing an example of a mixed approach, where standardization of simple functionalities is combined with gatekeeper libraries included into the competitors IM clients.

In both considered cases (standardization as well as documented provision of interfaces), we draw the conclusion that preserving end-to-end security for interoperable communication is achievable with available technical building blocks. For this, our focus in this study is the composition and connection of IM networking protocols and cryptographic protocols that enable functional interoperability with the required level of confidentiality and authenticity between the communicating users.

Since the timeline for deployment of interoperable IM required by the DMA is relatively short, we also discuss how quickly the different technical solutions can be realized and how far short-term solutions are compatible with possibly more sustainable, long-term solutions. Seemingly, the most realistic short-term solution is that gatekeepers open interfaces to their architecture, document their protocols, and provide libraries that the competitors' client applications embed. These libraries take plaintext messages and translate them into ciphertexts of the gatekeeper's protocols. Besides the required trust in the gatekeeper in case the above solution is based on the gatekeeper's library, this solution is generally only viable as long as a single or only few gatekeepers are declared as such. Otherwise, each competitor would need to implement one stack of protocols for interoperability with each gatekeeper, which increases complexity to a barely manageable level. Therefore, in the long run, a standardized interoperability protocol suite seems to be the most sustainable solution. However, since the DMA currently permits the gatekeepers to specify how competitors can interoperably communicate with them, it is yet unclear if they would voluntarily adopt such a standard.

1 Introduction

1.1 Overview

Online communication services such as messaging services have become an integral part of everyday life and are now as widespread as traditional telecommunications services. A consumer survey by the German Federal Network Agency [24] shows that 88 % of the German population regularly use such services provided over the Internet for communication. Among German users, the messaging services WhatsApp (93 %), Facebook Messenger (39 %) and Instagram Direct Messages (25 %) are the most widespread. All three services are provided by Meta Platforms, Inc. (formerly Facebook). Competing services from other providers are much less widespread. However, almost three quarters of respondents use at least two different services in parallel (multihoming).

The observable market concentration in the messaging sector can be attributed to so-called network effects. Users of a particular messaging service benefit from a higher number of users of the same service, as typically only these users can communicate with each other. On the other hand, it is easily possible to use several messaging services in parallel.

In order to increase competition, interoperability obligations for providers of messaging services are being discussed[1]. The aim of such regulation is to break up the market power of dominant providers and reduce dependencies. The demand is to make it obligatory for providers of competing messaging services to open up the communication networks, which up to now have been predominantly closed. This interoperability between different messaging services enables users to communicate across providers.

The amendment to the German Telecommunications Act (TKG) has included messenger services in parts of the regulatory regime since December 2021 as so-called number-independent interpersonal communications services (NI-ICS). The law provides for the possibility of imposing interoperability obligations for messenger services under certain conditions.

The Digital Markets Act (DMA) also contains interoperability obligations for providers of central platform services who hold a gatekeeper position. This also applies to providers who provide NI-ICS. These must enable cross-provider communication with competitor services. Implementation of the DMA will be phased in after a provider's gatekeeper status is confirmed:

- After 6 months, basic functions such as message exchange, sending images and voice messages between two users (1:1 communication) must be enabled interoperably.
- After 2 years, these basic functions must also be available interoperably for group communication (1:n communication).
- After 4 years, (video) telephony functions must be designed to be interoperable both as 1:1 and as 1:n communication.

Gatekeepers are required to prepare a reference offer with regard to the exact technical interoperability conditions. Data security must not be reduced by the interoperability obligations, i.e., existing end-to-end encryption must still be possible. Users of the services should continue

[1] See Bundesnetzagentur (2021): Interoperability between messaging services. An overview of potential and challenges. www.bnetza.de/InteropMessenger

to be able to decide for themselves whether they want to use the interoperable communication options.

A central challenge in the implementation of interoperability commitments is to ensure the highest possible level of data protection and data security. This applies to both the processing of communication data and its encryption. In particular, end-to-end encryption ensures a confidential exchange of messages between users. However, there are different encryption methods that are assessed differently in terms of their security. End-to-end encryption is now implemented by many providers of NI-ICS, but there have been hardly any analyses to date of how end-to-end encryption can be implemented technically and organizationally for messenger services that are committed to interoperability.

Goal of this Study The primary goal of this study is to explain and discuss possible technical approaches to realize interoperability obligations for messaging services against the backdrop of the DMA. The focus of this study is on data security in general, and end-to-end encryption (E2EE) in particular.

Methodology The contents of this study are based on a comprehensive review of the scientific literature related to real-world cryptography and IM security, and on technical documentation provided by IM providers. When describing and discussing existing solutions, we prioritize scientific papers over documentations from the considered providers, because by nature the former are more objective towards their study target.

Structure of this Study This study has 3 parts:

- **Part 1: Real-World Cryptography.** In this part, the state-of-the art in cryptography used in real-world applications like file encryption, TLS and e-mail encryption is explained and documented. It starts with cryptographic building blocks like encryption algorithms and concludes with overviews on complex cryptographic protocols like key exchange.
- **Part 2: Instant Messaging.** In this part, cryptographic constructions are explained that are used to implement the basic functionality of IM apps. This includes key management for E2EE of text messages, file attachments (e.g., images or audio), group communication, and real-time communications (i.e., live audio and video calls).
- **Part 3: Interoperable Instant Messaging.** Here we first give an overview of different options how to implement interoperability in general, and select the two options which are most compliant with DMA regulations for deeper inspection. These two options are the specification of interfaces to the gatekeeper's infrastructure, and, alternatively, standardization. The former means that a new protocol for interoperable communication is developed from scratch. And the latter means that one involved provider (e.g., the gatekeeper) opens its architecture to other providers (e.g., competitors), which is supported by extensive documentation and, possibly, a client library that these other providers can embed in their client application.

1.2 Related Work

Several published works discuss and analyze different aspects regarding Article 7 of the DMA [69]. We categorize these works by their authors and list them chronologically:

The Electronic Frontier Foundation published blog posts [107, 114, 29] in which they generally explain and discuss the effects of the DMA. The messaging project Matrix posted articles on their blog [60, 38, 37, 32, 39] with similar explanatory content, also discussing how security in interoperable IM can be preserved. The technical blog posts by Rescorla [72, 71, 70] primarily focus on understanding and proposing solutions for multiple components needed for implementing interoperable protocols. Brown [22] published a study for the OpenForum Academy in which he discusses multiple legislative, organizational, and technical aspects that need to be regarded for the implementaiton of the DMA. Blessing and Anderson [19] list a number of challenges and problems that implementations of interoperable messaging protocols will have to face. Len et al. [48] translate the security and privacy requirements of the DMA into a more technical, cryptographic language for three components of IM protocols: identification of users, communication protocol, and abuse prevention. For each of these components, they define generic protocol flows based on abstract interfaces, describe existing concepts, and then propose a unified construction. This approach is related to the API approach that we consider in Section 4.3. (For transparency reasons, we note that this work [48] is independently co-authored by one of the authors of the present study.) The IETF MIMI Working Group [27] started to develop and standardize an interoperable messaging protocol that can be commonly used to implement the DMA. This approach is related to the standardization approach that we consider in Section 4.4.

2 Instant Messaging: Shared Cryptographic Mechanisms

Kerckhoffs' principle Before describing the building blocks necessary to understand modern protocols for instant messaging, we briefly introduce a fundamental principle of modern cryptography.

For this, we note that until the 1970s, military users tried to keep both the encryption algorithms and the encryption key secret. However, since the algorithm had to be executed by many people, or it was implemented in many devices, relying on the secrecy of the algorithm is a bad idea. An evident example is the story of the Enigma: In World War 2, the German army assumed that the Enigma encryption machine, a mechano-electrical device, was unknown to the allied forces. However, Polish intelligence had already acquired some machines before the war, which were handed over to the British intelligence. The algorithm was reverse-engineered and broken, and this fact wasn't disclosed to the German government until the 1970s.

To prevent this type of attack, Auguste Kerckhoffs proposed what is since then known as *Kerckhoffs' principle* already in 1883 [46]:

> It [the system] should not require secrecy, and it should not be a problem if it falls into enemy hands.

This is a generally accepted principle today. All details of the encryption algorithms themselves are published and therefore 'known to the enemy'. Only the encryption keys must remain secret, and then can easily be changed. In all security notions introduced in the following, we therefore always assume that the algorithm itself is known to the attacker.

2.1 Building Blocks

In this section we cover all concepts of applied cryptography that are necessary to understand the specific cryptographic building blocks used in IM apps. This includes symmetric and public-key encryption. For more details, we refer the interested reader to established text books on (applied) cryptography: [15, 59, 45, 20].

2.1.1 Symmetric Encryption

Symmetric encryption requires that the sender and the receiver of a message, and only these two, have a shared secret, the so-called *key*. The sender encrypts payload data, turning that data into a so called *ciphertext*, and the receiver decrypts the ciphertext to obtain the original payload data. We note that anyone with access to the key can be sender *and* receiver.

Mental Model We can visualize the symmetric encryption of a message, as shown in Figure 1, as sending a message locked in a safe: To open the safe, you need the same key that was used to lock it. Two aspects of the security of a symmetric encryption method are illustrated in this mental model: the security of the algorithm itself through the thick steel walls of the safe and the locking mechanism in the door, and the complexity of the key, which must not be easy to reproduce.

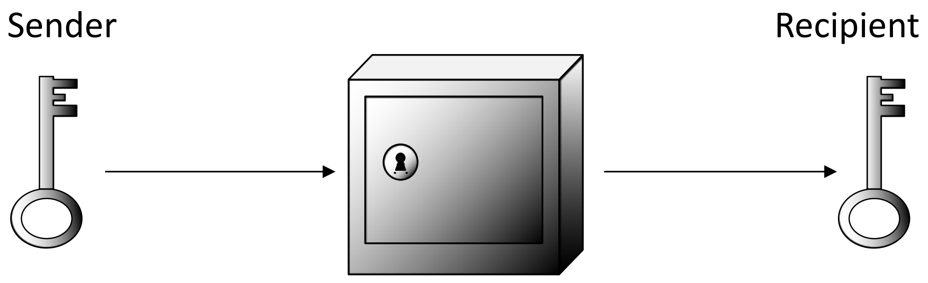

Figure 1: Mental model of symmetric encryption.

Notation In the following, we will use the notation

$$c \leftarrow \mathsf{Enc}_k(m)$$

to describe that evaluating an *encryption algorithm* Enc parametrized with a *key* k takes the *message*/the plaintext m and results in ciphertext c. Intuitively, this ciphertext reveals no information about the message to anyone who has no access to the key. Yet, using the key, the message can be recovered from the ciphertext. Please note that both the plaintext m and the ciphertext c are treated as sequences of bits here, since each 'message' has a representation as such a sequence of bits, depending on the character set used.

Thus, the plaintext m is recovered from the ciphertext c by evaluating a *decryption algorithm* Dec with the same key k:

$$m \leftarrow \mathsf{Dec}_k(c)$$

When encrypting a message, a random value can be included, resulting in a *probabilistic* encryption algorithm. This is often desired to achieve certain security goals. In this case, a dollar sign above the arrow indicates that the encryption algorithm can produce different outputs for the same input:

$$c \overset{\$}{\leftarrow} \mathsf{Enc}_k(m)$$

The decryption must always be deterministic since the result should be the same plaintext that the sender encrypted. So the following equation should always be valid:

$$m = \mathsf{Dec}_k(\mathsf{Enc}_k(m))$$

Security notions for symmetric encryption There is no 'absolute' security—only security relative to well-defined preconditions and protection goals.

The preconditions define the starting point of the attack: Do attackers only know the ciphertext, or do they also know (parts of) the plaintext? Do they have access to a black-box machine which allows them to encrypt or decrypt messages without learning the key? Preconditions are not always realistic, but exaggerated: We want an encryption algorithm to be 'secure' even if the attackers have access to a black box which decrypts arbitrary ciphertexts. The reasons for these exaggerations lie in the fact that such preconditions may be partly fulfilled in practice. For example, the assumption that attackers know the *complete* plaintext for a given ciphertext may be exaggerated, but in practice it often occurs that they know *a part* of the plaintext: In e-mail encryption, an attacker always knows the plaintext of the first ciphertext block, since standardized message encodings—the *Multipurpose Internet Mail Extensions* (MIME) headers—are encrypted there.

The protection goal depends on the preconditions: If attackers already know the plaintext to a given ciphertext, then it is pointless to require that this particular plaintext remains confidential; however, the secret key should not be computable even if plaintext and ciphertext are known. The strongest protection goal in symmetric encryption is that attackers can learn *nothing* from a ciphertext. This is formalized as *indistinguishability of ciphertexts*: If two plaintexts m_0 and m_1 are known to the attacker, and one of these plaintexts is randomly chosen and encrypted with a secret key k, the attacker, given the resulting ciphertext c, should not be able to determine which one was encrypted.

Standard security goals are often abbreviated in a two-part notion: The first part indicates the protection goal, and the second part the preconditions for the attack. We provide a non-exhaustive list to give some examples [108]:

- **OW-CO** One-way (OW) denotes the security goal to decrypt the plaintext from the ciphertext, and ciphertext-only (CO) denotes the fact that the attacker only knows the ciphertext and nothing else.
- **OW-KPA** KPA stand for *known plaintext attack*. Even if attackers know one or more ciphertext-plaintext pairs (KPA), they shouldn't be able to compute the plaintext of another ciphertext.
- **IND-CPA** Even if attackers have access to a black box which accepts plaintexts chosen by the attacker (*chosen plaintext attack*, CPA) and outputs corresponding ciphertexts, they shouldn't be able to distinguish which of two plaintexts was actually encrypted (*indistinguishability*, IND).
- **IND-CCA** Even if there is a black box which accepts arbitrary ciphertexts chosen by attackers (*chosen ciphertext attack*, CCA), the attackers should not be able to distinguish which of two plaintexts was actually encrypted in a ciphertext c. To make this definition sound, technically the black box must refuse to decrypt this single target ciphertext c.

The preconditions and security goals can be combined and extended almost arbitrarily. The most basic security goal is that the secret key k must be protected from being computed by the attacker.

Block Ciphers Cryptographic constructions for symmetric encryption can be divided in two classes: block ciphers and stream ciphers.

When using a *block cipher* for encryption, the plaintext m is divided into smaller plaintext blocks m_i of fixed length. A typical block length is 128 bits, which equals 16 bytes. For each block, the algorithm is applied once, either directly (Figure 2), or using some feedback from previous block encryptions (different *modes of operation*). The algorithms needs a key k, which is a bit sequence of fixed key length.

The most important block ciphers are AES (Advanced Encryption Standard [58]), and, for legacy applications, DES (Data Encryption Standard [57]). In modern application like IM apps, DES is

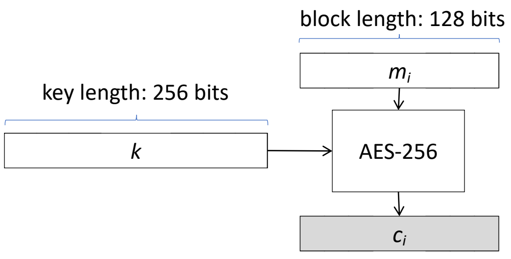

Figure 2: Parameters of the block cipher AES-256.

no longer used. AES may be used with three different key lengths: 128, 192 and 256 bits. AES-256 is depicted in Figure 2.

Each block cipher may be used in different *modes of operation*. The security of block ciphers heavily relies on the correct use of these modes. For more information see [108].

Padding in block ciphers If the length of the plaintext m is not a multiple of the block length of the block cipher, there may be an 'incomplete' plaintext block left at the end. For this, all practical applications of block ciphers indicate the length of the padding in the last byte. The padding itself may then be deterministic (RFC 2440, TLS) or probabilistic (SSL 3.0).

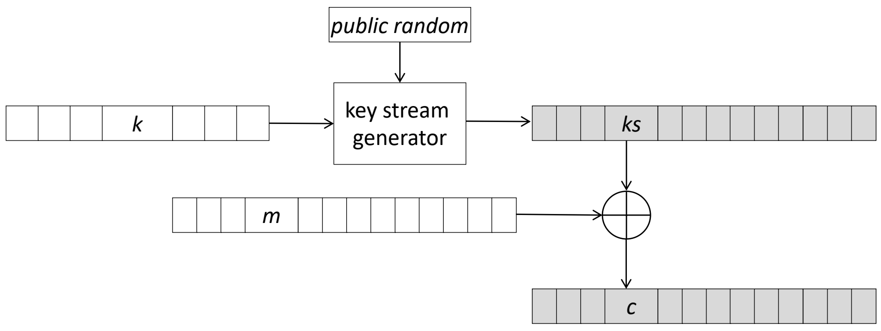

Figure 3: Encryption of a message m with a stream cipher.

Stream ciphers With *stream ciphers*, the plaintext m can be encrypted as a whole. A *key stream generator algorithm*, using as input the secret key k and randomized public data (e.g., frame counters, transmitted random numbers), generates a key stream *ks* which has the same length as m. For each bit in the plaintext m and the corresponding bit in the key stream *ks*, the XOR operation (Table 1) is then applied, resulting in a single bit of the ciphertext. Important stream ciphers are ChaCha20 [94], and, for legacy applications, RC4 [67].

(Un-)Authenticated Encryption Encryption may only protect the *confidentiality* of the plaintext, but not its integrity. This is obvious from the fact that all stream ciphers, and many block cipher modes of operation, use the XOR operation (Table 1) to combine ciphertext and plaintext. In these cases, inverting a bit in the ciphertext also inverts the corresponding plaintext bit. However, in 2002 it became clear that by exploiting the missing integrity of ciphertexts, also the confidentiality of the plaintexts can be compromised.

b	**c**	**b ⊕ c**
0	0	0
0	1	1
1	0	1
1	1	0

Table 1: The bitwise XOR of bits **b** and **c**.

Attacks against Confidentiality In 2002, Serge Vaudenay published an attack [118] on the then state-of-the-art CBC encryption mode of operation which exploited a weakness of the RFC 2440 padding to decrypt the plaintext—a so called *padding oracle*. The attack is very efficient—it is linear in the number of ciphertext bytes—and it works for all key lengths. Thus, this attack shows that weak integrity protection can be exploited to also undermine the confidentiality protection of encryption. Looking ahead (see the following paragraphs), modern authenticated encryption mechanisms solve this problem.

Hash functions Hash functions are cryptographic "fingerprints" which compress bit sequences m of variable length ($\{0,1\}^*$ denotes the set of all finite bit sequences) to a *hash value* of fixed length ℓ:

$$h \leftarrow H(m), h \in \{0,1\}^\ell, m \in \{0,1\}^*,$$

where $\{0,1\}^*$ refers to the set of of all finite bit sequences, and $\{0,1\}^\ell$ to the set of of all bit sequences of length exactly ℓ.

Hash functions should not be confused with error detecting or error correcting codes. These codes can locate single or multiple bit errors, but often messages can be modified without invalidating such codes. In contrast, hash values do not help to locate a bit error, but simply indicate that one or more bits were changed in a message. It should also be hard to modify a message in such a way that the hash value remains the same.

Security properties of hash functions can be memorized using the mental model of a human fingerprint:

- Collision Resistance: It should be hard to find two persons who have exactly the same fingerprint.
- 2nd Preimage Resistance: Given a person and their fingerprint, it should be hard to find a second person with exactly the same fingerprint.
- Preimage Resistance: Given only a fingerprint, it should be hard to find a person with this fingerprint.

SHA-2 is a modern, widely used hash functions that produced hash values of length 224, 256, 384 and 512, resp.; its successor SHA-3 outputs hash values of length 224, 256, 384 and 512 bits. Whenever possible, a hash function from the families SHA2 or SHA-3 should be used. Yet, MD5 and SHA-1 are two still widely used, but outdated hash functions. They produce hash values of length 128 and 160 bits, resp. For both of them it could be shown that they are not collision resistant, which makes various (not all) cryptographic systems based on them vulnerable to attacks.

Message Authentication Codes (MAC) A *message authentication code* (MAC) is a keyed cryptographic checksum, which can be computed either over the plaintext, or over the ciphertext.

A MAC tag τ for a message m is calculated using the MAC function $\mathsf{MAC}_{()}()$ and a MAC key k. The message m may have an arbitrary length, and the tag τ and the key k have a fixed length:

$$\tau \leftarrow \mathsf{MAC}_k(m)$$

A MAC tag can only be verified by receivers who know the secret key k:

$$\mathsf{TRUE}/\mathsf{FALSE} \leftarrow \mathsf{MAC.Verify}_k(m, \tau)$$

For lack of a better mental model, a MAC can be imagined as an encrypted fingerprint of a person: If a fingerprint has been taken from a person, it can only be verified if the matching key k is present.

In the literature, many different MAC schemes have been proposed. In practice, two paradigms are dominant: the calculation of MACs by iteration of a block cipher (CBC-MAC [1], CMAC [80]) or by applying a hash function to keys and data (HMAC; RFC 2104 [73]).

Authenticated Encryption Encryption and MAC computation can be combined in different ways [14]:

- **MAC-then-Encrypt** Here the MAC is computed over the plaintext, and then the combination of plaintext, (padding,) and MAC is encrypted. This combination is used, for example, in TLS up to version 1.2. If padding is applied after MAC computation, this still leaves some room for padding oracle attacks.

- **MAC-and-Encrypt** A MAC over the plaintext is computed, but only the plaintext is encrypted. The MAC is then appended to the ciphertext. This construction was used in the Secure SHell protocol (SSH).

- **Encrypt-then-MAC** The plaintext is first encrypted, and then the MAC tag is computed over the resulting ciphertext. This is the strongest construction, and generally recommended today. In specialized modes of operation like Galois/Counter Mode [54], the MAC is computed alongside the ciphertext, so a single pass on the message is sufficient.

Today, the usage of Encrypt-then-MAC is recommended for encryption.

Authenticated Encryption with Associated Data A MAC can be computed over any data sequence. So the input to the MAC computation is not limited to the plaintext or the ciphertext, but may include *associated data* (aka. *additional data*). Encryption schemes that protect the authenticity and confidentiality of ciphertexts as well as the authenticity of such associated data are called AEAD ciphers.

2.1.2 Public-Key Encryption

The terms *public key algorithms* and *asymmetric algorithms* describe the same concept: A *public key* is publicly accessible to all parties in a communication system. In contrast, the secret or *private key* is only known to the (few) receiver(s) which is typically a single entity. In an attempt to refute the speculations of Diffie and Hellman, the probably most famous public key algorithm was born in 1978: initially named *MIT algorithm* by Ron Rivest, Adi Shamir, and Leonard Adleman [105], it is known today as the *RSA algorithm*. Due to its inefficiency and motivated by attacks against several implementation vulnerabilities, a more important public-key encryption scheme today is the ElGamal algorithm [33]. Since the RSA algorithm is not used in modern IM apps, we refrain from presenting it in detail.

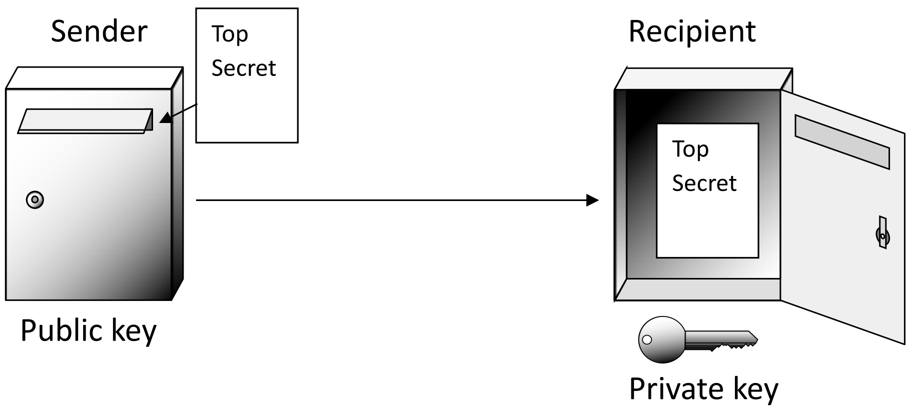

Figure 4: Mental model of public-key encryption.

Mental Model A mental model for *asymmetric encryption* is depicted in Figure 4: Everyone may encrypt a message by dropping it into a publicly accessible mailbox, i.e., by using the *public key*. Only the owner of the mailbox, who has the matching *private key*, can open it and read the message.

Diffie-Hellman Key Exchange (DHKE)

Before giving an example for a public-key encryption scheme, we introduce the Diffie-Hellman Key Exchange (DHKE). Based on the latter, we can then describe and explain the aforementioned ElGamal algorithm.

The primary goal of a key exchange protocol is to compute a shared secret between two (or more) users via an insecure communication channel. Using such a shared secret with a symmetric encryption scheme, one can then protect the confidentiality and integrity of subsequent messages. In 1976, Whitfield Diffie and Martin Hellman describe a procedure [28] that achieves the goal that two parties can agree on a secret key over an insecure communication channel.

In Figure 5, Alice and Bob want to agree on a common secret key k, but only a public, insecure channel (such as a wireless connection or the Internet) is available for communication. Both use a common prime number p (e.g., a standardized one), which defines a mathematical group $G = (\mathbb{Z}_p^*, \cdot)$, and an element $g \in G$. This group has $p - 1$ elements, and the element g generates a subgroup of order $q|(p-1)$. In the original DHKE, Alice randomly selects an integer a from

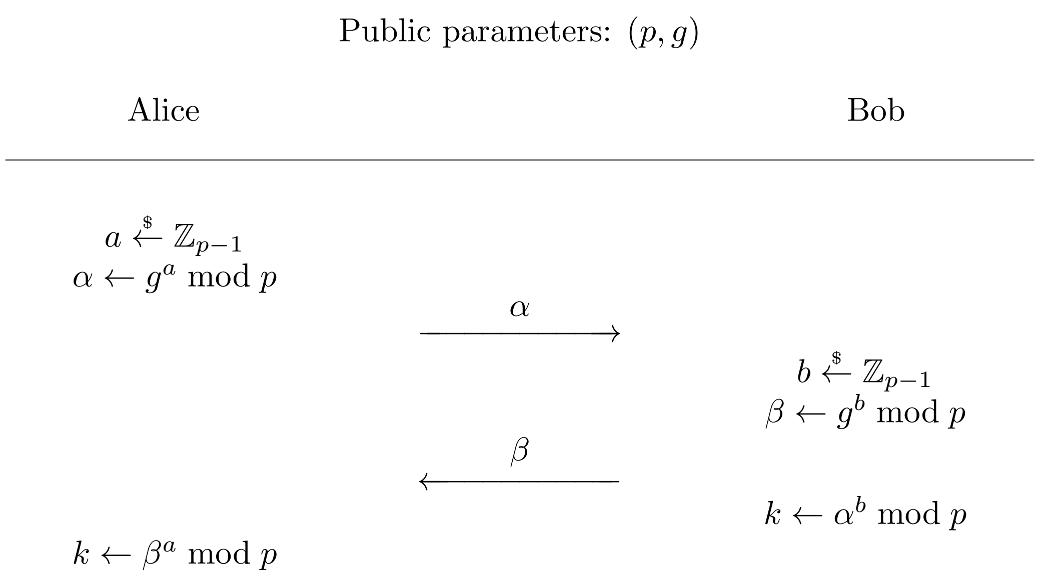
Public parameters: (p, g)
Alice
Bob
$a \xleftarrow{\$} \mathbb{Z}_{p-1}$
$\alpha \leftarrow g^a \bmod p$
α
$b \xleftarrow{\$} \mathbb{Z}_{p-1}$
$\beta \leftarrow g^b \bmod p$
β
$k \leftarrow \alpha^b \bmod p$
$k \leftarrow \beta^a \bmod p$

Figure 5: Diffie-Hellman key exchange (DHKE) in the multiplicative group $\mathbb{Z}_p^*$.

$\mathbb{Z}_{p-1} = \{0, 1, 2, 3, ..., p-2\}$, and computes $\alpha = g^a$ mod p. (It is sufficient to choose a randomly from the smaller set $\mathbb{Z}_q$.) Bob does the same with b and computes $\beta = g^b$ mod p. Then Alice and Bob exchange α and β. From the exchanged values, they can now compute a common shared secret $k = g^{ab}$ mod p since

$$\beta^a = (g^b)^a = (g^a)^b = \alpha^b \pmod{p}.$$

In real-world cryptographic protocols, this value k is used as the basis for the derivation of symmetric keys. DHKE works in any mathematical group. That is, a set equipped with a binary operation (like addition, multiplication, or point addition) which is associative, has a neutral element, and inverse elements. DHKE is only secure, i.e., k is only secret to outsiders, in groups in which the *discrete logarithm assumption* (DL) holds. This assumption states that given only some group element $\alpha = g^a$ mod p, it is impossible to compute a. The name of the assumption stems from the fact that in multiplicative groups, a can be viewed as the (discrete) logarithm of α to the basis g. Not all groups satisfy this assumption: Consider the group $(\mathbb{Z}_q, +)$, where addition is modulo q. This group has q elements. However, for any DH value $\alpha = a \cdot g$ (since the group is additive, we have multiplication instead of exponentiation here), the 'discrete logarithm' can easily be computed as $a = \alpha \cdot g^{-1}$ mod q.

Elliptic Curves

Many different secure groups have been proposed for DHKE, most important today are *elliptic curve groups*. Elliptic curves are point sets $\{(x, y)\}$ which satisfy an equation of the form

$$y^2 = x^3 + ax + b.$$

where the coefficients a, b are from some mathematical field, e.g., rational numbers $(\mathbb{Q}, +, \cdot)$, real numbers $(\mathbb{R}, +, \cdot)$, or *finite fields*.

On such sets, *point addition* can be defined, where a third point is derived given any two points on the curve. Figure 6 illustrates point addition on an elliptic curve which is defined over the field

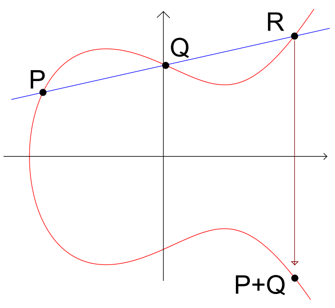

Figure 6: Point addition on an elliptic curve defined over the field of real numbers.

of real numbers $(\mathbb{R}, +, \cdot)$: Two Points P and Q are added by computing the third intersection point R of the line through P and Q with the elliptic curve, and then mirroring this point R on the x-axis to get the sum $P + Q$ of the two points. This point addition is associative, i.e.,

$$(P + Q) + R = P + (Q + R).$$

In cryptography, elliptic curves over finite fields are used. By varying these finite fields $GF(p^n)$, which exist for all numbers of elements p^n where p is a prime number and n a natural number, and by varying the coefficients a, b from these fields, many different elliptic curve groups can be defined. Not all of them are suitable for use in cryptography, so collections of recommended curves have been published. For example, in 1999 NIST recommended 15 elliptic curves for use in cryptography [47]. These recommended curves have q elements, where q is a prime number (typically different from the prime number p used to specify the underlying finite field $GF(p^n)$).

All cryptographic building blocks defined over a multiplicative group $(\mathbb{Z}_p^*, \cdot)$ can also be defined over an elliptic curve. This includes the ElGamal public key algorithms (encryption, key encapsulation mechanism, digital signature), signature standards like the *Digital Signature Algorithm* (DSA), zero-knowledge proofs, and DHKE itself. We will use this latter example to illustrate the differences, which are mainly in notation. Due to their better resistance to known attacks, elliptic curve algorithms play an increasingly important role in modern cryptographic implementations.

Elliptic curve Diffie-Hellman key exchange (ECDH) Since DHKE only requires a mathematical group, it can also be performed over an elliptic curve group. Its structure remains unchanged, however its syntax changes from multiplicative notation to additive notation.

This may be confusing, therefore Figure 7 illustrates ECDH in its entirety. Alice and Bob must agree an a publicly specified elliptic curve $EC(a, b)$ over a finite field $GF(p^n)$, and a base point P on this curve. The specification of the group also contains the group order q, i.e., the number of points on this curve. Alice then chooses an integer a randomly from $\{0, 1, 2, ..., q - 1\}$. The notation aP indicates that the point P is added a times:

$$\underbrace{P + P + ... + P}_{a}.$$

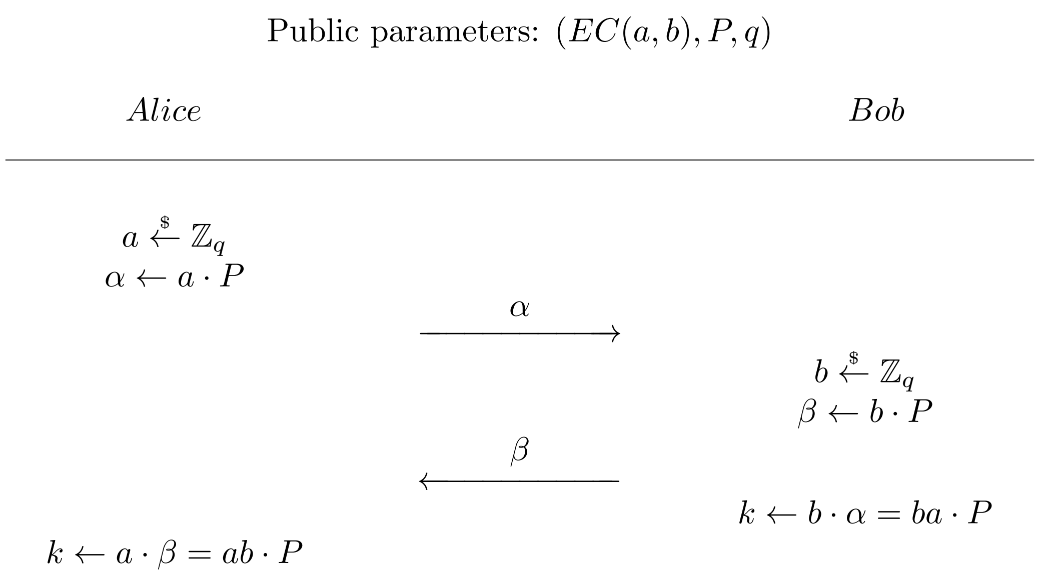
Public parameters: $(EC(a,b), P, q)$
$Alice$
Bob
$a \overset{\$}{\leftarrow} \mathbb{Z}_q$
$\alpha \leftarrow a \cdot P$
α
$b \overset{\$}{\leftarrow} \mathbb{Z}_q$
$\beta \leftarrow b \cdot P$
β
$k \leftarrow b \cdot \alpha = ba \cdot P$
$k \leftarrow a \cdot \beta = ab \cdot P$

Figure 7: Elliptic curve Diffie-Hellman key exchange (ECDH).

There are efficient formulas to compute aP. Bob does the same, and the values α and β are exchanged—in contrast to 'normal' DHKE, these are now points (pairs of numbers) instead of numbers. To compute the shared secret as in DHKE, Alice *adds* the point β a times to obtain $a \cdot \alpha = ab \cdot P = k$.

ElGamal Key Encapsulation Mechanism

If we separate DHKE into two consecutive phases, we get the *ElGamal Key Encapsulation mechanism*. In Figure 8, Bob starts DHKE by randomly selecting b and computing β. However, this time β is not sent directly to Alice, but is published as long-term a public key. Bob keeps the corresponding secret key b and stores it securely for long-term use.

Whenever some other person, in our case Alice, wants to agree on a shared secret k with Bob, she can use this public key. She may retrieve it from the database of the CA, as depicted in Figure 8, or Bob (his mail client, his web server) may send it to her (secure e-mail, TLS). After having received the certificate, Alice can retrieve β from the certificate and in some sense 'complete' the Diffie-Hellman Key Exchange: She selects a random exponent x, and computes a secret k. To share this secret k with Bob, she has to send him the missing DH share $X = g^x$. After receiving X, Bob can also compute k, and both can now use k to secure their communication.

If k is used to encrypt a message m using a symmetric encryption algorithm, the the ElGamal KEM is denoted as the ElGamal public key encryption algorithm. In Figure 8, this is shown by including the computations and messages in square brackets. If Alice already knows the X.509 certificate of Bob, she can thus use his public key β to encrypt any message m to him without prior interaction with Bob.

Digital signatures A digital signature scheme lets the signer create a key pair of secret *signing key* sk and public *verification key* pk. Using the signing key, the signer can create a digital

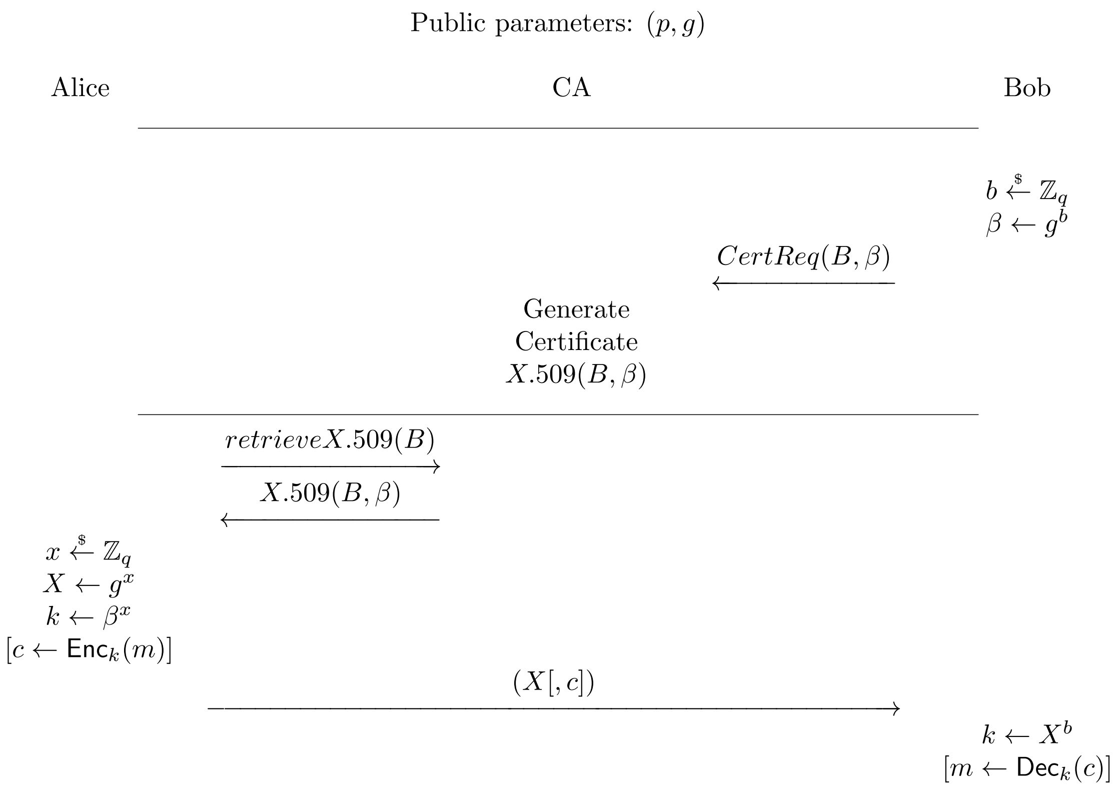
Public parameters: (p, g)
Alice
CA
Bob
$b \xleftarrow{\$} \mathbb{Z}_q$
$\beta \leftarrow g^b$
$CertReq(B, \beta)$
Generate Certificate $X.509(B, \beta)$
$retrieveX.509(B)$
$X.509(B, \beta)$
$x \xleftarrow{\$} \mathbb{Z}_q$
$X \leftarrow g^x$
$k \leftarrow \beta^x$
$[c \leftarrow \mathsf{Enc}_k(m)]$
$(X[, c])$
$k \leftarrow X^b$
$[m \leftarrow \mathsf{Dec}_k(c)]$

Figure 8: ElGamal KEM and PKE. Modular reductions are omitted. Messages in square brackets only belong to ElGamal PKE.

signature σ for some message m such that a verifier can use the verification key to check if signature and message belong together:

$$\sigma \leftarrow \mathsf{SIG.Gen}(\mathsf{sk}; m)$$

$$\mathsf{TRUE/FALSE} \leftarrow \mathsf{SIG.Vfy}(\mathsf{pk}; m, \sigma)$$

An secure signature scheme guarantees that this verification is only successful for signature-message pairs processed by the original signer.

More intuitively, the concept behind digital signatures can best be memorized with the mental model shown in Figure 9, which is known from official announcements of local governments. If, e.g., a road will be renewed, the mayor of a city can announce this by putting the renewal announcement on display on a public pin-board which is protected with a glass window. The glass window can only be opened with a private key owned by the mayor, so all passengers know that the announcement is authentic. The two most important digital signature algorithms are RSA signatures and DSA signatures.

RSA signatures For RSA signatures, key generation is identical to RSA public key encryption (Section 2.1.2). After completing key generation each user has a public key (e, n), where n must be the product of two unique prime numbers nowhere used in any other RSA modulus, and a private key d.

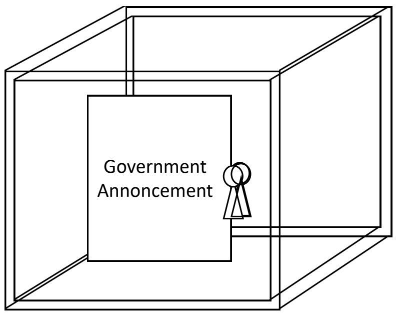

Figure 9: Mental Model for digital signatures.

Textbook RSA Again, we first describe the *textbook* variant of RSA signatures, and then show how to mitigate its weaknesses by properly encoding the message to be signed. To create a digital signature, the private key is needed:

$$\sigma \leftarrow \text{SIG.Gen}((d,n);m) = m^d \text{ mod } n$$

If the message m is longer than the RSA modulus, its hash value $h(m)$ is used instead. Since this description very much resembles the encryption of a message with the public key, in older literature the process of signing a message is sometimes wrongly referred to as 'encrypting the message with the private key'. This notion is misleading, as we will see later.

Verifying a digital signature involves the message itself, the signature, and the public key of the signer.

$$\text{TRUE/FALSE} \leftarrow \text{SIG.Vfy}((e,n);m,\sigma)$$

This verification returns TRUE if and only if

$$\sigma^e \text{ mod } n = m$$

Again, the message m can be replaced by its hash value if it is longer than the modulus n.

Textbook RSA is not EUF-CMA secure A basic requirement for digital signatures is that a valid signature for any message m, i.e. a signature that validates to $TRUE$ when using the public key (e,n), should only be computable by persons who know the corresponding private key. In cryptography, this basic requirement is formalized as *existential unforgeability under chosen-message attacks* (EUF-CMA). To satisfy this requirement, no attacker should be able to generate a valid signature on any message m (EUF), even if he is allowed to get valid digital signatures for *other* messages m^*. This formalized requirement is slightly stronger than what we would expect, namely EUF-KMA. Here, KMA stands for *known message attack*, and this is exactly what happens in practice: An attacker may learn many valid message-signatures pairs for a given public key (e.g. through collecting X.509 certificates from a public CA), but he should nevertheless be unable to produce a valid signature (e.g., a trusted X.509 certificate). This example with X.509 certificates serves us well to motivate the slightly stronger EUF-CMA requirement: For public CAs, an attacker can also request signatures on messages of his own choice, by sending a certification request for his chosen message m to the CA.

It is easy to see that textbook RSA signatures violate this security requirement, although in a rather 'random' way. Consider two valid signatures $\sigma_1 = \text{SIG.Gen}((d,n);m_1)$ and $\sigma_2 =$

SIG.Gen$((d, n); m_2)$. We claim that if we compute their product, the result will be a valid signature on the message $m_1 \cdot m_2$:

$$\sigma_1 \cdot \sigma_2 = (m_1^d \bmod n)(m_2^d \bmod n) = (m_1 m_2)^d \bmod n = \text{SIG.Gen}((d, n); m_1 m_2)$$

Therefore textbook RSA digital signatures are not used in practice. Instead, RSA-PKCS#1 encoding is used, which was shown to be secure in [43].

ElGamal Signatures The second important class of digital signature schemes was derived from the ElGamal signature scheme. In contrast to the deterministic RSA scheme, ElGamal signatures are probabilistic.

The public parameters and the key generation are similar to the ElGamal KEM. All signers may use the same public parameters, which consist of a (very) large prime number p (from about 2048 to 4096 bits in size), and a group element g from $(\mathbb{Z}_p^*, \cdot)$. The only difference is in the optimal choice of the prime number p: While for the use with the ElGamal KEM $p-1$ should have only very few prime factors (ideally only 2 and $\frac{p-1}{2}$), for the ElGamal signature $p-1$ may have many dividors, and we only require that g lies in a 'large' subgroup.

To be able to sign messages, Bob picks a random element b from $\{0, 1, 2, ..., p-2\}$ and publishes $B = g^p \bmod p$ as his public key. This is typically done, as with KEMs, in an X-509 certificate.

Bob now has a single advantage over all other participants: He knows the discrete logarithm b of B. When signing a message m, he can use this advantage as follows:

- He computes the hash value $h(m)$.
- He randomly selects a number r from $\{0, 1, 2, ..., p-2\}$
- He computes a check value $k \leftarrow g^r \bmod p$
- He computes $r^{-1} \pmod{p-1}$ by using the extended Euclidean algorithm.
- Finally he computes the signature value $s \leftarrow r^{-1}(h(m) - bk) \bmod p-1$

The signature value s can only be computed by Bob, since he is the only one who knows the private key b. The ElGamal signature now consists of the check value k and the signature value s:

$$(k, s) \overset{\$}{\leftarrow} \text{SIG.Gen}(b; m)$$

Please note that this signature generation can not be described as 'encryption with the private key'.

The fact that some computations are done modulo p, and some are done modulo $p-1$, can be explained as follows: The group $(\mathbb{Z}_p^*, \cdot)$, where the group operation is done modulo p, only has $p-1$ elements. Thus we know that for any group element h, we have $h^{p-1} = 1$. For any exponent that is larger than $p-1$, we can therefore remove all multiples of $p-1$ to speed up exponentiation, and this is exactly what the reduction modulo $p-1$ does.

Signature verification proceeds as follows:

- Compute the hash value $h(m)$
- Check if $B^k \cdot k^s = g^{h(m)} \bmod p$

If this check succeeds, the signature (k, s) is valid. For this check, only the public key B of Bob, the message m, and the signature (k, s) are needed.

Let us shortly explain why this check always succeeds if the signature is computed as described above. First, we have the following equation relating the values $h(m), s, k, r$ and the private key b:

$$s = r^{-1}(h(m) - bk) \pmod{p-1}$$

If we multiply this equation on both sides with r, and then add bk, we get

$$sr + bk = h(m) \pmod{p-1}$$

Since this equation holds, and all computations in the exponents are done modulo $p - 1$, we have

$$B^k k^s = g^{bk} g^{rs} = g^{bk+rs} = g^{h(m)} \pmod{p}$$

DSS and DSA The *Digital Signature Standard* (DSS, [47]) is an international standard that contains four different signature schemes—two variants of RSA signatures (including RSA-PKCS#1), and two variants of the ElGamal signature scheme. These variants of ElGamal are the *Digital Signature Algorithm* (DSA), and its elliptic-curve counterpart (ECDSA).

The DSA standard uses an observation made by Klaus Schnorr to make ElGamal signatures more compact. If the modulus p is 4096 bits long, an ElGamal signature has twice this length. The observation of Klaus Schnorr now runs as follows: If $p - 1$ has a smaller, but large enough prime factor q (say $|q| = 224$ bits), then there is also a subgroup G of $\mathbb{Z}_p^*$ of order q. If the element g is now taken from this subgroup G, then all computations in the exponents can be done modulo q (224 bits) instead of modulo $p - 1$ (4096 bits). By doing this cleverly, both components (k, s) of the signature can be reduced to q bits. Thus in our example, an ElGamal signature of length 8192 bis can be reduced to an DSA signature of length 448 bits.

Today, the ElGamal signature scheme is no longer used, but has been replaced by its DSA variant. Since the security of DSA relies on the security of the Discrete Log assumption (b cannot be computed from B), other groups where similar assumptions hold can also be used. In the case of elliptic curve groups, this results in the ECDSA algorithm.

2.2 Real-World Applications

2.2.1 Authenticated Key Exchange (AKE)

The need for authenticated key exchange The Diffie-Hellman Key Exchange is secure against passive adversaries who only observe the communication between protocol participants, but not tamper with it.

The situation is very different for active adversaries. Such an adversary may not only record exchanged messages, but may also modify or delete these messages, or insert new messages of his own choice. If an active adversary is able to control the communication between Alice and Bob, then he is denoted as a *Man-in-the-Middle* attacker. Such an attacker may break the security of DHKE, as depicted in Figure 10: The DHKE share α is intercepted by the adversary, who instead forwards his own share X to Bob, and also replies with X to Alice. The answer β

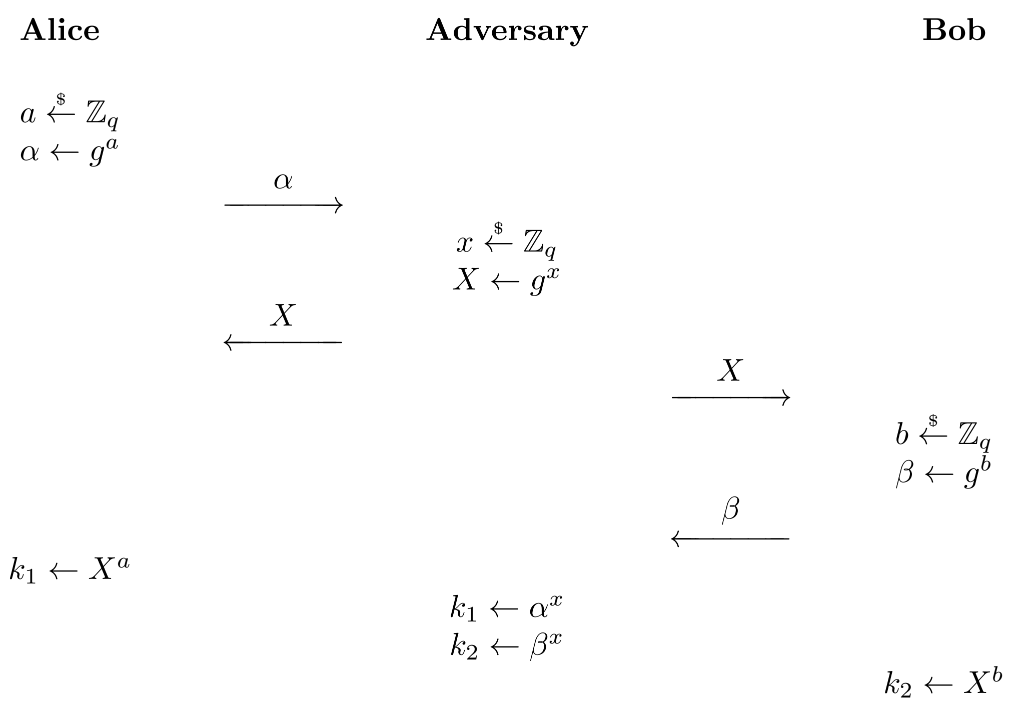

Figure 10: Man-in-the-Middle attack on DHKE.

from Bob is also blocked. Since the adversary knows x, he can compute both DHKE values k_1, k_2 and thus shares a secret value with both Alice and Bob. Any message that Alice encrypts with k_1 can be decrypted by the adversary, who then re-encrypts the message with k_2 and forwards it to Bob. The same can be done for the other direction. The adversary is thus able to read all encrypted traffic between Alice and Bob.

The reason for this 'failure' of DHKE in the presence of an active adversary is that neither of the values α, β is *authenticated*—they may originate from any user or adversary on the Internet.

A first proposal to solve this issue was to digitally sign both α and β—this solution is known as *signed DHKE*. Although it prevents the attack described above, it was never widely adopted. The reason for this is that if each party only signs its own messages, the other party can never be convinced of the *freshness* of the message—a signed value α may be recorded, together with its signature, and replayed days, months or years later in another attempt to break the signed DHKE protocol. In practice, key exchange protocols were instead combined with authentication protocols to form *authenticated key exchange* (AKE) protocols.

Challenge-and-response protocols The most important class of authentication protocols is the class of challenge-response protocols.

In Figure 11, a protocol is depicted where Alice authenticates to Bob by providing a MAC for a challenge randomly chosen by Bob. In this symmetric setup, both Alice and Bob need to share a secret k_{AB}. Bob then randomly selects a challenge $chall$ and sends it to Alice. Alice uses the shared secret k_{AB} to compute a MAC on $chall$ which serves as the response. Bob can verify this response by recomputing the MAC, and comparing it with the received challenge. After this comparison, Bob is convinced that the person 'at the other end of the communication

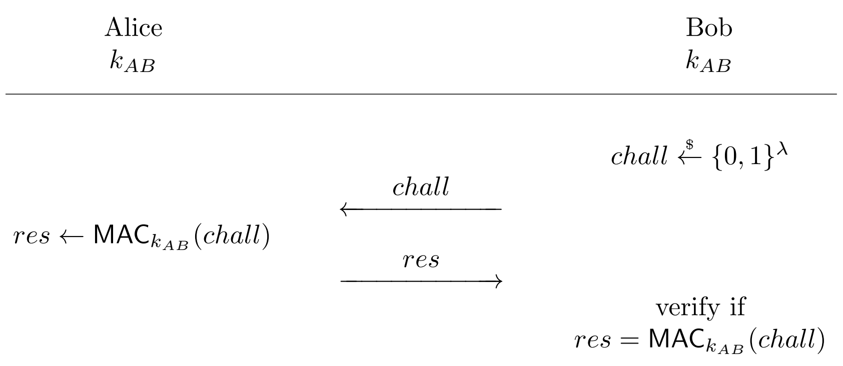
Alice
k_{AB}
Bob
k_{AB}
$chall \xleftarrow{\$} \{0,1\}^\lambda$
$chall$
$res \leftarrow \mathsf{MAC}_{k_{AB}}(chall)$
res
verify if
$res = \mathsf{MAC}_{k_{AB}}(chall)$

Figure 11: Challenge-and-response protocol with symmetric key setup.

line' knows k_{AB}, and since only Alice and Bob know this secret value, that this person must be Alice.

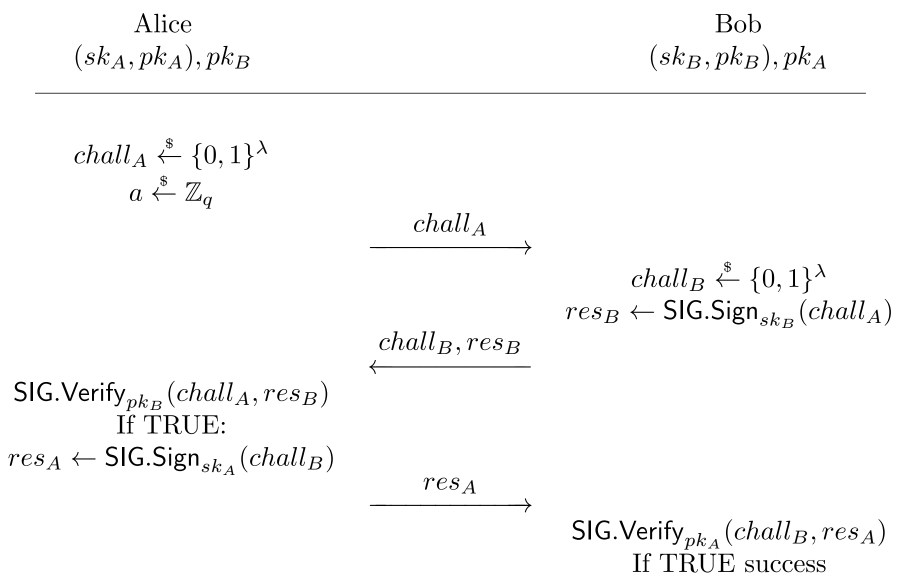
Alice
$(sk_A, pk_A), pk_B$
Bob
$(sk_B, pk_B), pk_A$
$chall_A \xleftarrow{\$} \{0,1\}^\lambda$
$a \xleftarrow{\$} \mathbb{Z}_q$
$chall_A$
$chall_B \xleftarrow{\$} \{0,1\}^\lambda$
$res_B \leftarrow \mathsf{SIG.Sign}_{sk_B}(chall_A)$
$chall_B, res_B$
$\mathsf{SIG.Verify}_{pk_B}(chall_A, res_B)$
If TRUE:
$res_A \leftarrow \mathsf{SIG.Sign}_{sk_A}(chall_B)$
res_A
$\mathsf{SIG.Verify}_{pk_A}(chall_B, res_A)$
If TRUE success

Figure 12: Mutual challenge-and-response protocol with public-key setup.

By interleaving two challenge-response protocols, mutual authentication can be achieved. Symmetric MACs may be replaced by asymmetric digital signatures. Figure 12 shows an example of a mutual authentication in a public-key setup. Alice has a signature key pair (sk_A, pk_A), and the public key is already known by Bob. Equally, Bob's public key of his key pair (sk_B, pk_B) is known to Alice. Again, Alice chooses a random challenge $chall_A$ and sends it to Bob. Bob computes a signature res_B on Alice's challenge, and returns this signature and his own challenge $chall_B$ to Alice. Alice first verifies this signature. If this verification returns *TRUE*, she answers with her own signature res_A on Bob's challenge $chall_B$. If this signature is successfully verified by Bob, the mutual authentication is completed.

Authentication alone is insufficient in open networks An authentication protocol alone does not protect against active adversaries on the Internet. The attack is even simpler than the one on DHKE: A Man-in-the-Middle attacker simply forwards all messages of the authentication protocol unchanged. Yet, it intercepts the communication by acting as an honest Alice towards Bob and as an honest Bob towards Alice. After successful completion of this protocol, the attacker is free to read and manipulate all subsequent messages exchanged.

Combining key exchange and authentication: Authenticated key exchange (AKE) There are many different ways how key exchange and authentication protocols can be combined in practice. Here, we will sketch one solution based on the building blocks described above, which incorporates some ideas from real-world protocols like TLS.

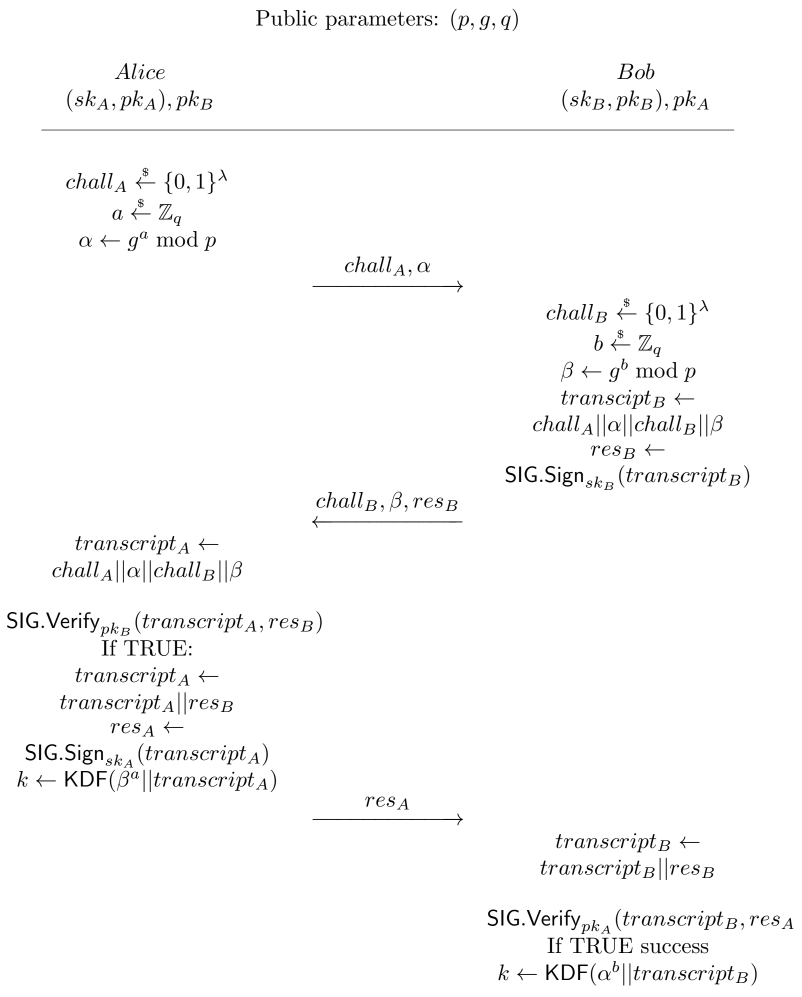
Public parameters: (p, g, q)
Alice $(sk_A, pk_A), pk_B$
Bob $(sk_B, pk_B), pk_A$
$chall_A \xleftarrow{\$} \{0,1\}^\lambda$
$a \xleftarrow{\$} \mathbb{Z}_q$
$\alpha \leftarrow g^a \bmod p$
$\xrightarrow{chall_A, \alpha}$
$chall_B \xleftarrow{\$} \{0,1\}^\lambda$
$b \xleftarrow{\$} \mathbb{Z}_q$
$\beta \leftarrow g^b \bmod p$
$transcipt_B \leftarrow chall_A||\alpha||chall_B||\beta$
$res_B \leftarrow \mathsf{SIG.Sign}_{sk_B}(transcript_B)$
$\xleftarrow{chall_B, \beta, res_B}$
$transcript_A \leftarrow chall_A||\alpha||chall_B||\beta$
$\mathsf{SIG.Verify}_{pk_B}(transcript_A, res_B)$
If TRUE:
$transcript_A \leftarrow transcript_A||res_B$
$res_A \leftarrow \mathsf{SIG.Sign}_{sk_A}(transcript_A)$
$k \leftarrow \mathsf{KDF}(\beta^a||transcript_A)$
$\xrightarrow{res_A}$
$transcript_B \leftarrow transcript_B||res_B$
$\mathsf{SIG.Verify}_{pk_A}(transcript_B, res_A$
If TRUE success
$k \leftarrow \mathsf{KDF}(\alpha^b||transcript_B)$

Figure 13: Authenticated key agreement (AKE) protocol which combines DHKE with signature-based challenge-and-response.

Figure 13 shows a simple academic AKE protocol. It combines DHKE with the challenge-and-response protocol from Figure 12 to achieve mutually authenticated key exchange. Some details have been adapted to reflect recent trends and insights into AKE protocols.

We assume that Alice and Bob have agreed on public parameters defining the DHKE group, that both possess a signing key pair, and that they each know the public key of the other party. In practice, negotiating public parameters and exchanging public keys (via X.509 certificates) will be part of the protocol.

Alice starts the protocol by randomly selecting a challenge $chall_A$ and a DH share α. Both are sent to Bob.

Bob also selects a challenge $chall_B$ and a DH share β. However, he does not only use the challenge of Alice to compute the response, but the complete transcript of their conversation so far. This transcript consists of the messages Alice already has sent, and of the messages Bob will send in response. The transcript must exclude the response res_B itself, otherwise an infinite loop in computing the digital signature would be created.

After the response of Bob arrives at Alice, she re-computes the transcript. This is necessary since we assume an active Man-in-the-Middle attacker, who may modify all messages exchanged in both directions. If the signature verification returns *TRUE*, this convinces Alice that no attack has taken place, and that β indeed originates from Bob. This prevents attacks like the one described in Figure 10. To systematically sign all of the transcript, Alice then appends the signature res_B to the transcript. This is an approach chosen by protocols like TLS, especially TLS 1.3. Other protocols like SSH only sign 'important parts of the transcript'—this would include both challenges and both DH shares. After singing this extended transcript and sending it to Bob, Alice can now derive a secret key. Again, this key derivation is slightly extended to reflect novel trends in AKE protocol design, especially TLS 1.3: The secret key k, which can be used to derive further encryption and MAC keys, is derived from the DH value β^a concatenated with the transcript.

Bob, after receiving the signature res_A, will also extend his transcript $trans_B$, and check the signature. If this check returns *TRUE*, Bob is also convince that no single bit of the exchanges messages was tampered with, and that α indeed originates from Alice. He now applies the same key derivation function to get k.

Security Goals Modern authenticated key exchange protocols are designed to meet the following security goals, which are also important for IM applications:

- **Indistinguishability of the Exchanged Key** Ideally, not a single bit of the exchanged, shared key should be computable by an adversary. This is modeled by requiring that the resulting key should be *indistinguishable* from a random value of the same length.
- **Authentication of Entities and Keys** Both protocol participants should know with whom they are exchanging confidential data. In certain application scenarios, this goal can be modified—e.g., in a client-to-server communication, it usually suffices that the server is authenticated. To keep data confidential for the authenticated entities, the exchanged symmetric key must also be authenticated. This requires that key agreement and authentication are closely interwoven in the AKE protocol.
- **Authenticated Channel Establishment** If the exchanged key is used for establishing a secure communication channel, this channel should (at least) provide confidentiality and authenticity for the communicated payload. This means that outsiders cannot read the transmitted payload, nor tamper with the transcript to manipulate the received payload.
- **Forward Secrecy** This requirement stems from the fact that long-term keys of parties may at some time in the future be compromised. Real-world examples for key compromises are accidental backup of secrets on cloud storage, theft of devices, installed viruses, etc. Another possibility is that a government requests the private key through legal actions, to perform digital surveillance. An AKE protocol having *forward secrecy* (FS) guarantees

that such surveillance is only possible for future message exchanges, and not for past message exchanges.

- **Post-Compromise Security** This property can be considered as the counterpart of FS: It requires that protocols can recover from compromises such that surveillance remains possible only for a short time. Hence, future communication becomes secure again after a compromise.
- **Metadata Hiding** In many scenarios not only the actual payload data needs to be communicated confidentially but also metadata such as sender identity, receiver identity (or even receiver identities in group communication), session identifier, membership operations in group conversations, etc. Several messaging services implement mechanisms to derive, transmit, and store as little metadata as possible. Looking ahead, we consider metadata hiding an important property for interoperable messaging.

2.2.2 Real-World AKE Protocols

Real-world AKE protocols like TLS typically consist of two components: An *handshake*, where some protocol like the one described in Figure 13 is executed, and an *encryption layer*, where application layer data is encrypted and authenticated with the key k negotiated in the handshake.

Without going too much into detail, in this section we will describe the most important real-world AKE protocols. In particular, we will describe how their handshake deviates from the basic model given in Figure 13, and how application data is encrypted and authenticated.

Transport Layer Security (TLS) TLS was invented, under the name *Secure Sockets Layer* (SSL), in the 1990s by Netscape Inc. SSL version 1 was never published, and SSL version 2 was flawed and soon replaced by SSLv3 [87]. This version was completely broken at the encryption layer/TLS record layer by the Poodle attack [55].

When IETF took over standardization, the protocol was renamed to *Transport Security Layer* (TLS). Currently, the two main versions of the protocol are TLS 1.2 [84] and TLS 1.3 [96]. They differ significantly, both at the encryption/record layer and the handshake. The motivation behind the development of Version 1.3 was to reduce latency, and to improve the security of the protocol with respect to published attacks.

Figure 14 depicts the layout of the TLS handshakes in use today. Both versions use the *ServerHello* and *ClientHello* messages to negotiate cryptographic algorithms and parameters. The basic unit for negotiation are the TLS ciphersuites. They bundle key exchange algorithm, server authentication, symmetric encryption algorithm, and hash algorithm (to be used in key derivation and MAC computation) within a single two-byte value. For each version of TLS, new ciphersuites were defined, so there are hundreds of different ciphersuites. A complete list of ciphersuites[2] is maintained by the *Internet Assigned Numbers Authority* (IANA). Additional parameters (e.g. the mathematical group used for DHKE) can be negotiated using different extensions to these messages. For both ciphersuites and extensions, the TLS client sends a list of possible values in the *ClientHello* message, and the TLS server selects one item from each

[2]https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml

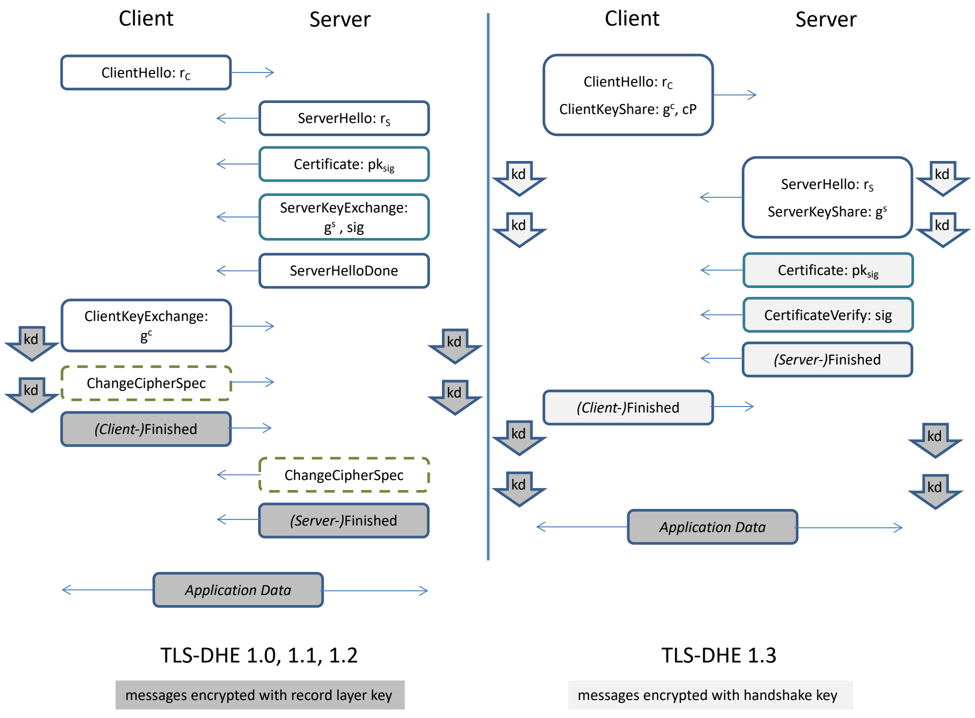

Figure 14: TLS-DHE Handshake, for versions 1.0 to 1.3.

list in the *ServerHello* message. After the exchange of these messages, both client and server know which cryptographic algorithms and parameters to use.

Key agreement in TLS 1.2 results in a shared secret called *PremasterSecret*. Using a key derivation function specified by the chosen ciphersuite, first a *MasterSecret* is derived, and from this value four different encryption layer keys are derived—one encryption and one MAC key for each direction. In TLS 1.2 and all its predecessors, there are three different families of key agreement protocols: TLS-RSA, TLS-DH, and TLS-DHE.

- In TLS-RSA, RSA-PKCS#1 public key encryption is used. The client chooses a 48 byte *PremasterSecret*, and encrypts it with the public RSA key of the server taken from its X.509 certificate. This ciphertext is sent in the *ClientKeyExchange* message to the server.

- In TLS-DH, an ElGamal KEM is used by the client, with the static DH share of the server taken from the server's X.509 certificate. The client randomly chooses an ephemeral DH share g^c and sends this share in the *ClientKeyExchange* message to the server. The *PremasterSecret* is the DH value.

- In TLS-DHE, DHKE is used. The X.509 certificate of the server only contains a signature verification key, and an additional *ServerKeyExchange* message is necessary to transport the server's ephemeral DH share. Again, the *PremasterSecret* is the DH value.

For TLS-DH and TLS-DHE, elliptic key variants TLS-ECDH and TLS-ECDHE have been defined. In all DH and DHE variants, the server sends the first DH share.

In TLS 1.3, only TLS-(EC)DHE has been retained for key agreement. The order of DHKE shares has been changed—now the client sends the first DH share. In order for the *ClientHello* and *ServerHello* messages to be backwards compatible to TLS 1.2, the DH shares are sent in extensions—the *ClientKeyShare* and *ServerKeyShare* extensions.

To protect against active Man-in-the-Middle adversaries, a MAC over the transcript of all handshake messages is computed, both by the client and the server. The MasterSecret serves as the MAC key, and the MACs are exchanged in the *ClientFinished* and *ServerFinished* messages.

The TLS 1.3 handshake is faster—it only needs 1.5 Round Trip Times (RTT), compared to 2 RTT for TLS 1.2. In TLS 1.3, keys can be derived earlier in the handshake, so more handshake messages are encrypted. There are many additional extensions and options for all TLS handshakes, which are summarized in [15].

In TLS 1.2 and earlier versions, the encryption layer/TLS record layer uses the MAC-then-PAD-then-Encrypt paradigm. The record layer accepts a continuous stream of bytes, which is first separated into different parts which are called *records*. For each plaintext record, a MAC is computed. Then, Padding is applied to extend the combination of plaintext and MAC to a multiple of the block cipher block length. Finally, this MAC-then-PAD plaintext is encrypted. This is a weak form of authenticated encryption—although the plaintext is integrity protected by a MAC, the padding may still be modified if the encryption mode is malleable. This allows for padding oracle attacks on the record layer.

In TLS 1.3, the PAD-then-Encrypt-then-MAC paradigm is used. This is a strong form of authenticate encryption, since the integrity of the ciphertext is preserved. Currently, no attacks are known against this type of encryption.

X3DH The *X3DH Key Agreement Protocol* [50] performs up to four Diffie-Hellman key exchanges to derive a secure shared key between two parties. The primary goal of this protocol is to let the two users execute the protocol asynchronously such that Alice can derive a shared key whenever she wants to, and Bob can compute the same key whenever he receives information from Alice, but without depending on whether Alice is online at the moment or not. For this, Bob prepares a pre-key pair (g^b, b) as well as a couple of one-time keys (g^β, β). Bob shares the public components (g^b, g^β) via a server (i.e., the server distributes them to other users on demand). In addition to that, Alice and Bob share the public parts g^A and g^B of their long-term key pairs (g^A, A) and (g^B, B) via that server, respectively. Whenever Alice wants to exchange a symmetric key with Bob, she uses her long-term key A and a freshly generated ephemeral key pair (g^a, a) as well as the mentioned public values of Bob (g^B, g^b, g^β) and mixes them via Diffie-Hellman key exchanges and a *key derivation function* KDF to obtain the shared key:

$$k = \mathsf{KDF}(g^{Ab}, g^{aB}, g^{ab}, g^{a\beta}) = \mathsf{KDF}((g^b)^A, (g^B)^a, (g^b)^a, (g^\beta)^a).$$

After sending her public ephemeral key g^a to Bob, Bob can compute the same shared key. Due to mixing these different keys via a key derivation function, the protocol achieves strong forward-secrecy guarantees: If for any of the pairs $((A, b), (a, B), (a, b), (a, \beta))$ both secrets are kept secure, then the shared key is secure, too. Since both users Alice and Bob regularly renew some of their contributed secrets, the probability that all pairs have at least one of the secrets revealed is reduced. In addition to that, the protocol is highly efficient and, as mentioned above, works in fully asynchronous settings. That means, Alice can start the protocol whenever she wants to and compute the shared key without interacting with Bob. After receiving a corresponding ciphertext from Alice, also Bob can compute the shared key without any further (live) interaction with Bob. (In contrast to this, several other key exchange protocols like TLS and Quic rely on *synchronous* interaction between both participating users who are required to be online simultaneously.)

X3DH is relevant in the context of (interoperable) messaging because many apps implement it to initiate secure messaging sessions. For example, Signal uses this protocol to compute the initial session secrets between users when they start their conversation.

Noise Framework The *Noise Protocol Framework* [62] can be seen as a modular extension of the simple X3DH protocol: Instead of consisting of a single (fixed) computation routine, it describes multiple ways how Diffie-Hellman key exchanges can be performed and combined. This makes the Noise protocol framework versatile and applicable to various use cases.

The primary application of Noise protocols is the establishment of a secure channel between users who share a joint configuration. (E.g., in most messaging apps, the providers specify a configuration that all users of the same app share; in contrast, when users browse the Internet for arbitrary websites, the configuration of the website's server is usually independent of the one of the browsing users; as a result, channel establishment protocols on the *open* Internet have to include some form of interactive configuration negotiation that closed settings like messaging environments can dispense with.)

In contrast to the X3DH protocol, all protocols in the Noise framework use the exchanged keys directly to establish a secure channel via which payload can be transmitted. Depending on the exact mix of Diffie-Hellman key exchanges performed in a particular Noise protocol, the resulting secure channel offers different security guarantees: e.g., confidentiality, authenticity, forward secrecy, metadata hiding, etc.

As in the case of X3DH, a major advantage of Noise protocols is their practical efficiency and simplicity. Therefore, IM applications such as WhatsApp use a Noise protocol to protect the client-to-server communication—replacing heavier alternatives like TLS.

2.2.3 Push Protocols

AKE protocols were designed for synchronous data exchange: In TLS, a client connects to a server and continuously exchanges HTTP requests and HTTP responses until the TLS session is terminated. In SSH, configuration commands from the client to the server, and the server's responses, are exchanged; when done, the SSH connection will be terminated. An IPsec connection is usually only active as long as IP packets need to be exchanged between two hosts. In all three cases, data is *immediately* delivered to the other party, and both parties are always online as long as a connection is running—in other words, communication between two endpoints is *synchronous*.

There is another large class of protocols where communication is *asynchronous*: One endpoint may send a message at some point in time, but this message is only delivered later to the other party, without any immediate response from the receiver. This class conssists of the following *push protocols*:

- **Historical:** Postal letters and (Morse code) telegrams are push services.

- **Short Message Service (SMS):** The ability to send short text messages to receivers identified by their mobile phone number was one of the major inventions in cellular telephony, and an unexpected success story. An SMS text message is sent in the clear, and may only be protected on the radio interface using GSM, UMTS, LTE or 5G encryption.
- **Secure E-Mail:** E-Mail is a push protocol where individual messages are pushed along a sequence of SMTP server until they reach their destination mailbox, identified through an e-mail address. Most e-mails are transmitted as plaintext; similar to SMS, such plaintext transmission may be protected only on some communication links, e.g., using SMTP-over-TLS. In contrast to SMS, secure E2E encryption standards exist: OpenPGP and S/MIME. The principles behind these standards will be described in this section.
- **Instant Messaging (IM) protocols:** These protocols will be described in Section 3.

Secure E-Mail E-mails can be secured by applying encryption and digital signatures to their body. E-mail headers are typically unprotected. Two standards for protecting e-mails exist: OpenPGP [83], and S/MIME [97]. Both standards use the same cryptographic construction—*hybrid encryption*—to provide confidentiality.

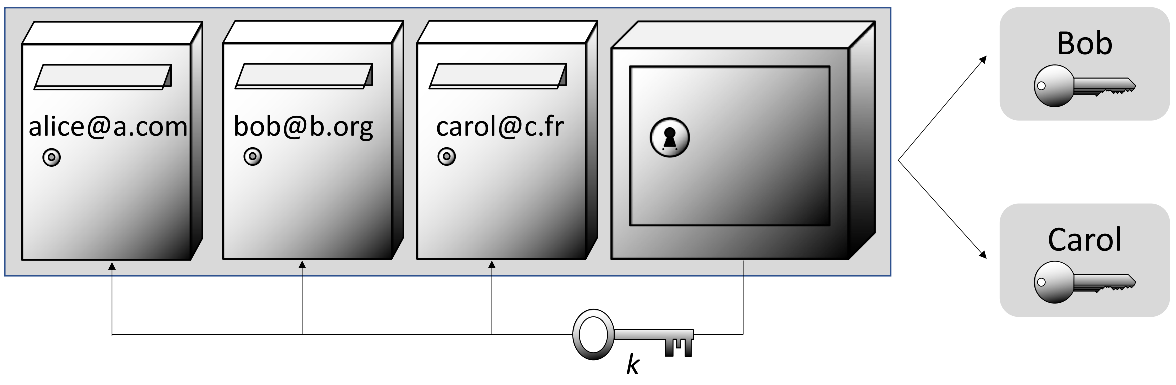

Figure 15: Hybrid encryption is used to encrypt an e-mail.

Figure 15 illustrates the basic principle behind hybrid encryption. When Alice wants to send an e-mail to Bob (mailbox `bob@b.org`) and Carol (mailbox `carol@c.fr`), she selects a random symmetric key k to encrypt the body of the message. This ciphertext is visualized as a safe in Figure 15—the safe is opened and closed with the same key k.

She then uses the public keys of Bob and Carol to encrypt the key k in such a way that only these two recipients can decrypt k, and thus also those two recipients can read the e-mail body. Additionally, she encrypts k for herself—in her 'Sent' mailbox, the e-mail will be stored in encrypted form, and without encrypting k with her own public key, she will no longer be able to read messages she once has sent. Public key encryption is visualized as a mailbox—anyone can insert the key k into the mailbox, but only the owner of the private mailbox key can retrieve k from the mailbox.

The three mailboxes and the safe are 'glued together' using one of the two standard data formats—either OpenPGP or S/MIME—and are sent to both Bob and Carol. If Bob receives this e-mail, he checks the mailboxes attached for the one mailbox he is able to open. To facilitate this, the mailboxes are labeled with the e-mail address of the sender and the two receivers. Once Bob has found the mailbox labeled with his e-mail address, he can use his private key to open it, extract the symmetric key k, and decrypt the body of the e-mail.

OpenPGP vs. S/MIME There is no real cryptographic difference between the two standard data formats—both use hybrid encryption, and both use digital signatures. They only differ in data format details, but still these differences prevent them from being compatible. They offer roughly the same degree of (in-)security, as the EFAIL attacks [64, 56] have shown.

S/MIME data formats are based on PKCS#7 data formats [74]. Updates of these data formats were managed by IETF under the name *Cryptographic Message Syntax* (CMS, [85]). PKCS#7 itself is based on the ASN.1 meta-specification language. CMS is also used, e.g., in PDF signatures.

OpenPGP defines its own data format. Besides for e-mail, OpenPGP is also used for signing binary software bundles.

Secure E-Mail uses stateless encryption The hybrid encryption scheme depicted in Figure 15 is *stateless* in the following sense: To decrypt any encrypted e-mail, only the private key of one of the recipients or the sender is necessary. No additional state must be kept and updated to do so. This has advantages and disadvantages:

- **Advantages:** Encrypted e-mails can be stored as ciphertext throughout their lifetime, and on any storage system (SMTP servers, IMAP servers, local e-mail storage).
- **Disadvantages:** During transport, security guarantees are lower than with current AKE protocols or with IM ratcheting. E.g., there is no forward secrecy: A Man-in-the-Middle attacker can record all e-mails addressed to Carol, and if he later retrieves the private key of Alice, he can decrypt all of them.

2.2.4 Trust Establishment

To authenticate entities and keys, *trust anchors* are necessary. In everyday life, we trust people because we know them personally, because they were recommended to us by a trustworthy person, or because they have some official role that involves trust (e.g., a teacher or a government official). Trust can also evolve over time. For example, if we buy vegetables from a new shop, and if these vegetables are always fresh, in our opinion the reputation of this shop will rise and we will trust it to offer fresh products all the time. These forms of trust anchors are mirrored in the digital world by the concepts listed here.

Manual configuration of secrets A typical deployment scenario of SSH is the remote administration of a small group of servers by a small group of administrators. These administrators also have direct access to the servers, so they can manually install and configure their client's public signature verification keys and mark them as trustworthy. Each server will then have a small list of public keys, and will trust all digital signatures that can successfully be verified with these keys. A similar configuration is done on the client side: Each administrator installs and configures as trustworthy all public keys of the servers to be managed. By verifying the digital signature sent during the SSH handshake the administrator can be sure that he is connected with one of the servers, and not with a Man-in-the-Middle attacker. Manual configuration of secrets may also occur in other scenarios, e.g., in IPsec IKE or in a web application during registration, where the password is set up manually. A crucial limitation of this method can be its lack of scalability due to the required amount of (manual) work.

Trust on first use (TOFU) If the manual configuration of the secret cannot be done directly with physical access to the device (as in the web application example above), there may be a small time frame where attacks cannot be excluded technically. For example, when a new user performs registration at a web application, the application does not know if the user is a real human person, or if it is a bot. Another example is the pairing of Bluetooth devices: It is well-known that an attacker can learn the shared secret if he acts as a Man-in-the-Middle exactly during this pairing protocol, but not later. The security of technical systems, where attacks are only possible during initial setup, but not later, is often described as *trust on first use* (TOFU). This is the best option when no common trust anchor is available.

However, there are many systems which are not even TOFU secure—this security is lost when too much automatism is involved in the setup of secrets. To give just one example: The OpenPGP community invented a mechanism called *Autocrypt* to improve the usability of e-mail encryption. Instead of manually configuring the public keys of users, each user may include his own public key in the Autocrypt e-mail header. An Autocrypt-enabled e-mail-client will then automatically install this public key, to encrypt all messages for this recipient in the future. The problem with this mechanism is that an attacker can easily overwrite this public key with his own key—he just needs to send another e-mail, with a fake FROM address, and a new Autocrypt header to the victim. From this point on, every message sent by the victim to the original recipient will be encrypted with the attacker's key.

Trust anchors and Public Key Infrastructures By using digital signatures, trust can be extended from a *trust anchor* to other objects. This is the idea behind *public key infrastructures* (PKI).

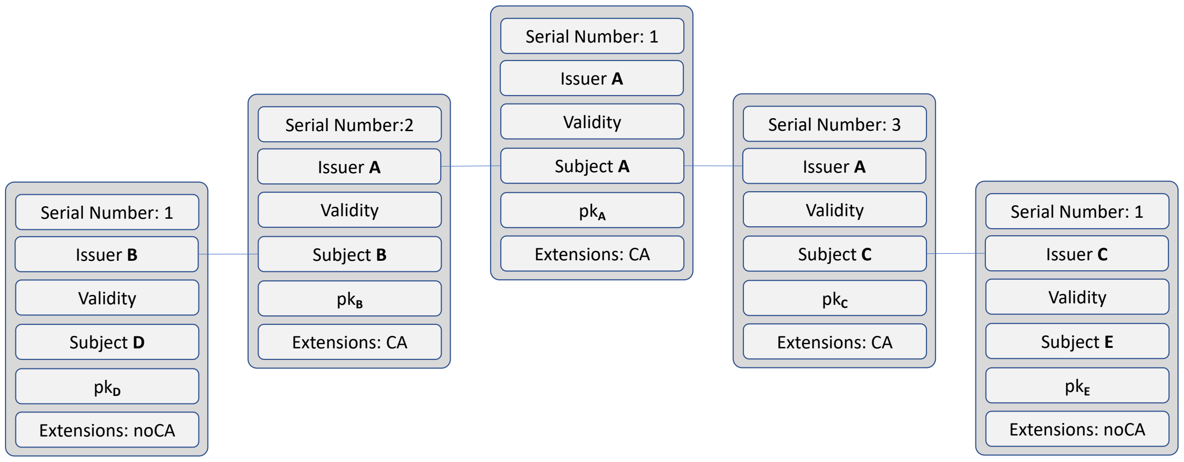

Figure 16: Public key infrastructure based on digital signatures.

In Figure 16, a simplified PKI is depicted. Each data set, which is a simplified version of an X.509 certificate, has the same sequence of contents. The *serial number* allows to distinguish different certificates issued by the same certification authority (CA). The name of the CA is given in the *issuer* field. The trust statement is made for the entity listed in the *subject* field, and is valid for the time period given in the *validity* field. The entity can use the trust received to issue trustworthy signatures, which can be verified with the public key pk_X. The transferred trust may be limited and described by *extensions*.

The trust anchor is the uppermost data set/certificate, also called *root element*. Here CA A asserts trust to itself and its own public key pk_A—this is clearly an invalid circle argument. Therefore trust in the root element must always be established by out-of-band means. Let's give some examples:

- To implement the German Signaturgesetz [109], the Bundesnetzagentur (BNetzA) was responsible for issuing a trust anchor for digital signatures in the form of a root certificate. This root certificate had to be renewed regularly. Trust in each new root certificate was established by publishing the public key (pk_A in Figure 16) in a trusted print medium, the German *Bundesanzeiger*.
- Mozilla includes a database of trusted root certificates in the source code of its web browser Firefox. Any certificate that can be linked by a signature chain to one of these trusted certificates is considered trustworthy. Criteria for inclusion or exclusion of certificates in this trust store are discussed in the CA/Browser forum.
- In order to use the worldwide university WLAN system eduroam[3], students at the Ruhr University Bochum must configure their WIFI network setting with a certificate issued by Ruhr University Bochum. This manual configuration establishes trust in the certificate.

Once a trust anchor is established, trust from the anchor can be transferred to other certificates by signing them with the private key of the trust anchor (not contained in the certificate). E.g., in Figure 16, trust is transferred to entity B by signing the certificate issued by A with serial number 2. This trust may be extended further, e.g., to the leftmost certificate, by signing this certificate with the private key of B. This is possible because the use of the public key in certificate no. 2 issued by A to B to sign additional certificates is explicitly allowed by the extension CA. The leftmost certificate may not be used to issue additional certificates, because the $noCA$ extension is contained there.

X.509 Certificates Most certificates in use today follow the X.509 standard [40], with some exceptions in the context of smartcard based applications.

The structure of an X.509 certificate is shown in Figure 17. Each certificate starts with the (static) version number 3. This is followed by the *serial number*—together with the data from the *issuer* field this number uniquely identifies each certificate. The signature algorithms (including the hash algorithm) is specified next. This allows to compute the hash value needed for signature verification while parsing the certificate. The digital identity of *issuer* and *subject* are specified using fields from another standard, *distinguished name* fields from X.500 [42]. Originally, distinguished names were intended to structure a digital identity according to geographical locations: country name (CN), organization or company (O), subunit of the organization (OU), location of this subunit (L), etc. Today, these fields are used freely, or in compliance with certificate interoperability profiles.

For being usable in different applications, the *subject* field must contain digital identifiers used within these applications. E.g., for TLS server certificates, the subject field will contain the

[3]https://eduroam.org/

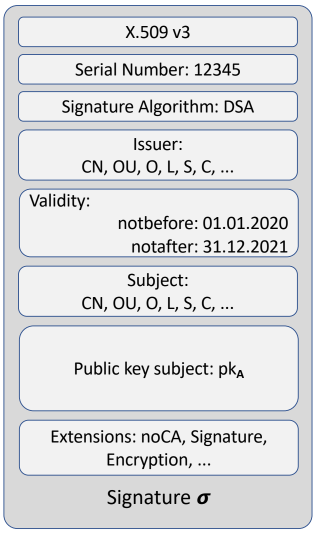
X.509 v3
Serial Number: 12345
Signature Algorithm: DSA
Issuer: CN, OU, O, L, S, C, ...
Validity: notbefore: 01.01.2020 notafter: 31.12.2021
Subject: CN, OU, O, L, S, C, ...
Public key subject: pk_A
Extensions: noCA, Signature, Encryption, ...
Signature σ

Figure 17: Structure of an X.509 certificate.

DNS domain name of the server, and for e-mail certificates, an e-mail address will be included here. *Validity* is stated as an interval. The *public key of subject* field contains a public key for a specified signature algorithm (RSA, DSA, ECDSA), and this key can be used to transfer trust from the certificate to applications: To server authentication in TLS, to additional certificates in a CA, or to authentication of the e-mail contents in e-mail applications. *Extensions* were defined for version 3 of X.509 to allow to differentiate the use of a specific certificate. For example, although an RSA public key can be used to encrypt data or to verify digital signatures, by putting only the *signature* extension in the certificate, the use of the key for encrypting messages is disabled.

Trust is transferred from the issuer to the subject by the issuer signing all data contained in the certificate.

Web of Trust In the OpenPGP community, another trust concept was heralded, the *web-of-trust*. In theory, this was intended to allow transitive propagation of trust along any signature chain, not only from a trust anchor down to the leaves of a PKI. Suppose user Alice trusts user Bob and has co-signed the public key of Bob and configured it as trustworthy. Suppose user Bob has done the same for user Carol and has co-signed her public key. Then the web of trust would imply that Alice may also trust Carol, since the trust relationship should be transitive.

In practice, however, the web-of-trust concept was only applied in small, technically versed sub-communities. OpenPGP users tend(ed) to download public keys from insecure key servers, and the Autocrypt mechanism mentioned above places trust in any public key, without any signature web behind it. Nevertheless the web-of-trust is still an interesting idea, but it lacks tools to support it in a usable way.

Trust based on company ethics In the context of IM, another concept of trust must be mentioned—the trust in a company to protect the confidentiality of the messages of its users. End-to-end (E2E) encryption in IM apps like WhatsApp is easily usable, because there is nothing to configure—during installation, WhatsApp will generate signature and DHKE keys, creates signatures and stores signed prekeys on a server accessible to all WhatsApp users. The end user doesn't note anything, because this process is conducted invisibly in the background. On the downside, the user also completely looses control over this process—if the company would expose private keys to other actors, or if the company simply switches off E2E encryption altogether, is outside their control. Users tend to trust E2E encryption in this context simply because they believe that protecting the privacy of the messages is important to the company, without understanding any details.

3 Instant Messaging: Novel Cryptographic Mechanisms

In this section, we describe the basic cryptographic concepts and building blocks implemented in most instant messaging applications. This includes:

- **Cryptography specific to simple text messaging:** Stateless and stateful encryption to build a secure communication channel as well as mechanisms that continuously update key material to protect such secure channels against the temporary corruption of encryption keys.
- **Encrypted file transfer:** Storage of encrypted files (e.g., image, audio, or video files) and the corresponding key management.
- **Group messaging:** Group key exchange, continuous updates of group keys, the upcoming group messaging standard *Messaging Layer Security* (MLS), synchronous vs. asynchronous group management, and, as an alternative, bundling multiple pairwise communication channels to realize group communication.
- **Real-Time IM functionality:** Voice calls, video calls, standardized protocols like SRTP [78], WebRTC [98, 100, 101], and key management for the encryption of the corresponding streamed data.
- **Overview of protocols** implemented in Signal, WhatsApp, Wire, Matrix, Telgram, and iMessage from the scientific literature.

3.1 Identity Discovery, Key Distribution, Trust Establishment

Identity discovery Instant messaging applications generally need identities for their users to efficiently deliver messages sent to them. In IM applications, the primary identities are in most cases mobile phone numbers. Yet, there are alternative identity concepts such as random strings (e.g., implemented in the Threema messenger) or e-mail addresses (e.g., implemented in iMessage), and, to handle multiple devices, messaging services may assign sub identities to each client device of a user.

To contact other users, identities may be entered manually into an IM app, or may be discovered automatically. The adoption of the latter strategy was one of the reasons why WhatsApp was so successful—by scanning the locally stored contact list of each WhatsApp user for the primary identity 'mobile phone number', all potential WhatsApp contacts could be identified.

Trust establishment and initial key distribution To initiate *authenticated* end-to-end encryption, trust in a set of initial keys must be established. Again, this can be done manually or automatically.

The prototypes for manual key establishment are secure e-mail standards like OpenPGP and S/MIME.

- In OpenPGP ecosystems, typically both keys and trust must be established manually. In the past, social gatherings called PGP key parties were held, where users could exchange keys locally, and establish trust by manually checking the digital fingerprints of the exchanged keys. Later, in an effort to enhance usability, key retrieval was automatized using

PGP key servers, but trust was still assigned manually. By attaching digital attestations of (manually established) trust relations between users to keys stored on these key servers, a trust network can be established. However, due to the sparse use of PGP, this trust network is essentially ineffective for most users.

- In S/MIME ecosystems, trust establishment is done via hierarchical public-key infrastructures (PKIs). Keys wrapped in X.509 certificates are typically exchanged via signed e-mails, but can also be pulled from centralized directories (e.g., LDAP [81, 82]) of CAs. The trust for this is established by having a selection of public trust anchors (e.g., a certificate authority) that certify subordinate trust entities that themselves certify further subordinate trust entities or user public-keys. At initial certification of each user public-key, a certifying entity needs to make sure that the certified identity is the one it claims to be and holds the secret key of the certified public key. Depending on the use case, this can be a manual or automated process.

The manual aspect of these trust and key establishment mechanisms have been heavily criticized for their lack of usability (see [121] and the follow-up papers). However, the user retains full control over their trust relations and decisions.

Trust establishment in IM Trust establishment in most IM apps is realized as a hybrid of the two described mechanisms: The IM providers serve as a (trusted) certificate authority that certify and distribute public-key material of their users. Instead of requiring a manual identity verification, the providers typically check the mapping between certified identity and public-key by contacting the user via an external channel that is bound to that identity. Usually, this is done via SMS or phone call to their identity phone number or via e-mail to their identity e-mail address.

In addition to this, most IM apps offer manual comparison of digital fingerprints of the communication partner's public key as in the above described PGP setting. IM apps typically support this process by showing QR codes on one user's device that can be scanned with the other user's device.

Since the automated certification process runs in the background, and the manual process of fingerprint comparison is primarily made available for experienced users, this approach supports usability for end-to-end encryption (E2EE).

Example: WhatsApp To give an example, WhatsApp [120] implements this process as follows:

1. On installation, the WhatsApp client generates three types of public-key pairs:
   - A single long-lived *identity key pair*. This is a Curve25519 key pair used for signing other public-keys as well as for exchanging the first shared secret when initiating a conversation.
   - A single medium-lived *signed prekey*. Together with the identity key, this Curve25519 key pair is used for initializing the key exchange during the communication establishment; its public key is signed with the identity key.
   - A list of *one-time pre keys*. These keys are also used for the the initial key exchange. However, while the former two key types are used for every initialization during their

lifetime, each one-time pre key is only used for a single initialization and removed thereafter.

2. When a user's primary device (usually their mobile phone) is registered at the WhatsApp servers, the public identity key and a list of public pre keys are transmitted to and stored at the server. Trust in these keys is, however, not primarily established through the described self-signatures via the identity key, but through the ability of the WhatsApp client to communicate via the given mobile phone number. For this, the WhatsApp servers send an SMS to the user's phone number that contains a random authentication string. On receipt of this SMS, the user's client sends this string together with the public key material to the WhatsApp servers, which binds this key material to the phone number. However, when users manually compare the digital fingerprints of their conversation, the identity keys become the primary trust anchor for the established session.

3.2 Text Chats and Ratcheting

Text messages in secure IM apps are transmitted via a secure channel between the sender and the recipient. This channel is established when one of the two users—the initiator—sends their first message to the other one—the responder. For this, the initiator identifies the responder as described in the prior Section 3.1 and queries the provider for the responder's public key material. Using this key material, the initiator executes an initial key exchange protocol that does not require the responder to be online. Non-interactive key exchange protocols like the Diffie-Hellman key exchange or one-pass key exchange protocols like the X3DH protocol [50] are suitable for this. Using the resulting established symmetric key, the initiator uses a secure channel protocol to encrypt the first (and all subsequent) message(s) to the responder and the responder uses this channel to respond.

Different Types of Channel Protocols While encrypting all messages with the established symmetric key may suffice to provide simple confidentiality, modern IM apps implement more sophisticated channel protocols. As we will discuss at the end of this part, regularly updating the shared key via a so called *ratchet* is a widely adopted technique to strengthen confidentiality against temporary compromises of the shared key. The intuition of such a ratchet (and the reason for its name) is that the key is continuously refreshed like a chain that is pulled through a ratchet. Old keys that were used already are deleted and fresh key material is added to the ratchet state.

The the most prominent design of this concept is Signal's Double Ratchet Algorithm [63]. Since it is adopted by several widely used IM apps, it is discussed as the de facto standard for secure channel protocols with key updates. It incorporates many novel security features, and its practicality has been demonstrated through its use in messaging services like WhatsApp and Signal. It will be one of the main tools when discussing interoperability of IM applications.

Compromise of Secrets: Forward Secrecy & Post-Compromise Security As sketched above, *ratcheting* is the name for a novel type of *continuous authenticated key agreement*, which works in asynchronous communication scenarios. This means that users can send messages encrypted with the established key and update this key without interaction of their communication partner. The basic idea is to regularly update and renew the secrets used for encrypting payload such that compromising a communication participant remains as harmless as possible. In the cryptographic literature, the corresponding security goals are called *forward secrecy* and *post-compromise security*. Although briefly introduced before, we recall these two core properties of modern messaging (and their ratcheting) protocols:

A messaging protocol provides *forward secrecy* if a compromise of Alice or Bob does not affect the confidentiality of *earlier* exchanged messages. This means that the messaging protocol has to 'forget' the key material used to en- resp. decrypting these old messages shortly *after* sending resp. receiving them.

A messaging protocol provides *post-compromise security* if a compromise of Alice or Bob does not affect the confidentiality of messages that will be exchanged in the *future*. This means that the messaging protocol has to add unpredictable randomness to the key material used to en- resp. decrypting these upcoming messages shortly *before* sending resp. receiving them.

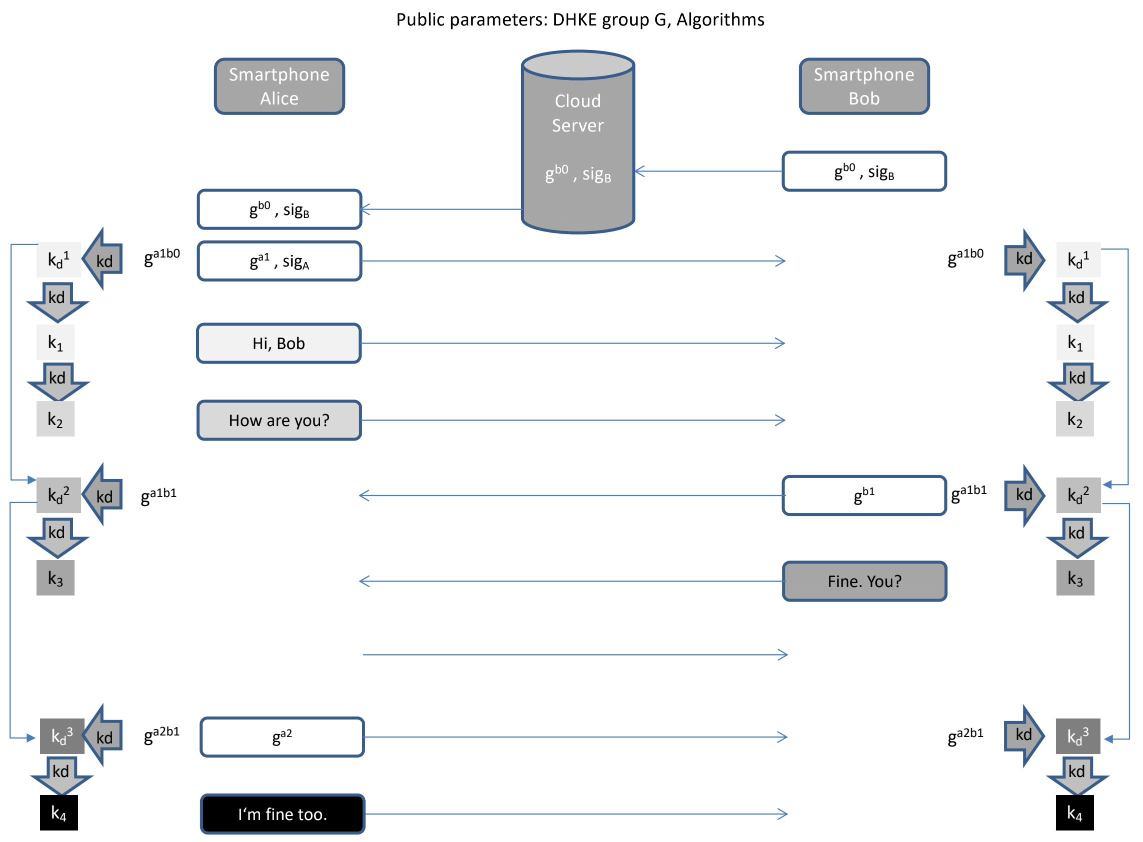

Figure 18: Simplified description of the double ratchet protocol.

Figure 18 depicts a simplified ratcheting protocol. It consists of building blocks described in the previous section, combined in a novel way. The basic ideas are based on Signal's Double Ratchet Algorithm [63], which was inspired by the Off-the-Record protocol[4].

[4]https://otr.cypherpunks.ca/

- Alice opens an IM chat with Bob, which will be E2E encrypted from the beginning. To enable this, both Alice and Bob have stored signed DH shares on a server owned by the IM provider.
- To initiate the chat, Alice retrieves one signed DH share g^{b0}, called *prekey*, of Bob. She also selects a fresh DH share, and derives a first key derivation key k_d^1 from the resulting DHKE value. The fresh DH share is then sent to Bob.
- Up to this point, the process resembles the ElGamal KEM (Figure 2.1.2). But now Alice derives additional keys k_1, k_2, in a deterministic way, for each chat message she sends. The DH share and the subsequent ciphertexts of the chat messages can be cached by the IM server, should Bob not be online (i.e., for asynchronous communication). DH shares are sent as plaintext. For each chat message, a different symmetric key is used for encryption. In Figure 18, this is depicted by using different shades of grey.
- As soon as Bob goes online, he receives the cached DH share g^{a1} from the IM server, together with two ciphertexts. He can then compute the DH value, derive the first key derivation key k_d^1, and the two encryption keys k_1 and k_2 to decrypt the two ciphertext chat messages.
- When he replies, another novel feature of ratcheting is used—the continuous exchange of DH shares. Bob selects and sends his own fresh DH share g^{b1}, and derives a second key derivation key k_d^2. There are a lot of options on how to derive the second key derivation key. Typically, the key k_d^2 is derived from the last DH value g^{a1b1} and from the previous key derivation key k_d^1. Since a value stored in a local variable of the IM app contributes to the derivation of keys, the internal state of the app must be kept synchronized between any two communicating IM instances. Therefore the encryption of IM chat messages is *stateful*[5].
- The first chat message sent by Bob is now encrypted with the new key k_3.
- When Alice receives g^{b1}, she can also statefully derive the second key derivation key k_d^2 and the decryption key k_3. This allows her to decrypt Bob's chat message.
- When Alice answers, i.e., when the communication direction changes again, the continuous DHKE continues. Alice selects another DH share g^{a2}, derives a third key derivation key k_d^3 (typically from g^{b1a2} and k_d^2), and a fourth encryption key k_4. With this key, the fourth chat message from Alice is encrypted.
- When Bob receives g^{a2}, the same key derivation steps are performed, and the fourth chat message can be encrypted.

As long as both IM apps can keep the state, i.e., as long as they know which two DH shares have been exchanged last and which key derivation key k_d^x was derived last, this continuous DHKE can go on forever. When one of the two parties loses the state (e.g., because a new smartphone is used and the IM app is installed freshly), both parties have to restart the session as described in Figure 18.

The novel aspects of ratcheting can best be explained when contrasting it with other AKE protocols and secure messaging standards.

[5]This is in direct contrast with the encryption of e-mails, which is *stateless*.

- **Synchronous vs. asynchronous communication.** Communication over DHKE-based AKE protocols like TLS and SSH is always *synchronous*: It does not make any sense to send an HTTPS request to a web server that is offline, or to try to administer a server via SSH if this server is switched off. In this aspect, IM closely resembles E2E e-mail encryption standards like OpenPGP or S/MIME. In both cases, there are intermediate servers which cache the encrypted messages until the receiver is online: For e-mail these are the intermediate SMTP servers and the IMAP server where the mailbox of the receiver is hosted. For IM, these are servers operated by the IM provider. In both cases, the servers are unable to decrypt the ciphertext.

- **Stateful vs. stateless encryption.** E-mail encryption is, for both standards OpenPGP and S/MIME, *stateless*. This means that for each e-mail to Bob, the same public key is used. Neither sender nor receiver have to store a variable state, they only have to store the static public key of the other party. The statefulness of the IM ratcheting encryption improves security during transport, but restricts the security of chat messages when stored in the IM apps: E-mail messages can be stored in encrypted form, because they can always be decrypted using the static private key of the recipient. IM chat messages, on the other hand, *must* be stored in cleartext, because each message is encrypted with another key derived from many different key derivation keys k_d^x, and as soon as these keys are deleted, the ciphertext can no longer be decrypted.

- **Statefulness in AKE protocols vs. statefulness in IM ratcheting.** An important security goal in AKE protocols is *forward secrecy* (FS). Phrased a little bit differently, this goal states that decryption of a recorded communication between two AKE hosts should only be decryptable when knowing a variable state, which is deleted from time to time—in other words, encryption and decryption should be *stateful*[6]. However, for AKE protocols, this state variable remains static for the duration of a protocol session, whereas in IM, it changes with every sent and received message.

Stateful encryption in ratcheting Modern ratcheting protocols require the session participants to constantly update an internal state. In a minimal configuration, this state consists of the following values: The last DH share sent, the last DH share received, and the last key derivation key. Because of the asynchronicity of the message exchange, it can be necessary to store more than these minimal values; i.e., one symmetric key derivation key for each communication direction. If any of the two communicating parties looses this state (e.g., because the app crashed, or because one party uses a new smartphone with a freshly installed app), both protocols must start from the beginning.

Asynchronous key updates Ratcheting allows to update keys asynchronously. This is necessary since chat messages and the corresponding key updates via DH shares may be delayed because one of the two parties is offline. We explain this using Figure 18.

Suppose Bob turns his smartphone off, or becomes unreachable because of missing mobile network connectivity, after receiving the message ‘Hi, Bob’. Bob may guess that Alice is politely asking him if he is OK, and he may prepare his answer ‘Fine. You?’ before his smartphone reconnects. Bob’s message will then be put in the sending queue of the smartphone, and will

[6]This property does, e.g., *not* hold for TLS-RSA, where a recorded session can be decrypted from a single *static* state, namely the private RSA decryption key of the server.

be delivered to the IM server once he reconnects to the network. In this case, the message 'Fine. You?' is prepared and encrypted before Alice's message 'How are you?' is received.

This change in message ordering is not a problem since by adding some meta information to the ciphertext, e.g., the time when the message was encrypted or the epoch/index of the key derivation key used, the exact same keys can be derived.

3.2.1 Privacy Preserving Wrapper: Sealed Sender

The Signal messenger adds a second encryption layer around the Double Ratchet to hide meta-data of the E2EE ciphertexts and, thereby, reach better privacy properties. This additional encryption layer is called *Sealed Sender* [44]. Intuitively, the sender of a ciphertext c_{DR} encrypts that ciphertext again with this wrapper, using a simple form of public-key encryption to the long-term identity key of the sender ipk_B. The special property of this second encryption wrapper is that it also uses the sender's secret identity key isk_A to protect authenticity of the final packet.

This final Sealed Sender packet consists of the receiver identifier—for efficient delivery via the server—and the (double-encrypted) ciphertext that neither reveals the identity of its sender nor of its receiver. Furthermore, this packet can contain a *proof of sender permission* π with which the server can verify that the sender is actually permitted to send sender-anonymous packets to the receiver:

$$c' \leftarrow \mathsf{ID}_B \| \mathsf{SealSenEnc}(\mathsf{isk}_A, \mathsf{ipk}_B, c_{\mathsf{DR}}) \| \pi$$

That means, the full packet consists of the recipient identifier, the proof, and the encryption of ciphertext c_{DR} that uses the sender's secret key and the recipient's public key.

Spam Filtering and Abuse Prevention Technically, the *proof of sender permission* is a random bit string that the receiver secretly generates and uploads to the server. Furthermore, the receiver shares this bit string with all their communication partners. Whenever one of these partners sends a sender-anonymous Sealed Sender packet to the receiver, they attach the random bit string to prove to the server that they have an ongoing session with that receiver.

If the receiver wants to block the communication with one of their prior communication partners, the receiver freshly generates a new bit string, uploads it to the server (to replace the prior one), and shares it with the remaining communication partners.

This variant of the Sealed Sender mechanism requires at least one non-anonymous ciphertext sent at communication initialization from a new sender to the receiver. The reason for this is that a new sender does not know the receiver's random bit string when initializing a conversation. The receiver will only share that bit string in their subsequent response. However, receivers can also indicate to the server that they accept sender-anonymous packets from anyone. While this option protects sender-anonymity for all ciphertexts, it prevents the server from actively blocking spam.

Privacy Guarantees Sealed Sender iphertexts stored on the (Signal) servers only reveal their receiver identity but not their sender identity. This partially hides the social graph between communication partners. A crucial limitation is that the sender always authenticates to the server when sending a ciphertext. Thus, the server can essentially observe the social network.

Looking ahead, we describe in Section 4.3.5 how interoperable communication can exploit the fact that two competing providers are involved in the message delivery to effectively hide the social graph.

3.3 File Transfer

Chat messages represent very small files, which are *pushed* to the receiving IM app by the IM servers. Pushing larger files using the same mechanisms is impractical: the user may be in an area with low network connectivity, so pushing a large file would block communication until the file is loaded. Instead, the solution shown in Figure 19 can be used.

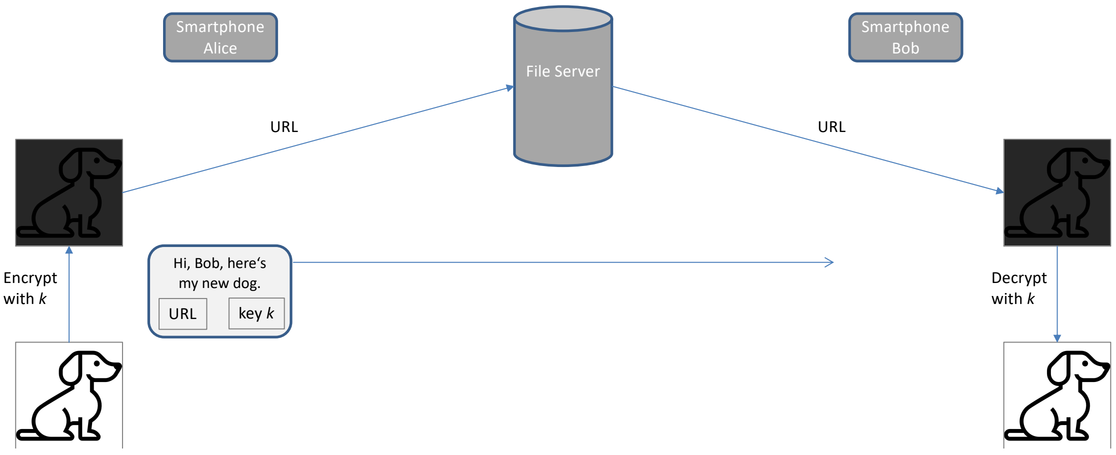

Figure 19: Encrypted file transfer. The symmetric key to decrypt the file and the URL to retrieve it are sent via the chat message channel.

Here, Alice wants to send a photo of her new dog to Bob. She selects the photo with her IM app. The app will then sample a random key k, encrypt the photo with this key, and upload the ciphertext to a file server. The file server returns an URL where the ciphertext can be retrieved later. Alice' IM app then adds this URL and the random key as metadata to her chat message to Bob. Both the text of the chat message and the metadata are encrypted with a new ratcheting key which only Bob can compute.

After receiving Alice' chat message, Bob's IM app may ask Bob if the 'attached' file should be loaded. If Bob has bad network connectivity, he can postpone this until he is at home and has WLAN connectivity. If he has good connectivity, he may instruct the IM app to download the file immediately. The app can the retrieve the ciphertext using the given URL, and decrypt it using the given key k.

The same mechanism can be used for non-interactive audio and video files. Please note that this method offers advantages in low network connectivity areas, compared to pushing the file along with the text message: The receiver may decide to download the file immediately, or to download it later in a network with higher bandwidth.

Only for real-time communication, such as voice calls or video calls, other solutions are needed.

3.4 Group Messaging

Synchronous group key management Classical group key management evolved from generalizations of the Diffie-Hellman key exchange protocol. In [41], DHKE was generalized to an arbitrary number of protocol participants; however, the number of rounds in which the participants need to interact and the number of messages to be exchanged before the group key is computed depends on the number of group members. In [25], the number of interactive rounds was reduced to 2, and $2n$ broadcast messages had to be exchanged to agree on a common key. Various other protocols were proposed afterwards, which are summarized in [65]. All these protocols have in common that key agreement has to be completed with *all* participants interacting before messages could be encrypted. In contrast to this, IM protocols require *asynchronous* group key agreement without online interaction because some group members may never be online at the same time, but encryption of group messages should nevertheless be possible.

Asynchronous group key management Based on the above description of file transfer and the following description of real-time communication in IM apps, it is evident that by having a secure, asynchronous (group) chat message channel, these advanced communication features can be secured, too: Using the secure text channel, symmetric keys can be securely sent to the communication partners; these keys are then used to encrypt attached files or the audio and video streams of (group) calls. In this section, we describe three widely used methods for implementing group communication in IM apps. For a detailed, systematic analysis of widely deployed group messaging protocols in major IM apps, we refer the interested reader to [106]. Although not yet implemented by large IM apps, we also briefly summarize the upcoming MLS standard [13].

The group manager selects the static group key In this solution, a static group key is selected by one of the participants—the group manager. In Figure 20, the group manager Alice

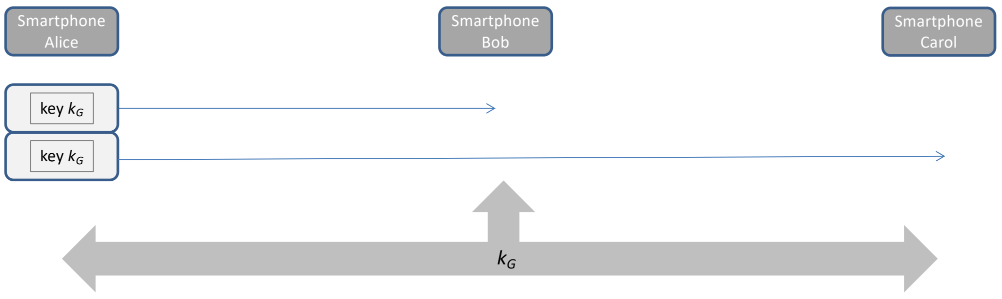

Figure 20: The group manager Alice selects a static group key k_G.

randomly selects a static group key k_G. She then distributes this key to the other group members Bob and Carol, using the pairwise chat messaging channels already established. Once all group members have received the key, they can encrypt and decrypt chat messages sent to the whole group. Group messaging can be asynchronous: If, e.g., Carol is offline, Bob can nevertheless start sending group messages, because the key management message from Alice will be cached together with all future group messages until Carol is again online. Key updates can be made by the group manager by sending around new group keys to all members, through the individual chat channels. New members can be added to the group by Alice sending them the group key k_G. A member X can be excluded from the group by Alice sending an new group key k'_G to all group members except X.

This approach offers basic confidentiality and authenticity with a communication overhead that is independent of the group size. To also achieve forward secrecy (FS) and post-compromise security (PCS), the group manager has to regularly distribute a fresh group key through the pairwise channels—the pairwise channels have to provide FS and PCS for this, too. Such updates, however, induce a communication overhead that is linear in the number of group members. Thus, this approach only scales well for small group sizes.

Each group member uses a different group key (Sender key) Here, each group member performs the same action as the group manager in the first solution. In Figure 21, Alice selects a

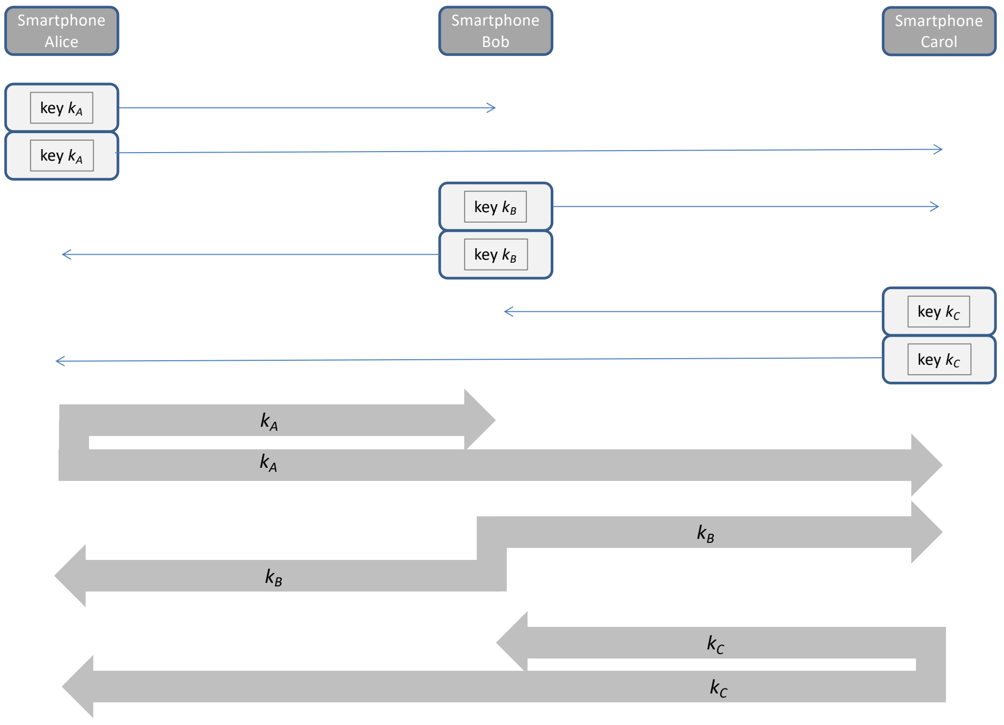

Figure 21: Each member of the group selects a different group key.

group key k_A which she uses to encrypt all her messages to the group. She also receives group keys k_B (from Bob) and k_C (from Carol), which she can use to decrypt incoming group messages. Each member can now update their group key individually, by either sending new keys to all other group members or by using a deterministic update method for the group keys. If the group size is n, each member has to store n actual group keys. To add a new group member Y, each 'old' group member must sent their group key to Y. To exclude a group member X, all remaining group members W must send new group key k'_W to all members except X.

This approach offers basic confidentiality and authenticity, too. Using a deterministic key update mechanism, FS can be achieved without increasing the communication overhead. If senders regularly send fresh sender keys through the pairwise channels, PCS can be achieved. Yet, such explicitly sent updates induce a linear communication overhead.

Individual ratcheting channels are used If an IM app employs ratcheting to continuously update keys in the pairwise channels, the advanced security properties (FS and PCS) of ratcheting are lost when using static group keys without updates as described in the two options above. In the

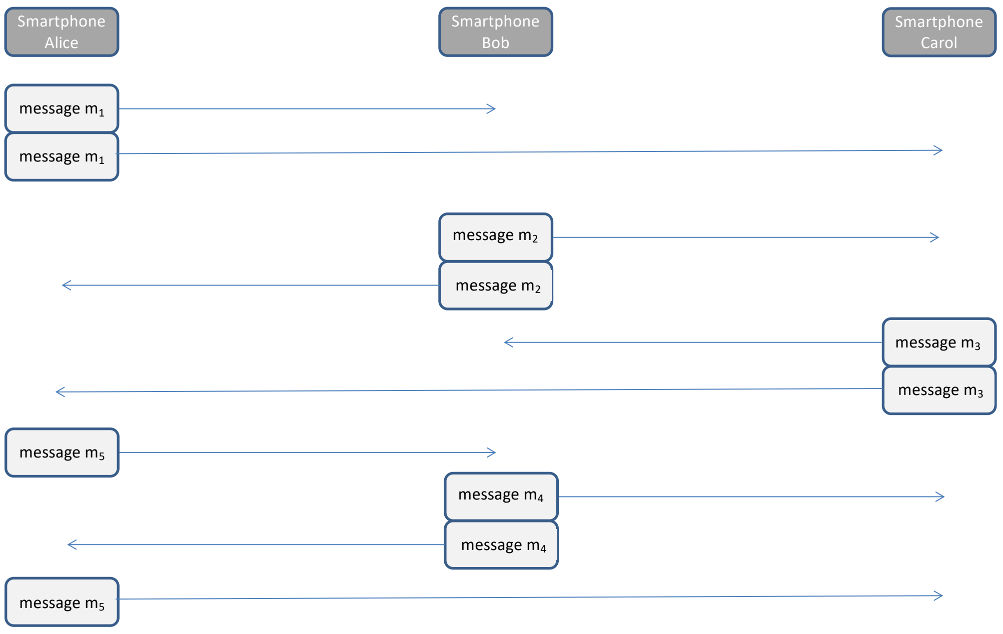

Figure 22: Individual ratcheting channels are used to distribute group messages.

solution from Figure 22, no group keys are used at all. Each message to a group of n members is sent as $n-1$ individual chat messages to the other members of the group. This results in a linear communication overhead. Since chat texts are usually short, this overhead mainly results from the ratcheting key update messages—these updates are however necessary to guarantee the stronger security of ratcheting. Large files will not be pushed along with these messages, but will be stored in encrypted form at a server. As described in Section 3.3, only a URL and the decryption key will be pushed to the other group members.

MLS The IETF Messaging Layer Security (MLS) working group [13] currently finalizes an eponymous standard protocol for secure group messaging. The goals of this standard are: reaching strong security goals—FS and PCS—with practical performance—logarithmic instead of linear communication overhead—and fixing an open standard—although Signal's Double Ratchet is considered a *de facto* standard, it was specified by Signal's cryptographers and its implementations in multiple IM apps differ minimally but crucially.

The MLS protocol realizes ratcheting with all group members based on a simple tree structure. In this tree, each group member is leaf and the shared group key is the root. Each node in that tree contains a pair of public key and secret key. Every user only stores the secret keys on the path from their leaf to the root. When a user executes an operation, they refresh the secret keys along the path(s) of one (or more) leaves to the root, encrypting these secrets to the siblings along the path(s). Thus, in most cases, the number of encrypted ciphertexts is at most log n, where n is the number of group members.

By regularly updating the secrets known by each group member, this protocol achieves similar security guarantees as ratcheting protocols in the two-party case: a compromise of user secrets only affects a short time period of the overall communication. Due to many analyses of the cryptographic building blocks of MLS (e.g., [5, 4, 16, 18, 6, 23, 119]), the research community substantiated the claimed security guarantees.

3.5 Real-Time Communication

The chat message channel of an IM application is not designed to handle real-time communication: It is designed to handle asynchronous message exchange, not synchronous, time critical data exchange. Most likely, the continuous key update of ratcheting mechanisms would introduce audible/visible delays in the audio/video signal.

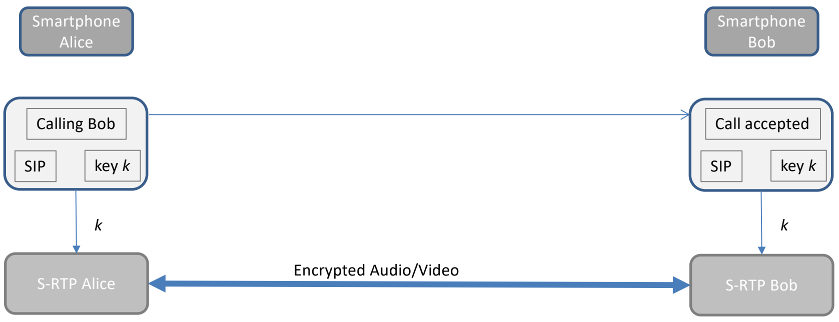

Figure 23: Audio/Video call key management using key exchange in chat metadata.

However, a similar solution as for file transfer can easily be implemented. In Figure 23, a generic solution is depicted. The IM client Alice and Bob are using a software module for encrypted real-time communication. A standard method is to use the *Secure Real-Time Protocol* (SRTP, [78]) for this purpose. When Alice wants to call Bob, a text message via the ratcheting channel containing only metadata is sent to Bob's IM app. This message signals that Alice wants to establish a real-time communication, i.e., a voice or video call. Standard protocols can be embedded here, e.g., the *Session Initialization Protocol* (SIP) [75, 76] for establishing Voice-over-IP (VoIP) calls. Additionally, a randomly chosen key k is included.

When Bob accepts the call, control is handed over to the real-time module, denoted as SRTP. This module now synchronizes real-time data exchange in such a way that audible or visible delays avoided. All SRTP data packets are encrypted using the key k chosen by Alice's IM client, which Bob's IM client did retrieve from the encrypted chat message.

This approach offers basic confidentiality and authenticity of the real-time data stream.

WebRTC To standardize the full stack of protocols that are necessary to establish secure real-time communication and, thereby, simplify the implementation and configuration of such a protocol stack, the *WebRTC* [98, 100, 101] framework was developed. This standard was in particular aimed to support real-time communication via web browsers and, therefore, comes with a high level of interoperability between different implementations. Thus, it is similarly well suited to support interoperability for real-time communication between IM clients of different providers.

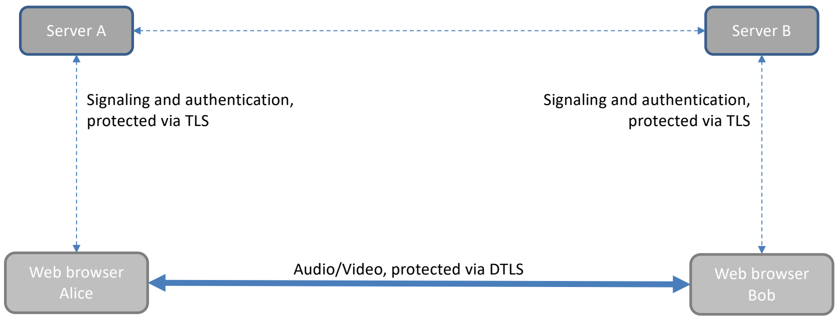
Server A
Server B
Signaling and authentication, protected via TLS
Signaling and authentication, protected via TLS
Web browser Alice
Audio/Video, protected via DTLS
Web browser Bob

Figure 24: Audio/Video call key management using WebRTC.

While WebRTC can also be used in smartphone apps other than a web browser, we explain it in Figure 24 using web browsers as an example. In this example, Alice's web browser may set up a direct data connection to Bob's browser via one or two web application servers. Both browsers must be registered at some web application supporting WebRTC. Then Alice can request addressing information from the web servers hosting this application. She also can transmit identifying information to Bob's web browser, e.g., a symmetric key which is later used in for encryption in DTLS [103]—the latter being a secure channel protocol that, in contrast to TLS, is designed for unreliable network connections via which data streams are sent. Once she knows the actual IP address of Bob's browser, she can start a UDP/DTLS connection, over which audio and video information can be transmitted directly. So far, there has been little research on how exactly the WebRTC framework is used in IM apps [111, 112, 113], and implementations may be subject to change here. Table 2 gives an overview on current approaches.

Generally, WebRTC is considered to offer confidentiality and authenticity of the streamed data just as SRTP.

3.6 Overview of Protocols

For illustration purposes, we provide a brief overview of the messaging protocols of popular IM applications in Table 2. Since the main aim of providing this overview is to emphasize that the implemented techniques are highly heterogeneous, we defer explaining the technical details to the subsequent paragraphs. However, we want to point out that, although Signal, WhatsApp, Facebook Messenger, Wire, and Matrix use the Double Ratchet Algorithm [63] for two-party communication, all five exact implementations are crucially different: Wire uses a different encryption algorithm, Signal uses a different payload encoding scheme, and WhatsApp leaves most meta data unencrypted. Furthermore, messages transmitted via Facebook Messenger are equipped with an additional abuse-tracing mechanism, called Message Franking. Finally, the key derivation implemented in Matrix differs crucially from the one in Signal's original Double Ratchet Algorithm.

IM Protocol	Two-Party	Group	Real Time
Signal [110]	Double Ratchet (DR)	DR	WebRTC
WhatsApp [120]	DR	Sender Key (SK)	SRTP
Facebook Messenger [34]	DR with Message Franking	SK with Message Franking	undocumented
Wire [122]	Proteus ($\approx$ DR)	Proteus ($\approx$ DR)	SRTP
Matrix [53]	Olm ($\approx$ DR)	Megolm ($\approx$ SK)	WebRTC
iMessage [7]	Public-key encryption	Public-key encryption	SRTP
Telegram [116]	MTProto	Unencrypted	MTProto

Table 2: Overview of messaging protocols.

In the following, we elaborate further on the techniques implemented in these IM apps based on publicly available information. This includes the published security whitepapers [110, 120, 34, 122, 53, 7, 116] as well as analyses published in the academic literature that we reference below. All undocumented and irrelevant implementation details are omitted in our description, in which we primarily focus on pointing out the crucial differences between the protocols of these IM apps.

3.6.1 Signal-Protocol related IM apps

In addition to the Signal messenger itself, several other IM apps implement protocols that are based on building blocks that were designed for and by Signal. Most prominently, all of the IM apps discussed in this first part implement Signal's Double Ratchet Algorithm [63]. To give an overview of the similarities and differences between these IM apps, we describe the building blocks implemented by the IM apps from the innermost protocol layer to the outermost (i.e., we begin with the payload message that communication partners exchange, continue with the message encoding and encryption, and end with the outermost client-to-server encryption layer).

Signal All IM apps encode the actual payload messages (including emojis, formatting information, certain metadata etc.) in a predefined format. The Signal messenger uses *Protocol Buffers* [35] for this.

The resulting encoded plaintexts are then encrypted with the Double Ratchet Algorithm in Signal. While the Double Ratchet Algorithm, being a two-party protocol, naturally works for encrypting two-party conversations, Signal also uses this protocol for group chats: Signal encrypts and sends a group message as $n-1$ copies via the $n-1$ pairwise Double Ratchet channels between the sender and the remaining $n-1$ group members individually. As a consequence, the resulting Double Ratchet ciphertexts all look the same and do not reveal whether they are sent in a two-party or group context. On a lower technical layer, Signal's Double Ratchet implementation uses AES-256 in CBC mode with PKCS#7 padding for encryption, and HMAC for integrity protection.

The Double Ratchet Algorithm is a session protocol that relies on an initial key exchange mechanism that establishes a shared secret between the session participants. For this, Signal uses the X3DH [50] protocol that establishes such a shared secret whenever a user starts a new conversation.

While the actual payload in Signal is treated identically for two-party messages and group messages, an additional group management protocol [26] protects the groups' membership structure for group chats. This protocol hides the group structure towards the service provider and cryptographically protects the consistency of group management operations (i.e., membership additions and removals).

To hide metadata, Signal wraps Double Ratchet ciphertexts (i.e., the packets that result from encrypting the encoded payload messages with the Double Ratchet) within so called *Sealed Sender* packets [44]. These packets hide all explicit metadata from the visible traffic, in particular the sender's and receiver's identifiers towards Signal's servers. Yet, for sending a message to a user Bob, Alice tells the Signal servers that she is the actual sender and that Bob is the actual receiver of that message. Thus, wrapping ciphertexts with the Sealed Sender mechanism for hiding metadata has only limited effect in Signal's deployment setting.

Whenever a live audio or video call is started by a Signal user, the WebRTC [98, 100, 101] protocol suite is used. For this, a symmetric secret is sent by the initiator of the live call via the above described Double Ratchet channel to the recipient. After this, a peer-to-peer SRTP channel is established between initiator and recipient.

Finally, all traffic between Signal clients and Signal servers is protected by TLS channels [106].

All in all, Signal's protocol stack offers forward secrecy, post-compromise security for two-party chats, and forward secrecy for group chats. Group chats also offer a slightly weakened form of post-compromise security: While the encryption protocol effectively renews the key material such that the communication recovers from prior key compromises, the group management protocol does not. As a result, a strong attacker could compromise a group member in order to add themselves to the group at a later point in time. Signal's protocol also hides certain metadata: In principle ciphertexts of group messages look the same as two-party messages; yet, since the sender sends all copies of a group message at the same time, the server can identify group messages as such. Furthermore, due to using the Sealed Sender mechanism, after the ciphertexts were uploaded to the server, these ciphertexts do not reveal which user sent them.

WhatsApp The payload encoding in WhatsApp is based on XMPP [88, 89, 90, 106]. From the resulting encoded payload of a *two-party chat*, only the actual message (i.e., not any metadata) is encrypted by WhatsApp via the Double Ratchet Algorithm [63]. The technical implementation of the Double Ratchet equals the one used by Signal [120].

For *group chats*, WhatsApp uses the Sender-Key mechanism [12] to encrypt the messages of the encoded payload. This mechanism avoids the linear communication overhead of Signal's pairwise-channel approach at the cost of post-compromise security for group messages. The membership management in WhatsApp is conducted in plain by WhatsApp's servers [106]. Thus, the WhatsApp server effectively decides who is added to and removed from the set of group members.

While the two-party Double Ratchet channels are initiated using the X3DH [50] protocol (as in Signal), WhatsApp group chats are initiated by sending the so called sender keys encrypted via the established pairwise Double Ratchet channels between all members.

For establishing live audio and video calls, WhatsApp uses SRTP [78] by sending an initial shared secret, that is used for encrypting the payload, via the pairwise Double Ratchet channels.

Finally, instead of using TLS channels for protecting client-to-server communication, WhatsApp uses the Noise [62] framework for this.

Due to using the Double Ratchet for two-party chats, WhatsApp provides forward secrecy and post-compromise security. However, the Sender Key mechanism for group chats only offers forward secrecy, and due the plain, server-based group membership management the overall group protocol is insecure against compromised servers. Furthermore, the payload encryption in WhatsApp only protects the actual messages but no metadata.

Facebook Messenger End-to-end encryption of messages in Facebook's Messenger is an optional feature. Thus, the following description only applies to chats for which users explicitly enable a "secret conversation".

The encryption protocol of encoded messages in Facebook [34] is based on the same implementation of the Double Ratchet Algorithm, Sender Key mechanism, and X3DH protocol as the one in WhatsApp.

On top of this, messages are processed by a so called Message Franking mechanism [30]. This mechanism guarantees that users who receive abusive content can effectively report this. By letting the sender of every message commit to the sent content (without revealing the content to anyone besides the designated recipient), the service provider can retrospectively verify the validity of an abuse report.

The exact payload encoding, the selection of encrypted (meta-)data, the group membership management, the protocol for live conversations, and the client-to-server transport layer protection are not publicly documented. Thus, it remains unclear which overall confidentiality, authenticity, and metadata hiding properties are fulfilled by the protocol stack of Facebook Messenger.

Wire The Wire messenger uses CBOR [102] for payload encoding. For encrypting the encoded payload, Wire uses an adapted version of Signal's Double Ratchet called Proteus. One of the technical changes in Proteus is the use of ChaCha20 for encryption (instead of AES in CBC mode in Signal). For group chats, Wire uses the same approach as Signal by sending a group message as $n-1$ copies via the pairwise channels between the sender and all remaining group members [122]. Yet, updates in the Wire Github repository[7] suggest that their group messaging protocol will be based on MLS [13] in the near future.

For live audio and video calls, Wire uses a composition of DTLS, SRTP, and channel authentication based on the pairwise Proteus communication channels. All client-to-server communication is protected by TLS.

[7] https://github.com/wireapp/core-crypto/blob/develop/docs/ARCHITECTURE.md

Due to the lack of comprehensive security analyses and the distributed nature of the documentation of Wire, we cannot conclusively assess the overall security provided by Wire's protocol stack. Nevertheless, the architecture is very similar to Signal's protocol stack, suggesting that both IM apps offer similar security guarantees.

Matrix While all of the above IM apps are based on a centralized server infrastructure, Matrix is designed to work in a federated infrastructure. This means that users can have different service providers and yet communicate with each other, using the Matrix protocol. (We want to note that, if all messaging platforms would agree on speaking the Matrix protocol, this would be a candidate solution for interoperable messaging; yet, this approach seems currently unlikely; see Chapter 4 for the full discussion.)

The payload message encoding in Matrix is based on the JSON format [95]. For encryption, Matrix uses a similar approach as WhatsApp: two-party conversations are encrypted using a variant of the Double Ratchet Algorithm called Olm [52] and group chats are encrypted using a variant of the Sender Key mechanism called Megolm [51]. The changes implemented in these variants are minimal but crucial: For example, the key derivation in Megolm uses different input strings than the one in the original Sender Key mechanism such that the keys output by these two protocols are effectively independent of and incompatible with each other.

Similar to WhatsApp, the group membership management is partially processed in plain by the Matrix servers.

The initial key for an Olm session is computed with a key exchange protocol that is related to Signal's X3DH protocol [50] and the initial key for Megolm sessions is distributed via the existing pairwise Olm sessions between all group members.

Matrix uses WebRTC to realize live audio calls and all client-to-server communication is protected with TLS—HTTPS more specifically.

A recent security analysis [2] identified several attacks against Matrix's protocols. Although Matrix implemented fixes and the authors of the analysis confirmed the effectiveness of these fixes, this cannot be considered a proof of security. Nevertheless, the overall protocol stack of Matrix is similar to the one of WhatsApp, suggesting that both IM apps offer similar security guarantees. However, since Matrix stores used key material, it does not offer comparable flavors of FS and PCS.

3.6.2 Other IM Protocols

The following widely used IM apps implement protocols that are entirely independent of the protocol components developed for the Signal messenger—in particular, these IM apps do not implement variants of Signal's Double Ratchet Algorithm.

iMessage For end-to-end encryption of messages, iMessage uses simple public-key encryption [7, 9]: The sender encrypts their messages to the receiver using the receiver's static public key. For group messages, the sender simply replicates this encryption step to send every group member an individually encrypted copy of the sent messages. The client-to-server communication in iMessage is protected with TLS. For live audio and video calls, FaceTime uses SRTP [8].

Since the key material used for encryption is static, corrupting an iMessage user's key material will break the confidentiality of this user's entire iMessage communication. Thus, iMessage only provides basic confidentiality but neither forward secrecy nor post-compromise security. It is unclear from the available documentation to which degree iMessage protects the conversations' metadata.

Telegram As for Facebook Messenger, Telegram two-party conversations are only end-to-end encrypted on demand. Beyond this, group chats in Telegram are never end-to-end encrypted at all. Thus, the following description only applies to two-party conversations for which users explicitly choose to have a "secret chat" [116].

For encoding payload messages, Telegram uses a proprietary format that is based on fixed-length and dynamic-length byte arrays. The subsequent encryption of encoded messages is based on a symmetric protocol called MTProto 2.0 [116]. The initial symmetric secret for an MTProto session is established using a Diffie-Hellman key exchange (DHKE). The two participants of an MTProto session regularly derive new "initial" secrets by executing fresh DHKEs, which renews the symmetric session secrets.

For voice and video calls [115] as well as for the protection of client-to-server communication, Telegram also uses proprietary protocols based on their MTProto 2.0 channel protocol [117].

Two recent analyses [3, 10] found attacks against MTProto and its implementations that were fixed subsequently. Based on the public documentation, Telegram provides confidentiality with a coarse notion or forward secrecy due to regularly re-initiating the MTProto sessions. (Although Telegram does not claim to provide post-compromise security, we want to mention that, in principle, such session re-initializations can suffice to reach this goal.)

3.6.3 Observations

The primary goal of the above survey is to emphasize that none of the messaging protocols is currently compatible with another one. Furthermore, the differences between these protocols are so manifold and divers that an attempt to provide interoperable messaging by converging the current protocols is pointless. Developing an entirely new protocol for interoperable messaging (based on existing building blocks) or extending one of the existing protocol stacks for this are seemingly far more promising options.

4 Interoperable Instant Messaging

4.1 Requirements of the DMA

To define the requirements that every proposal for interoperable messaging needs to fulfill, we quote and interpret the most relevant paragraphs from Article 7 of the DMA [69].

Functionality Before we consider the security requirements of the DMA, we begin with the fundamental, functional meaning of *interoperability* that is defined in Article 2:

> "*'interoperability' means the ability to exchange information and mutually use the information which has been exchanged through interfaces or other solutions, so that all elements of hardware or software work with other hardware and software and with users in all the ways in which they are intended to function*", Article 2 (29), DMA [69]

Based on this definition, gatekeepers have to implement measures to enable interoperable communication with other providers upon request:

> "*[The] gatekeeper [...] shall make the basic functionalities of its number-independent interpersonal communications services interoperable [...] by providing the necessary technical interfaces or similar solutions that facilitate interoperability, upon request, and free of charge.*", Article 7 (1), DMA [69]

More concretely, gatekeepers have to document the technical details with which other providers can implement interoperable messaging:

> "*The gatekeeper shall publish a reference offer laying down the technical details and general terms and conditions of interoperability with its number-independent interpersonal communications services, including the necessary details on the level of security and end-to-end encryption. The gatekeeper shall publish that reference offer within the period laid down in Article 3(10) and update it where necessary.*", Article 7 (4), DMA [69]

We want to emphasize that the DMA does not specify any technical approach based on which gatekeepers provide interoperable access to their services. Thus, many different options are possible. To clarify which options are practical and likely, this study considers several alternatives in the upcoming sections.

It is equally important to note that the gatekeepers can freely choose how they implement interoperability. Therefore, when discussing alternatives for possible implementation approaches, we keep in mind that the gatekeepers may follow their own interest.

End-to-End Confidentiality Paragraph 3 of the DMA explicitly requires that interoperable communication protocols offer at least as strong confidentiality properties as their (non-interoperable) intra-provider counterparts:

> "*The level of security, including the end-to-end encryption, where applicable, that the gatekeeper provides to its own end users shall be preserved across the interoperable services.*", Article 7 (3), DMA [69]

Paragraph 3 of the DMA is complemented by Paragraph 9 that permits, after specifying a secure proposal for interoperable communication, that the gatekeeper can add additional measures[8] for *integrity*, *security*, and *privacy*, which are required to be implemented by the competitors, too:

> "*The gatekeeper shall not be prevented from taking measures to ensure that third-party providers of number-independent interpersonal communications services requesting interoperability do not endanger the integrity, security and privacy of its services, provided that such measures are strictly necessary and proportionate and are duly justified by the gatekeeper.*", Article 7 (9), DMA [69]

As discussed in Section 3.6, all widely used IM apps offer some form of end-to-end confidentiality for their internal communication. Nevertheless, end-to-end encryption is not always the default and in some cases even unavailable: two IM apps (Facebook Messenger, Telegram) only enable end-to-end encryption on demand and Telegram provides no end-to-end confidentiality for group chats. However, since WhatsApp, being the most widely used IM app in the EU [24], offers end-to-end encryption for two-party and group messaging by default, we consider end-to-end confidentiality obligatory for protocol proposals.

Flavors of Confidentiality Beyond the mere availability of end-to-end confidentiality, our comparison in Section 3.6 shows that the exact "*level of security*" provided by these IM apps differ significantly: iMessage provides a relatively weak form of confidentiality as it uses static key material (i.e., no forward secrecy or post-compromise security); Telegram updates and replaces key material in a coarse yet regular frequency, which offers some form of resilience against future corruptions of user secrets (i.e., forward secrecy); multiple IM apps—those that are based on Signal's Double Ratchet—implement key update mechanisms that lead to strong confidentiality guarantees even against temporary past and future corruptions of user secrets (i.e., forward secrecy and post-compromise security).

We emphasize that the above quoted paragraph of the DMA explicitly requires that "*the level of security*" is preserved. Thus, also more fine-grained nuances of security properties must be maintained for interoperable communication. Since many of the mentioned widely used IM apps implement measures to uphold confidentiality even against corruptions of user secrets, we consider the corresponding cryptographic properties **Forward Secrecy** and **Post-Compromise Security** essential for protocol proposals.

Beyond these two flavors of confidentiality, one may extend the interpretation of the "level of security" even further.

Authenticity Typically, "end-to-end encryption" is associated with the cryptographic goal of confidentiality. Nevertheless, equally important and clearly also covered by the "level of security" is the protection of end-to-end authenticity.

Since all end-to-end encryption protocols considered in our survey from Section 3.6 implement measures to achieve authenticity, we also consider this an obligatory requirement for protocol

[8]In this study, we focus on secure and private interoperable communication that can be provided using cryptographic protocols. We give a brief overview of other concepts of security and integrity, such as abuse prevention or spam filtering, in Section 4.5.4 that may also be captured by Paragraph 9, Article 7. These types go beyond the focus of this study.

proposals. This includes authenticity protection under corruption of user secrets (i.e., forward security and post-compromise security).

Privacy In Paragraph 8 of Article 7, the DMA explicitly requires that the gatekeepers collect and exchange as little data of the users as technically possible:

> *"The gatekeeper shall collect and exchange with the provider of number-independent interpersonal communications services that makes a request for interoperability only the personal data of end users that is strictly necessary to provide effective interoperability. Any such collection and exchange of the personal data of end users shall fully comply with Regulation (EU) 2016/679 and Directive 2002/58/EC."*, Article 7 (8), DMA [69]

Given that the General Data Protection Regulation (GDPR) [68] generally also applies to non-gatekeepers, equivalent requirements hold for them, too. Thus, we technically interpret Paragraph 8 as follows: All (meta-)data about users and their interoperable communication must be transmitted confidentially—even towards the involved servers—if revealing this data is not technically necessary.

Consequently, practical protocol proposals that demonstrate the functionality and efficiency of interoperability for messaging without revealing certain (meta-)data contribute to the list of (meta-)data that must be treated confidentially. Therefore, we see one contribution of our study and similar works in demonstrating the compatibility of functionality, practicality, and feasibility for privacy preserving solutions, by clarifying which data technically needs to be revealed and which data must be hidden.

Metadata Hiding One particularly relevant pair of metadata consists of the identifiers of senders and receivers of messages. These sender-receiver pairs are crucial because they reveal the social interaction in the (interoperable) messaging network. Whenever such a pair is revealed, an edge is added to the graph of the social relations in that network. We therefore consider it important to hide these pairs whenever possible. To achieve this, it suffices to always only reveal either the sender or the receiver identifier, but never both.

Looking ahead, we will see in our protocol proposals that the nature of interoperable messaging—involving (at least) two competing, possibly non-colluding providers—supports technical solutions that hide the social network.

4.2 Preliminaries

We first give a brief overview on different interoperability approaches for end-to-end encryption, with their advantages and disadvantages. We do not consider metatdata exchange and routing of messages in this section. Please note that in all cases, interoperable IM clients must share *minimal* metadata of the two communication endpoints. E.g., the client ID of the recipient must be known to at least their own provider to efficiently deliver messages to them.

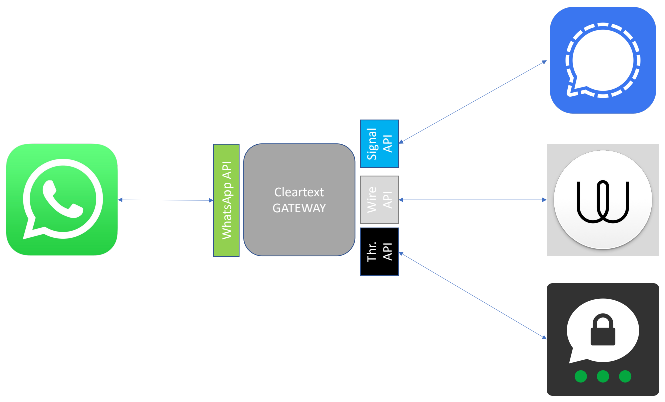

Figure 25: A gateway implements all cryptographic interfaces of gatekeepers and competitors.

4.2.1 Gateway Approach

In a gateway approach (aka. *Sever-Bridge*; see Figure 25), a single gateway server would be used to translate the different cryptographic and metadata message formats of all IM apps in the market. As a necessary consequence, all metadata and all cleartext will be visible to the gateway.

- **Pros:** Requires few changes to IM apps compared with other approaches (e.g., clients must understand the IDs of other IM systems).
- **Cons:** No end-to-end encryption possible—cleartext and metadata visible to gateway; enormous implementation and maintenance effort to implement and update many detailed APIs.

Since a gateway approach weakens the security of all IM systems by disabling E2EE in interoperable communication, we do not consider this approach any further.

4.2.2 Standardization Approach

In a standardization approach (Figure 26), each IM client would implement a standardized API for all basic functionalities.

- **Pros:** Same client implementation effort for all parties; transparent security.
- **Cons:** Standardization takes a lot of time; IM apps with simple, stateless crypto (e.g. iMessage) will be forced to implement new concepts—possibly also on the server side.

Standardization does not violate any of the DMA requirements, but it may not fit into the given time frame—recall that interoperability must be implemented less than two years after the gatekeeper is declared as such. However, since it offers advantages in situations where there is more than one gatekeeper in the market, we consider it further in Section 4.4.

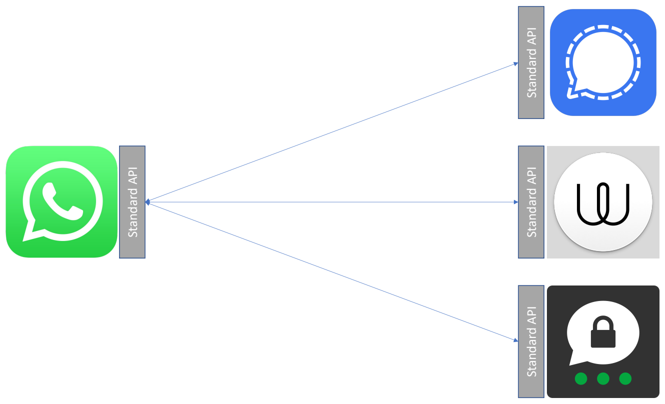

Figure 26: Each IM app implements a single standardized cryptographic API for interoperability, in addition to proprietary cryptographic interfaces.

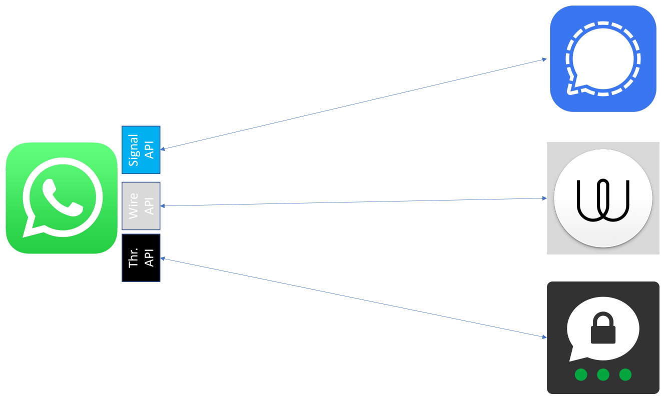

Figure 27: Gatekeepers implement the cryptographic APIs of all competitors.

4.2.3 Gatekeeper-Side API Approach

In a gatekeeper-side API approach (Figure 27), each gatekeeper would implement the APIs of all competitors in its own IM client.

- **Pros:** Security level for communication unchanged; gatekeeper has resources to implement this solution.
- **Cons:** High effort for gatekeeper—competitors can block programming resources with unclear specifications and frequent API updates.

Since the competitors' protocols would be used for the entire interoperable communication, the security level for communication remains the same on the cryptographic layer. Yet, if the gatekeeper has to implement a large number of protocols, this increases complexity for its client application. This may increase the attack surface and lead to more security vulnerabilities.

Since the DMA explicitly allows the gatekeeper to choose solutions for interoperability, it seems rather unlikely that they would choose this option. Yet, our generic language in Section 4.3 also covers this concept.

4.2.4 Competitor-Side, Competitor-Implemented API Approach

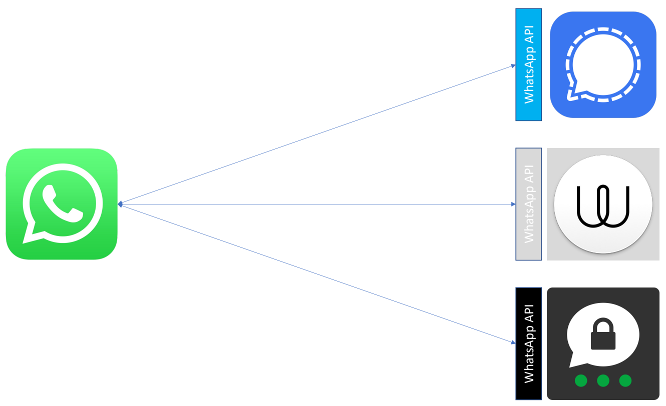

Figure 28: Gatekeepers specify a cryptographic API which the competitors must implement.

In a first competitor-side API approach (Figure 28), each gatekeeper would specify a detailed cryptographic and metadata API which each competitor must implement.

- **Pros:** Transparent security of the gatekeeper through de facto open specifications.
- **Cons:** High effort for competitors—they would have to implement complex, changing specifications; if specification was fixed, this would block innovations.

Similar to the *gatekeeper-side approach*, the level of security is preserved on a cryptographic layer due to using the same protocol for intra-provider and interoperable communication. However, also the attack surface in the competitors' client applications may be increased when they implement multiple protocols for interoperability with multiple different gatekeepers.

Although, this approach either imposes high costs on competitors, or could block innovation at the gatekeeper if changing their protocol is hampered, our generic language in Section 4.3 also covers this idea.

4.2.5 Competitor-Side, Gatekeeper-Implemented API Approach

In a second competitor-side API approach (Figure 29), each gatekeeper would specify only a cleartext and metadata API. Key exchange, data formats and encryption would then be handled by a software library available for inclusion in the competitor's IM clients. The competitors would only have to implement the relatively simple cleartext and metadata API.

- **Pros:** Medium implementation effort on all sides; simple distribution of protocol and implementation updates.
- **Cons:** Intransparent security of the gatekeeper which can easily be changed via software updates; executable code of other providers embedded in the competitors' applications.

This approach seems to be most suitable to satisfy the ambitious timeline of the DMA. We therefore investigate it in Section 4.3.

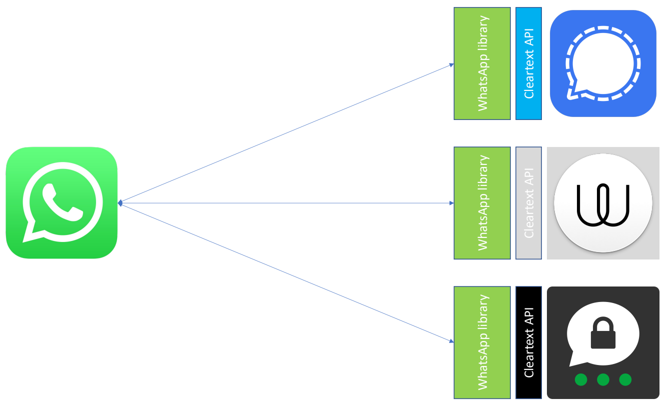

Figure 29: Gatekeepers specify a cleartext API and implement all cryptographic details in a software library available to the competitors.

4.3 API Approach

The first option we consider, called *API Approach*, can dispense with the standardization of a global interoperability protocol. The reasons for avoiding standardization of a protocol can be manifold: organizing a transparent process that brings together all relevant stakeholders is complicated, standardization may take a long time—which can be crucial for the short DMA implementation timeline—, the IM app landscape still quickly evolves, competition between IM providers led to the development of secure and practical protocols, prior standardization initiatives like those for TLS or even MLS have shown that agreeing on a single protocol requires many—sometimes undesirable—compromises, and standardization occasionally leads to legacy decisions that can almost not be reverted in re-standardization attempts of future versions. While a standard interoperable protocol may also have multiple advantages, we want to present this API Approach as an alternative.

Remark: All concepts discussed in this section realize E2EE. In particular, the considered server gateways/interfaces take ciphertexts—not plaintexts—as inputs. To enable correct processing by the servers, these ciphertext inputs are enriched with metadata. Since metadata will not be standardized in the API approach, we present different approaches for metadata processing, transmission, and avoidance.

4.3.1 Basic Idea

For the API Approach, one or more provider(s)—called *specifier(s)*—specify interfaces that define access to their original, intra-provider architecture. The other provider(s)—called *connectors*—can then implement their interaction with these interfaces such that interoperability is enabled via this connection between the providers at the interfaces. As we describe it in Section 4.2 and will elaborate below, the API can be specified at different locations on the delivery route between the users who communicate interoperably. The two main options are:

1.

The API defines access to the specifiers' servers that take ciphertexts from the connectors' users. As a result, the connectors have to implement the protocol between their own users and the specifier's servers. (2) Alternatively, the API is located inside the clients of the connectors' users. Thus, the connector's client app can send and receive plaintexts through this API. Encrypting the plaintexts, decrypting received ciphertexts, and processing the delivery happens behind this API—as a black box from the connector's perspective. One option to realize this is to embed a library behind the API that is developed by the specifier and distributed to the connectors' client apps.

Consequently, instead of designing and fixing a full standard messaging protocol, the API Approach allows participating providers to implement and use large parts of existing protocols, even when users communicate to users of different providers. On top of this, this approach can be enriched by standardizing the involved APIs. For example, when the APIs are located at the connectors' client application, the standardized portion of the APIs could enable fundamental messaging features like sending two-party messages, group messages, establishing a call, etc. Such an API would almost only be functional and aims to avoid commitments to how the functionalities are implemented behind the interfaces.

4.3.2 Location of API

As mentioned above, there are several options how and where to locate and implement the APIs. We present and discuss some of the most important alternatives.

4.3.3 Selection of Specifier

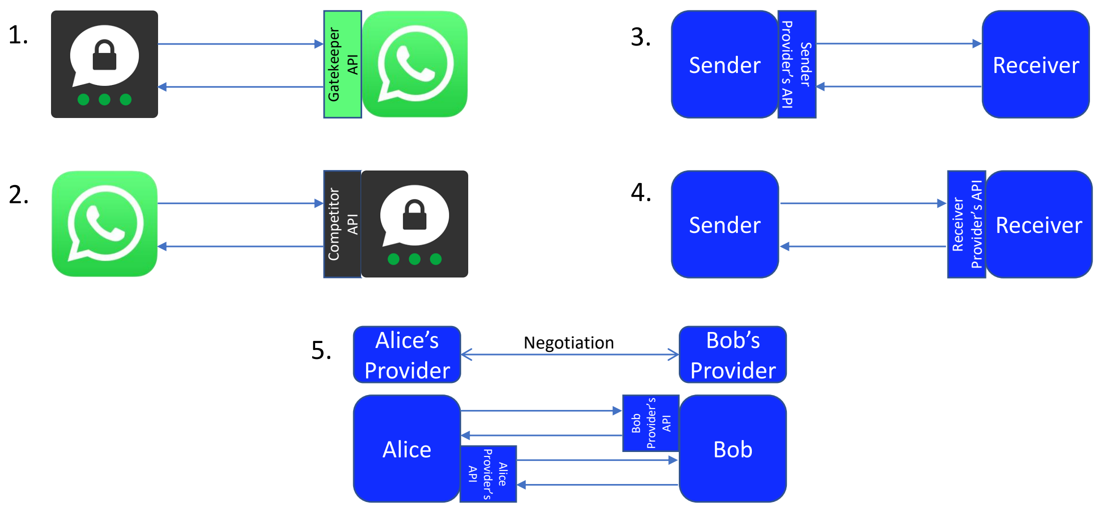

Figure 30: Illustration of options for specifier selection using WhatsApp and Threema as examples for gatekeeper and competitor. Arrows indicate the interaction between the connector and the specifier's API.

An important question for this approach is: who will be the API Specifier and who will be the Connector; more concretely, who will define how to access their own internal architecture and who will implement the protocols to access these interfaces from the outside. This decision does not yet determine whether the API is located at the Specifier's servers or on the Connector's client, which is discussed hereafter.[9] We consider five options that are illustrated in Figure 30:

- Gatekeeper dominance: The gatekeeper is always the specifier and the competitors are always the connectors.
- Competitor dominance: The competitors are always the specifiers and the gatekeepers are always the connectors.
- Sender dominance: The sender always speaks via their own provider's API such that the corresponding receiver's provider is the connector, and, hence, the receiver is required to listen at all connected APIs for receiving new messages.
- Receiver dominance: The sender is always in the connector role, speaking to the API of the receiver.
- Dynamic negotiation: Each pair of providers negotiates which of both will specify the API to which the other one connects. This negotiation can be conducted once and then fixed statically or conducted dynamically for every interoperable session initialization (i.e., every time a user starts a conversation with a user from the other provider). Thus, both providers may adaptively need to switch between the specifier and connector role

While the model of *gatekeeper dominance* is most likely based on the requirements of the DMA, we think that the other options are meaningful (at least in theory) as well.

Only Client API For the first option, only a plaintext API located at the connector's client is specified. If the competitor is the connector and the gatekeeper is the specifier, we call this option *competitor-side, gatekeeper-implemented API Approach* in Section 4.2. For broad and quick adoption in a setting with multiple gatekeepers, standardizing this client-located API would be beneficial; note that standardizing only the API is much simpler the standardizing an entire protocol stack.

As sketched above, the API takes plaintexts as inputs from the connector's application (e.g., when sending a message to a user of the specifier) and also responds with plaintext outputs (e.g., when receiving a message from a user of the specifier). From the connector's perspective, the translation between these input and output plaintexts into ciphertexts based on the specifier's protocol is invisible behind the API. Therefore, if the specifier does not additionally open and document its server API, the connector would rely on embedding a library that is provided by the specifier. Embedding such a library that consists of (executed) code of other providers could, however, raise security concerns. Moreover, it is unclear if only specifying a plaintext API and providing a library for embedding satisfies the requirement of the DMA that says: "*The gatekeeper shall publish a reference offer laying down the technical details and general terms and conditions of interoperability with its number-independent interpersonal communications*

[9] Having the API at the Specifier's servers means that the Connector has to implement the Specifier's full protocol whereas having the API at the Connector's client means that the Connector's client app only has to handover plaintexts to the Specifier's black box that is located on the Connector's client.

services [...].", Article 7 (4), DMA [69]. Nevertheless, we emphasize that this approach seems to simplify an initial realization of interoperable communication.

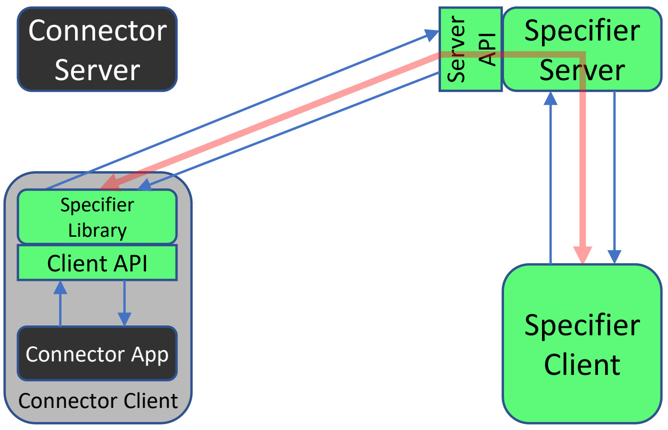

Figure 31: Illustration of API locations with possible specifier library. Blue arrows indicate communication between the entities; the red arrow indicates the E2EE channel between the two clients.

Only Server API For the second option, the specifier only defines an API at their servers with which the connectors can interact. If the competitor is the connector and the gatekeeper is the specifier, this option is called *competitor-side, competitor-implemented API Approach* in Section 4.2 since the connector would need to implement the entire protocol that translates plaintexts into ciphertexts that the specifier's server API expect. Again, standardizing this API among all specifiers would be ideal to allow for simple adoption and scaling. However, since different messaging protocols may need different types of interfaces with the involved servers, the API might be protocol dependent—or at least a core API may need to be extendable.

This approach requires that the connector implements the specifier's protocol for interoperable communication. If one provider is connector to multiple specifiers, implementing all their protocols requires a substantial effort. Thus, while this approach seemingly satisfies the above quoted requirements of the DMA, it may complicate the adoption of the DMA due to the implementation overhead.

Client and Server API Combining the two above approaches means that the specifier defines a ciphertext API for its own servers as well as a plaintext API for the connectors' clients. The latter defines the interaction between the connector's application and a black box (possibly a library) that implements the specifier's messaging protocol. Thus, this black box translates between the input and output plaintext on the one side and the specifier's protocol on the other side.

The benefit of specifying APIs on both levels is that it combines the advantages of the two prior options: Simple and quick adoption is facilitated because a library can be embedded by the connectors' client applications; and, if the connectors do not trust the specifiers' library code, they can refrain from embedding it and, instead, implement the protocol themselves.

We want to mention that the black box in the connector's client application will actually need two APIs: one for the app-to-black box interaction and one for the black box-to-Internet interaction. Since it is pointless to harmonize the latter (black box-to-Internet API) for different connector-specifier pairs, we refrain for elaborating on it in the following. We only want to point out that this black box-to-Internet API will need to realize connections to the specifier's servers as well as peer-to-peer connections to the specifier's clients (e.g., for real-time communication that enable live audio and video calls).

Routing via both Providers For functionality, it suffices that the connector's clients directly talk to the specifier's servers as illustrated in Figure 31—independent of which API is specified. However, for security and privacy, it can be beneficial that the connector's clients route their interoperable traffic through the connector's servers who forwards them to the specifier's servers, which is illustrated in Figure 32. Looking ahead, we indeed see the two following substantial benefits of including the connector's servers on the route: (1) The two providers can cooperate for preventing abuse and filtering spam by using their individual knowledge about their own users—without explicitly sharing that knowledge. (2) The metadata leaked for routing and delivering the ciphertexts from one user to another one can be reduced by giving each server on the route only the information necessary for them—and particularly not more than that. We will see that routing the traffic via both involved providers' servers significantly strengthens privacy, using highly efficient tools that are widely deployed in today's IM apps.

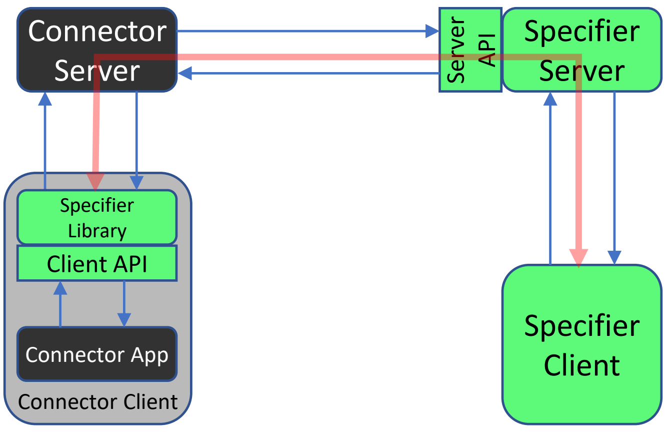

Figure 32: Illustration of routing via both users' providers. Blue arrows indicate communication between the entities; the red arrow indicates the E2EE channel between the two clients.

Additional Remarks Before describing the technical details of the API Approach, we briefly summarize its impact: Firstly, only defining APIs and not fixing a global standard protocol would allow specifiers to keep updating their protocols (up to the constraints of the specified APIs). Furthermore, it would allow for a quick initial adoption if the specifiers' libraries are embedded. If the APIs are generic enough this can be an initial step for gradually moving towards a standardized protocol.

Since purely specifying a client API—and requiring the connectors to embed the specifier's library in their client—seemingly violates the DMA's requirements, we continue discussing the latter two approaches: Only Server API as well as Client and Server API.

4.3.4 Identity Management, Key Distribution, Trust Establishment

Before looking at the protocol components that directly communicate payload between users, we consider the auxiliary protocols that are necessary to initiate the communication and verify that it is actually secure.

To start a conversation, the initiating user has to identify their communication partner. Since the naming schemes of user identities in existing IM apps are different and partially incompatible, an additional mechanism for *interoperable identification* needs to be added. After identifying their communication partner, the user who starts the communication has to derive public key material from their partner. Thus, an *interoperable mechanism for key distribution* needs to be deployed. Finally, after the communication is established, both communication partners need to be able to verify that they are actually communicating with each other, and that no person-in-the-middle attack is mounted that intercepts the two honest partners' communication. Thus, the interoperable protocol needs to provide some form of *session partner verification*.

While user identification and session partner verification can be implemented on the client side, interoperable key distribution requires effort on the server side. For this, we observe that both users' providers already have a key distribution service for their internal messaging protocol. While the specifier may reuse their key distribution service for interoperable communication, the connector cannot. There are two reasons for this: First, the connector's public key material may not be compatible with the specifier's messaging protocol such that it cannot be used for interoperable communication. Second, even if the connector's key material was compatible with the specifier's protocol, reusing it in a different context may lead to so called cross-protocol or protocol-confusion attacks (see, e.g., [21, 11, 61]). Thus, at least the connector needs to add another key distribution service for domain separation; in the worst case, this requires one additional key distribution service for every specifier they are connected with.

The simplest form of session partner verification is visual comparison of so called fingerprints. Equality of such fingerprints displayed on each of the communication partners' devices proves that only these two users can read the communicated payload; different fingerprints, in contrast, indicate that an attacker intercepted the communication. While this simple, yet manual form of authentication via visual verification is implemented in most IM apps, other communication protocols on the Internet achieve the same goal via automated approaches, for example Certificate Transparency [104]. We will not further elaborate on this, but we consider replacing the tedious, error-prone, manual verification with an automated approach advantageous.

Only Server API

If the specifier only documents its server API, the following interfaces need to be available to enable interoperability for identity management, key distribution, and session partner verification.

ID Management Based on our interpretation of the DMA, users of gatekeepers and users of competitors must be able to start a conversation with users of the respective counterpart. Thus, independent of which of both providers has the specifier and connector role, both providers have to offer some form of interface via which the counterpart's users can query for partner identifiers. More concretely, the specifier has to provide and document the following interface:

- SPEC.ID.query(*user_id*) $\rightarrow$ ($\texttt{true}/\texttt{false}$): Interface via which the connector's users can query for identifiers of the specifier's users.
  For simplicity, we treat *user_id* as a plain input but the connector's client could also encrypt the query such that only the specifier's servers could internally decrypt it; in this case, *user_id* would be a ciphertext that does not reveal the queried identifier to anyone besides the connector's client and the specifier's servers.

The connector has to provide the same interface:

- CONN.ID.query(*user_id*) $\rightarrow$ ($\texttt{true}/\texttt{false}$): Interface via which the specifier can query for identifiers of the connector's users. The same comment as above about the plain vs. encrypted query of *user_id* applies here, too.

While it seems to be outside the scope of the DMA, a useful feature would be to extend the above two interfaces such that users can search for sub-strings of or strings similar to user identifiers. With this feature, the user who starts an interoperable communication session would not need to know the exact user identifier of their partner but could adaptively search for it. (To prevent scraping attacks, rate limiting measures should be implemented at these interfaces.)

Key Distribution Both providers have to offer the following interface:

- SPEC.KD.query(*user_id*) $\rightarrow$ *keys* resp. CONN.KD.query(*user_id*) $\rightarrow$ *keys*: Interface via which users of one provider can query for key material of users of the other provider. The same comment as above about the plain vs. encrypted query of *user_id* as well as the output *keys* applies here, too.

We emphasize again that the connector has to add a replication of its existing key distribution service by letting its users generate key material that is compatible with the specifier's messaging protocol. The key generation and upload of key material will need further interfaces at the connector's servers. However, since these interfaces are independent of the specifier, we refrain from detailing them here.

Trust Establishment Using simple, visual fingerprint comparsion, as implemented in almost all widely used IM apps, does not need any interaction between servers and clients. Thus, no server interfaces are necessary for this. Yet, establishing more sophisticated mechanisms like certificate transparency will require the participation of both involved providers' servers.

Client and Server API

If the specifier does not only define an API for its own servers but also for the connector's client, the following additional interfaces would need to be defined for the client API.

ID Management The black box (e.g., the specifier's library) that is embedded in the connector's client and that interacts with the connector's messaging application would need to provide the following interface to the app:

- LIB.ID.query(*user_id*) $\rightarrow$ ($\texttt{true}/\texttt{false}$): Interface via which the connector's application can query for identifiers of the specifier's users.

An extension to search for sub-strings or similar strings as mentioned for the server API could be added to the client API, too.

Key Distribution Since key distribution is a purely technical component that is necessary to initiate an (interoperable) session, there is no need for an explicit interface between client app and specifier's black box.

Trust Establishment For manual verification of the session partners via visual comparison of fingerprints, the specifier's black box has to provide the following interface:

- LIB.SV.fingerprint(*user_id*) $\rightarrow$ *fingerprint*: Interface via which the connector's application can query for a string that represents a fingerprint which captures who the participating users are. The input *user_id* refers to the communication partner's identifier.

Note that such an a posteriori verification extends the simple trust-on-first-use approach.

4.3.5 Two-Party Text Chats

The actual communication protocol is the main component for interoperable messaging. Based on our overview from Section 3.6, we recall that the existing landscape of intra-provider messaging protocols shows significant differences between both the general design ideas and the low level technical implementations. Despite this heterogeneous landscape, we will see that the specifications of abstract server and client APIs for capturing the existing protocols of arbitrary specifiers can be surprisingly simple. In principle, these APIs define interfaces that simply take input and output plaintexts (for the client API) or ciphertexts (for the server API).

After discussing the basic APIs, we discuss how to integrate the connector's servers into the routing of ciphertexts from senders to receivers. This will strengthen the privacy guarantees and minimize revealed metadata.

Only Server API

The server API for two-party text chats is straight forward:

- SPEC.MS.send(c) $\rightarrow$ ($\texttt{true}/\texttt{false}$): Interface via which (users of) the connector can send ciphertexts to users of the specifier. If the specifier's servers need the sender's or receiver's identifiers for processing the ciphertext, these identifiers can be encoded in that ciphertext.

Additionally, the specifier needs to implement a so called push service that informs the connectors' users (or the connector in place of them) about new ciphertext sent to them by the specifier's users. For this, either the connector's servers or the connector's client need to provide the following interface:

- CONN.MS.receive(c) $\rightarrow$ ($\texttt{true}/\texttt{false}$)
  resp. APP.MS.receive(c) $\rightarrow$ ($\texttt{true}/\texttt{false}$): Interface via which (users of) the connector can receive ciphertexts from users of the specifier. If the connector's client or servers need the sender's or receiver's identifiers for processing the ciphertext, these identifiers can be encoded in that ciphertext.

Alternatively, the specifier's servers could provide an interface for a so called pull service via which the connector's users or servers could query for new ciphertexts sent to them; yet, since IM apps aim to avoid unnecessary latency, realizing ciphertext delivery via pull services that only detect new ciphertexts in a coarse pull frequency is impractical.

Client and Server API

The client API for two-party text chats is also straight forward:

- LIB.MS.send$($*user_id*$, m) \rightarrow (\texttt{true}/\texttt{false})$: Interface via which the connector's application can send plaintexts to a user *user_id* of the specifier.

To support quick adoption and extension for the case that more than one provider is declared as a gatekeeper, the message encoding scheme would ideally be standardized.

For receiving plaintext messages, the application needs to open an interface to which the black box pushes decrypted payload:

- APP.MS.receive$($*user_id*$, m) \rightarrow (\texttt{true}/\texttt{false})$: Interface via which the connector's application can receive plaintexts from a user *user_id* of the specifier.

Since IM apps are usually deployed in unreliable, mobile networks, reliable transmission is provided by explicitly acknowledging the receipt of messages. These receipt (and read) acknowledgments are visually realized in most messaging applications by displaying tick marks for the received (and read) messages. To add this reliability feature for interoperable messaging, the client application has to add the following (push) interface: For receiving plaintext messages, the application needs to open an interface to which the black box pushes decrypted payload:

- APP.MS.receiveAck$($*ack*$) \rightarrow (\texttt{true}/\texttt{false})$: Interface via which the connector's application can receive receipt and/or read acknowledgments from users of the specifier. Input *ack* may contain a reference to the message whose receipt or read status it acknowledges.

Routing via both Providers

The description of the APIs so far reads as if the connector's client directly speaks with the specifier's servers—note, however, that the APIs do not require this. Alternatively, the connector's client can interact with the connector's servers (only) and the connector's servers interact with the specifier's servers. We will describe a protocol solution in the next section that utilizes this indirect forwarding approach for better privacy of the communicating users.

To add the connector's servers to the routing of traffic, these servers have to provide the following interface:

- CONN.MS.send$($*spec_id*$, c) \rightarrow (\texttt{true}/\texttt{false})$: Interface via which users of the connector can send ciphertexts to the connector's server that it will then forward to the servers of specifier *spec_id*.

Simple, Privacy Preserving Routing

Using the indirect traffic routing that includes the connector's servers, we describe a simple, privacy preserving protocol that transports ciphertexts interoperably from one user to another one without changing the specifier's original messaging protocol. For this a routing protocol is wrapped around the specifier's original messaging protocol. This routing protocol idea is inspired by a recent work of Len et al. [48].

The protocol works the same in both directions: connector-client–to–specifier-client and vice versa. For simplicity, we only describe the protocol flow for sending a message from the connector's client to a user of the specifier; an illustration of this protocol flow is provided in Figure 33. Messages sent in the opposite direction are processed analogously.

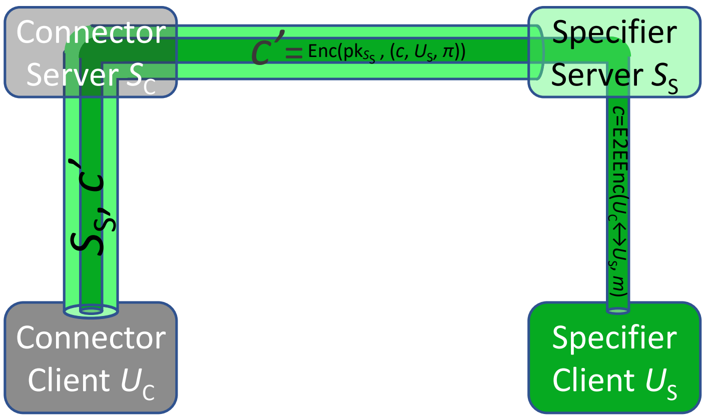

Figure 33: Illustration of privacy preserving wrapper protocol, using routing via both users' providers.

The user sending the message is called U_C (for <u>u</u>ser at <u>c</u>onnector), the connector's server is called S_C, the receiver is called U_S (for <u>u</u>ser at <u>s</u>pecifier), and the specifier's server is called S_S. We assume that U_C and U_S have an ongoing interoperable messaging session and both servers have public keys to which one can encrypt. The protocol works as follows:

1. U_C encrypts message m to U_S using the specifier's messaging protocol, which yields ciphertext c.
2. U_C then encrypts (c, U_S, π) to the public key of S_S, which yields c'.
   Included value π is a proof that attests that U_C is permitted to send messages to U_S. A simple way to implement this proof is based on a mechanism deployed with Signal's Sealed Sender [44]: U_S randomly samples a string π and shares it with S_S when U_S start accepting incoming interoperable messages. During their first interaction with U_C, U_S shares π with U_C. As a result, only those users who share a session with U_S know π and can, thereby, prove their permission to send to U_S.
   For the first message sent from U_C to U_S, U_C has to reveal their identity to S_S in order to prove that U_C was not blocked by U_S before. This is done by setting $\pi \leftarrow U_C$.
   In case U_S did not block any users yet and does not want any filters to be applied on their incoming ciphertexts, U_S could inform S_S that they accept ciphertexts without a proof π. Thereby, already the initial ciphertext in a conversation does not (need to) reveal U_C's identity to S_S.

3. U_C calls interface CONN$_{S_\mathsf{C}}$.MS.send(*spec_id* $= S_\mathsf{S}, c'$) of S_C.

4. S_C forwards c' to S_S, possibly executing abuse prevention and spam filtering routines.

5. S_S decrypts c' to obtain (c, U_S, π). If π is the correct permission proof for U_S or if $\pi = U_\mathsf{C}$ was not blocked by U_S before or if π is empty but U_S accepts sender-anonymous ciphertext from anyone, S_S forwards c to U_S.

6. U_S processes c by executing the specifier's messaging protocol to obtain message m.

Privacy of the Protocol We observe that the sender's servers never learn the receiver's identity. Similarly, the receiver's server do not learn the sender's identity if ciphertext c does not reveal it by itself—with the exception that the first ciphertext in a session may reveal it. Practical solutions to hide the sender identity from ciphertexts are deployed in practice [44] and enhanced protocols for this are developed in the literature [31, 17]. It is remarkable that this solution works without changing the specifier's messaging protocol. Therefore, it can be easily added on top of the existing protocol stack.

We recall that our interpretation of Article 7 (8) of the DMA [69] is that metadata needs to be hidden and protected whenever technically possible. Since the described protocol only uses practical, widely deployed tools that are common in the messaging environment (e.g., simple public-key encryption or, if used with authentication and proof of permission, Signal's Sealed Sender [44]), we think that this protocol serves as a demonstration that always revealing both involved users' identities is not technically necessary. In contrast, hiding this data is indeed technically possible.

4.3.6 File Transfer

Real-world messaging protocols attach files to text messages by encrypting these files with a symmetric encryption scheme, uploading the resulting ciphertext to the server, and sharing the used symmetric encryption key via the normal messaging channel. This simple approach can be adopted for interoperable messaging by providing the following API.

Only Server API

The specifier has to grant access to its storage to which the encrypted files are uploaded:

- SPEC.FS.upload(c) $\rightarrow$ ($\mathtt{true}/\mathtt{false}$, *url*): Interface via which users of the connector can upload encrypted files on the specifier's servers.

- SPEC.FS.download(*url*) $\rightarrow c$: Interface via which users of the connector can download encrypted files from the specifier's servers.

Client and Server API

The client API would be extended with the following interface:

- LIB.FS.sendAttach$(\textit{user_id}, m, f) \rightarrow (\texttt{true}/\texttt{false})$: Interface via which the connector's application can send plaintexts with attached files to a user *user_id* of the specifier.

To receive attachments, the push service would be extended as follows:

- APP.MS.receiveAttach$(\textit{user_id}, m, f) \rightarrow (\texttt{true}/\texttt{false})$: Interface via which the connector's application can receive a plaintext message with an attached file from a user *user_id* of the specifier.

4.3.7 Group Messaging

So far, all described APIs are protocol agnostic, meaning that they abstractly capture inputs and outputs of a large variety of specifier protocols. To capture group messaging, this may not hold: Several deployed group messaging protocols rely on a server that actively participates in the membership management or in the processing of group communication (see Section 3.6). As a result, we consider three levels of server interaction: (1) No interaction because membership management and group communication are conducted on the client side, (2) Server interaction that is based on plaintext information provided by the involved users, and (3) Server interaction that is only based on ciphertexts provided by the involved users.

Only Server API

The main difference between the three types of server involvement is reflected in the server API.

Pairwise Channel Based Protocols that manage membership and group communication exclusively on the client side do not need additional server interfaces.

To give a simple example how that could work, we consider a prior version of Signal's group messaging protocol (cf. [106]): This protocol sends every group message as $n-1$ individual copies that are transmitted via the pairwise channels between the sender and the remaining $n-1$ group members. To make sure that only group members receive group messages, that only group members can send to the group, and that only member can change the set of members, every message contains a secret, random group identifier. Whenever a new user is added to the group, the existing members share the current group identifier with that new member. Based on this, a received message is accepted as a group message only if the contained group identifier is correct. Using the group identifier as a proof of membership, changes of the member set only take effect if they are sent with that proof.

The main advantages of this approach are its simplicity and the fact that the server does not need to be involved. The two main disadvantages are the linear communication overhead and the lack of conflict handling in case of concurrent contradictory membership changes. Due to the communication overhead, the group size may need to be limited as in the intra-provider setting (cf., group size limits enforced in, e.g., Signal).

Server Group Management Plain For specifier protocols that rely on a plain group management processed by the server, the following interface must be provided:

- SPEC.GM.manage$(\mathit{user_id}, \mathit{operation}, \mathit{USER_IDs}) \rightarrow (\texttt{true}/\texttt{false})$: Interface via which users of the connector can initiate operations to manage the set of group members on the specifier's servers. This may include creating a group, adding users, removing users, leaving a group, or promoting users as administrators. The first input is the querying user, the second input is the operation, and the third input is the set of affected users.
- SPEC.GM.update$(c) \rightarrow (\texttt{true}/\texttt{false})$: Interface via which users of the connector can provide cryptographic material based on which the specifier's server executes cryptographic operations for the group. One example for this could be the so called propose-and-commit approach with which the MLS protocol [13] updates key material of the group members.

To inform the connector's users about conducted membership operations or cryptographic calculations, the connector's servers or clients have to provide the following interfaces:

- CONN.GM.changed$(\mathit{user_id}, \mathit{operation}, \mathit{USER_IDs}) \rightarrow (\texttt{true}/\texttt{false})$ resp. APP.GM.changed$(\mathit{user_id}, \mathit{operation}, \mathit{USER_IDs}) \rightarrow (\texttt{true}/\texttt{false})$: Interface via which (users of) the connector can receive applied group membership changes from the specifier's server.
- CONN.GM.update$(c) \rightarrow (\texttt{true}/\texttt{false})$ resp. APP.GM.update$(c) \rightarrow (\texttt{true}/\texttt{false})$: Interface via which (users of) the connector can receive cryptographic material from the specifier's server that initiates the execution of cryptographic group operations.

Server Group Management Encrypted If the server only executes cryptographic operations for the group members but does not process plain membership management, the above described API can be reduced by removing the plain interfaces SPEC.GM.manage, CONN.GM.changed, and APP.GM.changed.

Client and Server API

On the client side, it suffices to replicate the plaintext interfaces:

- LIB.GM.manage$(\mathit{operation}, \mathit{USER_IDs}) \rightarrow (\texttt{true}/\texttt{false})$: Interface via which the connector's application can initiate operations to manage the set of group members on the specifier's servers. This may include creating a group, adding users, removing users, leaving a group, or promoting users as administrators.
- APP.GM.changed$(\mathit{user_id}, \mathit{operation}, \mathit{USER_IDs}) \rightarrow (\texttt{true}/\texttt{false})$: Interface via which the connector's app can receive applied group membership changes from the black box.

The two latter approaches do not induce a significant communication overhead by themselves. Thus, it depends on the specified underlying group messaging protocol if they scale well for large groups (as in MLS) or if the group size needs to be limited (as in, e.g., Signal). A longer discussion about interfaces and implementations for group messaging protocols deployed in practice can be found in [106].

4.3.8 Real-Time Communication

To implement live audio and video calls, the servers and clients have to interact to establish low-latency communication channels that are typically realized on a peer-to-peer layer. Since all IM apps considered in Section 3.6 implement some form or subset of the WebRTC standard [98, 100, 101], using this standard for interoperable live communication seems most practical. We refrain from detailing the exact API defined in the WebRTC standard(s) and instead provide an abstract idea for the necessary interfaces.

Only Server API

On the server side the following interface needs to be provided:

- SPEC.RT.call$(user_id) \to (\texttt{true}/\texttt{false})$: Interface via which users of the connector can initiate a call with a user of the specifier.

In addition to this, the specifier's clients have to provide an interface for receiving a media stream:

- SPAPP.RT.receive$(stream_{\text{conn_to_spec}}) \to (stream_{\text{spec_to_conn}})$: Interface via which the specifier's application can receive a (peer-to-peer) media stream from a user of the connector, to which it responds with such a media stream in the opposite direction.

Finally, the connector's client has to provide this interface, too:

- APP.RT.receive$(stream_{\text{spec_to_conn}}) \to (stream_{\text{conn_to_spec}})$: Interface via which the connector's application can receive a (peer-to-peer) media stream from a user of the specifier, to which it responds with such a media stream in the opposite direction.

Depending on who sends the media stream first, this user will not respond with a second media stream when receiving the media stream from the partner.

We emphasize that this description is an extreme simplification of the actual API necessary to realize real-time communication. Yet, in principle, the abstract idea is captured here.

Client and Server API

For the plain app-to-black box API, the following interfaces need to be provided:

- LIB.RT.call$(user_id, stream_{\text{conn_to_spec}}) \to (stream_{\text{spec_to_conn}})$: Interface via which the connector's application can initiate a media stream with a user of the specifier, to which the black box responds with such a media stream in the opposite direction as soon as it is established.

- APP.RT.receivePlain$(user_id, stream_{\text{spec_to_conn}}) \to (stream_{\text{conn_to_spec}})$: Interface via which the connector's application can receive a media stream via the black box from a user of the specifier, to which the application responds with such a media stream in the opposite direction.

Ideally, the media coding formats are standardized for interoperable real-time communication.

To the best of our knowledge, no systematic performance analysis of end-to-end encrypted real-time protocols has been published. Thus, estimating the performance limitations of such real-time protocols is hard. All IM apps considered in Section 3.6 implement real-time communication based on peer-to-peer protocols. For group real-time communication, the corresponding media streams are simply sent as $n-1$ individual pairwise streams from the sender of each stream to the remaining $n-1$ group members. Using these protocols for interoperable real-time communication will not change the performance. Thus, similar scaling limitations for interoperable group real-time communication will need to be enforced as for the intra-provider counterparts. It is notable that Zoom[10]—probably not being characterized as a standard messaging service—uses the provider's server infrastructure to solve scaling and performance issues. Using the techniques implemented in Zoom with effective peer-to-peer end-to-end encryption appears to be a complex open problem.

4.3.9 Observations, Remarks, and Preliminary Conclusion

The critical advantage of the API approach is that it works (almost) without developing new protocols. It only requires the implementation of existing protocols—possibly supported by embedding simple libraries. Thus, at an early stage of adoption of the DMA, this approach seems to require least effort.

A crucial aspect for a long-term implementation of interoperability is to allow for protocol updates. If the protocols are deployed via the gatekeeper's library, this either means that the specifiers have to review updates of this library or that the specifiers and their users have to trust the gatekeeper. Instead, if the specifiers implement protocol updates themselves, this induces a continuous work load on their end.

The more gatekeepers are declared, the higher the implementation overhead caused by implementing multiple protocols on the competitors' clients based on the (most likely variant of the) API approach. As a result, the standardization approach, considered in subsequent Section 4.4, seems to be a more sustainable and suitable solution in the long run. We still want to note that the two approaches are non-exclusive and the API approach may gradually lead to the standardization approach by standardizing one used component after another.

4.4 Standardization Approach

In this section, we consider *standardization* as an option to implement IM interoperability. Each IM client would then implement two protocols: a proprietary IM protocol with advanced features, and a basic, standardized IM protocol to connect to different IM clients.

[10] https://zoom.us/; Zoom lets the sender of a stream create and send multiple versions of that stream with different quality levels. The server then selects an individual quality level for each recipient depending on the quality of their Internet connection.

4.4.1 IETF MIMI Working Group

Early 2023, the IETF initiated the *More Instant Messaging Interoperability (MIMI)* working group [27]. Its charter states:

> *"The More Instant Messaging Interoperability (MIMI) working group will specify the minimal set of mechanisms required to make modern Internet messaging services interoperable. [...] The standards produced by the MIMI working group will allow for E2EE messaging services for both consumer and enterprise to interoperate without undermining the security guarantees that they provide."*

Since this perfectly aligns with the goals of the DMA, we will refer to the progresses of this group below. However, Internet drafts are still in a very early stage (revision 00 to 02) with very sparse technical documentation so far. The MIMI WG considers the following building blocks.

IM interoperability problems The draft `draft-mahy-mimi-problem-outline-02` gives a general overview on problem areas for which standardization is required:

- **Naming schemes** Different IM implementations use different naming schemes. A general URL-based syntax to unify naming schemes is proposed in `draft-mahy-mimi-identity-01`.
- **End-to-end identity verification** End user verification is done in a proprietary way in each IM scheme. Some IM systems use signature keys generated during app installation, others use manual verification of individual key fingerprints. This problem is addressed in `draft-barnes-mimi-identity-arch-00`.
- **Discovery** Two flavors of discovery are mentioned: Service discovery (where DNS SRV records [79] are proposed as a solution), and user/key material discovery (where user IDs and related identification keys must be discovered across different IM services).
- **Enabling E2EE** Contrary to the description in the charter, X3DH and the Double Ratchet algorithm are considered as possible solutions here, alongside MLS.
- **Content negotiation** In pull protocols like HTTP, client and server may negotiate which type of content they support—e.g., which image file formats the browser is able to display. This problem also arises in interoperable communications between different IM clients. Content negotiation is discussed in the MLS WG.
- **Content format** For both information (text, rich text, audio, video) and meta-information (delivery notifications, replies, reactions), common message formats must be standardized. This problem is discussed in `draft-mahy-mimi-content-02`.
- **Transport protocol** In `draft-rosenberg-mimi-protocol-00`, a REST API is proposed as transport protocol, but XMPP [88] and a variation of the Matrix protocol are also considered.
- **Real-time communications** Here, only the current state of the art is described, where DTLS-SRTP, WebRTC, and SIP are considered.

- **Setup between IM providers** Beyond data formats, algorithms, and protocols, IM providers must also agree on policies before starting interoperable communications. This point is only mentioned, no solution is proposed.
- **Authorization** The question is raised if there is a need for interoperable authorization, in addition to authentication. So far, the question did not lead to meaningful answers or proposals yet.

Two areas related to security are worth mentioning in more detail.

Key agreement/group ratcheting The MIMI WG considers the future MLS standard [13] as the only option for key agreement, both in one-to-one and group communication. However, MLS does not consider entity authentication, so the MIMI charter lists adding this as one of the goals of the WG.

Entity authentication In `draft-barnes-mimi-identity-arch-00`, the MIMI WG present a general (well-known) framework for identity management, and three instantiations for it: manual verification of key fingerprints, X.509, and verifiable credentials. In `draft-mahy-mimi-identity-01`, a generic syntax to describe different IM client identifiers in the unified language of URLs is discussed, along with known trust establishment techniques like PKIs, Web of Trust, and cross-signing; also known cryptographic mechanisms to implement trust inheritance (X.509, JSON Web Tokens, Verifiyble credentials) are considered.

4.4.2 Identity Discovery, Key Distribution, Trust Establishment

Interoperable naming scheme A fundamental prerequisite for standardized interoperability is a unique naming scheme across all IM providers. This standardization step is beneficial and relatively simple, even if other solutions for interoperability are favored, like the API approach described in Section 4.3. The two most important naming schemes today are:

- **International telephone numbers** Historically, national telephone companies could be identified by the country code prefix of the international telephone number. With the liberalization of the telecommunication markets, a larger share of the telephone number prefix was used to identify the different providers. Today, since customers are allowed to keep their mobile phone number even if they change providers, large central databases replace this prefix/suffix naming scheme.
- **E-mail addresses** An e-mail address like `mail@hackmanit.de` consists of two parts: A local mailbox name (`mail`), and a public domain name (`hackmanit.de`), separated by the @ symbol. This enables interoperable message transfer in e-mail systems: An e-mail is first routed to a mailserver of the provider using the domain name; this server is then responsible for placing the e-mail in the correct mailbox.

In `draft-mahy-mimi-identity-01`, the IETF MIMI WG considers solutions based on URLs. Formally, e-mail addresses can be used as parts of an URL: `mailto:mail@hackmanit.de` is a URL, where `mailto:` indicates that the SMTP protocol should be used to push the e-mail to its mailbox, where the mailbox is located at `hackmanit.de`, and the local name of the mailbox is `mail`. The protocol identifier like `mailto:` is not part of the client identity, but indicates that a standardized protocol (here:

SMTP) should be used together with the client identity. Thus, we do not consider protocol prefixes here.

An interoperable naming scheme for IM client identities could therefore use either prefixes or suffixes to identify different IM providers. Based on this, we propose to use special suffixes added to individual IM client identities to distinguish between different IM providers, and to push IM messages to the correct receiving provider. Client ID and suffix should be separated by an ASCII character not used in any individual IM naming scheme (e.g., `@` if this character is not used). Suffixes should be special domain names (e.g., `interop.whatsapp.com`) to leverage on the exiting DNS system for interoperability.

In an API approach, this naming scheme could be used to discover the correct client-side API for sending a message: A message to `id1@interop.whatsapp.com` would be sent using the WhatsApp gatekeeper API, and a message to `id2@interop.messenger.com` would be sent using the Facebook Messenger gatekeeper API.

Identity discovery One of the main reasons for the success of WhatsApp was its automated identity discovery: By using mobile phone numbers as parts of the identity of primary client devices, WhatsApp clients are able to search the address book of a mobile phone for *candidate identities*, and then query a WhatsApp server if these candidate identities are indeed real identities.

If a user consents to this use of their address book, this strategy can be adapted to interoperable identity discovery, but only for IM naming schemes based on mobile phone numbers. In that case, the local IM client would scan the local address book for candidate identities, and then query the identity servers of other IM providers for real identities. This strategy of course implies GDPR issues, since many servers learn all mobile phone numbers of the client. A similar strategy could be adapted to address books of e-mail clients, if e-mail addresses are part of the IM naming scheme.

Besides that, users may exchange identities at personal meetings, e.g., via QR codes, or by standardized push messaging schemes like SMS or e-mail.

Key distribution and trust establishment The most general case for establishing a trust relationship between two IM clients, and thus suitable for standardization, seems to require a long-lived signing key pair. A finite list of mandatory signing algorithms can be standardized, as it is the case, e.g., for TLS, OpenPGP or S/MIME.

The signature key pair should be generated during installation of the IM client, and must be securely associated with the IM identity of the client. If this IM identity involves standard naming schemes like, e.g., mobile phone numbers, the correctness of these naming parts should be guaranteed through an appropriate protocol. Since this secure association of public signing key and IM identity only happens within a specific IM system, there is no need for standardization here.

However, afterwards, the validity of the association of the signing key with the IM identity must be verifiable by any other IM provider. Four different approaches may be considered for standardization, three of which are mentioned in `draft-barnes-mimi-identity-arch-00` and `draft-mahy-mimi-identity-01`:

- **Manual verification/installation** Each client presents the generated signature verification key in a human perceptible, machine readable form. QR codes may be a good choice for this—each user will have to scan he QR code of the other user, presented on their phone. For human users, scanning a QR code implies that messages addressed to this user will be pushed to the mobile device which displayed the QR code. For the IM client, the IM client ID of the other device will be bound to a public signature verification key that can be used to initiate key exchange.
- **X.509 certificates** A classic way to bind identities to public keys are X.509 certificates. Trust is established between two IM systems by sharing a common set of root certificates, which are trusted by both IM providers. The combination of client ID and public key will be signed by a certification authority (CA), and this signature can be validated using one of the trusted root certificates, possibly via intermediate certificates. The main challenge with this approach is that CAs issuing certificates for some IM client ID must implement means to verify that a certain public key belongs to a certain client ID.
- **Cross-signing** This is a slightly modified approach also using X.509 certificates. However, here each IM provider acts as its own CA, and issues X.509 certificates. Trust between two IM providers is established by each of the two parties signing the root certificate of the other party (cross-signing).
- **Verifiable credentials** Verifiable credentials (VC) provide a more flexible data format than X.509 certificates, but serve the same purpose—they should bind an IM client identity to a public signature verification key. Historically, SAML assertions [66] can be considered to be VC and, in a narrower sense, JSON-based signed data structures according to the W3C VC standard [49].
- **DNS records** Given the proposed structure of interoperable IM addresses, with a DNS domain name as suffix, the DNS system itself can be used to establish trust. Here the combination of IM client address and signature verification key will be signed with a private key of the IM provider, and the public key to verify this signature is stored in the DNS. This method is currently used to validate DKIM signatures, where a valid DKIM signature protects an e-mail from being classified as spam.

The concept of *Web of Trust* known from PGP is not suitable for IM systems, since transitivity of trust is not implemented: Public signature verification keys cannot be cosigned. Moreover, this kind of manual involvement of users is known to generate severe usability issues.

These public signature verification keys should then be signed by a trusted authority, within a data structure that also includes the mobile phone number. This trusted authority can either be an IM provider or a root certificate authority (CA).

4.4.3 Text Chats and Ratcheting

Regarding a standardized (two-party) messaging protocol, the IETF MIMI WG has not made substantial progress yet. Nevertheless, obvious, advanced options are mentioned (but not discussed) for key management, like X3DH, Double Ratchet, and MLS.

E2EE The basic security feature of the text chat channel is end-to-end encryption (E2EE). However, starting at this point, standardization may become controversial, since a number of options have to be discussed:

- **E, AE or AEAD** The term *E2EE* only implies that *encryption* is used, but no authentication. However, encryption alone may not even be enough to guarantee the confidentiality of messages—standardization bodies like IETF and W3C therefore add cipher modes that guarantee *authenticated encryption* (AE) or *authenticated encryption with associated data* (AEAD) to their standards. This may increase chat message length (the MAC may be longer than the ciphertext), but since popular modes like Galois/Counter Mode and ChaCha/Poly exist, and since message length does not seem to be a limiting factor in text messaging anymore, a suitable AEAD algorithm should be standardized.

- **Stateful vs. stateless encryption** Synchronizing a common state with another IM application is challenging, since communication between the two applications may be asynchronous. Therefore, it seems easier to implement *stateless* encryption, where a constant key is used to encrypt and authenticate the exchanged messages. Nevertheless, IM applications like Signal and WhatsApp have shown that *stateful* encryption can also be used in asynchronous communication. However, there is no systematic research on how these ratcheting applications behave if a state is lost and how long they keep older states to compensate for the asynchronous communication. Here, input from these companies would be helpful to standardize practical solutions.

- **Key management parameters** In addition to the encryption algorithm itself, public key algorithms and parameters, key agreement protocols, and key derivation mechanisms must be standardized.

Message encryption layer: AEAD All actual versions of encryption-related standards include at least an optional AEAD algorithm. Thus, in any novel standard, it is mandatory to only allow AEAD encryption schemes, like Galois/Counter Mode, or ChaCha/Poly.

Stateless E2EE For IM applications, initial key agreement based on a preshared symmetric key can only be used if manual QR code scanning is conducted for key agreement and trust establishment *before* the actual communication begins—which is highly impractical in most cases. Limiting a future standard to only this option would severely reduce usability. We therefore do not consider this option further.

Instead, each IM client generates a public/private key pair used for public-key encryption (PKE) or key encapsulation (KEM). The public key of this key pair can be authenticated by signing it with the private signing key of the IM client. There are several options on how to use this key pair.

- **Public key encryption for each message** The public key can be used to encrypt a text message directly, or in a hybrid approach (by also employing symmetric encryption). In both cases, authenticity of each text message must be achieved by additional means, e.g., by using digital signatures.

- **KEM for each message** Similar to the above, using hybrid encryption.

- **Static DHKE/NIKE** Rather surprisingly, Threema used static DHKE to derive a symmetric key for encryption between two IM clients. This solution—just as the prior two variants based on PKE and KEM—offers weak security guarantees: If the private key of a participant is revealed, all messages sent to or from the participant can be decrypted later. Threema recently replaced this mechanism and we only list it for the sake of completeness.

- **KEM/PKE for key agreement** The key pairs can be used to agree on an authenticated symmetric key at the start of a communication. This approach is taken in TLS-RSA and TLS-(EC)DH.

Stateless E2EE for push messages is used in both e-mail encryption standards, S/MIME and OpenPGP. With stateless E2EE, chat messages can be stored in encrypted form on the device, since the decryption key is always available. However, stateless E2EE does not provide *forward secrecy* nor *post-compromise security*: If the private key of an IM client is revealed, all messages encrypted to this client can be decrypted afterwards.

Given the current state of discussion in the research and standardization community, the lack of forward security clearly outweighs the ability to store messages in encrypted form. Therefore, this option will probably not gain support in standardization.

Stateful E2EE In this approach, the encryption key is constantly updated. This may happen at regular time intervals, in a synchronized manner; e.g., this is done in TLS whenever a new TCP connection is established. Or this may happen continuously, in an asynchronous manner; this approach is called *ratcheting* in general. In both cases, two IM clients need a public/private key pair for initial establishment of an authenticated symmetric key. For all subsequent key updates, this initial authenticated symmetric key can be used as a basis. For synchronized updates, this is, e.g., used in TLS session resumption. For asynchronous updates, older key material is used in the key derivation for newer key material.

- **Synchronous key updates** Here, the (set of) symmetric AEAD encryption key(s) is updated at certain events (e.g., the closing of the TCP network connection), or at regular intervals (e.g., every 10 minutes). For IM clients, this could happen when both clients are online simultaneously. Each client only has to store a single session state at each point in time. Before a key update, all encrypted messages must be received and updated. Examples with which keys can be re-exchanged or updated include:
  - **X3DH protocol** Deriving a fresh symmetric key by using long-lived DH keys for implicit authentication, and short-lived DH shares for forward secrecy.
  - **Interactive key agreement** Similar to above but based on (interactive) DHKE and digital signatures to agree on a symmetric key. This approach is, for example, taken in TLS-(EC)DHE. Due to the required interaction between users, this approach is less relevant for asynchronous IM settings.
  - **Deterministic key derivation** A secret shared by both clients is used to derive a subsequent key. Based on the one-way guarantee of the key derivation, this approach also offers forward-secrecy without the need for communication between the involved clients.

- **Asynchronous key updates** Here, the (set of) symmetric AEAD encryption keys is updated regularly by the sender of a message, without any synchronization with the recipient. Each client has to store multiple states, since encrypted messages may arrive out-of-order.
  - **Double Ratchet** The Double Ratchet algorithm [63] is a two-party protocol (see Figure 18). The sender of a message updates the encryption key for each message, using a symmetric key derivation function. Whenever the communication direction changes, i.e., on each reply to a sent message, a new DH share is sent to update the state on both sides in a non-deterministic manner. Initially, keys are authenticated using X3DH; later, keys are authenticated by including previous authenticated key material when deriving a new encryption state. If the state is lost, two IM clients may resynchronize using signed prekeys stored on the server with X3DH.
  - **IETF MLS** IETF Messaging Layer Security (MLS) [13] is a group protocol for multiple parties, generalizing older ideas for synchronous group key agreement to asynchronous settings. Similar to the Double Ratchet, MLS regularly lets group members update their secret key material. First applications started to implement and deploy MLS[11], yet little empirical data about user behavior and practical usability is available so far. Since key updates depend on the behavior of members in the group, frequency of these updates, the resulting real-world performance, as well as effects of state loss may, therefore, be an important open issue.

Both synchronous and asynchronous key update mechanisms are used by different IM applications.

While both synchronous and asynchronous key update mechanisms may be suitable for standardization, public discussion both in research and standardization concentrates on the asynchronous case as it works seamlessly for the IM setting. From the two asynchronous options considered here, both have disadvantages with respect to standardization: IETF MLS is in a very early deployment phase and it is based on several synchronicity assumptions (e.g., a central server is needed to establish a global order for technical messages sent in a group). While the high-level protocol documentation of the Double Ratchet algorithm is placed in the public domain [63], the Signal code is GPL licensed, which caused a contention before[12]; furthermore, a post-quantum secure variant of the Double Ratchet algorithm was patented[13], making this variant ineligible for standardization.

Nevertheless, the combination of X3DH and (original) Double Ratchet seems to be the best option for an initial interoperability standard. Yet, we think that the standard should not fix and (over-)commit to a single communication protocol.

Double Ratchet Standardization Signal, WhatsApp, Facebook Messenger, Wire, and Matrix use the Double Ratchet algorithm [63] for two-party communication. Thus, we consider ratcheting state-of-the-art which supports this option for standardization. As discussed in Section 3.6, the exact implementations of these protocols differ on a technical level: For example,

[11] See, e.g., https://blog.webex.com/hybrid-work/scalable-end-to-end-security-in-webex/ and https://www.ringcentral.com/whyringcentral/company/pressreleases/ringcentral-announces-comprehensive-end-to-end-encryption-solution-for-message-video-and-phone.html.

[12] https://wireapp.medium.com/axolotl-and-proteus-788519b186a7

[13] https://patents.google.com/patent/US10412063B1/en

different key derivation algorithms are used to derive the symmetric key chains and the authenticated encryption scheme for message encryption differ. Standardizing a unified scheme—probably in cooperation with the inventors—based on a variant of Signal's Double Ratchet protocol seems a reasonable option.

Standardization of key agreement parameters: Asymmetric Cryptography and Key Derivation The open secure IM standard must define mandatory algorithms to enable secure interoperable communication. These mandatory algorithms must be chosen with care, since they can only be updated by changing the standard.

- **DH group parameters** For non-deterministic updates of states, a variant of DHKE is used in the Double Ratchet algorithm. For this, mandatory groups must be standardized that offer the necessary security guarantees. Given the current state of discussion, these may should be chosen from the set of elliptic curve groups which are currently considered secure.

- **Alternative: KEM** Instead of following the Double Ratchet algorithm directly, a standardized variant could use key encapsulation mechanisms (KEM) to generalize from DHKE. The standardized KEM instantiations may include ElGamal based on elliptic-curve groups as well as post-quantum secure KEMs.

- **Key derivation algorithms** The direct output of the (Diffie-Hellman or KEM based) key update mechanisms are symmetric secrets. To mix these secrets with the prior state, a standardized key derivation function (KDF) is required. Currently, the HKDF algorithm [86] is favored in standardization.

These parameters should not pose any problems in standardization, since there are parameter choices available which are considered secure by both the academic and standardization community.

Management/Update of cryptographic algorithms Proprietary IM systems can update cryptographic algorithms by simply updating their source code. Therefore, semi-specifications issued for such proprietary systems often lack features to manage/update cryptographic algorithms.

For standardized secure IM systems, a mechanism must be defined to negotiate algorithms securely. This requires a clean approach to avoid vulnerabilities such as version downgrade attacks.

4.4.4 File Transfer

To be make sure that attached files encrypted by the app of one provider can be decrypted by another provider's app, a standardized encryption algorithm, a standardized data format for file encryption, and a standardized data format for the decryption key must be used.

Encrypted file formats Fortunately, several general-purpose data formats for file encryption exist and can be used. We briefly discuss their advantages and disadvantages here.

- **OpenPGP** [83] Files can be encrypted as a *Symmetrically Encrypted Data Packet (Tag 9)* [83, Section 5.7]. The most popular implementation, GnuPG, unfortunately does not have an API, but modern implementations exist. OpenPGP supports a weak form of authenticated encryption in all libraries, and may include full AE support in its novel version.
- **PKCS#7/CMS** [74, 85] Files can be encrypted as EnvelopedData [85, Section 6.1]. The most popular application of PKCS#7/CMS is S/MIME.
- **XML Encryption** [36] XML Encryption specifies meta information about encrypted XML files, and the encryption and decryption process. The standard is very flexible, and supports AEAD. However, it has a very rich set of features, from which a small set of mandatory options must be selected in the IM context. Also, due to the very rich feature set of XML, XML parsers can be vulnerable to several attacks, so they must be configured carefully.
- **JSON Web Encryption** [93] Similar to XML, JSON is a text-based data format, whereas CMS and OpenPGP are binary data formats. In contrast to XML Encryption, JSON Web Encryption has a reduced feature set and stricter encryption and decryption processing. It supports AEAD.

Of course it is possible to add AEAD data formats to any other non-cryptographic data format. The drawback of this approach is that libraries which securely implement these formats must be developed first. If one of the standard cryptographic data formats mentioned above are used, these libraries already exist and can be integrated.

We propose to use JSON Web Encryption for encrypting files, since lightweight JSON parsers exist, AEAD is supported, and JSON is now a standard, platform-independent data format in the web. The only drawback is that binary files must be encoded as ASCII.

Format of decryption key The format for the decryption keys (in which the key is included in the text chat message) must be standardized for different IM applications. Here, also JSON (without encryption) can be used.

4.4.5 Group Messaging

For IM, group chats are a standard feature. There are different approaches to manage encryption of messages within a group. The individual ratcheting channels between any two IM clients can be used to establish group keys, or to directly send the group message—i.e., using the pair-wise channels between all group members instead of establishing a single group key. An overview on group key management was given in [106]. We see the following options for standardization here.

- **MLS** Once the IETF Messaging Layer Security (MLS) standard [13] is finalized and deployed, it would be a natural candidate for the standardization of group chats.
- **Static group key** One group manager generates an AEAD group key and distributes it to the other group members via AEAD-encrypted text messages.

- **Sender key mechanism** Each group member generates a group sender key, and distributes this key to all other participants; the sender key may be updated using a key derivation mechanism after each send operation.

- **Pairwise chat channels** The IM applications Signal and Wire have shown that group communication can be based on pairwise chat channels, with different E2E encryption keys for each pair of members in the group. Since pairwise channels with E2EE need to be implemented anyway, we propose to use this simple approach as the preliminary option to implement interoperability in group chats.

Since deployment of MLS has only started very recently, and implementing static group keys or sender key mechanism would add another work item to standardization, we propose to implement interoperable group messaging through the standardized two-party text message channels as described below. Ideally, the standard is flexible enough to also cover MLS as a long-term alternative.

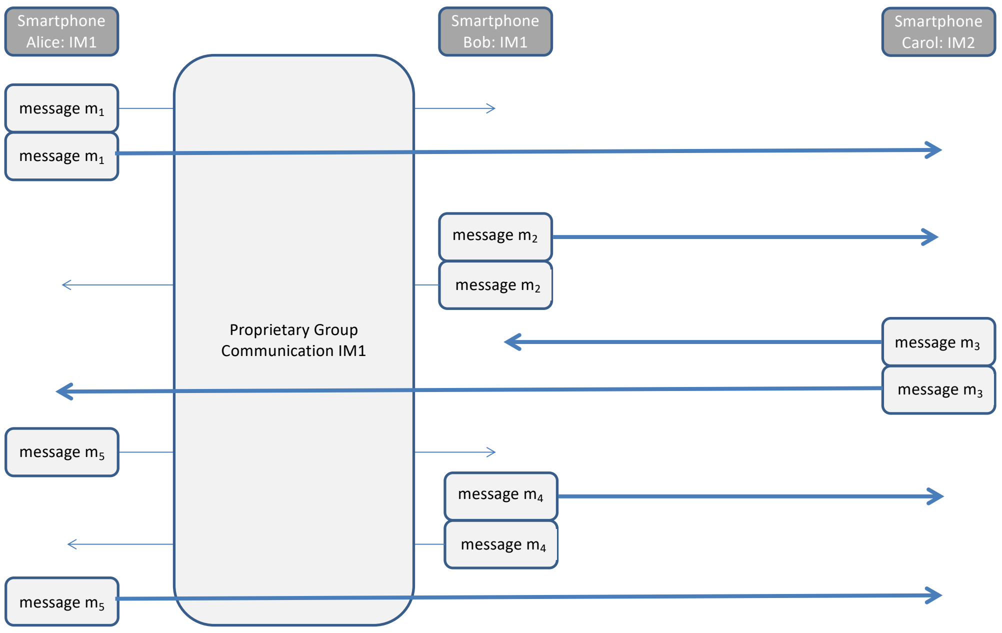

Figure 34: Devices using IM application IM2 can be included in groups defined in IM1 bey using pairwise channels which use E2EE.

In Figure 34, Alice, Bob, and Carol form a static IM group. Alice and Bob use the same IM application IM1, and can therefore use the proprietary group communication protocol of IM1, which was designed to handle any number of group members within IM1. Carol, and any future member of the group can be included in the group communication by using the standardized pairwise communication channels from Section 4.4.3. This is a lightweight solution that that only needs additional standardization for handling group management operations.

4.4.6 Real-Time Communication

Existing standards Real-time audio or video calls require synchronization and jitter control. This can be provided by the secure real-time protocol (SRTP, [78]), or by the real-time protocol (RTP, [77]) over DTLS [91, 103].

WebRTC WebRTC is a complex suite of standards [98, 99, 100] jointly developed by IETF and W3C. It was mainly developed to enable direct browser-to-browser communication, with a signaling channel over webservers. This signaling channel is the most complex part of WebRTC, and it is not needed for IM applications, since a direct communication channel via E2EE encrypted text messages exists. Therefore, it seems overly complex to include the whole set of WebRTC standards in a future IM standard—certain parts of it may however be used.

SRTP SRTP is a full cryptographic solution to encrypt audio or video calls. SRTP only needs a *master key* and a *master salt* [78, Section 4.3.1] from an external key management application. Streams can be encrypted end-to-end, in which case the standardized chat protocol (Section 4.4.3) can be used to agree on a master key/salt. The only standardization requirement would be to agree on a common, standardized data format for this master salt/key.

RTP-over-DTLS In this option, DTLS would be used to encrypt the RTP/UDP packets. However, RTP-over-DTLS comes with a significant overhead, since DTLS consists of a *handshake* and the DTLS *record layer*. Since no X.509 certificate infrastructure exists for this use case (and setting up such a PKI would be completely pointless), the DTLS handshake would be used in preshared key mode, where authentication is realized based on a shared symmetric secret. However, it is also possible to exchange serialized X.509 certificates over the text message channel, or hash values of these certificates. The PSK, which is comparable to the *master key* of SRTP, could be agreed upon using the standardized chat protocol (Section 4.4.3). Still, a DTLS handshake has to be performed. This results in the following standardization requirements:

- Standardization of DTLS preshared key format.
- Standardization of mandatory DTLS version, mandatory cipher suites, and mandatory DTLS options.

SRTP-after-DTLS In this option, DTLS is only used to negotiate SRTP keys, via its handshake. For encryption of real-time data, SRTP is used directly. Compared to RTP-over-DTLS, this solution offers the possibility to inspect synchronization data like timestamps when routing the real-time data.

The standardization of SRTP or of a small subset of WebRTC seems preferable since this involves least standardization overhead. The use of DTLS is obsolete in the IM setting due to existing E2E encrypted communication channels.

4.4.7 Summary: Interoperability through Standardization

The fact that standardization is a feasible option to achieve interoperability is documented by the existence of two IETF working groups: the *More Instant Messaging Interoperability (MIMI) Working Group* [27], and the *Messaging Layer Security (MLS) Working Group* [13]. While the

MLS WG is more focused on specifying and implementing a novel solution for asynchronous group key messaging, the MIMI WG is specifically addressing interoperability issues.

Nevertheless, standardization may not fit the tight DMA schedule to implement and deploy interoperability. This is documented in the long list of open issues in `draft-mahy-mimi-problem-outline-02`, and by the fact that all other drafts of this WG only contain problem statements with only few solutions so far.

Recommended standardization: Naming scheme and initial authentication

There are at least two areas where interoperability solutions would benefit significantly from standardization in general—i.e., also the API approach: A standardized naming scheme for user identities, and standardized end-to-end identity verification. Lets exemplify this with WhatsApp in the role of the gatekeeper.

Example: Gatekeeper WhatsApp wants to identify a Threema user WhatsApp distinguishes between two different types of clients: primary devices and companion devices. Primary devices can only be registered on smartphones or tablets which have their own mobile telephone number. A challenge-response protocol via SMS authenticates the primary device. Companion devices are registered using the primary device, especially its long-lived signing key pair.

This authentication method does not work for a Threema client: Threema does not require devices to have a mobile phone number. Hence, an SMS-based challenge-response protocol cannot be used for interoperable authentication.

Thus gatekeeper WhatsApp has two options: They can offer an unauthenticated interface to Threema users, where any Threema ID can be claimed by the sender of the message (and therefore any Threema ID could be used for malicious activities); or they have to agree on a (standardized) mechanism to authenticate the claimed Threema ID.

Handling non-standardized metadata has been discussed in Section 4.3. We believe that having a unified namespace for IM clients would significantly facilitate such metadata handling. An example of such a unified namespace for push services are e-mail addresses.

Optional standardization

Other interoperability problems can be solved either through an API-based approach, or through standardization.

- **E2EE of text chats** Here, the gatekeeper may provide a plaintext API combined with a cryptographic library to the competitor IM client, or both may use a standardized E2EE API. This includes key management for E2EE.
- **File transfer** Here, a standardized file encryption format may be used by both parties, or the competitor IM client may use the local gatekeeper API for obtaining the decrypted file attached to a text message.

- **Group messaging** A standardized group key protocol (group key, sender key, individual channels, MLS) may be used for group messages, or both gatekeeper and competitor IM client may use individual text message channels added to their internal groups.
- **Real-time communication** For individual calls, both a standardized solution or a cleartext real-time API (e.g., RTP) provided by the gatekeeper can be used. For group calls, a standardized solution may offer significant advantages, e.g., with speaker synchronization.

Benefits of Standardization

Standardizing the basic functionalities for interoperability would require developing a new protocol from scratch. However, since this interoperability protocol is (only) meant for communication between users of different providers, this approach is seemingly independent of the provider-internal communication protocols. Thus, each IM application can still use a proprietary interface to communicate with other apps from the same provider, but use the standardized interface to communicate with other apps. Innovation would not be hindered on the proprietary interface, but will be limited by the speed of standardization of the standard interface. Competitors could implement the standard interface on their own, or use a library provided by the gatekeeper or by third parties. Notably, if there are two more more gatekeepers in the market, competitors would still only have to implement a single additional interface for interoperability.

4.5 Summary

In this report, we describe and discuss paths to implementing the requirements of the DMA regarding interoperability of messaging services. After introducing cryptographic building blocks used in today's instant messaging applications and providing an overview of protocols implemented in these applications, we draw the conclusion that harmonizing differences between these protocols is not a viable approach for interoperability. Consequently, we propose two non-exclusive alternatives: specifying (standard) client and server APIs based on which interoperable protocols can be deployed—this alternative may be supported by client libraries—and standardizing a new, unified interoperable protocol.

Keeping in mind the tight time schedule mentioned in the DMA, and the responsibility assigned to the gatekeepers to enable interoperability, the API approach outlined in Section 4.3 seems to be the first choice in enabling interoperability. However, there are some basic problems that must be solved either way for which standardization seems the best choice, especially because these standardization cases are rather simple and can, therefore, specified quickly (i.e., the required discussion seems to provoke little controversy).

On a larger time schedule and, in particular, when there are multiple gatekeepers in the market, extended standardization of basic IM functionalities can enable interoperability with reduced complexity, better security, and less implementation overhead.

Table 3 summarizes a possible initial path to interoperability. We explain the proposals made below.

4.5.1 Recommended Standardization

We see two components for which we recommend some kind of standardization to enable interoperability even in a very early adoption stage of the DMA.

IM Client Identities Instant messaging providers use their own proprietary naming schemes. If these proprietary names are used across different IM systems, the same name (e.g., the mobile phone number) may be assigned to different IM clients, and IM systems may be unable to assign a foreign identifier to a specific IM system. We therefore propose to standardize a common namespace and a common reference structure. This may be done via a simple naming scheme, where an identifier of the IM system is added to the local name of each IM client. One possible approach follows the structure of e-mail addresses, by adding the domain name associated with the IM provider (e.g., `provider.push` or `interop.provider.tld`) after the local identity `localIM`, separated by a special character like `@`. This approach does not introduce privacy issues as long as the underlying local identity protects the privacy of the user (i.e., if the local identity does not reveal personal data of the user, then neither does the composed global identity).

Initial Key Distribution and Trust Establishment IM providers use their own authentication and identification schemes, which will not work for foreign IM identities. Providing unauthenticated access to any (standardized or gatekeeper-provided) API may lead to impersonation attacks with severe real-world impact. The simplest interoperable mechanism for authentication/trust establishment and initial key distribution between clients is using digital signatures

Interoperability Aspect	**API**	**Standardization**
IM Client Identities	Proprietary identity extensions	Standardization of identities (e.g., `localIMId@interop.IMprovider.tld`)
Identity Discovery	Directory of published identities at `directory.IMprovider.tld`	(JSON) Plaintext API
Initial Key Distribution and Trust Establishment	Directory of public key material published as JSON Web Signatures at `directory.IMprovider.tld`	Standardized JSON Web Signature containing an IM client identity and the client's public key
Two-Party Text Chats	Documented gatekeeper protocol supported by provision of cryptographic library, for inclusion in the competitor client	Plaintext API: Send and receive structured text messages
File Transfer	Documented gatekeeper protocol supported by provision of cryptographic library, for inclusion in the competitor client	Plaintext API: Send and receive attached cleartext files
Group Messaging	Documented gatekeeper protocol supported by provision of cryptographic library, for inclusion in the competitor client	Plaintext API: Group ID, text message
Real-Time Communication	Documented gatekeeper protocol supported by provision of cryptographic library, for inclusion in the competitor client	Plaintext API: RTP protocol with required list of payload formats

Table 3: Summary of API and Standardization Approach

to authenticate a pair consisting of an IM identity and some initial public key material; the latter is used for the key agreement between clients to establish a shared state for end-to-end encrypted communication. We recommend to standardize an authentication and trust establishment mechanism to enable interoperability. A suitable choice may be JSON web signatures, which are already standardized in RFC 7515 [92]. This approach should be complemented by a mechanism that does not rely on trust in the providers (e.g., manual, visual comparison of the partner's identity fingerprints supported by QR code scans as implemented in almost all widely deployed IM apps).

4.5.2 Optional Standardization

Plaintext API All basic IM functionalities discussed in this report would benefit from the standardization of a corresponding plaintext interface. There are two levels of standardization to be considered here.

1. **Common generic message format** One obvious choice for this would be JSON [95], but the comparison of encoding schemes deployed in messaging services in Section 3.6 lists more suitable alternatives.
2. **Specific data formats for specific functionalities** Such an approach could include structured message formats for file transfer, or the Real-Time Protocol (RTP, [77]) with a list of payload formats that are required to be supported for real-time functionalities.

4.5.3 Gatekeeper-provided Libraries and APIs

For all other basic functionalities, a cryptographic library implementing all details of the gatekeeper's protocols, together with a plaintext interface, can solve the interoperability problem. To reduce required trust in the gatekeeper, the protocol details would need to be documented clearly such that competitors can decide whether they want to embed the gatekeeper's library or implement the protocols themselves. The following plaintext interfaces would be needed:

- **Identity Discovery** If an IM user has learned the identity of another IM user in a GDPR-compliant way, this identity can be queried in their provider's database to learn if interoperable communication is possible.
- **Two-Party Text Chats** The client can send plaintext text messages together with the communication partner's client identity to a plaintext API, and the cryptographic library will manage key establishment, key updates, and AE encryption of the message, and will then send the resulting ciphertext to the partner's provider. The sender's provider may be used as a proxy that forwards the ciphertext from the sender's client to the receiver's provider.
- **File Transfer** The plaintext interface accepts a local file reference for text messages with attachments.
- **Group Messaging** The client can send the group ID and the plaintext message to the plaintext interface, and the cryptographic library will manage group keys, encryption, and sending of the ciphertext.

- **Real-time Communication** The plaintext interface will accept plaintext real-time data streams, and "respond" with plaintext streams from the communication partners.

If the gatekeeper updates the implementation of these basic functionalities, these updates can be provided to competitors through cryptographic library updates. Thus, innovations by the gatekeeper and by the competitors can be implemented with software updates only, without the need for standardization.

Long-term solution The described hybrid approach that is based on minimal standardization, plaintext APIs, and technical documentation—ideally supported by a library—is simple and quick on a short scale, but has several drawbacks in the long run. For effective implementation, the competitor apps initially may embed library code from the gatekeepers. However, this requires trust and increases the attack surface. The more gatekeepers a competitor connects to, the larger the attack surface and the bigger the required trust becomes. Therefore, we recommend to gradually increase the set of standardized components such that competitors (and gatekeepers) can avoid using the (possibly changing) gatekeeper protocols. Replacing the gatekeepers' protocol components with standards unifies the code needed to be implemented for interoperability. Beyond this, an increasing stack of standard protocols can be compiled into an open, single, unified library that all competitors (and gatekeepers) can embed. This reduces implementation effort and may join forces between the competitors.

Since the experience in industry and academia with messaging protocols—especially group messaging protocols—is still not mature, we recommend to keep the standard as generic as possible, allowing for multiple, different underlying (continuous) key exchange and encryption protocols. For example, the core of the upcoming MLS standard is based on an idea developed more than five years ago[14]; since then, many extensions and improvements have been proposed in the academic literature that the upcoming standard does not include. A standard for interoperable messaging should, therefore, not assume or commit too *much* to a particular architecture and, instead, leave room for innovation.

4.5.4 Further Considerations

Several aspects that are crucial for functional and secure interoperable messaging go beyond the focus of this study. Some of these aspects were discussed in Section 1.2 already. We briefly want to mention them here:

Transport Layer Security To protect the traffic between clients and servers as well as between servers of different providers, a secure transport layer protocol needs to be employed. Using available options like TLS as the most important, standardized secure channel protocol or less flexible but more lightweight alternatives like the Noise framework for this does not need any further discussion here as it is the usual approach.

Authorized Interface Access To make sure that only authorized entities can access the servers and clients through their interfaces—both in the API approach and in the standardization approach—, additional authorization and authentication protocols can be used. Servers may attest their users' clients towards other providers' servers that they are permitted to interact via these interfaces. This can be simplified if all interoperable traffic from one user is routed via

[14] https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8/

their own provider's server to the other provider's server that forwards it to the actual communication partner; the reason is that then only providers have to implement authorized access to the interfaces among each other.

Abuse Prevention Although users might be authorized by their providers, they can act harmfully and maliciously. Measures to mitigate such behavior may include simple rate limiting at the interfaces, behavioral scanning of interface accesses, spam filtering, etc. Beyond this, users may send abusive content. For this, the providers should implement report mechanisms and blocking procedures. Ideally, the providers cooperate for all these measures.

Organizational Procedures Beyond cooperation for abuse prevention, it is generally important to establish organizational procedures as well as technical provisions for realizing interoperability. This begins with the implementation of the interfaces, resp. standards, and protocols for which both, resp. all sides have to coordinate their development and test their code for compatibility. It continues with making sure that protocol updates, possibly standard updates, as well as implementation updates can be distributed in a coordinated way. Most importantly, security vulnerabilities have to be fixed efficiently and effectively.

Usability and Legal Framework To let users make use of their ability to communicate with users of other providers, it is important to design the user interface for this new option intuitively and comprehensibly. The design of user interfaces is ideally coordinated among the providers.

Furthermore, the entire process of the development and deployment needs to be accompanied by external actors with different backgrounds and perspectives who analyze the security of interop protocols, claim the rights specified by the DMA, prevent and track misbehaving providers, and make sure that the DMA actually achieves its goals.

Trust, Reviews, and Risk Reduction By regulating IM protocols based on the DMA, the IM apps are inevitably becoming dependent on external entities. Examples for these dependencies are: If the specifiers embed libraries of the gatekeepers, their users have to trust the gatekeepers to not implement backdoors; If the specifiers implement the gatekeepers' protocols themselves, they are dependent on the clarity of the gatekeepers' documentation and users who communicate interoperably have to trust two providers to securely implement these protocols; If providers agree on a standard protocol for interoperability, users have to trust the standardization process. It is, therefore, important to keep the processes of interoperability implementations as transparent as possible and their overhead as small as possible.

Security of Interoperable Messaging In both considered cases—standardization as well as documented provision of interfaces—, we draw the conclusion that preserving end-to-end security for interoperable communication is achievable with available, technical building blocks. For this, our focus in this study is the composition and connection of IM networking protocols and cryptographic protocols that enable functional interoperability with the required level of confidentiality and authenticity between the communicating users. Inspired by a recent work [48], we even demonstrate that interoperable communication can partially strengthen privacy guarantees for the communicating users.

It is important to mention that, for fully secure interoperability, also other dimensions than the protocol layer need to be regarded: secure implementation of these protocols, testing of the implementations, updatability of protocol specifications, distribution of the corresponding updated implementations, establishing effective organizational links between the involved providers, mitigating possibly adversarial behavior of involved providers, usability and user-interface design for the interoperability of client applications, etc. At several points in this study, we mention important aspects of these other dimensions. Since the timeline for deployment of interoperable IM required by the DMA is relatively short, we also discuss how quickly the different technical solutions can be realized and how far short-term solutions are compatible with possibly more sustainable, long-term solutions.

5 References

[1] ISO/IEC 9797-1. *ISO/IEC 9797-1 Information technology – Security techniques – Message Authentication Codes (MACs) – Part 1: Mechanisms using a block cipher*. https://www.iso.org/obp/ui/#iso:std:iso-iec:9797:-1:ed-2:v1:en. 2011.

[2] M. R. Albrecht et al. "Practically-exploitable Cryptographic Vulnerabilities in Matrix". In: *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 1419–1436. doi: 10.1109/SP46215.2023.00081. url: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00081.

[3] Martin R. Albrecht et al. "Four Attacks and a Proof for Telegram". In: *2022 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2022, pp. 87–106. doi: 10.1109/SP46214.2022.9833666.

[4] Joël Alwen, Daniel Jost, and Marta Mularczyk. "On the Insider Security of MLS". In: *Advances in Cryptology – CRYPTO 2022, Part II*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Vol. 13508. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2022, pp. 34–68. doi: 10.1007/978-3-031-15979-4_2.

[5] Joël Alwen et al. "Modular Design of Secure Group Messaging Protocols and the Security of MLS". In: *ACM CCS 2021: 28th Conference on Computer and Communications Security*. Ed. by Giovanni Vigna and Elaine Shi. Virtual Event, Republic of Korea: ACM Press, Nov. 2021, pp. 1463–1483. doi: 10.1145/3460120.3484820.

[6] Joël Alwen et al. *Security Analysis and Improvements for the IETF MLS Standard for Group Messaging*. Cryptology ePrint Archive, Report 2019/1189. https://eprint.iacr.org/2019/1189. 2019.

[7] Apple. *Apple Platform Security*. url: https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf.

[8] Apple. *FaceTime security*. url: https://support.apple.com/guide/security/facetime-security-seca331c55cd/web.

[9] Apple. *How iMessage sends and receives messages securely*. url: https://support.apple.com/guide/security/how-imessage-sends-and-receives-messages-sec70e68c949/web.

[10] Theo von Arx and Kenneth G. Paterson. *On the Cryptographic Fragility of the Telegram Ecosystem*. Cryptology ePrint Archive, Report 2022/595. https://eprint.iacr.org/2022/595. 2022.

[11] Matilda Backendal, Miro Haller, and Kenneth G. Paterson. "MEGA: Malleable Encryption Goes Awry". In: *44rd IEEE Symposium on Security and Privacy, SP 2023*. IEEE, 2023.

[12] David Balbás, Daniel Collins, and Phillip Gajland. "Analysis and Improvements of the Sender Keys Protocol for Group Messaging". In: *CoRR* abs/2301.07045 (2023). doi: 10.48550/arXiv.2301.07045. url: https://doi.org/10.48550/arXiv.2301.07045.

[13] Richard Barnes et al. *The Messaging Layer Security (MLS) Protocol*. Internet-Draft draft-ietf-mls-protocol-11. Work in Progress. Internet Engineering Task Force, Dec. 2020. 88 pp. url: https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-11.

[14] Mihir Bellare and Chanathip Namprempre. “Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm”. In: *Advances in Cryptology – ASIACRYPT 2000*. Ed. by Tatsuaki Okamoto. Vol. 1976. Lecture Notes in Computer Science. Kyoto, Japan: Springer, Heidelberg, Germany, Dec. 2000, pp. 531–545. doi: 10.1007/3-540-44448-3_41.

[15] Albrecht Beutelspacher, Jörg Schwenk, and Klaus-Dieter Wolfenstetter. *Moderne Verfahren der Kryptographie*. Springer, 2001.

[16] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. “On the Price of Concurrency in Group Ratcheting Protocols”. In: *TCC 2020: 18th Theory of Cryptography Conference, Part II*. Ed. by Rafael Pass and Krzysztof Pietrzak. Vol. 12551. Lecture Notes in Computer Science. Durham, NC, USA: Springer, Heidelberg, Germany, Nov. 2020, pp. 198–228. doi: 10.1007/978-3-030-64378-2_8.

[17] Alexander Bienstock, Paul Rösler, and Yi Tang. “Secure Messaging in Mesh Networks Using Stronger, Anonymous Double Ratchet”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023*. ACM, 2023.

[18] Alexander Bienstock et al. “On the Worst-Case Inefficiency of CGKA”. In: *TCC 2022: 20th Theory of Cryptography Conference, Part II*. Ed. by Eike Kiltz and Vinod Vaikuntanathan. Vol. 13748. Lecture Notes in Computer Science. Chicago, IL, USA: Springer, Heidelberg, Germany, Nov. 2022, pp. 213–243. doi: 10.1007/978-3-031-22365-5_8.

[19] Jenny Blessing and Ross Anderson. *One Protocol to Rule Them All? On Securing Interoperable Messaging*. https://www.cl.cam.ac.uk/~rja14/Papers/interoperability-spw23.pdf. 2023.

[20] Dan Boneh and Victor Shoup. “A graduate course in applied cryptography”. In: *Draft 0.5* (2020).

[21] Marcus Brinkmann et al. “ALPACA: Application Layer Protocol Confusion - Analyzing and Mitigating Cracks in TLS Authentication”. In: *USENIX Security 2021: 30th USENIX Security Symposium*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, Aug. 2021, pp. 4293–4310.

[22] Ian Brown. *Private Messaging Interoperability In The EU Digital Markets Act*. https://openforumeurope.org/wp-content/uploads/2022/11/Ian_Brown_Private_Messaging_Interoperability_In_The_EU_DMA.pdf. 2022.

[23] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. *Cryptographic Security of the MLS RFC, Draft 11*. Cryptology ePrint Archive, Report 2021/137. https://eprint.iacr.org/2021/137. 2021.

[24] Bundesnetzagentur. *Nutzung von Online-Kommunikationsdiensten in Deutschland, Ergebnisse der Verbraucherbefragung 2021*. https://www.bundesnetzagentur.de/SharedDocs/Downloads/DE/Sachgebiete/Digitales/OnlineKom/befragung_lang21.pdf. 2022.

[25] Mike Burmester and Yvo Desmedt. “A Secure and Efficient Conference Key Distribution System (Extended Abstract)”. In: *Advances in Cryptology – EUROCRYPT’94*. Ed. by Alfredo De Santis. Vol. 950. Lecture Notes in Computer Science. Perugia, Italy: Springer, Heidelberg, Germany, May 1995, pp. 275–286. doi: 10.1007/BFb0053443.

[26] Melissa Chase, Trevor Perrin, and Greg Zaverucha. “The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption”. In: *ACM CCS 2020: 27th Conference on Computer and Communications Security*. Ed. by Jay Ligatti et al. Virtual Event, USA: ACM Press, Nov. 2020, pp. 1445–1459. doi: 10.1145/3372297.3417887.

[27] Alissa Cooper and Tim Geoghegan. *More Instant Messaging Interoperability (mimi) Working Group*. https://datatracker.ietf.org/wg/mimi/about/. 2023.

[28] Whitfield Diffie and Martin E. Hellman. “New Directions in Cryptography”. In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654.

[29] Cory Doctorow and Electronic Frontier Foundation. *An Urgent Year for Interoperability: 2022 in Review*. https://www.eff.org/de/deeplinks/2022/12/urgent-year-interoperability-2022-review. 2022.

[30] Yevgeniy Dodis et al. “Fast Message Franking: From Invisible Salamanders to Encryptment”. In: *Advances in Cryptology – CRYPTO 2018, Part I*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10991. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2018, pp. 155–186. doi: 10.1007/978-3-319-96884-1_6.

[31] Benjamin Dowling et al. “Strongly Anonymous Ratcheted Key Exchange”. In: *Advances in Cryptology – ASIACRYPT 2022, Part III*. Ed. by Shweta Agrawal and Dongdai Lin. Vol. 13793. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Heidelberg, Germany, Dec. 2022, pp. 119–150. doi: 10.1007/978-3-031-22969-5_5.

[32] Element. *The Digital Markets Act explained in 15 questions*. https://element.io/blog/the-digital-markets-act-explained-in-15-questions/. 2022.

[33] Taher ElGamal. “A public key cryptosystem and a signature scheme based on discrete logarithms”. In: *IEEE transactions on information theory* 31.4 (1985), pp. 469–472.

[34] Facebook. *Security Whitepaper*. url: https://about.fb.com/wp-content/uploads/2016/07/messenger-secret-conversations-technical-whitepaper.pdf.

[35] Google. *Protocol Buffers*. https://protobuf.dev/. 2023.

[36] Frederick Hirsch et al. *XML Encryption Syntax and Processing Version 1.1*. W3C Recommendation. http://www.w3.org/TR/2013/REC-xmlenc-core1-20130411/. W3C, Apr. 2013.

[37] Matthew Hodgson and Matrix. *How do you implement interoperability in a DMA world?* https://matrix.org/blog/2022/03/29/how-do-you-implement-interoperability-in-a-dma-world. 2022.

[38] Matthew Hodgson and Matrix. *Interoperability without sacrificing privacy: Matrix and the DMA*. https://matrix.org/blog/2022/03/25/interoperability-without-sacrificing-privacy-matrix-and-the-dma. 2022.

[39] Matthew Hodgson and Matrix. *The DMA Stakeholder Workshop: Interoperability between messaging services*. https://matrix.org/blog/2023/03/15/the-dma-stakeholder-workshop-interoperability-between-messaging-services. 2023.

[40] *Information technology – Open systems interconnection – The Directory: Public-key and attribute certificate frameworks*. ITU-T, 2008. url: http://www.itu.int/rec/dologin%5C_pub.asp?lang=e%5C&id=T-REC-X.509-200811%5C-I!!PDF-E%5C&type=items.

[41] Ingemar Ingemarsson, Donald T. Tang, and C. K. Wong. “A Conference Key Distribution System”. In: *IEEE Transactions on Information Theory* 28.5 (1982), pp. 714–719.

[42] International Telecommunication Union. *The Directory — Overview of Concepts, Models and Services*. ITU-T Recommendation X.500. Nov. 1993.

[43] Tibor Jager, Saqib A. Kakvi, and Alexander May. “On the Security of the PKCS#1 v1.5 Signature Scheme”. In: *ACM CCS 2018: 25th Conference on Computer and Communications Security*. Ed. by David Lie et al. Toronto, ON, Canada: ACM Press, Oct. 2018, pp. 1195–1208. doi: 10.1145/3243734.3243798.

[44] jlund. *Technology preview: Sealed sender for Signal*. https://signal.org/blog/sealed-sender/. Oct. 2018.

[45] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2020.

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

[47] Cameron F Kerry and Patrick D Gallagher. “Digital signature standard (DSS)”. In: *FIPS PUB* (2013), pp. 186–4.

[48] Julia Len et al. “Interoperability in End-to-End Encrypted Messaging”. In: (2023). https://eprint.iacr.org/2023/386. url: https://eprint.iacr.org/2023/386.

[49] David Chadwick Manu Sporny Dave Longley. *Verifiable Credentials Data Model v1.1 - W3C Recommendation 03 March 2022*. url: https://www.w3.org/TR/vc-data-model-1.1/.

[50] Moxie Marlinspike and Trevor Perrin. “The x3dh key agreement protocol”. In: *Open Whisper Systems* 283 (2016).

[51] Matrix. “Megolm group ratchet”. In: (2022). url: https://gitlab.matrix.org/matrix-org/olm/blob/master/docs/megolm.md.

[52] Matrix. “Olm: A Cryptographic Ratchet”. In: (2019). url: https://gitlab.matrix.org/matrix-org/olm/blob/master/docs/olm.md.

[53] Matrix. *Security Documentation*. url: https://matrix.org/docs/guides/end-to-end-encryption-implementation-guide.

[54] David A. McGrew and John Viega. “The Security and Performance of the Galois/Counter Mode (GCM) of Operation”. In: *Progress in Cryptology - INDOCRYPT 2004: 5th International Conference in Cryptology in India*. Ed. by Anne Canteaut and Kapalee Viswanathan. Vol. 3348. Lecture Notes in Computer Science. Chennai, India: Springer, Heidelberg, Germany, Dec. 2004, pp. 343–355.

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

[56] Jens Müller et al. ““Johnny, you are fired!” - Spoofing OpenPGP and S/MIME Signatures in Emails”. In: *USENIX Security 2019: 28th USENIX Security Symposium*. Ed. by Nadia Heninger and Patrick Traynor. Santa Clara, CA, USA: USENIX Association, Aug. 2019, pp. 1011–1028.

[57] NIST. “Data Encryption Standard”. In: *In FIPS PUB 46, Federal Information Processing Standards Publication*. 1977, pp. 46–2.

[58] NIST. *Specification for the Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. 2001. url: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[59] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[60] Amandine Le Pape and Matrix. *Digital Markets Act and interoperability: Debunking the gatekeepers' myths*. https://matrix.org/blog/2022/02/03/digital-markets-act-and-interoperability-myth-debunking-of-the-gatekeepers-arguments. 2022.

[61] Kenneth G. Paterson, Matteo Scarlata, and Kien Tuong Truong. "Three Lessons From Threema: Analysis of a Secure Messenger". In: *Attack Website* (2022). url: https://breakingthe3ma.app/files/Threema-PST22.pdf.

[62] Trevor Perrin. *The Noise Protocol Framework (Rev. 34)*. http://www.noiseprotocol.org/noise.html. July 2018.

[63] Trevor Perrin and Moxie Marlinspike. *The Double Ratchet Algorithm*. https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf. url: https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf.

[64] Damian Poddebniak et al. "Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels". In: *USENIX Security 2018: 27th USENIX Security Symposium*. Ed. by William Enck and Adrienne Porter Felt. Baltimore, MD, USA: USENIX Association, Aug. 2018, pp. 549–566.

[65] Bertram Poettering et al. "SoK: Game-Based Security Models for Group Key Exchange". In: *Topics in Cryptology – CT-RSA 2021*. Ed. by Kenneth G. Paterson. Vol. 12704. Lecture Notes in Computer Science. Virtual Event: Springer, Heidelberg, Germany, May 2021, pp. 148–176. doi: 10.1007/978-3-030-75539-3_7.

[66] Nick Ragouzis et al. *Security Assertion Markup Language (SAML) V2.0 Technical Overview*. https://www.oasis-open.org/committees/download.php/27819/sstc-saml-tech-overview-2.0-cd-02.pdf. 2008.

[67] *RC4 Source Code release on Cypherpunks mailing list*. https://web.archive.org/web/20010722163902/http://cypherpunks.venona.com/date/1994/09/msg00304.html. Sept. 1994. url: https://web.archive.org/web/20010722163902/http://cypherpunks.venona.com/date/1994/09/msg00304.html.

[68] *REGULATION (EU) 2016/679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation))*. https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679&from=EN. 2016.

[69] *REGULATION (EU) 2022/1925 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 14 September 2022 on contestable and fair markets in the digital sector and amending Directives (EU) 2019/1937 and (EU) 2020/1828 (Digital Markets Act)*. https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32022R1925. 2022.

[70] Eric Rescorla. *Architectural options for messaging interoperability*. https://educatedguesswork.org/posts/dma-interop/. 2023.

[71] Eric Rescorla. *Discovery Mechanisms for Messaging and Calling Interoperability*. https://educatedguesswork.org/posts/messaging-discovery/. 2022.

[72] Eric Rescorla. *End-to-End Encryption and Messaging Interoperability*. https://educatedguesswork.org/posts/messaging-e2e/. 2022.

[73] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104 (Informational). RFC. Updated by RFC 6151. Fremont, CA, USA: RFC Editor, Feb. 1997. doi: 10.17487/RFC2104. url: https://www.rfc-editor.org/rfc/rfc2104.txt.

[74] B. Kaliski. *PKCS #7: Cryptographic Message Syntax Version 1.5*. RFC 2315 (Informational). RFC. Fremont, CA, USA: RFC Editor, Mar. 1998. doi: 10.17487/RFC2315. url: https://www.rfc-editor.org/rfc/rfc2315.txt.

[75] M. Handley et al. *SIP: Session Initiation Protocol*. RFC 2543 (Proposed Standard). RFC. Obsoleted by RFCs 3261, 3262, 3263, 3264, 3265. Fremont, CA, USA: RFC Editor, Mar. 1999. doi: 10.17487/RFC2543. url: https://www.rfc-editor.org/rfc/rfc2543.txt.

[76] J. Rosenberg et al. *SIP: Session Initiation Protocol*. RFC 3261 (Proposed Standard). RFC. Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026, 6141, 6665, 6878, 7462, 7463, 8217, 8591, 8760, 8898, 8996. Fremont, CA, USA: RFC Editor, June 2002. doi: 10.17487/RFC3261. url: https://www.rfc-editor.org/rfc/rfc3261.txt.

[77] H. Schulzrinne et al. *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550 (Internet Standard). RFC. Updated by RFCs 5506, 5761, 6051, 6222, 7022, 7160, 7164, 8083, 8108, 8860. Fremont, CA, USA: RFC Editor, July 2003. doi: 10.17487/RFC3550. url: https://www.rfc-editor.org/rfc/rfc3550.txt.

[78] M. Baugher et al. *The Secure Real-time Transport Protocol (SRTP)*. RFC 3711 (Proposed Standard). RFC. Updated by RFCs 5506, 6904. Fremont, CA, USA: RFC Editor, Mar. 2004. doi: 10.17487/RFC3711. url: https://www.rfc-editor.org/rfc/rfc3711.txt.

[79] J. Peterson. *Address Resolution for Instant Messaging and Presence*. RFC 3861 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Aug. 2004. doi: 10.17487/RFC3861. url: https://www.rfc-editor.org/rfc/rfc3861.txt.

[80] JH. Song et al. *The AES-CMAC Algorithm*. RFC 4493 (Informational). RFC. Fremont, CA, USA: RFC Editor, June 2006. doi: 10.17487/RFC4493. url: https://www.rfc-editor.org/rfc/rfc4493.txt.

[81] K. Zeilenga (Ed.) *Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map*. RFC 4510 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2006. doi: 10.17487/RFC4510. url: https://www.rfc-editor.org/rfc/rfc4510.txt.

[82] J. Sermersheim (Ed.) *Lightweight Directory Access Protocol (LDAP): The Protocol*. RFC 4511 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2006. doi: 10.17487/RFC4511. url: https://www.rfc-editor.org/rfc/rfc4511.txt.

[83] J. Callas et al. *OpenPGP Message Format*. RFC 4880 (Proposed Standard). RFC. Updated by RFC 5581. Fremont, CA, USA: RFC Editor, Nov. 2007. doi: 10.17487/RFC4880. url: https://www.rfc-editor.org/rfc/rfc4880.txt.

[84] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). RFC. Obsoleted by RFC 8446, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919, 8447, 9155. Fremont, CA, USA: RFC Editor, Aug. 2008. doi: 10.17487/RFC5246. url: https://www.rfc-editor.org/rfc/rfc5246.txt.

[85] R. Housley. *Cryptographic Message Syntax (CMS)*. RFC 5652 (Internet Standard). RFC. Updated by RFC 8933. Fremont, CA, USA: RFC Editor, Sept. 2009. doi: 10.17487/RFC5652. url: https://www.rfc-editor.org/rfc/rfc5652.txt.

[86] H. Krawczyk and P. Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869 (Informational). RFC. Fremont, CA, USA: RFC Editor, May 2010. doi: 10.17487/RFC5869. url: https://www.rfc-editor.org/rfc/rfc5869.txt.

[87] A. Freier, P. Karlton, and P. Kocher. *The Secure Sockets Layer (SSL) Protocol Version 3.0*. RFC 6101 (Historic). RFC. Fremont, CA, USA: RFC Editor, Aug. 2011. doi: 10.17487/RFC6101. url: https://www.rfc-editor.org/rfc/rfc6101.txt.

[88] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Core*. RFC 6120 (Proposed Standard). RFC. Updated by RFCs 7590, 8553. Fremont, CA, USA: RFC Editor, Mar. 2011. doi: 10.17487/RFC6120. url: https://www.rfc-editor.org/rfc/rfc6120.txt.

[89] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*. RFC 6121 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Mar. 2011. doi: 10.17487/RFC6121. url: https://www.rfc-editor.org/rfc/rfc6121.txt.

[90] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Address Format*. RFC 6122 (Proposed Standard). RFC. Obsoleted by RFC 7622. Fremont, CA, USA: RFC Editor, Mar. 2011. doi: 10.17487/RFC6122. url: https://www.rfc-editor.org/rfc/rfc6122.txt.

[91] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347 (Proposed Standard). RFC. Obsoleted by RFC 9147, updated by RFCs 7507, 7905, 8996, 9146. Fremont, CA, USA: RFC Editor, Jan. 2012. doi: 10.17487/RFC6347. url: https://www.rfc-editor.org/rfc/rfc6347.txt.

[92] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Signature (JWS)*. RFC 7515 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, May 2015. doi: 10.17487/RFC7515. url: https://www.rfc-editor.org/rfc/rfc7515.txt.

[93] M. Jones and J. Hildebrand. *JSON Web Encryption (JWE)*. RFC 7516 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, May 2015. doi: 10.17487/RFC7516. url: https://www.rfc-editor.org/rfc/rfc7516.txt.

[94] Y. Nir and A. Langley. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 7539 (Informational). RFC. Obsoleted by RFC 8439. Fremont, CA, USA: RFC Editor, May 2015. doi: 10.17487/RFC7539. url: https://www.rfc-editor.org/rfc/rfc7539.txt.

[95] T. Bray (Ed.) *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259 (Internet Standard). RFC. Fremont, CA, USA: RFC Editor, Dec. 2017. doi: 10.17487/RFC8259. url: https://www.rfc-editor.org/rfc/rfc8259.txt.

[96] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Aug. 2018. doi: 10.17487/RFC8446. url: https://www.rfc-editor.org/rfc/rfc8446.txt.

[97] J. Schaad, B. Ramsdell, and S. Turner. *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification*. RFC 8551 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Apr. 2019. doi: 10.17487/RFC8551. url: https://www.rfc-editor.org/rfc/rfc8551.txt.

[98] H. Alvestrand. *Overview: Real-Time Protocols for Browser-Based Applications*. RFC 8825 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Jan. 2021. doi: 10.17487/RFC8825. url: https://www.rfc-editor.org/rfc/rfc8825.txt.

[99] E. Rescorla. *Security Considerations for WebRTC*. RFC 8826 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Jan. 2021. doi: 10.17487/RFC8826. url: https://www.rfc-editor.org/rfc/rfc8826.txt.

[100] E. Rescorla. *WebRTC Security Architecture*. RFC 8827 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Jan. 2021. doi: 10.17487/RFC8827. url: https://www.rfc-editor.org/rfc/rfc8827.txt.

[101] J. Uberti and G. Shieh. *WebRTC IP Address Handling Requirements*. RFC 8828 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Jan. 2021. doi: 10.17487/RFC8828. url: https://www.rfc-editor.org/rfc/rfc8828.txt.

[102] C. Bormann and P. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 8949 (Internet Standard). RFC. Fremont, CA, USA: RFC Editor, Dec. 2020. doi: 10.17487/RFC8949. url: https://www.rfc-editor.org/rfc/rfc8949.txt.

[103] E. Rescorla, H. Tschofenig, and N. Modadugu. *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*. RFC 9147 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Apr. 2022. doi: 10.17487/RFC9147. url: https://www.rfc-editor.org/rfc/rfc9147.txt.

[104] B. Laurie, E. Messeri, and R. Stradling. *Certificate Transparency Version 2.0*. RFC 9162 (Experimental). RFC. Fremont, CA, USA: RFC Editor, Dec. 2021. doi: 10.17487/RFC9162. url: https://www.rfc-editor.org/rfc/rfc9162.txt.

[105] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”. In: *Communications of the Association for Computing Machinery* 21.2 (1978), pp. 120–126.

[106] Paul Rösler, Christian Mainka, and Jörg Schwenk. “More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema”. In: *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 2018, pp. 415–429.

[107] Christoph Schmon and Electronic Frontier Foundation. *EU Parliament Takes First Step Towards a Fair and Interoperable Market*. https://www.eff.org/deeplinks/2021/11/eu-parliament-takes-first-step-towards-fair-and-interoperable-market. 2021.

[108] Jörg Schwenk. *Guide to Internet Cryptography – Security Protocols and Real-World Attack Implications*. 1. Edition. Information Security and Cryptography. Springer, 2022.

[109] G Sig. “Gesetz über Rahmenbedingungen für elektronische Signaturen und zur Änderung weiterer Vor-schriften”. In: *Bundesgesetzblatt Teil I* 22 (2001), pp. 876–884.

[110] Signal. *Github Repository*. url: https://github.com/signalapp/.

[111] Natalie Silvanovich. *Exploiting Android Messengers with WebRTC: Part 1*. https://googleprojectzero.blogspot.com/2020/08/exploiting-android-messengers-part-1.html. 2020.

[112] Natalie Silvanovich. *Exploiting Android Messengers with WebRTC: Part 2*. https://googleprojectzero.blogspot.com/2020/08/exploiting-android-messengers-part-2.html. 2020.

[113] Natalie Silvanovich. *Exploiting Android Messengers with WebRTC: Part 3*. https://googleprojectzero.blogspot.com/2020/08/exploiting-android-messengers-part-3.html. 2020.

[114] Mitch Stoltz et al. *The EU Digital Markets Act's Interoperability Rule Addresses An Important Need, But Raises Difficult Security Problems for Encrypted Messaging*. https://www.eff.org/de/deeplinks/2022/04/eu-digital-markets-acts-interoperability-rule-addresses-important-need-raises. 2022.

[115] Telegram. *End-to-End Encrypted Voice and Video Calls*. url: https://core.telegram.org/api/end-to-end/video-calls.

[116] Telegram. *End-to-End Encryption, Secret Chats*. url: https://core.telegram.org/api/end-to-end/.

[117] Telegram. *MTProto Mobile Protocol*. url: https://core.telegram.org/mtproto.

[118] Serge Vaudenay. "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS..." In: *Advances in Cryptology – EUROCRYPT 2002*. Ed. by Lars R. Knudsen. Vol. 2332. Lecture Notes in Computer Science. Amsterdam, The Netherlands: Springer, Heidelberg, Germany, Apr. 2002, pp. 534–546. doi: 10.1007/3-540-46035-7_35.

[119] Théophile Wallez et al. *TreeSync: Authenticated Group Management for Messaging Layer Security*. Cryptology ePrint Archive, Report 2022/1732. https://eprint.iacr.org/2022/1732. 2022.

[120] WhatsApp. *Security Whitepaper*. url: https://scontent-muc2-1.xx.fbcdn.net/v/t39.8562-6/309473131_1302549333851760_6207638168445881915_n.pdf?_nc_cat=107&ccb=1-7&_nc_sid=ad8a9d&_nc_ohc=xPyxKXUARBcAX9UC73G&_nc_ht=scontent-muc2-1.xx&oh=00_AfAI4CtOmzHygHSuUJuBRaCEft_LlbCwlluNbPMcLp9QOw&oe=638BB35D.

[121] Alma Whitten and J. Doug Tygar. "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0". In: *Proceedings of the 8th USENIX Security Symposium, Washington, DC, USA, August 23-26, 1999*. 1999. url: https://www.usenix.org/conference/8th-usenix-security-symposium/why-johnny-cant-encrypt-usability-evaluation-pgp-50.

[122] Wire. *Security Whitepaper*. url: https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf.