PQSHIELD

Think openly, build securely

White paper:

Secure Messaging in a Post-Quantum World

Software | PQShield | August 2022

Executive Summary

This white paper discusses the security of secure messaging protocols in a post-quantum world. While this is a technical document, we also hope to provide more general audiences with an understanding of the stakes, challenges, and techniques of (post-quantum) secure messaging protocols.

We start by surveying the current state of secure messaging, the threat models, and the challenges. We then explain how these guide the definition of cryptographic notions that capture real-life security requirements while enabling a rigorous scientific analysis.

We then delve into technical matters. We explain how to add post-quantum security to the secure messaging protocol *Signal* in the case of conversations between two users. We conclude this white paper by highlighting and addressing the scalability challenges of secure messaging in a group setting.

The Need for Secure Messaging

The first section familiarizes a general audience with the main cryptographic notions studied in this document. For a scientific audience, this section explains how prevalent security notions (such as forward secrecy and post-compromise security) are justified by concrete threats.

This introductory section provides a bird's-eye view of the secure messaging landscape, showcasing the main actors (such as Signal, WhatsApp, Telegram, etc.), common threat models, and laying out a few security notions. We opt for a top-down approach, starting with broad and general considerations, then narrowing our focus to secure messaging *protocols*, which concentrates on the challenges related to post-quantum cryptography.

Making Signal Post-Quantum

The second section discusses how to provide post-quantum security to the secure messaging protocol Signal in the two-party setting. We chose Signal for a few reasons: it is commonly considered one of the most secure messaging protocols out there, and it is also one of the most involved from a cryptographic standpoint, making its analysis both valuable and interesting.

At a high level, Signal can be decomposed into two sub-protocols: X3DH for the initial handshake phase, and Double Ratchet for ongoing conversations. Unfortunately, both sub-protocols heavily rely on the *Diffie-Hellman key-exchange*, a cryptographic construction that is not post-quantum, and for which we don't have a clear post-quantum alternative. As a result, having a post-quantum version of Signal is non-trivial.

In the second part of this section, we explain how these technical challenges have been addressed. In a collaborative work accepted at PKC 2021, PQShield and academic researchers proposed the first generic, post-quantum alternative to X3DH [HKKP21]. Similarly, industry and academic researchers proposed a generic, post-quantum alternative to Double Ratchet [ACD19]. Together, both results paved the way for deploying a post-quantum variant of Signal.

Scalable Group Messaging Protocols

The third and final section discusses another challenge of secure messaging protocols: scalability when considering large groups of users. It turns out that *group* secure messaging protocols have the potential to quickly drain end users' mobile data, which can have highly undesirable consequences. This motivates a concrete need for group messaging protocols providing both efficiency and a high-security guarantee.

We compare the efficiency and security guarantees of *four* secure group messaging protocols: Sender Keys (WhatsApp), Pairwise Channels (Signal), TreeKEM (Messaging Layer Security – MLS) and Chained CmPKE (proposed by PQShield and academic researchers at CCS 2021 [HKP+21]). All four protocols are either post-quantum secure or easy to make so. Each of them explores a different corner of the design space of messaging protocols, and each has a complementary strength. We also briefly discuss how a ciphertext compression technique, presented by PQShield and academic researchers at ASIACRYPT 2021 [KKPP20], significantly decreases the bandwidth consumption of TreeKEM.

Contents

Acronyms

Table 1: Table of acronyms

Acronym	Expansion
AEAD	Authenticated encryption with additional data
AKE	Authenticated key-exchange
CmKEM	Committing mKEM
DH	Diffie-Hellman
E2EE	End-to-end encryption
FS	Forward secrecy
HMAC	Hash-based message authentication code
KDF	Key-derivation function
KEM	Key-encapsulation mechanism
mKEM	Multi-recipient KEM
MLS	Messaging Layer Security
PCS	Post-compromise security
PKE	Public-key encryption
PKI	Public-key infrastructure
PQS	Post-quantum security
SMA	Secure messaging application
SMP	Secure messaging protocol
SMPF	Secure messaging platform
TLS	Transport layer security
X3DH	Extended triple Diffie-Hellman

1 The Need for Secure Messaging

This section introduces the reader to the cryptographic notions necessary to understand the stakes, challenges, and eventually the design choices of secure messaging.

We first present the landscape of secure messaging (§1.1). We separate this landscape into roughly two components: secure messaging applications (§1.1.1) on one side, and the legal and technical tools leveraged to attack them (§1.1.2) on the other side.

We then show how we can map these real-world constraints to technical and cryptographic notions (§1.2). These notions are asynchrony (§1.2.1), end-to-end encryption (E2EE, §1.2.2), forward secrecy and post-compromise security (FS and PCS, §1.2.3), and post-quantum security (PQS, §1.2.4).

1.1 The Landscape of Secure Messaging

A messaging platform is an internet-based platform whose primary purpose is to allow its users to exchange messages instantaneously. The use of messaging platforms has skyrocketed this last decade, fueled by the increasing adoption of smartphones. In parallel, individuals have become increasingly concerned about the confidentiality and privacy of their conversations, prompting efforts to design *secure* messaging platforms.

On the other hand, state and private actors have developed a set of legal and technical tools to access the content of individuals' conversations without their consent. As is often the case in cybersecurity, solutions used by one side (either platforms or attackers) inform the design of tools developed by the other side. As a cryptography company, PQShield is naturally more inclined to adopt the viewpoint of messaging platform designers. However, understanding both perspectives goes a long way in comprehending why these platforms are designed the way they are.

1.1.1 (Secure) Messaging Applications

The most common way the users engage with messaging platforms is via mobile messaging applications (e.g., on iOS or Android), as opposed to browser or desktop applications. It is therefore unsurprising that the widespread adoption of smartphones during the last decade has been accompanied by a meteoric rise in messaging platforms. Fig. 1 presents some of the most popular applications: WhatsApp, iMessage, Signal, Telegram, etc. Due to this prevalence, the documented attack models, attacks, and mitigations of said attacks are largely tailored to mobile messaging applications. We also adopt this point of view.

While all of the messaging applications in Fig. 1 claim to be secure, we will see that some of them have an elastic definition of what "secure" means, leading to significant differences in the security guarantees they provide. These disparities are well-documented but sometimes overlooked by the general public, and we will discuss them in this section (see summary in Table 2, page 9). First, we lay down some terminology in Fig. 2.

Fig. 2 illustrates the size of the attack surface of a secure messaging platform: a weakness in the security of the protocol or the secure messaging application (SMA), or a compromise of the server

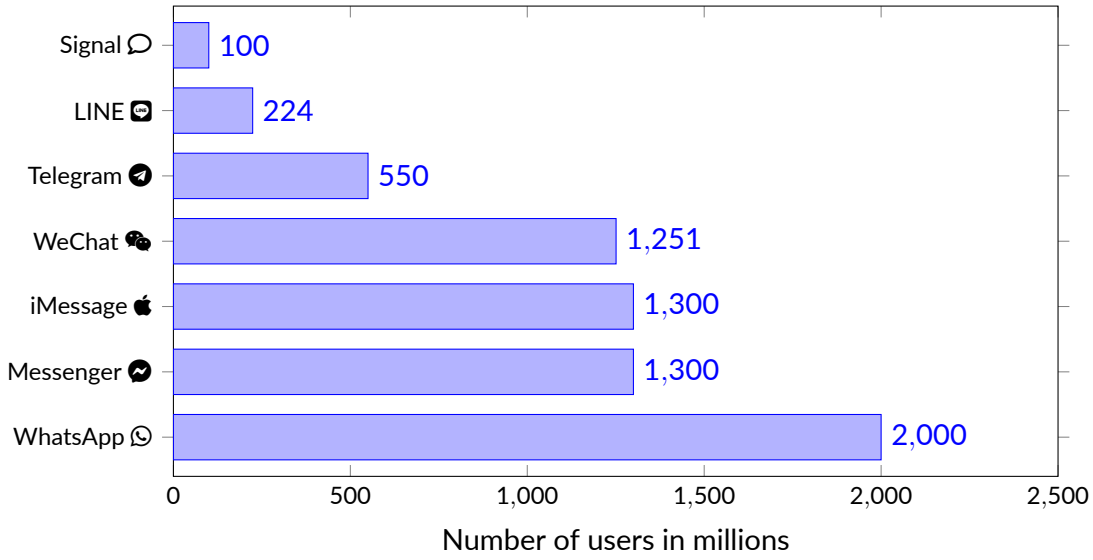


Figure 1: Estimated market share of some prominent messaging applications.

or a device, can potentially have dramatic consequences for the end users. In this document, we focus on the security of *secure messaging protocols*. Indeed, most of the cryptographic complexity of secure messaging concentrates on the protocol layer and raises significant challenges for post-quantum security. This makes it a natural study topic for PQShield since we specialize in post-quantum cryptography. Note that many secure messaging protocols are designed with mitigations in place for dealing with a potential compromise of the server and/or a device, as we will discuss in-depth in §1.2.

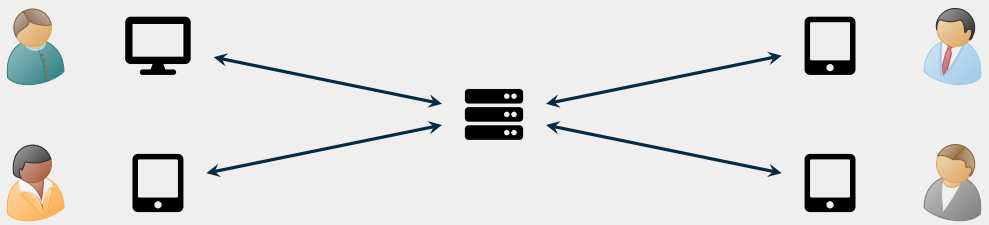


Figure 2: Each user has a *(secure) messaging application* (SMA) on their device (,).
The *(secure) messaging protocol* (⟷) specifies the communications between the server () and the devices at the cryptographic layer. The *(secure) messaging provider* (SMP) specifies the messaging protocol, operates the server and, typically, developed the messaging application.

1.1.2 Legal and Technical Attack Vectors

Since their popularisation about a decade ago, SMAs have been targets of choice for motivated attackers. Attackers may be state actors, companies, hackers, or sometimes a mixture thereof. We review some of the main documented attack vectors. Note that many of these may not only compromise conversations exchanged via SMAs but the entire contents of targeted devices.

Laws and Warrants (Legal)

A typical method employed by state actors is adopting laws that give state agencies the ability to compel secure messaging providers to hand over information about their customers. In Fig. 2, this

could correspond to providing all information transiting via the server (🖴).

A well-documented example is the so-called Yarovaya law that went into effect in 2018 in Russia. The subsequent and highly publicized stand-off between the Russian government and Telegram [EM21] shows that the concrete application of such laws is not necessarily straightforward.

However, in the USA, subpoena-backed access to server data seems common practice among law-enforcement agencies. In that regard, one enlightening source is an internal FBI document recently made public via a Freedom of Information Act (FOIA) request[1]. This document summarises the data that the FBI can request from major secure messaging providers, and we reproduce it on Page 45. The extent of this surveillance is a great motivation for design choices detailed in §1.2.2.

Finally, warrantless seizure and search of electronic devices at border crossings are frequent.

Device Hacking (Technical)

Another method consists of hacking user devices to gain access to their conversations: this can be done by brute-forcing user passwords, leveraging zero-day exploits, etc. This is becoming a common practice in state agencies around the world, to the point that specialised companies such as NSO or Cellebrite [Cel20] advertise this as part of their services. Sometimes the secure messaging application itself is the attack vector, as documented with a highly technical remote iMessage exploit by NSO [Zer21].

Backdoors (Legal and Technical)

Finally, a legal and technical method is to compel device manufacturers and/or messaging platforms to implement backdoors in their products to allow systematic access or monitoring of user content.

In Western countries, the discourse around this approach is heavily politicized, and backdoors are often reframed by their promoters as "responsible encryption" or "client-side scanning (CSS)" (see [AAB+21] for a technical discussion of CSS). To the best of our knowledge, there is no widespread adoption of such stringent measures in these countries, though we note that very recent EU law proposals might change this status quo[2].

On the other hand, an extensive series of reports by Citizen Lab[3] show that some form of large-scale automated surveillance and censorship seems to be deployed in the WeChat/Weixin application. We could not infer from the reports whether this censorship is performed by the WeChat platform or by the Chinese government[4].

1.2 How Real-World Constraints Impact Protocol Designs

We now discuss how real-world constraints, including the threats discussed in §1.1.2, can be translated into technical and cryptographic notions. Some prominent notions are given in Table 2, and we will discuss each of them in a dedicated section.

[1] https://propertyofthepeople.org/document-detail/?doc-id=21114562

[2] TechCrunch: *Europe's CSAM scanning plan unpicked* https://techcrunch.com/2022/05/11/eu-csam-detection-plan/

[3] Citizen Lab Research: *WeChat Surveillance Explained* https://citizenlab.ca/2020/05/wechat-surveillance-explained/

[4] WeChat is developed by Tencent and is based in China.

Table 2: Popular messaging applications and properties on underlying protocols: support of asynchrony, end-to-end encryption (E2EE), forward secrecy (FS), post-compromise security (PCS), and post-quantum security (PQS).
Taxonomy: not supported (✗), opt-in or partial support (✓), enabled by default (✓✓). For each application, we distinguish conversations between two users (👥) and group conversations (👥👥).

		WeChat		LINE		Telegram		Messenger		WhatsApp		Signal	
		👥	👥👥	👥	👥👥	👥	👥👥	👥	👥👥	👥	👥👥	👥	👥👥
Asynchrony	(§1.2.1)	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
E2EE	(§1.2.2)	✗	✗	✓	✓	✓	✗	✓	✗	✓✓	✓✓	✓✓	✓✓
FS	(§1.2.3)	✗	✗	✗	✓	✓	✗	✓	✗	✓✓	✓✓	✓✓	✓✓
PCS	(§1.2.3)	✗	✗	✗	✓	✓	✗	✓	✗	✓✓	✓	✓✓	✓✓
PQS	(§1.2.4)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

1.2.1 Asynchrony

The first constraint is purely functional. Participants in a conversation may not be online at the same time, and this has a higher chance of happening when multiple participants are involved and/or when they span several time zones. Fig. 3 illustrates this constraint and how to address it: an always-online server (🖴) operated by the secure messaging platform sits between the parties (🧍 and 🧍), storing messages sent by one party and relaying them when the other party comes online. This is a simple architectural choice, but it has several security implications (see §1.2.2).

Figure 3: Transport encryption and end-to-end encryption (E2EE). $\boxed{\text{DATA}}_k$ indicates that DATA is encrypted using the symmetric key k. We use distinct colors for end-to-end encryption and transport encryption. The server shares a transport encryption key k_A with Alice, another key k_B with Bob, and can therefore read the metadata MD. However, the end-to-end encryption key k is only known to Alice and Bob, hence the server cannot read the message MSG.

1.2.2 End-to-End Encryption (E2EE)

This second constraint stems from the choice made in §1.2.1. All messages transit through a server (🖴) operated by the secure messaging provider (SMP), which raises a natural question:

How much do we trust the SMP and the server it operates?

As highlighted in §1.1.2, there is a real possibility in many countries that state agencies can access the contents of the server of the SMP. This poses an obvious threat to the privacy of the end users of the messaging platform.

End-to-end encryption (or E2EE) provides a partial answer to this threat. The principle of E2EE is that the contents of a message sent between two endpoints (in our case, the end users) cannot be read or modified by any entity between the endpoints. This is illustrated in Fig. 3, where Alice () and Bob () exchange encrypted messages via a server (), and a layer of E2EE prevents the server from reading the messages. While E2EE is nowadays a fairly standard notion for secure messaging applications, some high-profile SMPs are still in the process of enabling it by default; for example LINE, Telegram and Facebook Messenger propose it as an opt-in feature (under the terms "*Letter Sealing*", "*Secret Chat*" and terms "*Secret Conversation*", respectively).

Transport Encryption and E2EE

In secure messaging, we need to consider two layers of encryption: *transport encryption* (orange in Fig. 3) and *E2EE* (green in Fig. 3). This document focuses on E2EE, since it encompasses most technical challenges. Indeed, transport encryption is fairly straightforward in comparison, as it deals with a setting similar to TLS: one server, one client and synchronous communications. Transport encryption protocols used by major providers, for example MTProto (Telegram) and Noise (WhatsApp), are indeed variants of TLS.[5] Post-quantum alternatives to TLS are well-studied [Lan18, Kwi19, SSW20], and we expect them to be easy to adapt for transport encryption.

Limitations of E2EE

For all its merits, E2EE still suffers from some limitations. A few of these are listed below, and merely illustrate two simple facts: (a) in secure messaging, not all data is necessarily protected by E2EE, and (b) data protected by E2EE can be leaked *outside* of the E2EE protocol.

- **Metadata collection.** As illustrated in Fig. 3, metadata such as the social graph of users ("who talks to whom") are typically only encrypted at the transport layer, not the E2EE layer. As such, they can easily be recovered from the server data by motivated attackers (see Page 45). Protocols such as *Sealed Sender* [Sea18] and *Private Groups* [Pri19] by Signal protect such metadata from the server.
- **Device compromise.** If the device of a user is compromised (e.g., via hacking or seizure by authorities), so will the contents of their conversation, regardless of how strong E2EE is. Forward secrecy and post-compromise security, discussed in §1.2.3, provide some partial mitigation when the scope of compromise is limited in time. However, they would not address more powerful threat models such as client-side scanning or backdoors.
- **Unencrypted cloud backups.** As often in cyber-security, the devil is in the details. Strong E2EE is pointless if the information it protects can leak via other means. For example, if E2EE is applied to conversations but not to cloud backups, then a state authority can request the contents of cloud backups and completely bypass E2EE conversations. This is already observed in the wild[6]. One could assume that E2EE cloud backups are standard practice, but this is far from being the case. For example, iMessage does not provide E2EE cloud backups, and WhatsApp has only rolled out this feature very recently[7].

[5] MTProto: https://core.telegram.org/mtproto. Noise: https://noiseprotocol.org/.

[6] Forbes: *Meet The Secretive Surveillance Wizards Helping The FBI And ICE Wiretap Facebook And Google Users* https://www.forbes.com/sites/thomasbrewster/2022/02/23/meet-the-secretive-surveillance-wizards-helping-the-fbi-and-ice-wiretap-facebook-and-google-users/

[7] Engineering at Meta: *How WhatsApp is enabling end-to-end encrypted backups* https://engineering.fb.com/2021/09/10/security/whatsapp-e2ee-backups/

We note that the geographical location of servers storing backups may depend on several factors (the location of the user, the secure messaging provider, financial incentives, etc.), and backups may be duplicated across several data centers ("geo-redundancy"). These points make the security implications of unencrypted cloud backups even murkier.

1.2.3 Forward Secrecy and Post-Compromise Security

E2EE is a good security baseline, but it does not suffice to address the unique security challenges arising in secure messaging. Unlike TLS sessions, conversations on messaging platforms can last months, if not years. Under these conditions, it becomes plausible that a participant in a conversation will see their device compromised at some point during the conversation. This can be caused by a hack or by the physical seizure of the device, and raises the following question:

To what extent can we protect our messages in face of a device compromise?

The response to this question is multi-pronged. At the application level, Telegram allows members of a group conversation the ability to erase messages, including those sent by other group members. Concurrently, one could set messages to disappear upon reading and/or after a fixed time limit (*disappearing messages*), an approach supported by Telegram, Signal, and WhatsApp among others.

To be meaningful, these application-level solutions must be completed by equally strong guarantees at the protocol level. In other words, the secure messaging protocol should have mechanisms in place that would limit an adversary's ability to read past and future messages, even if said adversary has compromised the server and one or more user devices. For instance, if the adversary stored all the encrypted conversations sent between the users, then a device compromise could allow him to decrypt all prior conversations, making any protection at the application-level ineffective.

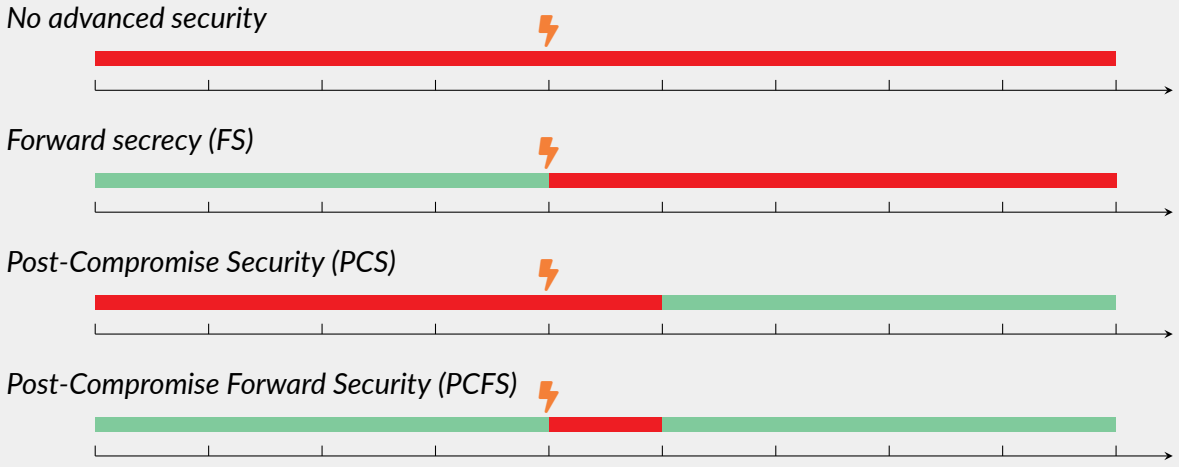


Figure 4: FS, PCS and PCFS. Assuming an attacker compromises the session at a given time (ϟ), each security notion guarantees that epochs in pistachio green (▬) remain secure. No security guarantee is made for epochs in red (▬). For PCS and PCFS, a delay is sometimes required before security is recovered.

Forward Secrecy (FS)

Informally, forward secrecy (FS) guarantees that if a user is compromised at a given time, then the contents of his prior conversations remain secure (see Fig. 4). This preserves security of the exchanged conversations happening *before* a compromise. As a concrete example, a protestor whose mobile phone is seized by the authorities may want to preserve the confidentiality of their past conversations.

At the application layer, FS is provided by *disappearing messages*, though we note that SMAs which propose it (Signal, WhatsApp, Telegram, etc.) often make it an *opt-in* feature. At the protocol layer, FS must guarantee that an attacker learning the cryptographic keys of a user cannot decrypt the contents of his prior conversations (even if the attacker learns long-term decryption or signing keys). Some high-profile SMPs provide E2EE but do not provide FS at the E2EE layer, see for example LINE [LIN20] and Threema [Thr21, Soa21]. In §2 we will discuss various means of achieving FS.

Post-Compromise Security (PCS)

Post-compromise security (PCS) is concerned with preserving security of the exchanged conversations happening *after* a compromise. This notion, first formally introduced in [CCG16], is complementary to FS that concerns security *before* a compromise. The long lifetimes of mobile devices and of the secure conversations they host provide natural motivations for PCS. As a first example, consider a user whose mobile device gets hacked by exploiting a zero-day vulnerability; the vulnerability later gets patched, and the hacker no longer has access to the mobile phone. As a second example, a traveling user may have their device confiscated by border customs for examination, then returned a few hours later. In both examples, there is a significant chance that the device content is compromised, including cryptographic keys. How can we *heal* from such a device compromise?

At first glance, it seems extremely difficult to preserve meaningful security guarantees in such situations. It turns out that if we consider the adversary to be *passive during a sufficiently long period after the compromise*, a reasonable level of PCS can be obtained by rotating cryptographic (public) keys in a specific manner during this period. This practice is sometimes called *(interactive) ratcheting*. We discuss interactive ratcheting for the classical two-party setting (§2.1.3), the post-quantum two-party setting (§2.2.2), and the group setting (§3). Note that interactive ratcheting can be bandwidth-expensive; Signal and WhatsApp currently are the only widespread SMAs who implement this practice and therefore achieve PCS.

1.2.4 Post-Quantum Security

Post-quantum security (PQS) guarantees that security properties such as confidentiality, integrity, etc., are still upheld in the presence of adversaries equipped with large-scale quantum computers. With the upcoming standardization of post-quantum cryptographic primitives by organizations such as NIST [NIS20], it becomes increasingly urgent to adapt existing protocols in order to integrate these quantum-safe primitives. We invite interested readers to read our white papers on post-quantum cryptography [PQS20] and one of the post-quantum cryptographic standards [PQS21].

Of all the secure messaging applications we have mentioned so far, none currently achieve PQS. Augmenting existing SMP designs with PQS raises several challenges in terms of feasibility, efficiency, and scalability. Thankfully, an increasing number of these challenges are being solved, either by members of PQShield or by external researchers. We discuss the challenges and how to solve them in §2 and §3.

2 Making Signal Post-Quantum

The *Signal* protocol is widely regarded as the gold standard for establishing secure messaging between two users. However, the cryptographic problem underlying its security is known to be easily solvable by quantum computers, and any adversary harvesting current communications would be able to uncover the exchanged messages in the future. This warrants concern to entities in the long-term security and motivates the need for a *post-quantum secure* (PQS) secure messaging protocol.

This section studies the Signal protocol and explains the main obstacles to making it PQS. It then shows several recent solutions to overcome these obstacles and illustrates how they can be combined into a PQS protocol that achieves the same functionality and (in fact, better) level of security as the Signal protocol.

2.1 Overview of the Signal Protocol

While there has been a range of secure messaging protocols, the Signal protocol [SIG] is typically considered the reference when it comes to secure messaging between two users (see Table 2). Not only is it used in the Signal application, but variations of the core protocol are also deployed within various other applications such as WhatsApp, Messenger's Secret Conversations, and Skype private conversations. Understanding the design principle of the Signal protocol will help us understand other secure messaging protocols — including those with weaker security guarantees — with relative ease.

Below, we dissect the Signal protocol and explain how appealing security properties like *forward secrecy* (FS, see §1.2.3) and *post-compromise security* (PCS, see §1.2.3) are guaranteed. We then explain in §2.2 the obstacles for turning the Signal protocol post-quantum secure and provide recent solutions by industry and academic researchers to overcome them. This section focuses on secure messaging protocols (SMPs) between two users, and the group setting is deferred to §3.

2.1.1 The Players and Goal of the Signal Protocol

We name (and) in the figures as Alice and Bob, respectively. Recall that an SMP required an always-online server to operate between Alice and Bob for them to communicate asynchronously (see §1.2). With this in mind, we can state the question solved by the Signal protocol:

> *How can Alice and Bob asynchronously exchange messages securely with the help of a* possibly malicious *server?*

Notice that considering a protocol to be secure against a malicious server provides strong security; for instance, the conversations remain secure even if state actors accessed the server data. Using slightly more technical terms, this question can be further divided into two questions.

(Q1) *How can Alice and Bob establish a shared key with a possibly malicious server in between?*

(Q2) *How can Alice and Bob guarantee FS and PCS even when the shared secret key gets compromised?*

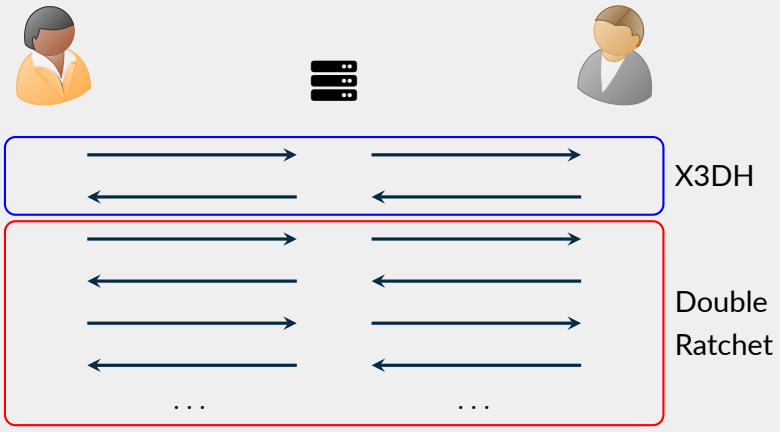


Figure 5: A modular view of the Signal protocol, decomposed as the X3DH and Double Ratchet protocols.

- **X3DH:** See Fig. 7 for a description, and Fig. 9 for a post-quantum variant.
- **Double Ratchet:** See Fig. 8 for a description, and Fig. 11 for a post-quantum variant.

Unlike the case of transport encryption (see §1.2.2), the answer to (Q1) is not as simple as "Use TLS" due to the asynchronicity between Alice and Bob. In the next subsections, we explain at a high level how the Signal protocol answers both (Q1) and (Q2). Specifically, **the Signal protocol is divided into two sub-protocols called the *X3DH* (for *Extended Triple Diffie-Hellman*) [MP16b] and *Double Ratchet* [MP16a] protocols, which answer (Q1) and (Q2) respectively.** This is illustrated in Fig. 5.

2.1.2 Solution to (Q1): The X3DH Protocol

The goal of the X3DH protocol is to *asynchronously* establish a secure key (which we call a *session key*) between Alice and Bob. The session key is used within the Double Ratchet protocol to exchange messages. We explain the X3DH protocol in two steps.

STEP 1. The X3DH protocol builds implicitly on the following Diffie-Hellman (DH) based *authenticated key exchange* (AKE) protocol $\mathsf{AKE}_{\mathsf{X3DH}}$ depicted in Fig. 6. In this figure, lpk_X (resp. lsk_X) denotes the *long term* public (resp. secret) key for user $X \in \{A \text{ (Alice)}, B \text{ (Bob)}\}$, KDF is a key derivation function (for example the HMAC-based HKDF [Kra10]), and K is the exchanged session key. Moreover, Sign and Verify are the signing and verification algorithms of an XEdDSA signature [Per16].

At a high level, Alice and Bob can be seen as running *three* (dependent) DH AKE protocols whose goal is to share the values g^{ay}, g^{bx}, and g^{xy}, respectively. $\mathsf{AKE}_{\mathsf{X3DH}}$ combines all three values into one session key K via a KDF. For our purpose, it suffices to understand that the signature σ_A allows Bob to *explicitly* know that Alice generated g^x. Now, informally, as long as one of g^{ay}, g^{bx}, or g^{xy} is unknown to the adversary, then the output of the KDF remains random over its output domain. For instance, even if both the long term secret keys lsk_A and lsk_B are compromised *after* Alice and Bob finished the key exchange protocol, an adversary cannot compute the term g^{xy} due to the hardness of the decisional Diffie-Hellman (DDH) problem. Since g^{xy} is unknown, the session key K will remain secure.

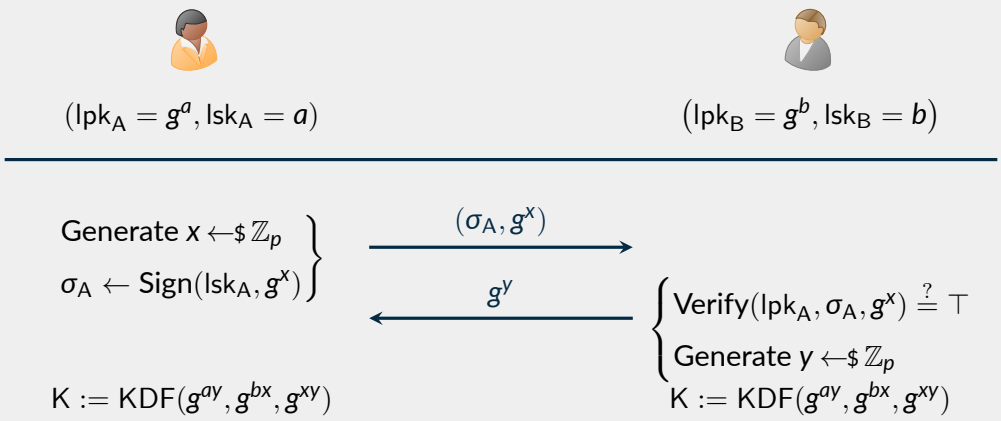


Figure 6: DH-based AKE protocol $\mathsf{AKE}_{\mathsf{X3DH}}$ underlying the X3DH protocol. The exchanged contents (σ_A, g^x) and g^y are sent over a non-authenticated insecure channel.

Other Types of DH-based AKE Protocols

Some natural questions are why g^{ab} is not fed to the KDF or whether there are any other ways to derive session keys from the tuple (g^a, g^x, g^b, g^y). Indeed, many other similar DH-based AKE protocols mainly differ in the derivation of the session key K from the tuple (g^a, g^x, g^b, g^y). Some famous AKE protocols are HMQV [Kra05], NAXOS [LLM07], and $\mathcal{TS}2$ [JKL04]. Depending on how the session key is derived, they offer better efficiency or, in some cases, provide stronger security properties. Notably, it could be possible to consider an alternative X3DH protocol starting from a different type of DH-based AKE protocol. For further reading, [CCG+19, Table 1] provides a list of known DH-based AKE protocols.

How Are Long Term Keys Exchanged?

One issue we glossed over in the above discussion is how Alice and Bob exchange their long-term public keys lpk; the computation of K implicitly relied on knowledge of the peer's lpk. Generally, this is solvable by a public key infrastructure (PKI), where an authenticated list of user long-term public keys is publicly stored. In case PKI is not part of the SMA ecosystem (as in the case of the Signal app), users may compare their lpk manually or by scanning a QR code using a different channel. Methods for such out-of-bound authentication are outside the scope of this document. See [Wha21, Vau05, RS18] for more details.

STEP 2. The X3DH protocol turns $\mathsf{AKE}_{\mathsf{X3DH}}$ into an asynchronous protocol by putting an always-online server in between Alice and Bob as depicted in Fig. 7. We can decompose the X3DH protocol in two phases, each corresponding to the respective values sent by Alice and Bob in Fig. 6.

- *Registration Phase*: All users upload to the server three cryptographic values: the long term public key lpk, a *pre-key signature* σ, and a *signed pre-key* g^x. These three values — called the *pre-key bundle* — correspond to $(\mathsf{lpk}_A, \sigma_A, g^x)$ in Fig. 6.
- *Session Key Establishment Phase*: When Bob wishes to start a secure messaging session with

Alice, Bob first fetches the pre-key bundle of Alice from the server. If Alice's pre-key signature σ_A is valid, then Bob sends g^y to the server as in Fig. 6. As explained in §2.1.3, Alice and Bob use the derived session key K to send encrypted messages via the Double Ratchet protocol. Alice can retrieve g^y from the server once online to establish the same session key K as Bob.

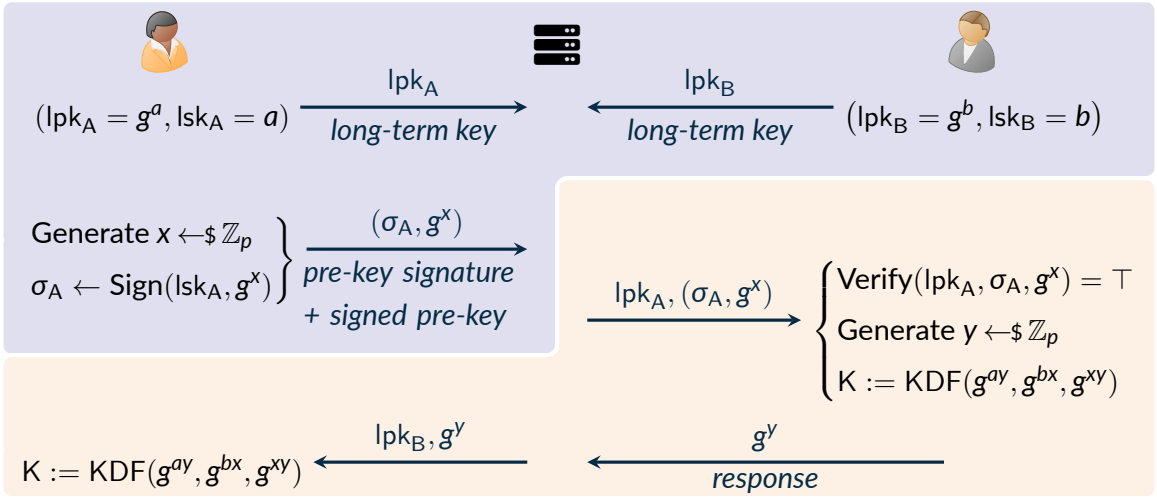


Figure 7: The X3DH protocol. Upon registration (in blue), users upload their pre-key bundle to the server. Users can upload multiple pre-key signatures and signed pre- keys (σ_A, g^x) for better security (see below for the detail). At any moment, Bob can fetch Alice's pre-key bundle and run $\mathsf{AKE}_{\mathsf{X3DH}}$ (in orange).

One salient property of X3DH is **receiver obliviousness**: users do *not* need to know who they will be communicating with at the registration phase. For instance, Alice's pre-key bundle can also be used by another user Charlie. This feature is essential to any *instantaneous* SMA since otherwise, users who wish to talk to Alice must wait till she uploads a pre-key bundle designated for them.

Security of the X3DH Protocol

The X3DH protocol (Fig. 7) inherits *almost* all of the strong security guarantees offered by the underlying AKE protocol $\mathsf{AKE}_{\mathsf{X3DH}}$ (Fig. 6). This is because AKE protocols are by design, supposed to remain secure even if an attacker (i.e., a malicious server) sits in between the users.

The only subtle difference between the two protocols is that while the pre-key bundle of Alice is never reused in $\mathsf{AKE}_{\mathsf{X3DH}}$, it is reused in the X3DH protocol. More precisely, Alice never uses the same (σ_A, g^x) in $\mathsf{AKE}_{\mathsf{X3DH}}$. The effect of this difference only shows up when the security of the AKE protocol *fails*.

When an adversary compromises *both* the long-term secret and the session-specific randomness $(\mathsf{lsk}_A = a, x)$, he can easily compute the session key K. However, since the content (σ_A, g^x) sent by Alice is never reused, the only session key that becomes compromised is the session that used g^x. On the other hand, for the X3DH protocol, since Alice's pre-key bundle is reused by many users, all of the session keys established by Alice become compromised. Put differently, by reusing the same pre-key bundle, the number of session keys that become insecure in case of a total compromise of Alice's device is more devastating for the X3DH protocol than the underlying AKE protocol $\mathsf{AKE}_{\mathsf{X3DH}}$.

There are two ways the X3DH protocol mitigates the scope of compromise. The first way is to minimize the number of sessions that reuse a particular pre-key bundle. This is accomplished by periodically refreshing and uploading a newly signed pre-key and pre-key signature, e.g., every week

or month. The other more secure way is to never reuse the same pre-key bundle. The X3DH protocol allows each user to optionally upload many *one-time pre-keys* of the form $g^{a_1}, g^{a_2}, \cdots$ that are deleted from the server once fetched by some user. This component (or more precisely $g^{a_i y}$) is then included in the KDF to derive the session key. Since this solution requires more storage for the server, there is a tradeoff between efficiency and security. More details on the specification and security guarantees of the X3DH protocol can be found in [MP16b].

We note that even if the exchanged session keys become compromised, the Signal protocol has an elegant way of *healing* the compromise, achieving PCS in the process. This is achieved by the Double Ratchet protocol explained next.

2.1.3 Solution to (Q2): The Double Ratchet Protocol

The Double Ratchet protocol allows users Alice and Bob to securely send messages by using the session key established via the X3DH protocol. The main innovation of Double Ratchet is to **update the session key for every message** to provide a fine-grained level of both FS and PCS. The Double Ratchet protocol can be broken up into an *local* ratchet protocol and an *interactive* ratchet protocol[8], each ratchet respectively takes care of FS and PCS. While we have discussed in §1.2.3 the informal definitions and motivations behind FS and PCS (the *what* and *why*), we now discuss *how* we can achieve these security notions.

FS and the Local Ratchet

FS guarantees that even if a shared secret key between Alice and Bob is compromised, the key cannot decrypt *past* encrypted messages (see §1.2.3). This is accomplished efficiently by updating the key in an irreversible manner using a KDF. Let us denote the current secret key as ck_1 (short for *chain* key). Alice can use a KDF to create two keys $(\mathsf{ck}_2, \mathsf{mk}_1) \leftarrow \mathsf{KDF}(\mathsf{ck}_1)$ where ck_2 is the updated secret key and mk_1 (short for *message* key) is the secret key used to encrypt message msg_1. Alice can repeat the process by feeding ck_2 into a KDF to obtain ck_3 and the secret key mk_2 to encrypt the following message msg_2. Since inverting the KDF is a hard problem, even a complete compromise of ck_3 does not expose previous $(\mathsf{mk}_i, \mathsf{ck}_i)_{i\in[2]}$, thus achieving FS.

Looking ahead, this is precisely what the local ratchet protocol does. We call it "local" since this protocol is internal to Alice. Namely, Bob with the shared secret key ck_1 can run the same deterministic process without Alice's help to maintain the same chain of secret keys.

PCS and the Interactive Ratchet

PCS guarantees that even if the shared secret key is compromised, the key cannot decrypt *future* encrypted messages (see §1.2.3). Put differently, PCS allows the user to *heal*. As a first attempt to realize PCS, Alice may inject fresh randomness to update the secret key. If the randomness is unknown to the adversary, then the resulting updated secret key is healed. However, this is not yet enough: Alice needs to tell Bob what kind of randomness was added for Bob to be able to maintain the same shared updated key. Clearly, Alice cannot send the randomness in the clear since the adversary can combine it with the compromised old secret key. For the same reason, Alice cannot encrypt the randomness using the old secret key since the adversary can simply decrypt it.

To solve this issue, we notice that while Alice's secret state is compromised, *Bob's isn't*. Alice can

[8] The former and latter are coined as "symmetric-key" ratchet and "Diffie-Hellman" ratchet in the Signal white paper [MP16a]. We choose the term local and interactive to make the distinction between the two protocols clear.

therefore take advantage of this asymmetry.[9] Assume Bob has previously sent $g^{\hat{r}_1}$. Then, Alice, whose secret state is compromised, samples randomness $\hat{r}_2 \leftarrow\$ \mathbb{Z}_p$ and updates the shared key K_1 as $K_2 \leftarrow \mathrm{KDF}(K_2, (g^{\hat{r}_1})^{\hat{r}_2})$. By the hardness of the DDH problem, $g^{\hat{r}_1 \cdot \hat{r}_2}$ looks random to an adversary without knowledge of $\hat{r}_2$, and hence, K_2 remains secure. Moreover, Alice can send $g^{\hat{r}_2}$ to Bob in the clear as in an ordinary DH key exchange protocol, where Bob can compute $g^{\hat{r}_1 \cdot \hat{r}_2}$ as $(g^{\hat{r}_2})^{\hat{r}_1}$ using his knowledge of the secret $\hat{r}_1$. With this, Bob can also derive the same updated secret key K_2. Note that we can swap the roles of Alice and Bob; in this case, Bob samples randomness $\hat{r}_3$ and proceeds as Alice did to achieve PCS and recover from a compromise.

Looking ahead, this is precisely what the interactive ratchet protocol does. We call it "interactive" since this protocol needs both Alice and Bob to participate.

Double Ratchet = Local Ratchet + Interactive Ratchet

With the local and interactive ratchet protocols explained, it remains to glue the two protocols together to obtain the Double Ratchet protocol. A high-level illustration of Double Ratchet is provided in Fig. 8. Below, we provide a step-by-step explanation of the protocol.

Process (1): Alice sends messages $(\mathrm{msg}_{2,i})_i$ to Bob. Here, we assume Alice and Bob already share a session key K_1 and Bob sent $g^{\hat{r}_1}$ in the previous round (see insert on page 20 for the initial round). In (1), Alice first runs the sender part of the interactive ratchet protocol: it samples a fresh randomness $\hat{r}_2 \leftarrow\$ \mathbb{Z}_p$ and computes $g^{\hat{r}_1 \cdot \hat{r}_2}$. She then runs $(K_2, \mathrm{ck}_{2,1}) \leftarrow \mathrm{KDF}(K_1, g^{\hat{r}_1 \cdot \hat{r}_2})$, where K_2 is the *updated* session key and $\mathrm{ck}_{2,1}$ is the chain key. Observe that this invocation of the KDF is how the local and interactive ratchets are combined.

Process (2): Alice runs the local ratchet protocol as in the figure to derive a message key $\mathrm{mk}_{2,i}$ to encrypt message $\mathrm{msg}_{2,i}$. Here, $\mathrm{mk}_{2,i}$ is used only once.

Process (3): Alice sends $g^{\hat{r}_2}$ generated by the interactive ratchet protocol and the encrypted messages $(\mathrm{Enc}_{\mathrm{mk}_{2,i}}(\mathrm{msg}_{2,i}))_i$ to Bob.[10]

Processes (4) and (5): Bob runs the receiver part of the interactive ratchet protocol to derive $g^{\hat{r}_1 \cdot \hat{r}_2}$. Using this and the old shared session key K_1, Bob can derive the secret keys that Alice can. This allows Bob to decrypt all the encrypted messages.

Processes (6): Changing the roles of Alice and Bob, Bob now sends messages $(\mathrm{msg}_{3,i})_i$ to Alice using K_2 and $g^{\hat{r}_2}$. Specifically, Bob replicates the processes (1) to (3).

Finally, as explained earlier, the Double Ratchet protocol inherits FS and PCS from the underlying local and interactive ratchet protocols, respectively.

[9] If Alice and Bob's internal states are compromised simultaneously, then nothing can be done.

[10] To be precise, the encrypted messages also include a message number as an associating data, and the Double Ratchet handles lost or out-of-order messages.

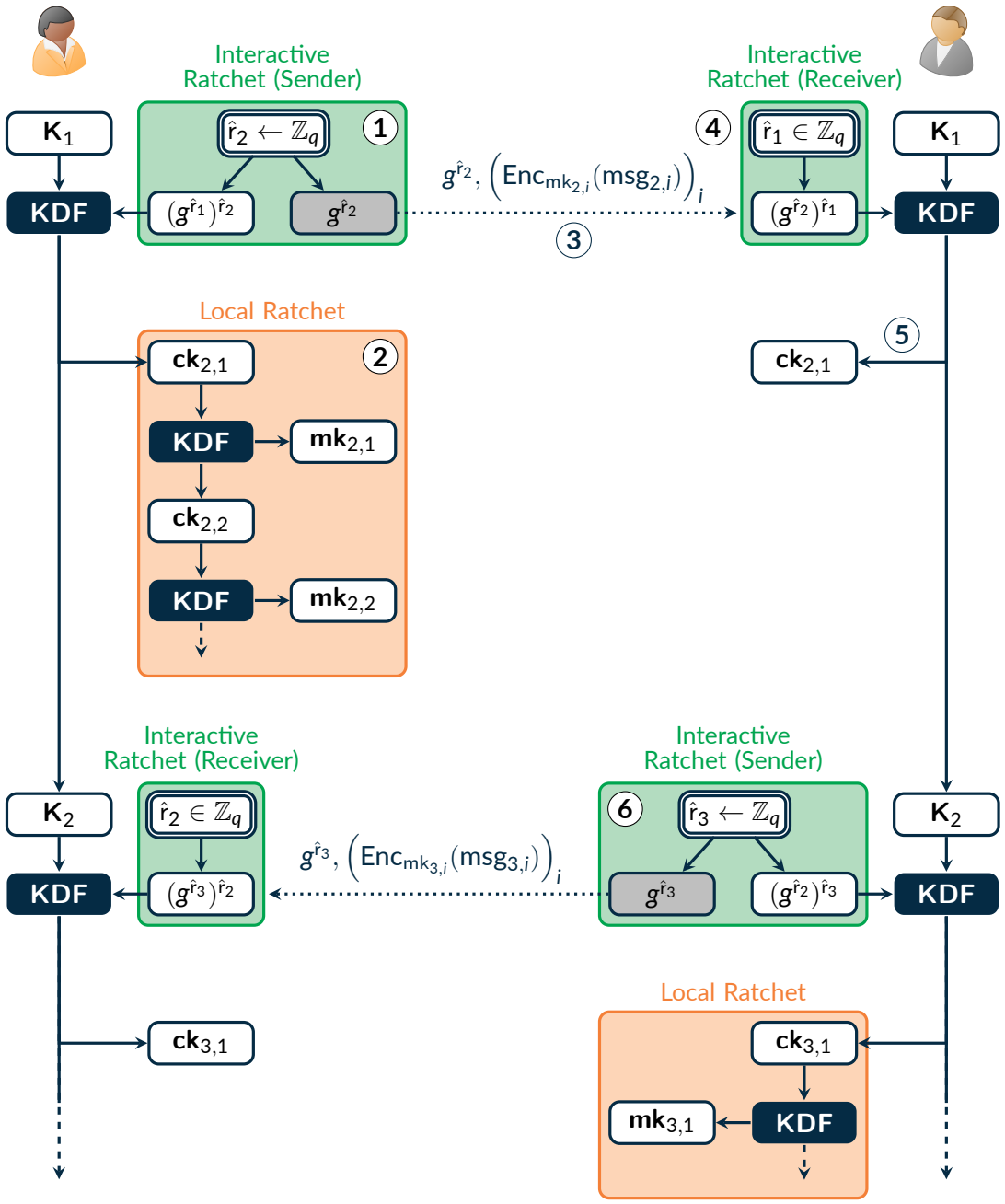


Figure 8: The Double Ratchet protocol. The circled numbers indicate the order of the data flow. Solid arrows indicate operations that are one-way (due to KDF or the DDH/DLOG assumptions); white boxes are secret keys; light gray boxes are public keys; black boxes denote KDF evaluation; double-edged boxes feature that the value is only known to one party, and the dotted arrows indicate the data transmitted between Alice and Bob where $\mathsf{msg}_{j,i}$ is the i-th message sent at epoch j.

Initial Interactive Ratchet

If this is the first round right after the X3DH protocol (i.e., Bob sends messages to Alice in Fig. 7 as part of the response), then there are several options for how $g^{\hat{r}_0}$ is derived. In the deployed version of the X3DH protocol by Signal, Bob uses the signed pre-key g^x included in Alice's pre-key bundle as $g^{\hat{r}_0}$. Alternatively, we can let users to upload several $g^{\hat{r}_0}$ during in the registration phase so that distinct $g^{\hat{r}_0}$ are used by each conversation. While the former provides better efficiency, the later provides stronger security in face of a compromise.

2.2 Making Signal Post-Quantum Secure

The X3DH and Double Ratchet protocols build heavily on the unique commutative property of the DH key exchange protocol. Without a proper intuition of their design rationales, one may even believe that the DH key exchange is an indispensable ingredient to these protocols.

In this section, we show how to mimic the functionality and security guarantee of the X3DH and Double Ratchet protocols while only relying on standard cryptographic tools such as KDFs, digital signatures, and key encapsulation mechanisms (KEMs). A post-quantum secure Signal protocol naturally follows by instantiating these tools in a post-quantum secure manner.

Generic/Post-Quantum Analog of Diffie-Hellman

Before diving into the main content, we take a slight detour and explain the distinctive feature of the DH key exchange. The commutativity of the operations in DH key exchange, namely $(g^a)^b = (g^b)^a$, gives it one of its main strength: *non-interactivity*. By using Alice's public key g^a, Bob can compute a key g^{ab} without explicitly communicating with Alice, and vice versa. This property is unique to DH-based protocols, and for instance, some impossibilities for achieving similar properties from popular post-quantum tools such as lattices and codes are known [GKRS20]. A non-interactive protocol is obtainable from supersingular isogenies [CLM+18, DG21] — another popular post-quantum tool. However, they are either inefficient or require users to prove the well-formedness of the long-term public key. Other works have tried to model the essence of the DH key exchange using a generic cryptographic primitive such as *split* KEMs [BFG+20] but so far, we only know how to construct them from the DH protocol.

In summary, a simple and efficient replacement of the implicit DH key exchange in the Signal protocol by post-quantum primitives is not a viable option. In §2.2.1 and §2.2.2, we explain how to replicate the X3DH and the Double Ratchet protocols without relying on the unique commutative property of the DH key exchange.

2.2.1 Post-Quantum Secure X3DH Protocol

Recall the two steps we took to explain the X3DH protocol in §2.1.2. Once we have a "nice" AKE protocol similar to $\mathsf{AKE}_{\mathsf{X3DH}}$ as explained in Step 1 (see Fig. 6), turning it into an X3DH-like protocol following Step 2 is straightforward. Thus, we only need to focus on constructing a generic AKE protocol with "nice" properties, which we elucidate below.

(P1) *Receiver Obliviousness:* The first value sent by the sender (i.e., Alice in Fig. 6) is independent of the receiver (i.e., Bob in Fig. 6). This was required since on time of registration to the messaging

application, the users may not know who they will be communicating with (see Fig. 7).

(P2) *Compromise Resistance:* The exchanged session key remains secure even if all *non-trivial* combinations of secret information are compromised. Note that if Alice's or Bob's long term and ephemeral secrets (a, x) or (b, y) are compromised, then an adversary can *trivially* compute the exchanged secret. A non-trivial combination points to anything else.

While there have been numerous generic constructions of AKE protocols [FSXY12, FSXY13, KF14, YCL18, XLL+18, HKSU20, XAY+20], including but not limited to post-quantum secure ones, all known constructions satisfying both (P1) and (P2) are limited to DH-based AKE protocols.[11]

Combining Digital Signatures and KEMs to Replicate the X3DH Functionality

PQShield and academic researchers together proposed at PKC 2021 [HKKP21], the first generic AKE protocol satisfying both (P1) and (P2). The AKE protocol only relies on KDFs, digital signatures, and KEMs, all of which we know how to instantiate efficiently in a post-quantum manner. The protocol is described in Fig. 9. KeyGen, Encap, and Decap are the key generation, encapsulation, and decapsulation algorithms of a KEM, and $(\mathsf{ek}, \mathsf{dk})$ is the encapsulation and decapsulation keys. Sign and Verify are signing and verification algorithms of a digital signature scheme, and $(\mathsf{vk}, \mathsf{sk})$ is the verification and signing keys.

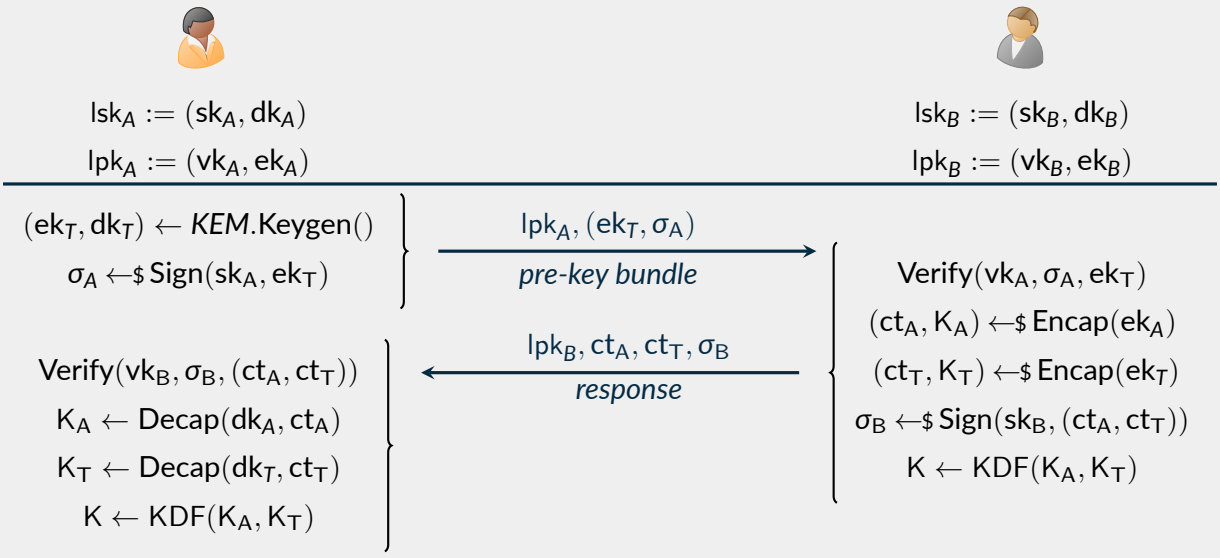


Figure 9: The AKE protocol underlying our generic X3DH protocol.

One can see from Fig. 9 that Alice and Bob share the same session key K, and the content sent by Alice is *independent* of Bob's identity as required by (P1).

Security

We explain how this protocol satisfies (P2). To this end, let us draw a parallel between the exchanged contents in the AKE protocol $\mathsf{AKE}_{\mathsf{X3DH}}$ underlying X3DH (Fig. 6) and our AKE protocol (Fig. 9). This is summarized in Fig. 10.

Classical X3DH. Recall in $\mathsf{AKE}_{\mathsf{X3DH}}$, the partial secret g^{xy} is shared as follows: Alice first sends a freshly sampled g^x; Bob samples y and sets the shared secret as g^{xy}; Bob sends g^y to Alice, and Alice

[11] There are protocols that satisfy the two properties if secure erasure of memory is possible. We seek solutions without making such a hardware assumption.

	$\mathsf{AKE}_{\mathsf{X3DH}}$	Ours	$\mathsf{AKE}_{\mathsf{X3DH}}$	Ours	$\mathsf{AKE}_{\mathsf{X3DH}}$	Ours
Alice (pub. / priv.)	g^x/x	$\mathsf{ek}_\mathsf{T}/\mathsf{dk}_\mathsf{T}$	$g^a/a \in \mathsf{lpk}_\mathsf{A}/\mathsf{lsk}_\mathsf{A}$	$\mathsf{ek}_\mathsf{A}/\mathsf{dk}_\mathsf{A} \in \mathsf{lpk}_\mathsf{A}/\mathsf{lsk}_\mathsf{A}$	g^x/x	n/a
Bob (pub. / priv.)	g^y/y	$\mathsf{ct}_\mathsf{T}/\mathsf{K}_\mathsf{T}$	g^y/y	$\mathsf{ct}_\mathsf{A}/\mathsf{K}_\mathsf{A}$	$g^b/b \in \mathsf{lpk}_\mathsf{B}/\mathsf{lsk}_\mathsf{B}$	n/a
Shared partial secret	g^{xy}	K_T	g^{ay}	K_A	g^{bx}	n/a

Figure 10: Relationship between exchanged values of $\mathsf{AKE}_{\mathsf{X3DH}}$ and our AKE protocol. (pub. / priv.) denotes the public and private values of Alice and Bob. For $\mathsf{AKE}_{\mathsf{X3DH}}$, the shared partial secrets g^{xy}, g^{ay}, g^{bx} are used to derive the shared session key K. n/a indicates that no such analog exists in our AKE protocol. Note that g^x/x and g^y/y may hold different purposes for different shared partial secrets — this is why there is an analog of g^x for g^{xy}, while there isn't for g^{bx}.

derives g^{xy} using x. This is replicated in our AKE protocol by replacing g^x with the freshly sampled encapsulation key ek_T, g^{xy} with the key K_T of the KEM, and g^y with the ciphertext ct_T. Looking at how the components relate to each other in Fig. 10, we see that as long as the sampled secrets (x, y) or dk_T are not compromised, then g^{xy} or K_T remain secure. The same analogy can be made for the partial secret g^{ay} and K_A by relating (a, y) with lsk_A. Specifically, the security properties of g^{xy} and g^{ay} in $\mathsf{AKE}_{\mathsf{X3DH}}$ are replicated by K_T and K_A in our AKE protocol.

The interactivity issue. Unfortunately, a similar analogy for g^{bx} fails. Notably, this is where $\mathsf{AKE}_{\mathsf{X3DH}}$ relies on the unique *non-interactiveness* of the DH key exchange. Even if Alice does not know Bob— as required by (P1) — Alice can prepare g^x ahead of time so that Bob can establish a key g^{bx} with Alice. If we tried to replicate this with KEMs, then Alice would need to know Bob's encapsulation key ek_B at registration, which contradicts (P1).

Our solution. Our AKE protocol solves this issue by asking Bob to sign ct_A and ct_T. Let us explain the high-level intuition. From Alice's perspective in $\mathsf{AKE}_{\mathsf{X3DH}}$, g^{bx} is a term that can only be computed by Bob. Moreover, since the session key K is derived by feeding g^{bx} to the KDF, if the communicating peer uses the same K as Alice, then she is *implicitly* convinced that the peer is Bob. Our AKE protocol performs this implicit check made by Alice *explicitly* by asking Bob to add his signature instead. Intuitively, a user capable of creating a valid signature for Bob's verification key vk_B can only be Bob. This is roughly why the security of g^{bx} in $\mathsf{AKE}_{\mathsf{X3DH}}$ is replicated by σ_B in our AKE protocol. Note that σ_B does not need to be fed to the KDF since as opposed to g^{bx}, σ_B explicitly convinces Alice that the communicating peer is Bob.

Conclusion. By relating the security of g^{xy}, g^{ay}, and g^{bx} to $\mathsf{K}_\mathsf{T}, \mathsf{K}_\mathsf{A}$, and σ_B, respectively, we are able to achieve the same (or in fact slightly stronger) security guarantee as $\mathsf{AKE}_{\mathsf{X3DH}}$. Our AKE protocol is based on KDFs, digital signatures, and KEMs, and can be used as a drop-in replacement of the X3DH protocol. For those interested in the complete security analysis of our AKE protocol and a more detailed comparison with the X3DH protocol, we refer to [HKKP21].

Deniability of Establishing Communication

One subtle difference between $\mathsf{AKE}_{\mathsf{X3DH}}$ and our AKE protocol is that the signature σ_B leaves evidence that Bob tried to engage in a conversation with Alice.[a] If necessary, there are several ways to add deniability to such a fact. At the application layer, Bob can periodically update his signing key and publish his old signing key sk_B. Once sk_B is public, then anybody could have generated σ_B, so Bob can no longer be held accountable.

At the protocol layer, Bob can derive a one-time-pad key $\mathsf{K}_{\mathsf{OPT}}$ as $(\mathsf{K}, \mathsf{K}_{\mathsf{OPT}}) \leftarrow \mathsf{KDF}(\mathsf{K}_A, \mathsf{K}_T)$ and send an encryption $\mathsf{ct}_{\mathsf{OPT}} = \mathsf{K}_{\mathsf{OPT}} \oplus \sigma_B$ of σ_B. Alice then verifies σ_B by first decrypting the ciphertext. We can further strengthen the type of achieved deniability by relying on anonymity enhancing tools such as *ring* signatures with relatively low overhead. For those interested, more details are provided in, for example, [MP16b, HKKP21, BFG+22].

[a] Note this is different from Bob being held accountable for his sent encrypted messages. Recalling that the messages are sent using symmetric key encryption in Double Ratchet, Alice sharing the same session key as Bob could have also generated the encrypted message.

2.2.2 Post-Quantum Secure Double Ratchet Protocol

Compared with the X3DH protocol, Double Ratchet was constructed more modularly. As explained in §2.1.3, it consists of local and interactive ratchet protocols, combined together by a KDF. Moreover, the local ratchet protocol only consists of running another KDF in a sequential manner. Since we have efficient KDFs using only symmetric cryptography, a post-quantum instantiation of the local ratchet protocol is simple. Therefore, **the only component of Double Ratchet that is not readily post-quantum is the interactive ratchet, due to its heavy reliance on the DH key exchange**.

Using KEMs to Replicate the Interactive Ratchet

The interactive ratchet is responsible for achieving PCS (see §2.1.3). A DH key exchange is performed in a ping-pong fashion between Alice and Bob to update their internal states sequentially. This is illustrated in processes (1), (3), and (4) in Fig. 8.

After going through the generic X3DH protocol in the previous section, it is easy to see that such a DH key exchange can be replicated by a KEM. Unlike X3DH, Double Ratchet does not use specific properties of the DH key exchange (such as its non-interactivity), and **replacing DH with a KEM can be done straightforwardly**. Specifically, we refer the readers to the second and third columns of Fig. 10 to recall the relationship between the DH key exchange and KEM. With this knowledge, we can instantiate the interactive ratchet protocol using KEMs as in Fig. 11.

Recall that to update the shared secret key K_1, Alice must update it using freshly sampled randomness and also needs to send this information over to Bob. This is accomplished in process (1) by first running $(K_1', ct_1) \leftarrow\$ \mathsf{Encap}(ek_1)$, where ek_1 is the encapsulation key Bob sent in the prior round (see the insert on page 20 for the initial round). Then, Alice updates K_1 by $(K_2, ck_{2,1}) \leftarrow\$ \mathsf{KDF}(K_1, K_1')$. Alice finally sends to Bob the ciphertext ct_1 along with a newly generated encapsulation key ek_2 for Bob to use in the next round. In the next round, Bob first decrypts ct_1 as in process (4) and retrieves the same updated session key K_2 and chained secret key $ck_{2,1}$ as Alice. Bob then switches roles with Alice and runs process (6), where he further updates the session key using Alice's ek_2.

Security

The above abstraction of the interactive ratchet protocol first appeared at EUROCRYPT 2019 [ACD19]. Alwen et al. provided the first complete security analysis of the Double Ratchet protocol in a modular fashion. As expected, the KEM-based interactive ratchet protocol achieves the same (or even slightly stronger) security as the DH-based interactive ratchet protocol. We invite readers interested in the complete security analysis of the generic Double Ratchet protocol to read the full article [ACD19].

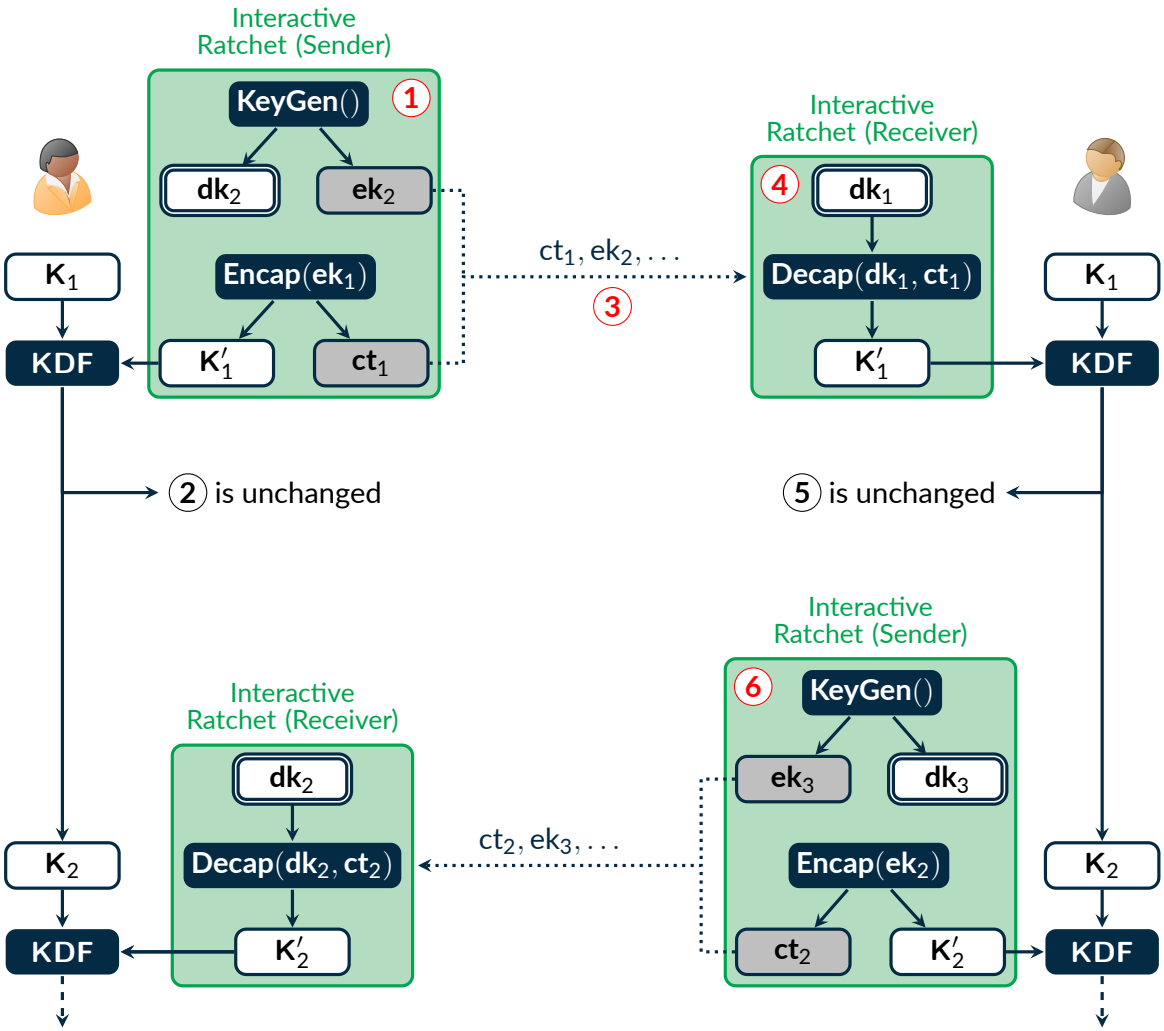


Figure 11: Replacing the DH-based interactive ratchet protocol by a KEM-based interactive ratchet protocol. The processes labeled by red circled numbers are the modifications and those labeled by black circled numbers remain identical to Fig. 8. The items related to the local ratchet protocol are identical to Fig. 8 and we thus omit them for simplicity.

3 Scalable Group Messaging Protocols

This final section focuses on secure *group* messaging protocols, which handle the security of group conversations.

In §3.1, we highlight the challenges that secure messaging protocols (SMPs) face in the group setting, in particular how security and bandwidth efficiency can have an adverse relationship. This will motivate the search for group SMPs with advantageous trade-offs in security and bandwidth consumption.

In §3.2, we present four prominent group SMPs, each exploring a different corner of the design space of group SMPs. After comparing the advantage and disadvantage of these protocols, we present the technical details of each protocol: Sender Keys (used in WhatsApp, §3.3), Pairwise Channels (used in Signal, §3.4), TreeKEM (used in MLS, §3.5) and Chained CmPKE (proposed by PQShield, §3.6).

3.1 Introduction

In §2, we demonstrated the feasibility and practicality of (post-quantum) secure messaging in the two-party setting. While constructing a (post-quantum) group secure messaging is feasible based on a two-party secure messaging, the resulting protocol will quickly become impractical as the number of group members grows.

To provide some reference on the size of a group, secure group messaging applications such as Signal, WhatsApp, and Line support group members ranging from 50 to 1,000. The almost finalized IETF draft standard for secure messaging, Message Layer Security (MLS) [OBR+21, BBM+22], aims to scale to groups as large as 50,000 members, typically including many users using multiple devices. Note that while Telegram supports up to 200,000 group members — an order of magnitude larger than other applications — the group conversations are not E2E encrypted (§1.2.2, Fig. 3) and can be read by the server.

The present section studies a *scalable* solution to group secure messaging. As we will see, scaling from two parties to a large number of parties raises questions not only in terms of scalability but also of security, and creates an interesting trade-off.

Scalability: Bandwidth Consumption as a Key Parameter in User Experience

To see why scalability becomes a critical issue for secure group messaging, let us discuss the bandwidth consumption of a mobile device. Using data from [Cab21], Fig. 12 illustrates the cost of mobile data around the world in 2021.

In many cases, mobile data is a scarce and expensive resource. According to [Cab21], the median cost of mobile data is on average $4.07/GB, mobile plans in 89 countries cost more than $20.00/GB, and for expensive countries "*[p]eople are often buying data packages of just a tens of megabytes at a time* [sic]." Reaching a data cap can have drastic consequences, such as blocking end-user access to

the mobile network or charging them a premium rate.

Secure messaging protocols can have a high toll on bandwidth consumption, especially in the post-quantum setting. Assuming a user sends and receives a total of 100 messages per day, the *two-party* Signal protocol would add an overhead of $32 \cdot 100 = 3200$ bytes, corresponding to the interactive ratchet (see Fig. 8), which provides PCS. In its post-quantum variation (Fig. 11), it would add about 42 kilobytes with the isogeny-based scheme SIKE [JAC+20], 153 kilobytes with the lattice-based scheme Kyber [SAB+20], and 25 megabytes with Classic McEliece [ABC+20], which demonstrates that the scheme can significantly impact bandwidth efficiency. In the group setting, this overhead may be further increased by the group size N, ranging anywhere from a dozen to tens of thousands.

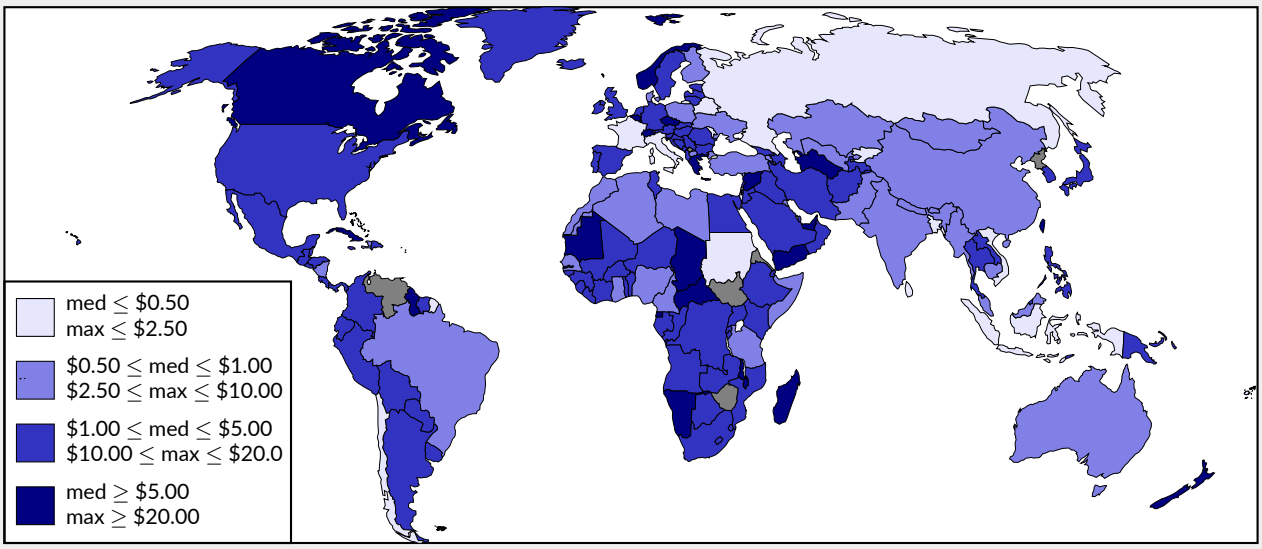


Figure 12: Prices of 1 GB of mobile data in the world in 2021. Methodology and detailed data are available at [Cab21]. med and max respectively stand for the median and maximum prices reported. A grayed out country means that the data is either missing or unreliable.

On the other hand, we expect time or latency considerations to have a minimal impact on the experience of end users, at least for Chained CmPKE. Let us analyze Table 3. The time bottleneck is the $O(N)$ cost to send a healing message. For all (post-quantum) instantiations of Chained CmPKE proposed in [HKP+21], uploading a healing message costs less than 50KiB for groups of at most 1024 users, which, even in countries with low uploading speed[12], is done in less than 0.2 seconds. In addition, we note that healing is a transparent operation for end users, as it is performed by their devices as a background task. As far as we know, similar computations are yet to be done for the other SMPs.

Security: Larger Groups Means Higher Risk of Compromise

As the number of group members grows, so does the risk of device compromise. More than in the two-party setting, this motivates the need for strong mitigations against user compromises in the group setting. Consider a secure group conversation in a group of N members. As a rough estimate, if each member has an (independent) probability ε of having their device compromised during a time

[12] As of July 2021, the slowest in Afghanistan, with 2.90 Mbps [Spe21].

unit, then the probability of compromise of at least one member grows as $Nt\varepsilon$ in time t. When $N = 100$, a device compromise is 50 times more likely than in a two-party conversation.

As discussed in §1.2.3, *disappearing messages* or the removal of compromised users at the application level can guarantee some form of security. However, to achieve *cryptographic* security (see FS and PCS in §1.2.3, Fig. 4), we need to generalize secure messaging to the group setting at the protocol level. Specifically, the Double Ratchet protocol (§2.1.3, §2.2.2) used in the two-party setting needs to be generalized to the group setting. Since achieving FS is relatively straightforward, this section focuses on how to efficiently generalize the *interactive ratchet*, which provides PCS, to the group setting. Indeed, the interactive ratchet is the primary source of bandwidth consumption of Double Ratchet, and constructing an efficient interactive group ratchet will be the key to a scalable group messaging protocol.

A Plethora of Design Choices

In the two-party setting (§2), the efficiency difference between any reasonable SMPs was small, so it made sense to only consider the most secure SMP (i.e., Signal). This is no longer the case in the group setting; **with more members in the group, simultaneously maintaining efficiency and a strong level of security becomes a non-trivial task.** Two types of group SMPs may achieve a different level of security while having vastly different efficiency — in such a case they are incomparable.

Ultimately, **the "best" group SMP depends on the goal of the application.** If the target is the highest level of FS and PCS security, then Pairwise Channels by Signal (§3.4) is a perfect candidate. On the other hand, if the target is to have better efficiency than Signal or be tailored for post-quantum security, then Chained CmPKE by PQShield (§3.6) is a good candidate. This tension between efficiency and security will be at the heart of our technical discussions.

3.2 Comparing Representative Group SMPs

We cover four representative group SMPs: **Sender Keys** (used in WhatsApp), **Pairwise Channels** (used in Signal), **TreeKEM** (used in MLS), and **Chained CmPKE** (proposed by PQShield). Each of them comes with distinctive characteristics. This section provides an overview of the efficiency and security provided by these group SMPs and summarizes their pros and cons. We refer the interested readers to Sections 3.3 to 3.6 for the technical details of each protocol.

3.2.1 Core Metrics to Evaluate Group SMPs

We provide a summary of the four protocols in Table 3. The table focuses on some of the core metrics used to evaluate group SMPs. Other possible metrics like metadata hiding and a decentralized server are outside the scope of this white paper (see §3.2.3 for a brief discussion).

- **Message cost.** This is the communication cost for a group member to send a single message to the group. Concretely, Pairwise Channels require $(N-1)$ symmetric ciphertexts; Sender Keys require a single symmetric ciphertext and signature; TreeKEM and Chained CmPKE require a single symmetric ciphertext and optionally a single signature.
- **Healing (and removal) cost.** This is the cost for a compromised group member to become healed. In the group setting, the group also becomes compromised when some member is *removed* from the group (see insert on Page 30). The (upload) column is the amount of data

Table 3: Comparison between representative group SMPs. N is the number of group members.

Scheme	Section	Message cost	Healing cost (upload)	Healing cost (download)	Healing cost (total)	FS	PCS	Type of shared Keys
Sender Keys (WhatsApp)	§3.3	$O(1)$	$O(N)$	$O(1)$	$O(N)$	✔✔	✔	per group member
Pairwise Channels (Signal)	§3.4	$O(N)$	$O(N)$	$O(1)$	$O(N)$	✔✔	✔✔	per group member-pair
TreeKEM (MLS)	§3.5	$O(1)$	$O(\log N)^{\dagger}$	$O(\log N)^{\dagger}$	$O(N \log N)^{\dagger}$	✔✔	✔✔	per group
Chained CmPKE (by PQShield)	§3.6	$O(1)$	$O(N)$	$O(1)$	$O(N)$	✔✔	✔✔	per group

† : This is the *optimal* healing cost achieved when the tree is balanced and no blank nodes exist (see §3.5 for more detail). In the worst case, they become $O(N)$, $O(N)$, and $O(N^2)$, respectively.

uploaded by a healing member, and the (download) column is the amount of data downloaded by each of the remaining group members in case *a single* member performs a heal. The (total) column is the total bandwidth consumption of a single member when every member performs a heal, defined as $1 \times$ (upload cost) $+ (N-1) \times$ (download cost).

- **FS and PCS.** Setting aside subtle differences, all four group SMPs achieve the same high level of FS denoted as (✔✔). In contrast, there is a distinction between the level of PCS achieved by Sender Keys and others. (✔) denotes that PCS is achieved only if *all* the group members heal; put differently, all group members become indirectly compromised when a single member gets compromised. (✔✔) denotes the best-case security where PCS is achieved once all the compromised group members heal.
- **Type of Shared Keys.** A shared key is typically viewed as a symmetric key[13] used to exchange messages securely. There are several types of shared keys we can consider:
  - *per group member*. Each group member x creates a symmetric key sk_x and shares it to all the other members. Each member therefore stores N symmetric keys in total. sk_x is used by member x to send a message.
  - *per group member-pair*. Each pair of group members x and y share a different symmetric key $\mathsf{sk}_{x,y}$. Each member therefore stores $(N-1)$ symmetric keys in total. $\mathsf{sk}_{x,y}$ is used by members x and y to exchange messages.
  - *per group*. All group members share a single symmetric key sk. All messages are exchanged using sk.

[13] The symmetric key can be a key for a symmetric encryption scheme or authenticated encryption scheme with additional data (AEAD). This distinction is unimportant for our explanation.

Removing Group Members Requires Healing

Healing is required whenever a group member becomes "compromised". One obvious compromise occurs when an *external* adversary compromises a group member, for example by hacking their phone — this is the only type of compromise needing consideration in the two-party setting. Interestingly, in the group setting, when a member is removed from a group, the resulting group is regarded to be compromised by viewing the removed member as an *internal* adversary. This is because the removed member can continue reading the encrypted conversation if the shared key is not updated. To make matters worse, the member may share the key with the server or state actors to broaden the scope of the compromise while giving the other members no way to detect such rouge activity.

3.2.2 Asymptotic vs Concrete Costs

To choose the right group SMP for an intended application, it is crucial to understand their concrete efficiency cost. The cost of sending a message is easy to compare (see the "Message cost" column of Table 3). Pairwise Channels requires $(N-1)$ symmetric ciphertexts, and other protocols require a single symmetric ciphertext and possibly a single signature. If the application handles a large group that frequently exchanges large messages, then a group SMP with an asymptotic $O(1)$ messaging cost could be ideal.

The healing costs are not as clear since the terms hidden in the big-O notations differ drastically between protocols. Table 4 summarizes the concrete healing costs of each group SMP. The first rows of Sender Keys and Pairwise Channels denote the healing upload and download costs of the currently deployed non-post-quantum secure versions. The second rows of Sender Keys and Pairwise Channels, and TreeKEM and Chained CmPKE assess the healing upload and download costs based on general cryptographic primitives, all of which can be instantiated efficiently in a post-quantum secure manner.

Notice that even though Sender Keys (PQ), Pairwise Channels (PQ), and Chained CmPKE all have the same asymptotic $O(N)$ healing upload costs, the concrete costs differ significantly. While Chained CmPKE only requires uploading a *single* KEM encapsulation key and key-independent part of an (m)KEM ciphertext, the other two protocols require $(N-1)$ of them. This efficiency gain comes from the use of *multi-recipient* KEMs (mKEMs) explored by PQShield and academic researchers in [KKPP20,HKP+21]. Plugging in concrete post-quantum KEM protocols satisfying $|\hat{ct}_0| \gg |ct_i|$, the healing upload cost of Chained CmPKE can be an order of magnitude smaller than Sender Keys (PQ) and Pairwise Channels (PQ). Interestingly, even if TreeKEM has an asymptotically smaller healing upload cost, the concrete upload cost of Chained CmPKE can be smaller for $N \lesssim 256$ [HKP+21, Figs. 6 and 7] since $|\hat{ct}_0| \gg |ct_i|$. This illustrates the importance of comparing the concrete efficiency of the protocols.

Table 4: Concrete healing cost of representative group SMPs. The terms ek, $\hat{ct}_0$, ct_i, sig, and sct stand for encapsulation key, key-independent part of a (m)KEM ciphertext, key-dependent part of a (m)KEM ciphertext, signature, and symmetric ciphertext, respectively. See §3.6 for more details on mKEMs. Informally, a standard KEM ciphertext can be decomposed as $ct = (\hat{ct}_0, ct_i)$, where $\hat{ct}_0$ (resp. ct_i) is *independent* (resp. *dependent*) of the member's encapsulation key.

Scheme	Upload					Download				
	ek	$\hat{ct}_0$	ct_i	sig	sct	ek	$\hat{ct}_0$	ct_i	sig	sct
Sender keys	$(N-1)^*$				$(N-1)$	1^*				1
Sender keys (PQ)	$(N-1)$	$(N-1)$	$(N-1)$		$(N-1)$	1	1	1		1
Pairwise channels	$(N-1)^*$					1^*				
Pairwise channels (PQ)	$(N-1)$	$(N-1)$	$(N-1)$			1	1	1		
TreeKEM	$\log_2(N)$	$\log_2(N)^\dagger$	$\log_2(N)^\dagger$	1		$\log_2(N)$	$\log_2(N)^\dagger$	$\log_2(N)^\dagger$	1	
Chained CmPKE	1	1	$(N-1)$	1		1	1	1	1	

$*$: More precisely, these are group elements (for example, points on elliptic curves).
$\dagger$: This is the *optimal* healing cost for uploads and downloads achieved when the tree is balanced and no blank nodes exist (see §3.5 for more detail). In the worst case, they become $O(N)$.

3.2.3 Summary

We provide a summary of the four representative group SMPs to aid the selection of the "best" protocol meeting the objective of an application.

Efficiency

In the classical setting, Sender Keys, TreeKEM, and Chained CmPKE all provide a similar level of efficiency and outperform Pairwise Channels. In contrast, **in the post-quantum setting, Chained CmPKE by PQShield outperforms all other group SMPs in many common scenarios.** We provide more details below.

Message Cost. Pairwise Channels require $(N-1)$ symmetric ciphertexts, and the other protocols require a single symmetric ciphertext and (optionally) a single signature (see Table 3). In the classical setting, a signature of XEdDSA [Per16] is only roughly 32 bytes, and Pairwise Channels has a noticeably larger message cost compared with the other protocols — especially when the size of the group or the exchanged message content is large. However, in the post-quantum setting, a signature could be an order of magnitude larger, and Pairwise Channels could offer a lower message cost for a small-sized group. The threshold depends on the concrete choice of the signature scheme and the size of the exchanged message.

Healing Cost. In the classical setting, all the protocols have roughly identical total healing costs in both concrete and asymptotic terms. When there is a specific demand to minimize the upload healing cost, TreeKEM offers the lowest cost (in the optimal case). In the post-quantum setting, Chained CmPKE provides the best total and download healing cost for all group sizes, which is made possible by the efficient mKEM studied in [KKPP20, HKP+21]. Moreover, for a medium-sized group,

i.e., $N \lesssim 256$, the upload healing cost of Chained CmPKE outperforms TreeKEM. More details are provided in [HKP+21].

Security

While the currently deployed versions of Sender Keys (WhatsApp) and Pairwise Channels (Signal) do not have post-quantum security (PQS), they can both be upgraded to have PQS as shown in Sections 3.3 and 3.4. Conditioning on all four group SMPs being post-quantum secure, **Pairwise Channels offers the strongest security** and Sender Keys offers the weakest security. **TreeKEM and Chained CmPKE offer trade-offs between efficiency and security** and can achieve the same security level as Pairwise Channels when opting for security.

Concretely, all group SMPs achieve the same level of FS. However, Sender Keys achieves a slightly lower level of PCS than the three other group SMPs (see Table 3), since the compromise of a single member requires all group members to heal for the group to be secure again. In a large group, this may leave the group insecure for an extended period depending on the healing schedule.

Signal's implementation of Pairwise Channels mandates that a heal is performed whenever a group speaker changes and offers the highest level of PCS by default. In contrast, the healing schedule of TreeKEM and Chained CmPKE is application dependent — if we opt for security, these two protocols become as secure as Pairwise Channels. However, we can trade security for better efficiency.

3.2.4 Roadmap of the Remaining Sections

The remaining sections explain the technical detail of the four group SMPs: Sender Keys (WhatsApp, §3.3), Pairwise Channels (Signal, §3.4), TreeKEM (MLS, §3.5), and Chained CmPKE (by PQShield, §3.6). Each section provides a summary of the protocol, and answers (a) how the group is initialized, (b) how group members send messages with the shared keys, and (c) how groups can recover from the compromise or removal of a member.

3.3 Sender Keys by WhatsApp

Summary

WhatsApp's *Sender Keys* [Wha21] runs on top of the two-user Signal protocol. The strength of this design lies in its **simplicity and smaller messaging cost** compared to Signal's Pairwise Channels. The messaging cost is almost identical to those of TreeKEM (§3.5) and Chained CmPKE (§3.6).

The downside is that it provides **noticeably weaker security guarantees amid a compromise**. Unlike in all other protocols, when a single member gets compromised, then all group members — rather than only the compromised one — must heal to make the group secure again. Since group healing is also required whenever a member is removed (see §3.2.1), the healing cost may become an efficiency and bandwidth bottleneck in a large active group.[14]

[14] WhatsApp seems to restrict at the application layer the ability for a group member to actively heal (see [Wha21, Section "Group Messages"]). Specifically, a member cannot heal unless somebody is removed from the group. Therefore, although Sender Keys has the capability of healing after a compromise, WhatsApp itself may not. In this document, we evaluate what can be achieved by Sender Keys.

Initialization

We assume a secure pairwise channel is established between all group members through the two-party Signal protocol (see Fig. 13b) described in §2. In the initialization phase, each group member x creates a symmetric encryption key sk_x called the *sender key* and sends it to all the other $(N-1)$ members through the pairwise channel. At the end of the initializing phase, every member has N sender keys in storage — one for each group member.

Sending Messages

To send a message msg to the group, member x does, in this order, (i) encrypt msg using sk_x, (ii) sign the symmetric ciphertext, and (iii) send the (ciphertext, signature) pair to the server (see Fig. 13a). The server then fans out this to all the group members. Each receiving member checks the validity of the signature and then decrypts the ciphertext using the shared sk_x. Since the server fans out the same content to all the members, a message can be sent to the entire group at the cost of one signature and one symmetric ciphertext. FS is achieved by "ratcheting" the sender key sk_x after encrypting a message analogous to the local ratchet protocol explained in §2.1.3.

Healing (and Removing Group Members)

Each group member holds the sender key sk of *all* other members. Therefore, compromising a *single* group member exposes *all* sender keys. To recover from such a compromise, all members must clear their sender keys and restart the initialization phase. Since each group member needs to send their new sender key through $N-1$ pairwise channels — each one implementing the Double Ratchet protocol — the healing upload cost (per member) is $O(N)$ public key values. While the healing download cost from a single member is $O(1)$, the total download cost (per member) becomes $O(N)$ since every member needs to heal despite not being explicitly compromised.

With the original Double Ratchet protocol (§2.1.3), each public key value is a group element g^r, which is 32 bytes when using Curve25519 [MP16a]. With the post-quantum variant (§2.2.2), each public key value is an encapsulation key and a ciphertext of a KEM, the sum of which is at least 433 bytes (see [PQS21] for detailed comparisons).

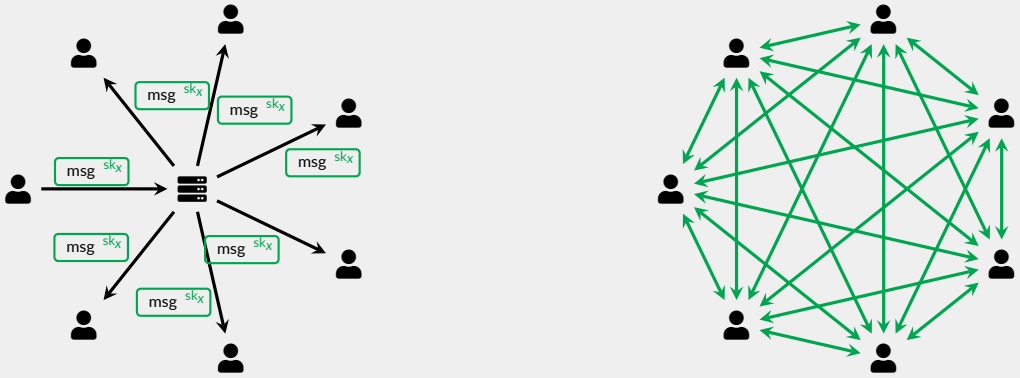
3.4 Pairwise Channels by Signal

Summary

Pairwise Channels by Signal naturally extend the two-user Signal protocol (§2) to the group setting, by running the two-party Signal protocol between all possible pairs of group members (see Fig. 13b). The strength of Pairwise Channels lies in its **strong security guarantees**. Concretely, the protocol mandates each member to heal whenever they speak up — this is in contrast to other protocols that allow members to choose the healing schedule, resulting in a trade-off between security and efficiency. The downside is that it has the **highest messaging cost among all the protocols**. Since group members send messages independently to each other via a pairwise channel, the message cost is $O(N)$ rather than $O(1)$. The message cost may become a bandwidth bottleneck for large groups or when large messages are frequently exchanged.

Initialization

As in Sender Keys (§3.3), we assume a secure pairwise channel is established between each pair of group members through the two-party Signal protocol. Recalling the Double Ratchet protocol (§2.1.3), this means each pair of group members x and y share a symmetric key $sk_{x,y}$. At the end of



(a) Sender Keys. Member x initially shares a symmetric key k_x with the other group members using Pairwise Channels (see Fig. 13a). Any message sent by x is encrypted under sk_x and subsequently signed by x's signing key, costing the sender $O(1)$ in bandwidth and computation.

(b) Pairwise Channels. A separate E2EE channel is created for each pair (i, j), hence sending an encrypted message in a group of N members entails a cost $O(N)$ for the sender.

Figure 13: Group messaging protocols *Sender Keys* (WhatsApp) and *Pairwise Channels* (Signal).

the initializing phase, every member stores $(N-1)$ symmetric keys, one for each pairwise channel.

Sending Messages

To send a message msg to the group, member x encrypts msg to all members separately by independently executing the Double Ratchet protocol. Since the same message is encrypted under $(N-1)$ different symmetric secret keys, the messaging cost is $(N-1)$ symmetric ciphertexts.

Healing (and Removing Group Members)

Since the Double Ratchet protocol (in particular the interactive ratchet) is executed every time a sender changes, a group member is healed by default whenever it talks after receiving a message. Moreover, by the property of the Double Ratchet protocol, when the compromised member heals, so does the entire group. This is in sharp contrast to Sender Keys where all the members, including the non-compromised members, have to heal for the entire group to become secure.

Each group member runs $(N-1)$ independent two-party Signal protocols, so the healing upload cost is $O(N)$, while the healing download cost is $O(1)$. The concrete cost of the Double Ratchet protocol is identical to those explained for Sender Keys (see §3.3). Since healing is performed whenever the speaker changes, the healing cost can become quite large for an active group. For instance, even in a short period if all the N group members speak up, then the total healing download cost becomes $O(N)$.

3.5 TreeKEM by MLS

Summary

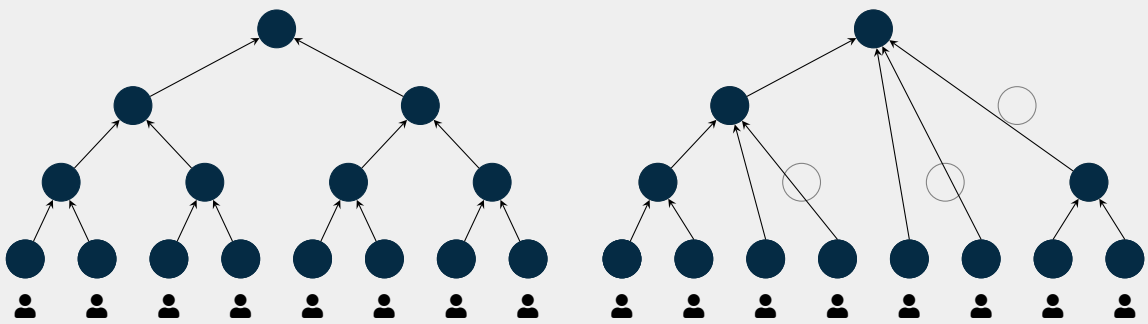
MLS [OBR+21, BBM+22] is an IETF draft standard for secure messaging aiming to make the healing cost (i.e., PCS) more scalable compared to Sender Keys and Pairwise Channels. As with other SMPs, MLS decomposes into modular components, where TreeKEM is the core novel protocol underlying MLS responsible for healing the group.

The healing *upload* cost of TreeKEM can be as small as $O(\log N)$. TreeKEM is the only protocol we cover that achieves a sub-linear healing upload cost. Moreover, unlike Signal, MLS offers a granular level of FS and PCS. While MLS can achieve the same level of security as Signal if members heal as frequently as in Signal, the members can trade security for efficiency by healing less frequently.

On the other hand, **the healing *download* cost is at least** $O(\log N)$ compared to the $O(1)$ achieved by the other protocols we cover. In some scenarios, the high download cost can outweigh the benefit of a low upload cost. For example, when all N users heal as frequently as Signal, the total download cost can grow as large as $O(N \log N)$, which is more than $O(N)$ achieved by the other protocols. Note that the healing download and upload costs can degrade from $O(\log N)$ to $O(N)$ when members are frequently removed, or some members do not heal frequently enough.

The Ratchet tree and the TreeKEM invariants

The core principle of TreeKEM is to arrange group members at the leaves of a so-called *ratchet tree* which we denote as T (see Fig. 14). We recall that the root of a tree is its topmost node. The path of a node x is the sequence of nodes connecting x to the root (including x and the root). For a visual illustration, consider any node in Fig. 14: its path can be visualized by starting from this node and following the arrows ($\longrightarrow$) until reaching the root.



(a) A complete tree **(b)** A tree with some blank nodes

Figure 14: Two configurations of a ratchet tree for a group of $N = 8$ members. Terminology:
1. Solid nodes (●) contain an encryption keypair (ek, s), blank nodes (○) are empty.
2. (a ⟶ b) means any user knowing the secret key of a also knows the secret key of b.

In nominal conditions, T is complete (Fig. 14a). Certain events, for example removing users, might cause some nodes of the ratchet tree to be blank (Fig. 14b). To each solid (i.e., non-blank) node i of the tree is associated an asymmetric encryption keypair (ek_i, s_i).[15] Blank nodes are empty. A critical part of TreeKEM is that the so-called *TreeKEM invariant* shall be satisfied at all times:

(T1) All group members know the encryption key ek_i of all solid nodes i.

(T2) An entity knows the decryption key of a solid node i of T if and only if: (a) this entity is a member of the group associated with T, and (b) i is in the path of the leaf node of this member.

Since the root is in the path of all nodes, (T2) implies that an entity knows the decryption key of the root if and only if this entity is a group member. By passing this key in a KDF, group members can obtain a shared symmetric key, which allows them to securely exchange encrypted messages.

[15] More precisely, s_i is the randomness used to generate an asymmetric encryption keypair $(\mathsf{ek}_i, \mathsf{dk}_i)$. For simplicity, we view s_i as the decryption key.

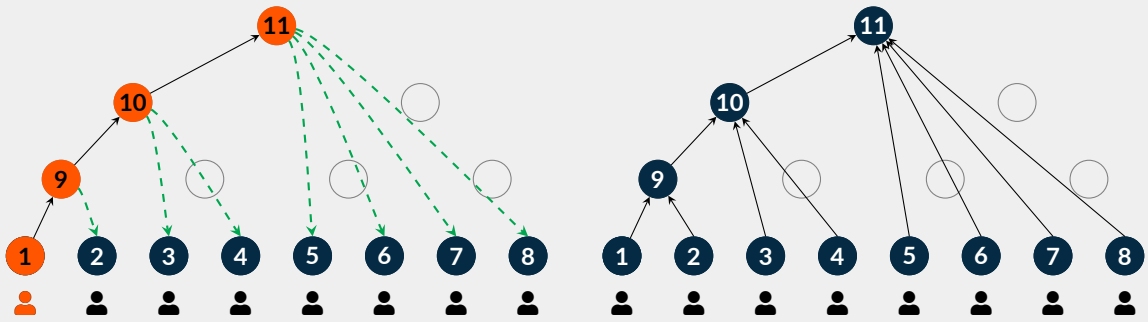
An additional, but less important requirement, is to minimize the number of blank nodes in the ratchet tree. The reason for that requirement is that the fewer blank nodes there are in the tree, the more efficient the protocol will be. In the rest of this section, we explain how TreeKEM performs common group operations (initializing a group, adding or removing members, etc.) while preserving the TreeKEM invariant and minimizing the number of blank nodes.

Initialization

Unlike Sender Keys or Pairwise Channels, one group member — Member 1 (👤) in Fig. 15a — is responsible for initializing the group. At the initialization, this member assigns each member to a distinct leaf of the ratchet tree, but all nodes are initially blank except for the leaves.

We recall that the goal of all members is to preserve the TreeKEM invariants (T1) and (T2) while minimizing the number of blank nodes. Member 1 can update (and therefore unblank) all nodes in his path, and these are the only ones he can update since (T2) would otherwise be invalidated. Therefore Member 1 generates new encryption keypairs for nodes $1, 9, 10, 11$.

At this point, we mention a nice optimization used by TreeKEM. Whenever the decryption key s_i of a node i is updated, the decryption key s_j of its parent j is obtained by passing s_i into a PRG: $s_j := \mathrm{PRG}(s_i)$. This implies that as soon as a group member knows the decryption key s_i of a freshly updated node i, he knows the decryption keys of all nodes that are ancestors of i. In fine, it makes it easier to enforce the invariant (T2).



(a) Initialization by Member 1 (👤) (b) Shared tree after initialization

Figure 15: Initialization of a TreeKEM tree for a group of $N = 8$ members. Additional terminology:

1. The number (x) indicates the order in which keypairs were generated.
2. Orange nodes (●) indicate nodes being updated by the active user (👤).
3. ((x) - - -> (y)) indicates that the secret key of (x) is encrypted using the public key of (y).

Let us go back to the initialization of the group, and break down what Member 1 needs to broadcast.

- In order to preserve the invariant (T1), Member 1 broadcasts all newly generated encryption keys ek_i (● in Fig. 15a).
- Preserving (T2) is more involved: Member 1 broadcasts in encrypted form all the newly generated decryption keys s_i (- - -> in Fig. 15a). Each s_i is being encrypted only to Members j that have i as an ancestor, for example s_9 is being encrypted only to Member 2. The PRG optimization described above reduce the necessary number of ciphertexts; for example, since $s_{10} = \mathrm{PRG}(s_9)$ and $s_{11} = \mathrm{PRG}(s_{10})$, Member 2 can infer s_{10} and s_{11} from the knowledge of s_9.

In Fig. 15b, decryption keys s_i's are encrypted directly under encryption keys of individual group

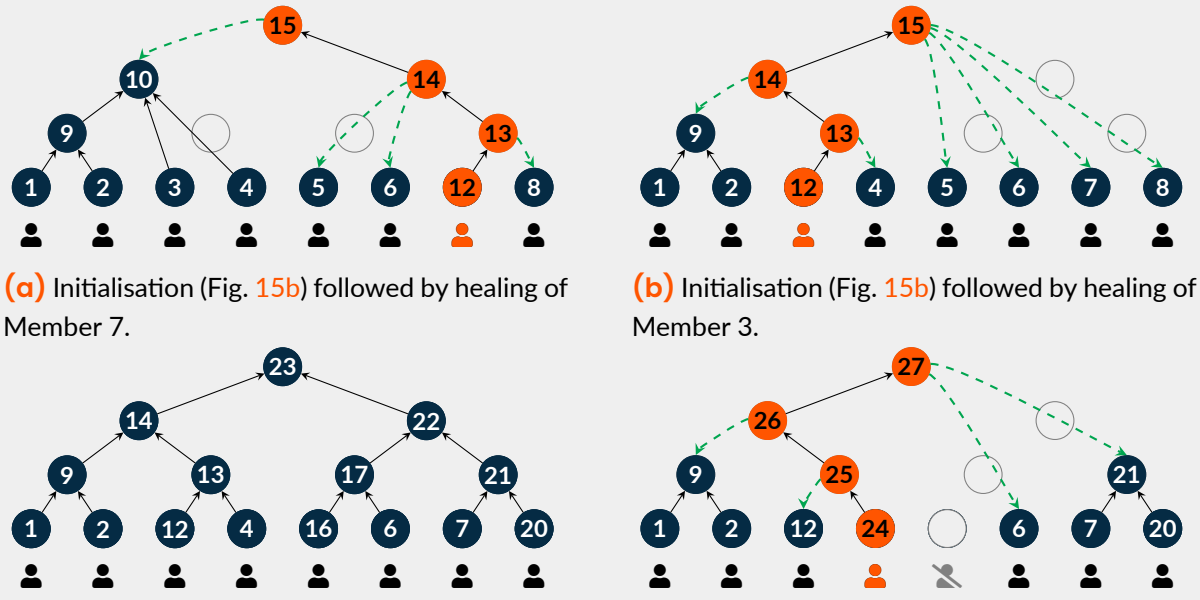
members. As more group members perform healing, the ratchet tree will have fewer blank nodes. This will allow to encrypt the same decryption key s_i to several group members simultaneously, by encrypting it under the encryption key of a node that is a common ancestor of said group members, who are all able to recover s_i thanks to the invariant (T2).

Sending Messages

When a group member sends a message msg to the group, it encrypts msg using the root secret key as a symmetric key. We recall that thanks to (T2), the root secret key is known by group members (and only them). In more detail, to achieve FS, the root key is first expanded into N-symmetric keys using a PRG, where each symmetric key sk_i is used by member i to send a message, and then locally "ratcheted" analogously to Sender Keys. Since the same content is sent to all the group members, the cost of sending a single message to the entire group is one SKE ciphertext. Unlike Sender Keys, MLS leaves it an option for the group members to sign the ciphertext; We can cryptographically tie the sent message to a specific group member using a signature. Otherwise, any member can potentially send messages on behalf of another group member leaving them deniable of the fact of sending messages.

Healing

Healing is handled similarly to the initialization process. We handle removing separately below since it is slightly more complex than healing. Fig. 16 illustrates how healing is performed. At a high level, the goal of healing is to arrive at a tree with no blank nodes as in Fig. 16c — this is when MLS's appealing $O(\log N)$ healing upload cost starts to kick in.



(a) Initialisation (Fig. 15b) followed by healing of Member 7.

(b) Initialisation (Fig. 15b) followed by healing of Member 3.

(c) Initialisation (Fig. 15b) followed by healing of Members 3 (Fig. 16b), 5 and 8. The tree is full.

(d) Fig. 16c followed by Member 4 removing Member 5. The tree is no longer full.

Figure 16: Figs. 16a and 16b are two possible key updates after the initialization phase. Fig. 16d is a key update with removal after the tree is in state Fig. 16c. See Fig. 15 for the explanation of the symbols.

When a member heals, the goal is to preserve the TreeKEM invariants while minimizing the number of blank nodes in the tree. Therefore, a member who wishes to heal does the following:

1. Update the encryption keypairs of all nodes in their path;
2. Broadcast the encryption keys of all freshly updated nodes;
3. For each freshly updated node, broadcast in an encrypted form the decryption key of this node to all members who have this node as an ancestor.

Let us discuss each step individually.

1. Minimising the number of blank nodes is good for global efficiency, however, a group member can only update the encryption keypairs of nodes in their path; updating all of these nodes is the best this group member can do.
2. The goal of the second step is to preserve the invariant (T1).
3. The goal of the third step is to preserve the invariant (T2). Two optimizations are applied. First, the decryption key of an updated node may be encrypted under the encryption key of a common ancestor node of group members instead of each member; this is made possible by (T2). Second, since the decryption key s_i of each freshly updated node is passed into a PRG to obtain the decryption key of its (freshly updated) parent node, it suffices to encrypt to each member the decryption key of the closest node to them.

For example, if Member 7 heals (Fig. 16a), he will update the keypairs $(\mathsf{ek}_i, s_i)_{i\in[12:15]}$, and broadcast four encryption keys (●) and as many ciphertexts (- - - →):

$$\left((\mathsf{ek}_i)_{i\in[12:15]}, \mathsf{Enc}_{\mathsf{ek}_{10}}(s_{15}), \mathsf{Enc}_{\mathsf{ek}_5}(s_{14}), \mathsf{Enc}_{\mathsf{ek}_6}(s_{14}), \mathsf{Enc}_{\mathsf{ek}_8}(s_{13})\right)$$

Here, an astute reader may have noticed that the efficiency and bandwidth saving depend highly on the position of the user healing. For the same initial configuration (Fig. 15b), Member 7 needs to send 4 encryption keys and 4 ciphertexts to heal (Fig. 16a), whereas Member 3 needs to send 4 encryption keys and 6 ciphertexts (Fig. 16b). The number of healings required to arrive at a complete tree depends on the position of the members' healing. After initialization, at least $N/2$ members must perform healing before the tree is complete Note that group members do not necessarily follow the optimal healing schedule, in which case we may need more time and bandwidth in practice.

When we arrive at a complete tree state as in Fig. 16c, the healing upload cost becomes *optimal*. It is easy to check that any group member healing with a complete tree only needs to send $O(\log N)$ KEM encapsulation keys, $O(\log N)$ PKE ciphertexts, and one signature. This is much more efficient than Sender Keys and Pairwise Channels that require $O(N)$ public key values. On the other hand, since the minimum healing download cost is identical to the minimum upload cost, MLS does not achieve $O(1)$ download cost as in other protocols.

Removing Group Members

To securely remove a group member, TreeKEM must delete all decryption keys known by the removed member to maintain Item (T2). Assume Member 4 removes Member 5 in a completely healed tree (Fig. 16d). Member 4 first blanks all the nodes leading from Member 5's node to the root. To fill in the now empty root node, Member 4 further performs a heal. The group is now effectively secure against the removed member since all the secret keys maintained by Member 5 are deleted from the tree, and the updated decryption keys are sent only to the non-removed members.

One side effect of the removing procedure is that blank nodes are reintroduced to the tree even after Member 4 heals. Notably, even though the group is secure against the removed member, other

members need to heal to regain the optimal healing upload cost of MLS. Even worse, when many members are removed from the group, the tree can no longer maintain a nice binary tree structure since the removed leaves remain blank (unless a new member is added to the empty leaves). In such a case, it may be more efficient to reinitialize the group and rearrange the remaining group members at the leaves of a new smaller binary tree.

Commits and Proposals

The recent iterations of TreeKEM (i.e., after version 8 on MLS [BBM+22]) follow a "propose-and-commit" flow, in which members of a group may propose to add new members, remove existing ones, or update their keys by sending *proposal* messages. These proposals only take effect when a group member initiates a new epoch by transmitting a *commit* message, which simultaneously validates a list of indicated proposals and heals itself. Previous to this iteration, a member needed to perform healing whenever adding or removing a member. The new iteration allows batching all these processes until some member decides to heal — offering a better tradeoff between efficiency and security.

3.6 Chained CmPKE by PQShield

Summary

Chained CmPKE [HKP+21] by PQShield — published by PQShield in collaboration with academic researchers at CCS 2021 — is a significant simplification of TreeKEM by MLS. In a bird's eye view, Chained CmPKE is TreeKEM with a *depth-1 N-ary* tree (i.e., a flat tree with only a root and leaf nodes) where the KEM is replaced by an *multi-recipient* KEM (mKEM).

The strength of Chained CmPKE is that it is achieving the best of Sender Keys and Pairwise Channels. While the asymptotic healing costs are identical to Sender Keys and Pairwise Channels, it also realizes the $O(1)$ messaging cost of Sender Keys and the strong PCS security of Pairwise Channels. While the concrete healing costs of Chained CmPKE, when instantiated with a DH-based mKEM and signature scheme, are almost identical to the vanilla Sender Keys and Pairwise Channels, the upload costs can be a few hundred times more efficient when moving to the post-quantum setting.

Chained CmPKE also compares very favorably to TreeKEM. In brief, unless minimizing the healing *upload* cost in a large group is the critical objective, Chained CmPKE is a better choice. The healing upload and download costs of TreeKEM are both $O(\log N)$ in the best case, while those of Chained CmPKE are always $O(N)$ and $O(1)$, respectively. While TreeKEM has a better asymptotic *upload* cost, Chained CmPKE behaves better for small N since the hidden constants behind the O-notation are much smaller. For example, in the post-quantum regime, the concrete upload cost of Chained CmPKE can be smaller for $N \lesssim 256$ [HKP+21, Figs. 6 and 7].

Moreover, Chained CmPKE always has a better *total healing cost*. If all N group members regularly heal, then the total healing costs is $O(N)$ for Chained CmPKE— just like Sender Keys and Pairwise Channels— and $O(N \log N)$ for TreeKEM.

Power of Multi-Recipient KEMs

Let us first motivate mKEM. An mKEM allows to securely send the same session key K to a group of N members. An obvious question is: "*Can't we do that by running N KEMs in parallel?*" The appeal

of mKEM lies in the large *concrete* bandwidth and computation saving compared to trivially running N KEMs in parallel, even though both solutions have a $O(N)$ asymptotic cost. At a high level, if a standard *KEM* ciphertext ct can be decomposed into an encapsulation-key-independent $\widehat{\text{ct}}$ and dependent ct_{ek} components, we can reuse $\widehat{\text{ct}}$ for every group member since it is not directly tied to a specific member. In case $|\widehat{\text{ct}}| \gg |\text{ct}_{\text{ek}}|$, we can get a great savings.

The idea of mKEM has been around since the early 2000's [Kur02, Sma05] — using Diffie-Hellman, running mKEM for N members is twice as efficient as running N KEMs for each member. At ASIACRYPT 2020 [KKPP20], we studied mKEM instantiations at a practical level using 9 post-quantum KEMs (which are lattice and isogeny-based NIST candidates and CSIDH) and show that our mKEM offers savings of at least one order of magnitude bandwidth and makes encryption time shorter by a factor ranging from 2 to 35.

Initialization

The main idea of Chained CmPKE is to take full advantage of mKEM by sending the same secret to all the group members. Chained CmPKE works like TreeKEM but with a flat tree structure. In the initialization phase, some group member — Member 1 (👤) in Fig. 17a — runs mKEM to generate the ciphertext $\text{ct} = (\widehat{\text{ct}}, (\text{ct}_{\text{ek}_i})_{i\in[2:8]})$, where ek_i is the KEM encapsulation key of Member i. Given $\text{ct}_i = (\widehat{\text{ct}}, \text{ct}_{\text{ek}_i})$, each Member i can decrypt the session key K. Member 1 further generates a new pair of KEM keys $(\text{ek}_1, \text{sk}_1)$ and then signs $(\widehat{\text{ct}}, \text{ek}_1)$. Finally, it sends the mKEM ciphertext, new encapsulation key, and signature $(\text{ct}, \text{ek}_1, \sigma)$ to the server.

The server then parses ct and only sends the relevant components $(\text{ct}_i, \text{ek}_1, \sigma)$ to each Member i. Importantly, while the mKEM ciphertext ct is of size $O(N)$, each Member i only needs to download a *part* of ct, which is of size $O(1)$. Using a "committing" mKEM [HKP+21], the group members can authenticate the content by only checking the authenticity of $\widehat{\text{ct}}$, rather than the entire ct. At the end of the initialization phase, all the members share the same session key assigned to the root node as in Fig. 17b.

(a) Initialization by Member 1 (b) Shared state after initialization

Figure 17: Initialization phase of Chained CmPKE. See Fig. 15 for the explanation of the symbols.

Sending Messages

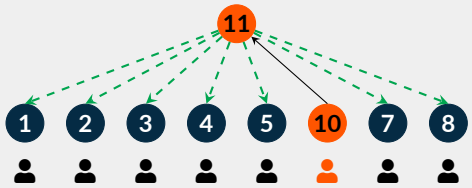
Since both Chained CmPKE and TreeKEM send messages using only the shared root secret key, sending messages is performed identically to MLS.

Healing (and Removing Group Members)

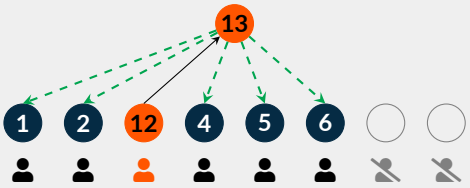
Healing and removing group members are significantly simple operations for Chained CmPKE thanks to the lack of blank nodes as in TreeKEM. Notably, healing is identical to the initialization phase. Let's say Member 6 makes the first heal after the initialization. This is illustrated in Fig. 18a. Member 6 runs *mKEM* with all the other group members as the recipients, updates its KEM encapsulation key,

signs them, and then sends it to the server. The server then dispatches the appropriate contents to each group member. At the end of the healing, all the members share a new root secret key, and Member 6's state is healed.

Removing a member is performed exactly like a heal, where the only difference is that we no longer create mKEM ciphertexts for the removed member. This is illustrated in Fig. 18b, where Member 3 removes members 7 and 8 and performs a heal afterward.



(a) Initialization (Fig. 17b) followed by Member 6 healing.



(b) Fig. 17b followed by Member 3 removing Members 7 and 8, then healing.

Figure 18: Key update Fig. 18a and removal Fig. 18b. See Fig. 15 for the explanation of the symbols.

Applying mKEM to Sender Keys, Pairwise Channels, and TreeKEM

A natural question is if we can replace KEMs by mKEMs in other protocols to boost efficiency.

- For Signal's **Pairwise Channels**, the idea seems difficult to apply since a group member sends a different content to each other members.
- For WhatsApp's **Sender Keys**, we can leverage mKEMs since a group member x sends the same sender key sk_x to all the members. However, since it needs to execute the Double Ratchet protocol to heal, the member x must send $(N-1)$ independent *KEM* encapsulation keys. Specifically, while we can roughly cut the healing cost by half, we won't truly benefit from mKEM due to the cost of sending $O(N)$ KEM encapsulation keys.
- For MLS's **TreeKEM**, this idea provides significant gains. Looking at Figs. 15 and 16, we can use mKEMs instead of KEMs, whenever a group member is sending the same node secret to more than one member. We show in [KKPP20] that we can optimize TreeKEM by modifying the tree structure to be an m-ary tree rather than a binary tree — this allows us to send the same node secret to more than two users and to benefit more from using mKEMs. This observation has been extended further in later work published at ACM CCS [AHKM21] by using a *multi-message* mKEM. We note that in the post-quantum regime, setting $m = N$ provides the best efficiency — this corresponds exactly to PQShield's **Chained CmPKE**.

References

[AAB+21] Hal Abelson, Ross J. Anderson, Steven M. Bellovin, Josh Benaloh, Matt Blaze, Jon Callas, Whitfield Diffie, Susan Landau, Peter G. Neumann, Ronald L. Rivest, Jeffrey I. Schiller, Bruce Schneier, Vanessa Teague, and Carmela Troncoso. Bugs in our pockets: The risks of client-side scanning. *CoRR*, abs/2110.07450, 2021. (Cited on page 8.)

[ABC+20] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions. (Cited on page 27.)

[ACD19] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019*, *Part I*, volume 11476 of *LNCS*, pages 129–158. Springer, Heidelberg, May 2019. (Cited on pages 2 and 24.)

[AHKM21] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. Server-aided continuous group key agreement. Cryptology ePrint Archive, Report 2021/1456, 2021. https://eprint.iacr.org/2021/1456. (Cited on page 41.)

[BBM+22] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-13, Internet Engineering Task Force, 2022. Work in Progress. (Cited on pages 26, 34, and 39.)

[BFG+20] Jacqueline Brendel, Marc Fischlin, Felix Günther, Christian Janson, and Douglas Stebila. Towards post-quantum security for signal's X3DH handshake. In Orr Dunkelman, Michael J. Jacobson, Jr., and Colin O'Flynn, editors, *Selected Areas in Cryptography*, pages 404–430, Cham, 2020. Springer International Publishing. (Cited on page 20.)

[BFG+22] Jacqueline Brendel, Rune Fiedler, Felix Günther, Christian Janson, and Douglas Stebila. Post-quantum asynchronous deniable key exchange and the signal handshake. To Appear at PKC, 2022. (Cited on page 23.)

[Cab21] Cable.co.uk. Worldwide mobile data pricing 2021 | 1gb cost in 230 countries, 2021. https://www.cable.co.uk/mobiles/worldwide-data-pricing/. (Cited on pages 26 and 27.)

[CCG16] Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. On post-compromise security. In Michael Hicks and Boris Köpf, editors, *CSF 2016 Computer Security Foundations Symposium*, pages 164–178. IEEE Computer Society Press, 2016. (Cited on page 12.)

[CCG+19] Katriel Cohn-Gordon, Cas Cremers, Kristian Gjøsteen, Håkon Jacobsen, and Tibor Jager. Highly efficient key exchange protocols with optimal tightness. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019*, *Part III*, volume 11694 of *LNCS*, pages 767–797. Springer, Heidelberg, August 2019. (Cited on page 15.)

[Cel20] Cellebrite. Cellebrite advanced services, 2020. https://cellebrite.com/wp-content/uploads/2020/11/SolutionOverview_AdvancedServices.pdf. (Cited on page 8.)

[CLM+18] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: An efficient post-quantum commutative group action. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018*, *Part III*, volume 11274 of *LNCS*, pages 395–427. Springer, Heidelberg, December 2018. (Cited on page 20.)

[DG21] Samuel Dobson and Steven D. Galbraith. Post-quantum signal key agreement with SIDH. Cryptology ePrint Archive, Report 2021/1187, 2021. https://ia.cr/2021/1187. (Cited on page 20.)

[EM21] Ksenia Ermoshina and Francesca Musiani. The telegram ban: How censorship "made in russia" faces a global internet. *First Monday*, 26(5), Apr. 2021. (Cited on page 8.)

[FSXY12] Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. Strongly secure authenticated key exchange from factoring, codes, and lattices. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 467–484. Springer, Heidelberg, May 2012. (Cited on page 21.)

[FSXY13] Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. Practical and post-quantum authenticated key exchange from one-way secure key encapsulation mechanism. In Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng, editors, *ASIACCS 13*, pages 83–94. ACM Press, May 2013. (Cited on page 21.)

[GKRS20] Siyao Guo, Pritish Kamath, Alon Rosen, and Katerina Sotiraki. Limits on the efficiency of (ring) LWE based non-interactive key exchange. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020*, *Part I*, volume 12110 of *LNCS*, pages 374–395. Springer, Heidelberg, May 2020. (Cited on page 20.)

[HKKP21] Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. An efficient and generic construction for signal's handshake (X3DH): Post-quantum, state leakage secure, and deniable. In Juan Garay, editor, *PKC 2021*, *Part II*, volume 12711 of *LNCS*, pages 410–440. Springer, Heidelberg, May 2021. (Cited on pages 2, 21, 22, and 23.)

[HKP+21] Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. A concrete treatment of efficient continuous group key agreement via multi-recipient PKEs. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1441–1462. ACM Press, November 2021. (Cited on pages 3, 27, 30, 31, 32, 39, and 40.)

[HKSU20] Kathrin Hövelmanns, Eike Kiltz, Sven Schäge, and Dominique Unruh. Generic authenticated key exchange in the quantum random oracle model. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part II*, volume 12111 of *LNCS*, pages 389–422. Springer, Heidelberg, May 2020. (Cited on page 21.)

[JAC+20] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, Geovandro Pereira, Koray Karabina, and Aaron Hutchinson. SIKE. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions. (Cited on page 27.)

[JKL04] Ik Rae Jeong, Jonathan Katz, and Dong Hoon Lee. One-round protocols for two-party authenticated key exchange. In Markus Jakobsson, Moti Yung, and Jianying Zhou, editors, *ACNS 04*, volume 3089 of *LNCS*, pages 220–232. Springer, Heidelberg, June 2004. (Cited on page 15.)

[KF14] Kaoru Kurosawa and Jun Furukawa. 2-pass key exchange protocols from CPA-secure KEM. In Josh Benaloh, editor, *CT-RSA 2014*, volume 8366 of *LNCS*, pages 385–401. Springer, Heidelberg, February 2014. (Cited on page 21.)

[KKPP20] Shuichi Katsumata, Kris Kwiatkowski, Federico Pintore, and Thomas Prest. Scalable ciphertext compression techniques for post-quantum KEMs and their applications. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part I*, volume 12491 of *LNCS*, pages 289–320. Springer, Heidelberg, December 2020. (Cited on pages 3, 30, 31, 40, and 41.)

[Kra05] Hugo Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 546–566. Springer, Heidelberg, August 2005. (Cited on page 15.)

[Kra10] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. Cryptology ePrint Archive, Report 2010/264, 2010. https://eprint.iacr.org/2010/264. (Cited on page 14.)

[Kur02] Kaoru Kurosawa. Multi-recipient public-key encryption with shortened ciphertext. In David Naccache and Pascal Paillier, editors, *PKC 2002*, volume 2274 of *LNCS*, pages 48–63. Springer, Heidelberg, February 2002. (Cited on page 40.)

[Kwi19] Kris Kwiatkowski. Towards post-quantum cryptography in tls. Cloudflare Blog, 2019. https://blog.cloudflare.com/towards-post-quantum-cryptography-in-tls/. (Cited on page 10.)

[Lan18] Adam Langley. CECPQ2. Imperial Violet, 2018. https://www.imperialviolet.org/2018/12/12/cecpq2.html. (Cited on page 10.)

[LIN20] LINE. Line encryption report, 2020. (Cited on page 12.)

[LLM07] Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *ProvSec 2007*, volume 4784 of *LNCS*, pages 1–16. Springer, Heidelberg, November 2007. (Cited on page 15.)

[MP16a] Moxie Marlinspike and Trevor Perrin. The double ratchet algorithm, November 2016. https://signal.org/docs/specifications/doubleratchet/. (Cited on pages 14, 17, and 33.)

[MP16b] Moxie Marlinspike and Trevor Perrin. The x3dh key agreement protocol, November 2016. https://signal.org/docs/specifications/x3dh/. (Cited on pages 14, 17, and 23.)

[NIS20] NIST. Nistir 8309 - status report on the second round of the nist post-quantum cryptography standardization process, 2020. https://csrc.nist.gov/publications/detail/nistir/8309/final. (Cited on page 12.)

[OBR+21] Emad Omara, Benjamin Beurdouche, Eric Rescorla, Srinivas Inguva, Albert Kwon, and Alan Duric. The Messaging Layer Security (MLS) Architecture. Internet-Draft draft-ietf-mls-architecture-06, Internet Engineering Task Force, March 2021. Work in Progress. (Cited on pages 26 and 34.)

[Per16] Trevor Perrin. The xeddsa and vxeddsa signature schemes, October 2016. https://signal.org/docs/specifications/xeddsa/. (Cited on pages 14 and 31.)

[PQS20] PQShield. An overview of post-quantum cryptography. 2020. https://pqshield.com/whitepapers/. (Cited on page 12.)

[PQS21] PQShield. Understanding the upcoming nist post-quantum cryptography standards. 2021. https://pqshield.com/whitepapers/. (Cited on pages 12 and 33.)

[Pri19] Technology preview: Signal private group system, December 2019. https://signal.org/blog/signal-private-group-system/. (Cited on page 10.)

[RS18] Lior Rotem and Gil Segev. Out-of-band authentication in group messaging: Computational, statistical, optimal. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 63–89. Springer, Heidelberg, August 2018. (Cited on page 15.)

[SAB+20] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions. (Cited on page 27.)

[Sea18] Technology preview: Sealed sender for signal, October 2018. https://signal.org/blog/sealed-sender/. (Cited on page 10.)

[SIG] Signal protocol: Technical documentation. https://signal.org/docs/. (Cited on page 13.)

[Sma05] Nigel P. Smart. Efficient key encapsulation to multiple parties. In Carlo Blundo and Stelvio Cimato, editors, *SCN 04*, volume 3352 of *LNCS*, pages 208–219. Springer, Heidelberg, September 2005. (Cited on page 40.)

[Soa21] Soatok. Threema: Three strikes, you're out. Dhole Moments, 2021. (Cited on page 12.)

[Spe21] Speedtest. Speedtest global index — internet speed around the world, July 2021. https://www.speedtest.net/global-index. (Cited on page 27.)

[SSW20] Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum TLS without handshake signatures. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1461–1480. ACM Press, November 2020. (Cited on page 10.)

[Thr21] Threema. Cryptography whitepaper, 2021. (Cited on page 12.)

[Vau05] Serge Vaudenay. Secure communications over insecure channels based on short authenticated strings. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 309–326. Springer, Heidelberg, August 2005. (Cited on page 15.)

[Wha21] WhatsApp. Whatsapp encryption overview v6, 2021. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf. (Cited on pages 15 and 32.)

[XAY+20] Haiyang Xue, Man Ho Au, Rupeng Yang, Bei Liang, and Haodong Jiang. Compact authenticated key exchange in the quantum random oracle model. Cryptology ePrint Archive, Report 2020/1282, 2020. https://eprint.iacr.org/2020/1282. (Cited on page 21.)

[XLL+18] Haiyang Xue, Xianhui Lu, Bao Li, Bei Liang, and Jingnan He. Understanding and constructing AKE via double-key key encapsulation mechanism. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part II*, volume 11273 of *LNCS*, pages 158–189. Springer, Heidelberg, December 2018. (Cited on page 21.)

[YCL18] Zheng Yang, Yu Chen, and Song Luo. Two-message key exchange with strong security from ideal lattices. In Nigel P. Smart, editor, *CT-RSA 2018*, volume 10808 of *LNCS*, pages 98–115. Springer, Heidelberg, April 2018. (Cited on page 21.)

[Zer21] Project Zero. A deep dive into an nso zero-click imessage exploit: Remote code execution, 2021. https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html. (Cited on page 8.)

(U//FOUO) FBI's Ability to Legally Access Secure Messaging App Content and Metadata

(U//LES) As of November 2020, the FBI's ability to legally access secure content on leading messaging applications is depicted below, including accessible information based on the applicable legal process. Return data provided by the companies listed below, with the exception of WhatsApp, are actually logs of latent data that are provided to law enforcement in a non-real-time manner and may impact investigation due to delivery delays.

UNCLASSIFIED // LAW ENFORCEMENT SENSITIVE

App	iMessage	Line	Signal	Telegram	Threema	Viber	WeChat	WhatsApp	Wickr
Information accessed									
Legal Process & Additional Details	• **Message Content: Limited** • **Subpoena:** can render basic subscriber information • **18 U.S.C. §2709(d):** can render 25 days of iMessage lookups to and from a target number[1] • **Pen Register:** no capability[1] • **Search Warrant:** can render backups of a target device; if target uses iCloud backup, the encryption keys should also be provided with content return; can also acquire iMessages from iCloud returns if target has enabled Messages in iCloud	• **Message Content: Limited*** • Suspect and/or victim's registered information (profile image, display name, email address, phone number, LINE ID, date of registration, etc.) • Information on usage * Maximum of seven days' worth of specified users' text chats (Only when E2EE has not been elected and applied and only when receiving an effective warrant; however video, picture, files, location, phone call audio and other such data will not be disclosed)	• **No Message Content** • Date and time a user registered • Last date of a user's connectivity to the service	• **No Message Content** • No contact information provided for law enforcement to pursue a court order. As per Telegran's privacy statement, for confirmed terrorist invectigations, Telegram now disclises IP address and phone number to relevant authorities	• **No Message Content** • Hash of phone number and email address, if provided by user • Push/Token, if push service is used • Public Key • Date (no time) of Threema ID creation • Date (no time) of last login	• **No Message Content** • Provides account (i.e. phone number) registration data and IP address at time of creation • Message History: time, date, source number and destination number	• **No Message Content** • Accepts preservation letters and subpoenas, but cannot provide records for accounts created in China • For non-China accounts, they can provide basic information (name, phone number, email, IP address) which is retained for as long as the account is active	• **Message Content: Limited*** • Subpoena: can render basic subscriber records • Court Order: Subpoena return as well as information like blocked users • Search Warrant: Provides address book contacts and WhatsApp users who have the target in their address book contacts • Pen Register: Sent every 15 minutes, provides source and destination for each message * If target is using an iPhone and iCloud backups enabled, iCloud returns may contain WhatsApp data, to include message content	• **No Message Content** • Date and time account created • Type of decide(s) add installed in • Date of last use • Total number of messages • Number of external IDs (email addresses and phone numbers) connected to the account, but not plaintext external IDs themselves • Avatar image • Limited records of recent changes to account setting such as adding or suspending a device (does not include message content or routing and delivery information) • Wickr Version Number

(U) Prepared by Science and Technology Branch and Operational Technology Division

7 January 2021

[1](U//LES) Apple provided logs only identify if a lookup occurred. Apple returns include a disclaimer that a log entry between parties does not indicate a conversation took place. These query logs have also contained errors.