Challenges in E2E Encrypted Group Messaging

Writing Assignment Spring 2020 in TM8107 -
Cryptographic Protocols and Their Applications

Tjerand Silde
tjerand.silde@ntnu.no

Department of Mathematical Sciences,
Norwegian University of Science and Technology

May 14, 2020

Abstract

This report consider *end-to-end encrypted* messaging applications, and study the setting of group conversations. Communication security are becoming increasingly mainstream, but even though secure 1-1 communication is a solved problem, encrypted group conversations are implemented in many flavors, offering different levels of *privacy*, *integrity*, *authentication*, *anonymity* and *efficiency*.

We describe these properties in details and discuss trade-offs between different solutions. Further, we study some of the mostly used applications in practice; Signal, WhatsApp, Keybase, iMessage, Crypho, Wire and Threema - and compare their features.

Contents

1 Introduction

Everyone with a phone, tablet or computer chat or send emails over the Internet every day, and there are several properties that are important to the senders and receivers when using these applications. These properties depend on the situation that the messages are sent, be it in between friends or family, work-communication, doctor-patient, journalist-source, activism, police-work, politics or something else. We might want the messages to be private (that the content is *confidential*), being able to verify the *integrity* of the messages (that they are not altered with), or the make sure that we are communication with whom we think we are talking to (that we can *authenticate* the sender and receiver). Sometimes we only need some of these properties, other times we need all. But there are also more. We might want that the services are reliable, that we can send messages whenever we need to, and that it is efficient. In some cases we might need stronger versions of the requirements listed, and we'll discuss them more in detail later.

However, these properties is needed also outside of the two-party setting, where there are one sender and more than one receiver for each message. In group communication we need all this, but we must make sure that all the messages are synchronized, so that every member of the group can participate in the conversation and receive all the necessary information being sent. This makes the situation more complex. Groups may be really large, up to hundreds or thousands of members, and this might influence both the efficiency, scalability and availability of the communication. Many groups are dynamic; sometimes new members are being added or current members are leaving. It is not always easy to extend 1-1 communication to this setting.

For *privacy*, there are three possible situations: 1) all messages are sent in the clear, 2) the messages are sent over an encrypted channel from the sender to the application server, and from the server to the recipients, or 3) the messages are encrypted by the sender, and only the recipients can decrypt and read the messages.

In the fist setting the users have no privacy whatsoever. It might still be important to verify the integrity of the message and authenticity of the sender of each message, say, in the case of a national emergency and the government need to send important but public information to all its citizens.

The second situation is called transport layer encryption, and it makes sure that no one else that the sender, receiver and application server can read the messages. However, this means that the parties involved need to trust the server, as it allows him to read and store all the messages sent. This could expose the messages in the cases where the police wants access, the server get hacked or the company running the service have untrusted employees.

The third situation is called end-to-end encryption (E2EE)(See [CCG$^+$18] for more details). Here, all the communicating members have a secretly shared cryptographic key to makes sure that they are the only ones that can send and receive messages. In particular, this means that the application server only forwards messages being sent, but can't read the content. Note, however, that this does not mean that the communication is unhackable. Now, a hacker must hack a users device to get access to his, but only his, conversations, instead of hacking the centralized server to get access to everyones conversations.

All three cases are normal. Email and SMS are in general not encrypted, unless you are using specialized services. Transport layer encryption is used basically whenever you connect to a website on the Internet, or when you send messages via Microsoft Teams or Slack, use Gmail to send emails or have a video call over Zoom. Lastly, end-to-end encryption are getting more widespread as we speak, by a lot of effort in this area the past few years from cryptographers and developers. Examples are Protonmail, Tutanota and Fastmail for email, Facetime and Skype (opt-in) for video, and Signal, WhatsApp, Keybase, iMessage, Crypho, Wire and Threema for chat. Some of these chat services also offer end-to-end encrypted video calls for two people or small groups. Facebook Messenger and Telegram offer end-to-end encryption for 1-1 conversations, but not by default, and only offer transport layer encryption for group communication.

In this project we'll study all of the previously mentioned properties for the group-setting, where the minimal requirement is that all communication are *end-to-end encrypted*. We will focus on messaging services sending chat messages over the Internet, like the ones mentioned in the previous paragraph. As noted earlier, it is not necessarily easy to go from 1-1 communication to group conversations, and the services have all solved these challenges in different ways. We will look into their design structures, study their features, and analyze the security of each of the applications. Finally, also make a short comparison with the ongoing IETF standardization effort called Messaging Layer Security (MLS).

2 Two-Party End-to-End Encryption

We start by looking at end-to-end encryption for two parties, as explained and visualized in Figure 1. This is a simplified version of Pretty Good Privacy (PGP), published by Zimmerman in 1991 [Zim91]. We can achieve a secure channel based only on public key cryptography, given that the sender has access to the public encryption key of the recipient and that he has access to the public verification key of the sender. However, this is not an assumption we in general can make. When connecting over TLS [Res18] to websites on the Internet, we have a public key infrastructure (PKI) consisting of several companies who's job is to verify the ownership of each website and their public keys, so that users safely can connect. This moves the verification process of a website into trusting these companies to do their job properly. In messaging applications, each system need some way of authenticating the individual parties. Some application solve this by assuming that the users have other ways of communicating as well (e.g in person or via friends, like in PGP), to be able to verify one another, while other applications assume trusted third parties to do this process. We'll look into some of the methods that are implemented in practice by the applications mentioned earlier.

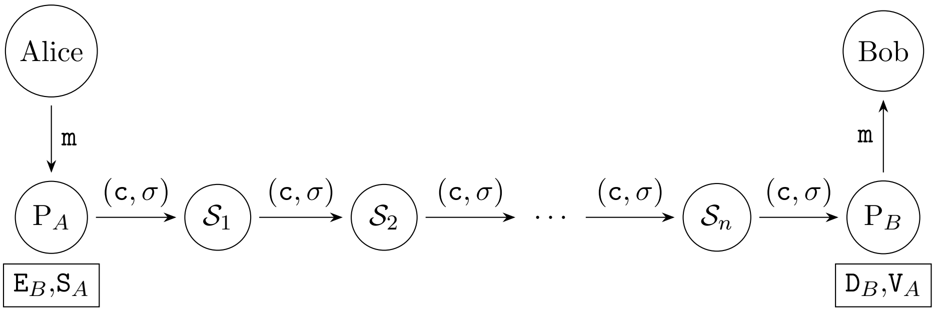

Figure 1: End-to-end encrypted channel between Alice's phone P_A and Bob's phone P_B, sending a message $\mathtt{m}$ from P_A to P_B as a ciphertext $\mathtt{c}$ together with a signature σ. The ciphertext and signature passes the n servers $\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_n$ on the way from the sender to the recipient. P_B verifies that the signature is valid before giving Bob the decrypted message. Here, $\mathtt{E}_B$ and $\mathtt{D}_B$ are Bob's public encryption key and secret decryption key, and $\mathtt{S}_A$ and $\mathtt{V}_A$ are Alice's secret signing key and public verification key. Also, we assume for simplicity that all the keys are stored in the phones memory.

Further, public key cryptography are slow and consists of a large overhead of data per ciphertexts. As in TLS, the way to solve this is to use public key cryptography to agree upon a secret shared key used for symmetric key cryptography, which is way more efficient. Any Authenticated Encryption with Associated Data (AEAD) [Rog02] scheme would do, removing the necessity of signatures for authenticating the messages after the key exchange.

However, this might still not be good enough. Assume that someone are listening in to the conversation and are storing all the ciphertexts being sent. If one of the devises are being hacked, and the hacker get hold of the secret key of the user, then the hacker would be able to decrypt all the ciphertexts ever sent between the two parties. This can be avoided. The idea of *forward secrecy* [BG19] is that each key is only used for a short period of time, being a so-called ephemeral key, before the two parties negotiate a new shared secret key used for the next time slot. In this case, the hacker can only decrypt and read the messages being sent in the current time slot, which might be as short as only a few minutes long. This way, we only use slow public key cryptography once per time slot to create a new key, while everything else is encrypted using fast symmetric key cryptography.

Also, assume that a hacker was able to take over a user's phone. Will the user be able to get back in control, or is the conversation poisoned for ever going forward? If the hacker is in full control of the device, there is nothing to do. However, if the attacker lose control of the device, say, if some malware was found and deleted, then the users should be able to agree upon a new and fresh key, and the conversation would again be private. This is called *post-compromise security* [CCG16]. If the users are creating new keys not only based on the long-term public keys, but also based on the transcript of messages, then the attacker can not get hold of the new session keys unless he already had access to everything from the past. This indicates that unless the attacker gain 100 % control over the user, then he'll lose control again whenever the conversation enters a new time slot.

One last property we'll discuss in detail for the two party setting is *deniability* [BBG04]. We do want the users to be able to verify whom they are talking to, but we don't necessarily want a user to be held responsible for a message she has sent. Deniability means that the receiver should be able to verify the authenticity of a received message, but he shouldn't be able to convince anyone else about this fact. In practice, this is implemented such that both parties holds a shared key for authenticated encryption, which means that both two parties involved could have produced any message in the conversation. Then both parties can be convinced about the authenticity of the messages, but any third party could only be convince that one out of the two parties sent a particular message, and hence, the receiver could himself have produced the message he's claiming to have received. Note that we don't have deniability when using signatures to authenticate messages.

We'll return to *forward secrecy*, *post-compromise security* and *deniability* when comparing the messaging protocols. Several of these properties was first implemented for group conversations in the Off-the-Record protocol [GUVC09], as a replacement for PGP (which offered none of these properties), and has been widely adopted by messaging applications since.

3 Challenges in the Group Setting

Let's consider the group setting. Ideally, we want the group conversations to inherit the same security properties as described for the two party setting. Some of the main challenges here are how to agree upon keys and how to send messages in between all the participants. Let's take a look at the three most common settings, and discuss their advantages and disadvantages in terms of communication and implementation complexity. We'll also discuss how metadata can be handled in different situation, both for message communication, group structures and lists of contacts.

3.1 Pair-Wise Channels

The simplest solution, in terms of implementation complexity, is to use pair-wise channels for group communication. Any application supporting end-to-end encrypted group communication already supports E2EE for 1-1 conversations. By leveraging these channels, and locally keeping track of who's in the group and not, we can encrypt each message using the shared keys earlier exchanged with each of the group members. This is visualized in Figure 2. Let the group consist of, say, four members, as in the figure. Then the sender encrypt his message under the shared secret key of each of the three other group members individually, and send the ciphertexts to the server, which then forwards the ciphertexts to the intended receivers. Note that the communication complexity per message in this setting is linear in the number of group members, which scales poorly for large groups.

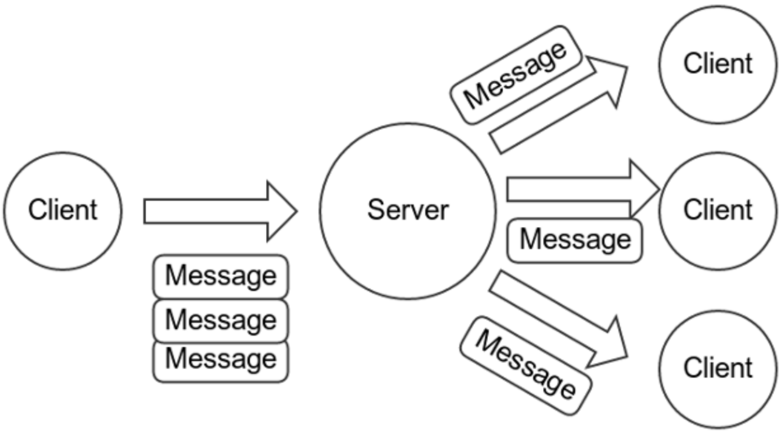

Figure 2: E2EE using pair-wise channels. From Signal Blog [Sig14].

It is obvious that in this setting, we inherit all the security properties from the 1-1 setting. If the 1-1 setting provides *forward secrecy*, *post-compromise security* and *deniability*, then the group setting does as well. Another advantage is that the server will be unaware about groups, as all communication looks like 1-1 conversations. It is however aware of the pair-wise channels.

3.2 Encrypted Message-Keys

The second solution is for the sender to choose a new random key for each message that he's sending. A key is most likely smaller than a message. This way, he can continue by encrypting the message under the new key, and then encrypt the key itself under using the keys agreed upon via the pair-wise channels. This way, the sender only have to send one copy of the encrypted message to the server, while sending a linear number of encrypted keys. The server then forwards the ciphertext in a fan-out fashion, while sending the encrypted keys to each individual group member, respectively. This is visualized in Figure 3. On average, this would lower the communication complexity of the group conversations, maybe except for very short messages. The implementation complexity should not increase by very much compared to only using pair-wise communication channels.

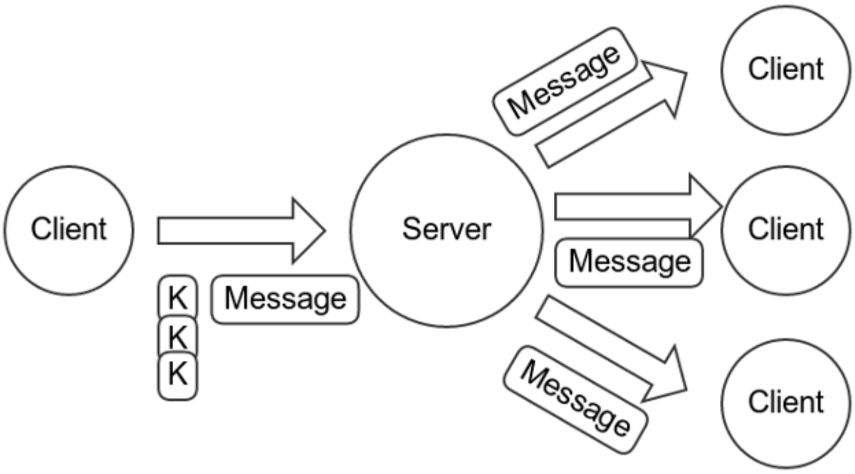

Figure 3: E2EE using encrypted message-keys. From Signal Blog [Sig14].

It is pretty clear that we also in this setting inherit all the security properties from the 1-1 setting, as the only difference is that we introduce ephemeral keys for each message, which is then shared via pair-wise channels. However, this puts more trust into the server. Now that the same ciphertext are being shared with all members of the group, it is obvious to the server that these users are having a group conversation.

3.3 Shared Groups-Keys

The last setting we'll discuss is group conversations with shared group-keys. In this case all the group members need to execute a group key-exchange, potentially involving both static and ephemeral public keys from each member of the group, to agree upon a single shared key. This key is then, implicitly, used to encrypt all messages sent by any user in the group to everyone else, as visualized in Figure 4. Now, only one ciphertext is sent to the server, which then use a fan-out mechanism to send a copy of the ciphertext to each user of the group. This lower the communication complexity to a minimum, it's actually constant per message, if we ignore the cost of the group key-exchange. However, the group key-exchange might be expensive both in terms of communication complexity and implementation complexity.

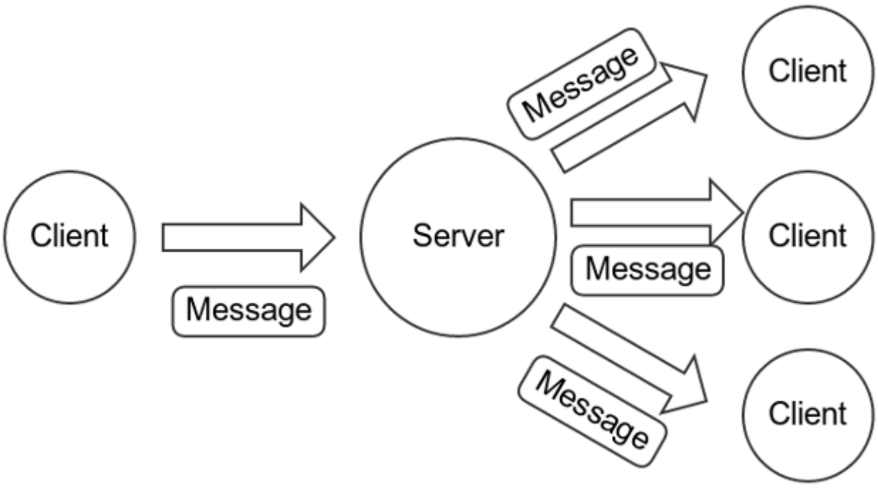

Figure 4: E2EE using shared group-keys. From Signal Blog [Sig14].

The security properties in this setting follows closely from the group key-exchange. Both *forward secrecy* and *post-compromise security* require the key-exchange to be repeated frequently, or to have a ratchet mechanism for deriving new session keys for future time slots. Further, *deniability* require the use of authenticated key-exchange protocol without using signatures, among other things. In this setting, the server also learns the group structure and the metadata regarding the users who are communicating together.

3.4 The Metadata Problem

Another issue in E2EE group communication, or any digital communication in general, is the leakage of metadata. Metadata is in our setting all information about the users in the system and the messages being sent. Examples are ip-addresses, usernames and other profile information, dates and timestamps for registration and being online, user contacts and social graph, when and to whom messages are sent and so on. This reveals quite a bit of information with the application server, and may leak in case it's being hacked, subpoenaed or misused by malicious or incompetent hosts running the servers. Some of the metadata might also leak to third parties controlling the infrastructure between the users and the server. This means that even if a system offer privacy, authenticity and integrity of the messages, it might not provide any anonymity at all, dependent on how the metadata is treated. This may or may not be important for the users.

One common way to prevent leaking metadata to third parties is to use TLS to secure the communication between the users and the client, as visualized in Figure 5. It still leaks which application is being used by the client, but it doesn't leak any information about the client user nor who he is communicating with via the messaging application.

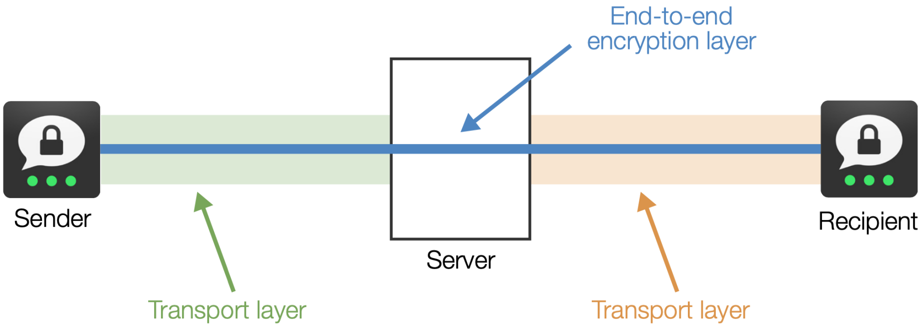

Figure 5: Protecting E2EE with TLS. From Threema Whitepaper [Thr19].

The other metadata is very dependent on the application structure. Some applications use phone numbers as usernames, and these are often closely tied to the real identity of a person. Email-addresses offer more anonymity, if one creates a new address for this application that can not be tied to a persons identity or other attributes. However, to be able to find other users on the application, the user has to share a list of contacts, that be phone numbers of email addresses, with the server. Hence, you share your social graph, and this can often uniquely identify all users. Other applications allow users to sign up by choosing only a username, but now the user need to find his friends and contacts by himself. Also, this only provides anonymity when identical usernames are not used across different applications, in case some of them ties the user to the real identity somewhere else.

One common way to handle lists of contacts is to hash each user in the list and send the hashes to the server. The server then compare the hashes with it's own list containing the hashes of all users, and returning the intersection. However, this is not really preserving anonymity; the sever has access to all users and can easily store a tuple of username and hash together in memory. Further, if the server only a hash of, say, every user's phone number, then one could more easily compare hashes in a secure manner. But also this approach have a downside, as any phone number doesn't have more than ten decimal digits, and one can easily brute force any hash in short time and extract the list of contacts. This is however the most common way of finding friends using the same application.

The best anonymity preserving methods are very complex. Private set intersection is an active area of research, and a huge amount of progress is made throughout the past few years. Unfortunately, most of the published protocols are still to impractical for most applications comparing huge sets of data. But there exists a solution: Signal [Sig20] amplifies the use of Software Guard Extensions (SGX) for this purpose. The SGX provides a *secure enclave*, which means that it can secret key-material and computations that not even the owner of the machine has access to, use this to decrypt incoming lists of contacts and do the comparison in a privacy preserving manner. In addition, the SGX provides *remote attestation*, which means that we externally can verify the code that is running on the SGX. Signal has published the code online, and everyone can then attest that the contact-matching procedure is working as it should. There are issues of practicality with this approach as well, but we'll refer to the documentation [Sig17] for the details.

Finally, lets discuss communication metadata. Initially, one could imagine that the server needs to know both the sender and the receiver of a message for it to be sent over the network. This applies both for the 1-1 and the group setting, but especially the group setting, where a a copy of the message potentially should be sent to several users at the same time. This is hard to avoid, and most messaging services allow the server to have access to this information. However, Signal [Sig19] have a solution to this as well. The idea of anonymous credentials was introduced by Chaum [Cha86] in 1986, for use in a system for digital cash, where a user can be assigned credentials at some point of time, and later prove the possession of some valid credentials without revealing his identity. This solution is unfortunately not practical, but more recent work by Chase et al. [CMZ14, CPZ19] for the setting of messaging applications give a feasible protocol. Here, the server creates a message-queue for each conversation in the system, and stores only an encryption of the list of users in each conversation. The user that created the conversation obtain credentials linked to the queue, and can ask the server to generate more credentials on the fly when new users are added or removed from the conversation. Then, any user with the correct credentials can prove to the server that he is a part of the conversation without revealing his identity, are allowed to fetch all unread messages, and can decrypt them locally. Signal also provides sealed senders [Sig18], where the name of the sender is encrypted via a public header key of the recipient, so that he can decrypt the header to find out who the sender is, and then decrypt the message with the appropriate key. This hides both the sender and recipient from the server.

4 E2E Encrypted Chat Applications

In this section we'll describe the features of some of the most used end-to-end encrypted messaging services.

Signal [Sig20]. An important part of the Signal protocol is the double ratcheting procedure [ACD19] used for authenticated key-exchange (AKE). This procedure takes as input a long term key and a short term key from each participant, and computes a shared key. Further, this key is used in a key-derivation ratchet procedure, that produces ephemeral session keys. When combining these procedures, Signal achieves both *forward secrecy* and *post-compromise security*. The AKE uses long term encryption keys, not signatures, which means that the key is not uniquely tied to the user, and hence, the protocol offers *deniability*. Each message is encrypted with a unique message key, which is derived from the message and the current session key. In the group setting, the initiator of the group creates a fresh key and share this key with each member of the group via pair-wise channels. The session key is a group session key, derived from the shared group key, and groups also use the double ratcheting procedure to create fresh session keys. Group keys are updated every time a new member are added to the group or a old member leaves. This way, the group inherit all the security properties from the pair-wise channels.

Messaging applications can't assume all users to be online at the same time, and hence, need to deal with online-offline key-exchanges. For static keys, this is easily solved by asking the server for the public keys of all recipients, but ephemeral key-exchanges inherently need fresh randomness. Signal solves this by having all users frequently upload one-time pre-keys to the server that can be fetched by other users. Then the online party can perform the key-exchange and start communicating with the offline party. The offline party complete the key-exchange when getting online, and the conversation can continue.

Signal allow users to pair-wise authenticate themselves via other channels using safety numbers and QR codes. Signal offers encrypted backups of all conversations, with randomly generated passwords of length 30 digits.

As previously explained, Signal empower SGX for private set intersections so that each users list of contacts stay private to the server. Metadata about groups are stored in encrypted from on the server, and members use anonymous credentials to fetch messages from the queue, where each message are encrypted in two layers. This hides both the sender and the recipient of a message from the server. Signal was subpoenaed in 2016 [Sig16], and all the information they were able to hand over to law enforcements was ip-addresses and information about when the user signed up and was active.

WhatsApp [Wha17]. Several other messaging applications are using the double ratcheting procedure from the Signal protocol as a backbone, and WhatsApp are one of those applications. Similarly, pair-wise channels are used to create group-keys, which are frequently updated, and hence, WhatsApp also achieve *forward secrecy*, *post-compromise security* and *deniability*.

WhatsApp also allow users to pair-wise authenticate themselves via other channels using safety numbers and QR codes. WhatsApp offers plaintext backups of all conversations to be uploaded and stored in cloud services.

One big difference between Signal and WhatsApp is that the WhatsApp server has access to all metadata; all phone numbers of contacts, group-memberships, statistics about sent messages etc.

Keybase [Key20]. This application is not based on the Signal protocol. In Keybase, each user have their own append-only chain of signed public keys, one key for each device. The chain updates whenever a new device is added or removed, or the user have created a paper-key for offline storage. All user chains form a large global Merkle-tree of public keys that are publicly available for everyone to analyze and verify. Each group forms it's own chain, and team names are public. Team-members may chose to publish team memberships on their public Keybase-profile. Group keys are shared symmetric keys that group admins rotate whenever a team member revokes a device, leaves the team, is removed from the team or resets his/her account. Then the group does a "lazy" update, in the sense that one of the admins generate a new key and share this with the other group members whenever he log on after the change in group structure.

Keybase offer users to send ephemeral exploding messages using one-time keys, but in general are all key-material static after the key-exchange. This means no *forward secrecy* nor *post-compromise security.* Every message sent it signed by the users public key, and hence, Keybase offer no *deniability.* We also note that Keybase allow the use of public keys to adopt the PGP infrastructure, using other channels (like email) to communicate messages, and using the Keybase client to encrypt and decrypt. This hides the metadata to the server.

Keybase offer a different way of authentication. They allow users to tie their account to other online accounts, like Twitter, Github, personal websites etc., by publishing their public keys and linking them together. This way, any user can be confident that the Keybase-user they are talking to, are the owner of these other accounts as well, which allow authentication via their online social graph. This may or may not break *anonymity*, depending on which accounts that are tied together.

As usual practice, the Keybase servers has access to all metadata in the network. Part of the metadata are even public, as a consequence of the user-structure, but the server have access to team memberships, users, roles and statistics about ciphertexts being sent through the network.

iMessage [App20]. Apple made iMessage initially to replace SMS, but it's also used as a purely online messaging application. iMessage works pretty similar to what is described in Figure 1, where each user creates it's own public/private-key pairs for encryption and signatures, and uploads the public keys to the Apple sever. Each user can then ask Apple for the public keys of another user, and start a conversation. Only static public keys are used, and hence, there is no *forward secrecy* nor *post-compromise security.* Group messaging is done via pair-wise channels, and is inherently inefficient. However, iMessage is usually not used for large groups, and in practice this works out nicely, and also makes the protocol complexity easier to handle. We also note that when sending multimedia, then iMessage use fresh message-keys to encrypt the file, upload the file to the Apple cloud, and send the encrypted message-key to the recipient together with a pointer to the file.

Apple offer no way of authenticating users, and users need to trust the sever when they ask for public keys (the so-called TOFU-model). In addition, one can store a plaintext backup of all conversation in iCloud if preferred.

Crypho [Cry20]. Crypho takes a different approach to generating keys. Instead of generating them randomly and storing them locally, Crypho let the user choose a password as secret. The password is input to the hash function *scrypt*, and the output is used as a secret key. The public key is then derived from this output. This allows for dictionary attacks etc., which they hope to mitigate by enforcing the user to activate 2-factor authentication (2FA). On the other hand, this allows for more flexibility for the user, where he more easily can go in between devices or browsers when using the service. Users can download and print a one-time key-recovery code in case password is forgotten or 2FA-device is lost/stolen.

Furthermore, the public keys are used to exchange symmetric keys used for pair-wise and group conversations, using the Stanford Javascript Crypto Library [SHB09], and hence, Crypho has implemented the group-key model for croup communication. However, because of static public keys, it follows that Crypho doesn't offer *forward secrecy* nor *post-compromise security*. It may offer *anonymity*, but that depends on the information individuals users registered when signing up.

Crypho let users compare safety numbers or QR codes for authentication, but also allows users to authenticate themselves via Bank ID (for Norwegian citizens), showing the name and date of birth of the user. This allows for strong authentication, to the price of anonymity.

Wire [Wir18]. Also this application uses the Signal protocol for key-exchange. The double ratchet protocol is used to generate session keys for pair-wise communication, and pair-wise channels are used for group conversations. The server has access to all the metadata about the groups and the messages being sent.

Wire-users can authenticate each other via safety numbers, and Wire offer password-protected backups of all messages.

Threema [Thr19]. This application uses the library NaCl [BLS12] for all cryptographic operations. A user generates secret keys locally, and published the public keys at the Threema server. The static keys are used for encryption and signatures, and hence, Threema offer no *forward secrecy* nor *post-compromise security*, but shared secret keys do offer *deniability*. All group communication are sent over pair-wise channels, but multimedia is only uploaded once and encrypted with a message-key which is distributed to the recipients via the pair-wise channels.

A user can sign up with username, phone number or email, and may or may not offer *anonymity*, based on the users choice. Authentication can be done by verifying safety numbers, but Threema also offer some trust in the sense that you can see if other people have verified security numbers with the user you are talking to. Users may download password-protected backup files from all their conversations.

Messaging Layer Security [IET20]. MLS is a new and ongoing initiative from The Internet Engineering Task Force (IETF). As we have learned, all group messaging applications are designing their own protocols for end-to-end encryption, with some similarities and some differences. However, there are no unified way of achieving security in this setting. Also, while the group-key setting is the most efficient in terms of communication of messages, it is linear in key-exchange complexity. For large groups, this can be a bottleneck. The MLS working group are trying to design a new protocol for group messaging, which have all the best security properties and communication properties as described earlier in this report, but with more efficient key-escrow. The goal of this work is to design a provably secure group messaging protocol with only logarithmic communication cost for key-exchange and key-updates, using binary trees. This work might influence how the described application will be working in the future.

5 Conclusion

We summarize and compare the applications in Table 1. All services have implemented end-to-end encryption, but offer different tradeoffs in terms of more robust privacy, anonymity and efficiency. Signal offer all the strongest privacy features, but require the users to sign up using phone numbers, which (often) uniquely ties to your real identity. They are trying to overcome this issue, but it is hard to do without revealing the users social graph (which they do keep private). On the other hand, Keybase and Crypho offer the option of strong authentication, linking the users either to their online social graph or using third party services like Bank ID. Keybase, Crypho, Wire and Threema let the users decide which information to share while signing up, allowing the user to choose their own order of anonymity while using the services. Note that more anonymity implies the user himself to connect with other users on the platform, as lists of contacts won't be helpful anymore.

Schemes:	FS	PCS	MP	D	A	PWC	EMK	GK
Signal	✓	✓	✓	✓	✗			✓
WhatsApp	✓	✓	✗	✓	✗			✓
Keybase	✗	✗	✗	✗	$			✓
iMessage	✗	✗	✗	✗	✗	✓	✓[†]	
Crypho	✗	✗	✗	✗	$			✓
Wire	✓	✓	✗	✓	$	✓		
Threema	✗	✗	✗	✗	$	✓	✓[†]	

Table 1: Comparing the E2EE group messaging applications. Notation: FS = Forward Secrecy, PCS = Post Compromise Security, MP = Metadata Privacy, D = Deniability, A = Anonymity, PWC = Pair-Wise Channels, EMK = Encrypted Message-Keys, GK = Group Keys, ✓= YES, ✗= NO, $ = Optional. Either PWC or GK communication is implemented, not both. Encrypted message-keys[†] are only used when sending multimedia.

References

[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, *Advances in Cryptology – EUROCRYPT 2019*, pages 129–158, Cham, 2019. Springer International Publishing.

[App20] Apple. Apple security, May 2020.

[BBG04] Nikita Borisov, Eric Brewer, and Ian Goldberg. Off-the-record communication, or, why not to use PGP. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES ’04, page 77–84, New York, NY, USA, 2004. Association for Computing Machinery.

[BG19] Colin Boyd and Kai Gellert. A modern view on forward security. Cryptology ePrint Archive, Report 2019/1362, 2019. https://eprint.iacr.org/2019/1362.

[BLS12] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology – LATINCRYPT 2012*, pages 159–176, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[CCG16] K. Cohn-Gordon, C. Cremers, and L. Garratt. On post-compromise security. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 164–178, 2016.

[CCG+18] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In *ACM CCS 18*, pages 1802–1819. ACM Press, 2018.

[Cha86] David Chaum. Showing credentials without identification: Signatures transferred between unconditionally unlinkable pseudonyms. In Franz Pichler, editor, *EUROCRYPT'85*, volume 219 of *LNCS*, pages 241–244. Springer, Heidelberg, April 1986.

[CMZ14] Melissa Chase, Sarah Meiklejohn, and Greg Zaverucha. Algebraic MACs and keyed-verification anonymous credentials. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 1205–1216. ACM Press, November 2014.

[CPZ19] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The signal private group system and anonymous credentials supporting efficient verifiable encryption. Cryptology ePrint Archive, Report 2019/1416, 2019.

[Cry20] Crypho. Crypho Security Whitepaper, May 2020.

[GUVC09] Ian Goldberg, Berkant Ustaoglu, Matthew Van Gundy, and Hao Chen. Multi-party off-the-record messaging. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *ACM CCS 09*, pages 358–368. ACM Press, November 2009.

[IET20] IETF. Messaging Layer Security, May 2020.

[Key20] Keybase. Keybase book, May 2020.

[Res18] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3, August 2018.

[Rog02] Phillip Rogaway. Authenticated-encryption with associated-data. In Vijayalakshmi Atluri, editor, *ACM CCS 02*, pages 98–107. ACM Press, November 2002.

[SHB09] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in javascript. In *2009 Annual Computer Security Applications Conference*, pages 373–381, 2009.

[Sig14] Signal. Private Group Messaging, May 2014.

[Sig16] Signal. Grand jury subpoena for Signal user data, October 2016.

[Sig17] Signal. Private contact discovery for Signal, September 2017.

[Sig18] Signal. Sealed sender for Signal, October 2018.

[Sig19] Signal. Signal private group system, December 2019.

[Sig20] Signal. Technical information and documentation, May 2020.

[Thr19] Threema. Threema Cryptography Whitepaper, January 2019.

[Wha17] WhatsApp. Whatsapp Encryption Overview, December 2017.

[Wir18] Wire. Wire Security Whitepaper, August 2018.

[Zim91] Philip R. Zimmermann. Why I Wrote PGP, June 1991.