Group Messaging for Secure Asynchronous Collaboration

Matthew A. Weidner
Churchill College

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: malw2@cl.cam.ac.uk

June 6, 2019

Declaration of originality

I, Matthew A. Weidner of Churchill College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. The word count excludes appendices.

Signed:

Date:

Acknowledgements

I would like to thank my supervisors, Dr. Alastair Beresford and Dr. Martin Kleppmann, for their time and valuable input on this project throughout the year, including feedback on multiple drafts of this dissertation. I also thank Michael Dodson for providing feedback on a draft of Chapter 4. I was supported by a Churchill Scholarship from the Winston Churchill Foundation of the USA.

Group Messaging for Secure Asynchronous Collaboration

Abstract

End-to-end encrypted applications improve users' privacy by making their data unreadable to anyone besides their intended recipients. In particular, their data is unreadable to application servers. Although end-to-end encryption is currently deployed only for messaging apps, recent academic work shows that it is possible to create end-to-end encrypted asynchronous collaborative applications, like Google Docs but without a trusted server, by layering Conflict-free Replicated Data Types (CRDTs) on top of a secure group messaging protocol. However, existing secure group messaging protocols are inadequate for CRDT-based collaboration, and they do not minimize the trusted computing base as much as possible.

This dissertation investigates secure group messaging for the purpose of supporting end-to-end encrypted asynchronous collaborative applications. Its research contributions are threefold: a novel taxonomy of existing secure group messaging protocols; Causal TreeKEM, a new protocol for managing shared encryption keys that provides strong security guarantees while supporting asynchronous group messaging without a central server; and the design of a secure group messaging protocol meeting the requirements of CRDT-based collaboration. The taxonomy improves on previous work surveying secure group messaging by dividing protocols into largely interchangeable parts instead of evaluating whole protocols, allowing one to easily create new protocols by combining parts. Causal TreeKEM improves on TreeKEM, a recent protocol, by eliminating the need for a linear order on state changes, such as adding or removing users. Finally, the secure group messaging protocol is unique in that it delivers messages in a consistent causal order to all users even in the face of malicious servers or group members, supports dynamic groups, and does not require a single central server.

In total, this dissertation represents an important step towards developing end-to-end encrypted asynchronous collaborative applications with a minimal trusted computing base.

Total word count: 14966, appendices excluded.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

End-to-end encryption is becoming the de-facto standard for messaging apps. WhatsApp[1] is a prominent example, with billions of users. End-to-end encryption improves users' privacy by making messages unreadable to anyone besides their intended recipients. In particular, messages are unreadable to the messaging server. This reduces the trusted computing base and protects against the possibility that the server may be compromised, for instance by rogue employees, hackers, or surveillance agencies. Real-world events show that server compromises are a legitimate threat, including the NSA's mass access to Internet companies' data through PRISM [1] and evidence of inappropriate employee access to data at Facebook [2] and Snap [3].

End-to-end encryption has not yet spread to applications besides direct messaging. In particular, collaborative applications, in which a group of users work together on a document without a need for manual conflict resolution, are not end-to-end encrypted. These applications are growing in importance; for example, Google Drive, which includes the Google Docs suite of collaborative applications, has nearly one billion users [4]. Adding end-to-end encryption to an application like Google Docs at first appears difficult, since the central server plays a key role in resolving conflicts between different users' edits.

However, recent academic work shows that collaborative applications can be implemented without the need for a server to mediate conflicts between users. In particular, *Conflict-free Replicated Data Types (CRDTs)* [5, 6] can be used to replicate a mutable data structure across a group of users, with the guarantee that all users see the same state once they have received the same set of updates, regardless of the order in which they received those updates. Thus only users need to read and process updates, not a central server, so that the updates and hence the shared state can be end-to-end encrypted.

In principle, it then becomes possible to build end-to-end encrypted asynchronous col-

[1]`https://www.whatsapp.com/`

laborative applications by layering CRDTs on top of a secure group messaging protocol, as described by Kleppmann et al. [7]. Here the secure group messaging protocol is responsible for broadcasting CRDT updates to the group of collaborators, so that they can maintain a shared state consisting of a monotonically growing set of updates. Each user's application then combines their set of updates to compute a shared document state. So long as the messaging protocol is secure, in that only collaborators can decrypt messages, the document state will be known only to collaborators.

However, existing secure group messaging protocols are inadequate for CRDT-based collaboration. For example, the protocols in deployed messaging apps do not include information about the causal order on messages, i.e., which messages a user received before authoring a particular message, while CRDTs require consistent causal ordering information. As another example, deployed messaging apps require all messages to pass through a central server. This rules out other desirable forms of communication, such as direct peer-to-peer connections, which enable use cases like LAN-local collaboration. Finally, while deployed messaging apps guarantee the confidentiality of messages even if the server acts maliciously, they do not guarantee other properties, like the property that all users see a consistent set of messages. It is desirable to minimize this trusted computing base, so that users can continue collaborating if servers or even other group members behave maliciously.

This dissertation investigates secure group messaging for the purpose of supporting end-to-end encrypted asynchronous collaborative applications. It does so by surveying and synthesizing existing work on secure group messaging, as well as by designing new protocols to solve problems unique to collaborative applications. Ultimately, this dissertation develops the theoretical foundation of a group messaging protocol for secure asynchronous collaboration with a minimal trusted computing base.

1.1 Contributions

I have made three main research contributions, divided into Chapters 3–5. The contributions all concern secure group messaging, specifically, *how a group of users, identified by their public keys, have a conversation with various security properties.* Note that this excludes *trust establishment*—how users learn each other's public keys and associate them with real-world identities. While trust establishment is an important part of deployed applications, with many existing protocols (see [8, §IV] for a survey), it is orthogonal to secure group messaging as defined here, and I consider it out of scope.

As my first contribution, I developed a taxonomy of existing secure group messaging protocols, presented in Chapter 3. I did this by dividing existing protocols into largely interchangeable parts that address different aspects of secure group messaging, then formulating properties that protocols for those parts may or not have and evaluating protocols with respect to those properties. In comparison to existing surveys of group messaging that evaluate protocols holistically, such as by Unger et al. [8], my taxonomy is more expressive: dividing protocols into parts with specific functions highlights more clearly what is currently possible and what may be good topics for future work, and new protocols can easily be designed at a high-level by combining parts from different existing works.

Based on this taxonomy, I identified a gap in the space of protocols for managing shared encryption keys in a group. I then filled this gap by creating a new protocol, Causal TreeKEM, which is the subject of Chapter 4. Causal TreeKEM lets a group of users efficiently share a group key while allowing users to be added or removed, and it achieves a strong security property called post-compromise security. In contrast to TreeKEM, an existing protocol on which it is based, Causal TreeKEM does not require a linear order on state changes, such as adding or removing users. This is necessary for asynchronous group messaging protocols without a trusted central server.

Finally, I designed a secure group messaging protocol meeting the requirements of CRDT-based collaboration, described in Chapter 5. The protocol combines Causal TreeKEM with other protocols identified in my taxonomy. Unlike any existing protocol, it delivers messages in a consistent causal order to all users even in the face of malicious servers or group members, supports dynamic groups, and does not require a single central server.

Chapter 2

Secure Two-party Messaging

In this chapter, we briefly describe the simpler case of secure two-party messaging. We also highlight the difficulties of moving to secure group messaging. Along the way, we introduce cryptographic concepts that serve as background for the later chapters. Our definitions follow Unger et al. [8, §V].

The fundamental goal of two-party secure messaging is to enable two users to have a secure conversation. Security in this context has multiple parts. *Confidentiality* is the property that only the two participants can compute the plaintexts of messages. *Integrity* is the property that only the two participants can send messages, i.e., the participants will reject messages that have been sent or tampered with by an outsider. *Authentication* is the property that the two participants agree on each other's identities, typically represented by public keys.

These properties are typically measured against an *active network adversary*, which can read and store all communications between the parties, send forged messages, and start parallel secure conversations, subject to realistic limits on its computational power. In the context of end-to-end encryption, servers are potentially active network adversaries.

Asymmetric encryption, as in PGP, achieves the above properties against an active network adversary. However, it does not achieve the advanced security properties of forward secrecy and post-compromise security, which are measured against an adversary who can also compromise users, learning all of their secret keys at a given point in time.

Forward secrecy is the property that an active network adversary who compromises some or all users should not be able to decrypt messages from before the compromise. Forward secrecy at the granularity of whole conversations can be achieved together with the above properties by using an authenticated Diffie-Hellman exchange to establish an *ephemeral* (short-lived) and *fresh* (not previously used) shared key for each conversation, then encrypting messages under the key using an Authenticated Encryption with Associated

Data (AEAD) scheme [9], as in TLS-DHE [10]. Forward secrecy at the granularity of individual messages can be achieved by additionally using *deterministic key ratcheting*, introduced by SCIMP [11]. In this approach, users encrypt every message with a unique key, derived from some shared key using a chain of hash function applications, and delete keys immediately after use.

Post-compromise security, also called backwards secrecy or self-healing, is the property that an active network adversary who temporarily compromises some or all users should not be able to decrypt messages indefinitely after the compromise is healed. This is useful in case an adversary temporarily accesses a device and makes a copy of its state, for example through malware or a physical inspection, but then loses access to the device. Post-compromise security can be achieved by continually refreshing the ephemeral shared key using new Diffie-Hellman exchanges, a technique introduced by OTR [12].

All of the above properties are achieved by the two-party *Signal protocol* [13, 14], which is used for two-party conversations in Signal[1] and WhatsApp. The Signal protocol is asynchronous in that users can send messages when the other user is offline, using a store-and-forward server to buffer messages. This is true even for the first message in a conversation: by downloading another user's prekey from an untrusted server, a user can establish an authenticated ephemeral shared key with no communication. Here a *prekey* is the first message of an authenticated Diffie-Hellman exchange; each user uploads a number of prekeys to an untrusted server in advance, so that other users can later use them to complete the Diffie-Hellman exchange and start a conversation immediately. The Signal protocol also achieves the property of *deniability*, which roughly states that one user cannot cryptographically prove that the other user sent any particular message.

2.1 From Two Parties to a Group

Moving from the two-party setting to arbitrary size groups introduces new challenges. A simple approach to achieve the above security properties is for users to send group messages over separate Signal protocol channels to each user, as in Signal group chats [15]; however, this is inefficient, since the sender must encrypt and send each message once per group member. Achieving properties like post-compromise security while still allowing message broadcast is a nontrivial challenge.

Additionally, moving to the group setting adds new desirable security properties. *Author authentication* is the property that users can verify who sent a particular message. Integrity and authentication imply that users always know that some intended group

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

member sent a message, but unlike in the two-party case, they do not know which group member it was, except that it was someone besides themselves. *Consistency* is the property that all users agree on what is happening in the group. This includes *participant consistency*, in which all users agree on the set of group members, and *transcript consistency*, in which all users agree on the set of messages sent so far and their ordering. Note that the adversaries for these properties may include malicious group members, who may try to forge message authors or break other users' consistency, in addition to active network adversaries, possibly including servers.

Ordering messages is also more difficult in the group setting. Typically, messaging applications display messages in a linear order. However, when there are more than two users, it is difficult to enforce a consistent linear ordering without a central server, and not all applications require a linear ordering. Thus we also consider two weaker orderings. The *causal order* is the partial order in which m precedes m' if the author of m received and processed m before sending m'. This includes the case that m and m' have the same author and m was sent before m'. The causal order is precisely what CRDTs require: to establish a consistent shared state, all users must receive messages in a consistent causal order, but they need not receive messages in the same linear order [5]. We also define the *per-sender order* as the partial order in which m precedes m' if m and m' have the same author and the author sent m before m'.

Chapter 3

Taxonomy of Secure Group Messaging Protocols

Many secure group messaging protocols already exist, both in the academic literature and in deployment. In this chapter, we survey existing protocols and organize them into a novel taxonomy.

As a starting point, we take Unger et al.'s discussion of secure group messaging protocols from 2015 [8]. Unger et al. give a comprehensive survey of secure messaging generally, reviewing academic and deployed protocols for trust establishment (verifying user identities), conversation security (the actions users take to communicate), and transport privacy (how the network layer delivers packets). Their survey includes definitions of various desirable security properties and large tables assessing existing protocols with respect to those properties. Secure group messaging protocols appear as a subset of their survey of conversation security protocols.

One downside to Unger et al.'s discussion is that they evaluate existing protocols as whole secure group messaging protocols. This disadvantages many works, which often focus on only one aspect of group messaging and ignore other aspects or implement them naively. Thus, as Unger et al. note, there is much to be gained by combining parts of different protocols. In addition, many existing protocols can be tweaked to add features or to cover use cases not considered by the original authors.

In this chapter, we aim to give a more complete picture of the secure group messaging landscape, by breaking down existing protocols into largely interchangeable parts. One can build a complete protocol by choosing their desired implementation of each part. For each part, we define a list of security and functional properties, in the style of Unger et al. We then present protocols from the literature, plus some extensions to these protocols, and evaluate them with respect to the properties. Finally, we present flowcharts for each part which include a series of questions about desired properties, ending with an optimal protocol part achieving those properties. This taxonomy should prove useful both to implementers of secure group messaging protocols and to researchers interested in designing new protocol parts.

3.1 Architecture

We adopt the high-level architecture shown in Figure 3.1. This is not the only way to create a secure group messaging protocol, but surveying the literature shows that it encompasses most reasonable protocols.

The parts are:

Messaging How are messages broadcast to the group? This involves choices like whether to use broadcast or pairwise channels, and whether there are servers to buffer messages.

Encryption key management How do users agree on symmetric keys that are used to ensure confidentiality and integrity? Note that these keys may change over time. This part includes users' authentication of other group members, so that they know who else shares the symmetric keys, but not authentication of individual messages' authors, which is handled by the next part. We assume all users know each others' long-term public signature keys through some trust establishment protocol.

Author authentication How do users prove to other users that they authored a particular message? A simple protocol for this part is to sign each message with the sender's long-term signature key; however, more complicated protocols exist which provide deniability.

Transcript consistency How do users ensure that they have consistent views of the conversation transcript? Note that depending on application requirements, users need not necessarily agree on the order of messages in the transcript.

3.2 Messaging

The messaging part provides the core functionality of a secure group messaging protocol: delivering messages that users send to the group. There are a number of ways to do this, but they differ mainly in network infrastructure or efficiency considerations, not security properties. For this reason, we discuss protocols for the messaging part only briefly, omitting systematic lists of properties and protocols.

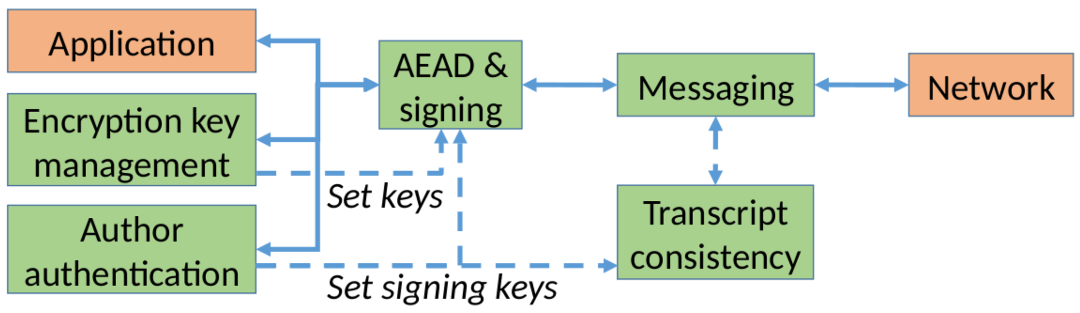


Figure 3.1: High-level architecture for secure group messaging protocols. Note that the encryption key management part functions by setting encryption keys used by the AEAD scheme, and the author authentication part may (but need not) function by setting signature keys used for message signing and/or by the transcript consistency part.

A typical protocol for the messaging part is to use a store-and-forward server to which users occasionally connect using TCP. When sending a message, a user can either give the server one copy which the server delivers to all other group members, as in WhatsApp, or they can give the server a separate direct message to deliver to each other group member, as in Signal. Alternatively, users can send messages directly over the network. If users are not always all online, they can use a *gossip protocol* (also called an *epidemic algorithm*) [16], in which pairs of users occasionally connect and synchronize their message sets over direct peer-to-peer connections. Gossip protocols are commonly used in distributed databases, such as Depot [17]; KleeQ [18] describes one for group messaging specifically.

3.3 Encryption Key Management

Encrypting messages to ensure confidentiality and integrity, so that only group members can read or generate messages, is the main characteristic distinguishing *secure* group messaging from ordinary group messaging. We now discuss how users manage encryption keys, i.e., how they negotiate, authenticate, and change them.

We assume that all users know each other's long-term public signature keys. Distributing and verifying these public keys is itself a difficult problem called *trust establishment*, but we regard it as out-of-scope; see Unger et al. [8, §IV] for a survey of approaches.

We discuss several aspects of encryption key management:

Encryption key type What type of encryption keys are used? E.g., is there a single key shared by all group members, a shared key between each pair of users, or something else?

Key negotiation How do users negotiate keys of the desired type while also authenticating the other group members? A large number of key negotiation protocols exist in the literature, going under the names *group key agreement* or *conference key agreement*; Boyd and Mathuria [19, Chapter 6] give a somewhat outdated survey.

To simplify our discussion, we choose only one key negotiation protocol for each encryption key type. Our choices are all optimized for the asynchronous setting. Specifically, we choose protocols in which users can start encrypting messages to the group immediately after creating the group or receiving a group creation message, without having to wait for round-trips with other users. Such protocols necessarily use prekeys. Also, where possible, we choose protocols that are used in existing asynchronous secure group messaging protocols.

State changes How do users change their keys at a point in time after key negotiation? We consider three kinds of state changes: adding users, removing users, and key refreshing. Changing keys is necessary when adding or removing users because removed users should no longer be able to decrypt messages, and added users should not be able to decrypt messages from before they were added. Key refreshing is the mechanism used to achieve post-compromise security. Also, when different users are assigned different keys for encrypting their own messages, key refreshing before every message is needed to achieve the property *message unlinkability preserving*—the protocol does not produce cryptographic proof that any two messages were sent by the same user. Key refreshing is used for this purpose in GOTR [20] and SYM-GOTR [21].

Deterministic key ratcheting Deterministic key ratcheting provides forward secrecy at low cost. The basic idea is to deterministically derive new encryption keys for each message, using a chain of key derivation function applications. After sending or receiving a message, a user can delete the corresponding key, which is only used once. Thus an adversary who later compromises that user cannot decrypt the message. A version of deterministic key ratcheting that supports out-of-order messages was originally designed for the two-party Signal protocol [13, §2.2]. WhatsApp uses it in the broadcast group setting [22, 15].

When used with aggressive key deletion, deterministic key ratcheting has the downside that malicious users can violate transcript consistency: if a user maliciously sends multiple messages encrypted under the same key, then each other user will only be able to decrypt one of the messages, and different users may decrypt different messages.

Participant consistency This is the property that all users agree on the set of group members. In general, this can be ensured by including group membership information in shared key derivation, like in MLS [23, §5.9], and by using authenticated pairwise channels during key negotiation.

3.3.1 Properties

We evaluate encryption key types and key negotiation protocols with respect to the properties listed below. We separate properties of encryption key types from properties of key negotiation protocols to emphasize that there may be multiple key negotiation protocols for a given encryption key type, even though Table 3.1 only evaluates one for each type.

The properties for encryption key types, including state changes but excluding negotiation, are as follows.

Allows message broadcast Plaintexts need only be encrypted under one key before being broadcast to the group. If this is false, the sender must instead encrypt the plaintext under a separate key for each other user, and then either send different messages to each user, or broadcast a single ciphertext together with its key encrypted under one key for each other user, as in Mobile CoWPI [24].

State change malicious user tolerance (group) For the protocol to function correctly, users must agree on some shared state, such as the value of shared group keys. However, for some encryption key types, a malicious user may initiate a state change (described above) that causes other users' states to diverge, causing a denial-of-service attack because the victims can no longer decrypt each others' messages. If such an attack is impossible, we give the protocol this property.

State change malicious user tolerance (sender) This is a weaker version of the above property in which for any honest user A, all users agree on the shared keys that A uses to encrypt messages, but they may disagree on the keys used by malicious users. The sender keys protocol, in which each user distributes their own sender key and uses that to encrypt messages, fulfils this property but not the previous one, since malicious users can send different versions of their own sender key to different users, but they cannot affect other users' sender keys.

State change ordering requirement The order in which users must generate and process state changes. For example, if this ordering is "Linear", there can only be a single sequence of state changes $1, 2, \ldots,$ and the user initiating state change $n + 1$ must have already processed state change n.

Key refreshing cost The broadcast communication cost of performing key refreshing for a single user. If only the refreshing user sends data, we state their communication cost; if instead key refreshing requires every user to send data, we state the cost per user and write "/user". If the protocol sends the same $O(1)$-size message to every user over pairwise Signal protocol channels, we count it as an $O(n)$-size broadcast.

Add user cost Same as key refreshing cost, but for adding a user. We exclude the cost of sending non-cryptographic group information to the added user (at least $O(n)$ to send the list of group members), as well as the broadcast cost of informing the group that a user is being added.

Remove user cost Same as add user cost, but for removing a user.

The properties for key negotiation protocols are as follows.

Malicious user tolerance (group/sender) Same as state change malicious user tolerance (group/sender), but it applies only to the initial key negotiation, not to later state changes.

Negotiation cost Same as key refreshing cost, but for the negotiation. We exclude the $O(n)$ cost of sending the list of group members and other non-cryptographic information.

3.3.2 Protocols

Table 3.1 evaluates protocols with respect to the above properties, with one choice of key negotiation protocol for each encryption key management protocol. Figure 3.2 presents the results in flowchart form, giving an algorithm that a group messaging protocol designer can use to select an encryption key management protocol given their requirements.

We now describe the evaluated protocols in detail.

Plain shared key This is a classic case considered by many key negotiation protocols in the literature.

Keys: The shared key material consists only of a single shared key.

Negotiation: The group creator generates a fresh key and sends it to each other group member over a pairwise Signal protocol channel (see Chapter 2). This is an example of a *key transport* protocol.

State changes: To refresh keys or remove users, the initiating user restarts the group with a fresh shared key, using the pairwise channels. To add a user, the adder generates a new shared key, broadcasts it to the group encrypted under the existing shared key, and sends it to the added user over a new pairwise channel.

Pairwise channels This key type is used by Signal [15], which also uses the negotiation and key refreshing protocols described here.

Protocol	**Properties (Key types)**							**Properties (Negotiation)**		
	Allows message broadcast	*State change malicious user tolerance (group)*	*State change malicious user tolerance (sender)*	*State change ordering requirement*	*Key refreshing cost*	*Add user cost*	*Remove user cost*	*Malicious user tolerance (group)*	*Malicious user tolerance (sender)*	*Negotiation cost*
Plain shared key	●	-	-	Causal	$O(n)$	$O(1)$	$O(n)$	-	-	$O(n)$
Pairwise channels	-	●	●	Per-sender	$O(n)$	$O(1)$/user	0	●	●	$O(n)$/user
Sender keys	●	-	●	Per-sender	$O(n)$/user	$O(1)$/user	$O(n)$/user	-	●	$O(n)$/user
Asymmetric key tree	●	-	-	Linear	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	-	-	$O(n) \ldots O(n \log(n))$

Table 3.1: Properties of encryption key types and key negotiation protocols for the encryption key management part of a group messaging protocol. "●": has the property; "-": does not have the property; "◐": partially has the property.

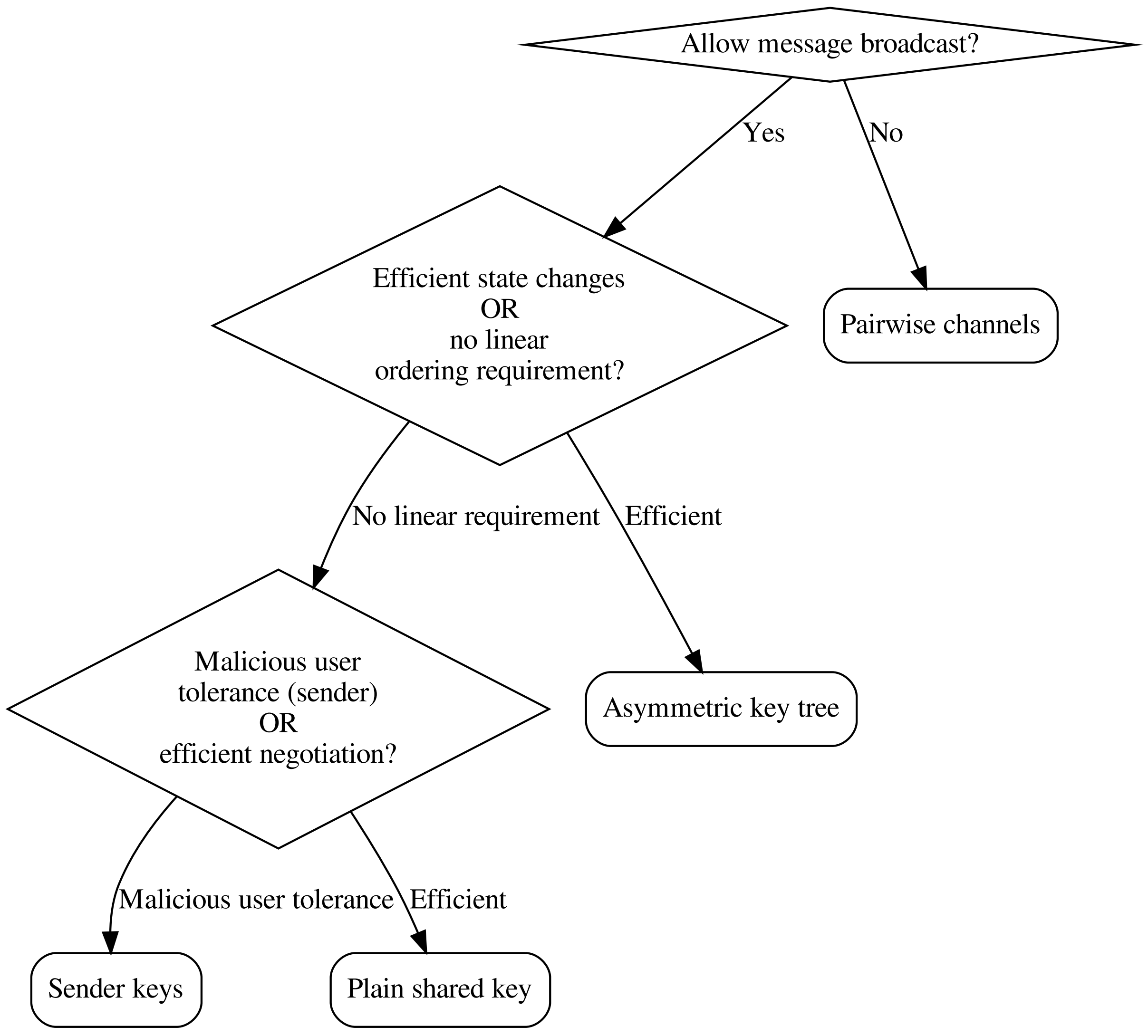


Figure 3.2: A flowchart of encryption key types, leading from desired properties to a protocol meeting those properties.

Keys: The shared key material consists of a shared key between each pair of users.

Negotiation: Each user creates a pairwise Signal protocol channel with every other user.

State changes: To refresh keys, a user refreshes the keys for each of their pairwise channels. To add a user, all other users create a pairwise Signal protocol channel with the added user. To remove a user, the remaining users stop sending messages to them.

Sender keys This key type is used by WhatsApp [22, 15], which also uses the negotiation and state change protocols described here, except that key refreshing is not used.

Keys: The shared key material consists of one *sender key* per user, known to all users but used to encrypt messages sent by that user only.

Negotiation: Before sending their first message, a user generates a fresh sender key and sends it to each other group member over a pairwise Signal protocol channel.

State changes: To refresh keys or remove users, every user (not just the refreshing user) restarts the group with a fresh sender key. To add a user, every user uses a new pairwise channel to send the added user their next message encryption key, from which the added user can compute all future message encryption keys via deterministic key ratcheting.

Asymmetric key tree A specific version of this key type, a Diffie-Hellman tree, was first described by Steer et al. [25]. Kim et al. later discussed how to handle dynamic groups in the synchronous setting [26]. Recently, Cohn-Gordon et al. proposed Asynchronous Ratcheting Tree (ART), a variant that permits asynchronous key refreshing [27]. ART was the first efficient and asynchronous post-compromise secure group key agreement protocol. ART's core ideas have since been developed into TreeKEM, which has better support for dynamic groups. TreeKEM forms the core of the Messaging Layer Security (MLS) draft IETF standard [23], and it is the source of the negotiation and state change protocols described below.

Keys: Users arrange themselves as the leaves of a rooted left-balanced binary tree. Each node of the tree, including the leaves but excepting the root, is labelled with an asymmetric key pair, with the private key known to all users below the node and the public key known to all users. The root is labelled with a symmetric key known to all users, which is used like a plain shared key. As a convenience when adding or removing users, some nodes may be blank, in which case they effectively function as the set containing their child keys.

Negotiation: The key tree starts out containing the initiator only. The initiator then adds the other users one at a time. This results in a key tree that is all blank except for the root and the leaves; the root is labelled with a key shared by all group members, while each user's leaf is labelled with an asymmetric key pair for which they alone know the private key. Over time, users perform key refreshings, eventually filling the key tree with non-blank values. The cost of initially adding the users is $O(n)$, while the key refreshings that fill the tree have a total cost of $O(n \log(n))$.

State changes: State changes adjust the tree structure to add or remove users. They also adjust keys on the path from the initiator to the root. We give a precise description of these state changes in the next chapter (Section 4.1).

3.4 Author Authentication

Unlike in the two-party case, in the secure group communication setting, message authors are ambiguous. In all of the protocols in Section 3.3 except pairwise channels, messages are sent encrypted and authenticated (e.g., with a Message Authentication Code (MAC)) under a symmetric key known to all group members. Thus a receiver knows that a message was authored by some group member besides themselves, but not which one. In this section, we discuss protocols for authenticating message authors. Note that the adversary here is a malicious insider, i.e., another group member who wishes to lie about a message's authorship. This is in contrast to most discussions of secure communication, in which group members are trusted and the only adversary is a malicious outsider.

A simple approach to author authentication is to use long-term signature keys to sign every message. However, this loses all deniability properties, motivating the other protocols below.

3.4.1 Properties

We evaluate protocols with respect to the following properties.

Authenticates authors The protocol allows users to verify message authors.

Unforgeable Malicious users cannot forge message identities even temporarily. A protocol can lack this property but still authenticate authors if users validate a message's author through a transcript consistency check after receiving the message. Users can either wait to display messages until their author is confirmed or display messages immediately and report an error if they are later found to be incorrect.

Deniability preserving Users cannot cryptographically prove to an outsider that another user authored a particular message. It may still be possible to prove that another user participated in the conversation's initial key negotiation. Note that deniability properties, such as this one and message unlinkability, are subtle and depend on the precise interactions between all protocol parts, and so these part-specific properties should be taken as guidelines only.

Message unlinkability preserving The protocol does not produce cryptographic proof that any two messages were sent by the same user. A protocol partially fulfils this property if some but not all messages may be cryptographically linked.

Malicious user tolerant keys For protocols that use signature keys, any subset of honest users agree on the signature keys of all other users, even if other users act maliciously. If this does not hold, then malicious users can give different public signature keys to different users. This does not allow them to forge authorship by honest users, but it allows them to send messages that only some subset of users consider valid. (One can use a transcript consistency protocol to ensure this property so long as the protocol is signature-free. That approach is used by mpOTR [28] but not by WhatsApp [22].)

Partition tolerant Any subset of users in a reliable network can authenticate message authors.

Asynchronous Users do not need to communicate with other users before accepting a message's author.

Cost The communication cost of the protocol compared to no author authentication. We do not include costs that are already accounted for by protocol parts listed in the restrictions property.

Restrictions Choices for other parts that must be used with this protocol.

3.4.2 Protocols

Table 3.2 evaluates protocols with respect to the above properties. Figure 3.3 presents the results in flowchart form, giving an algorithm that a group messaging protocol designer can use to select an author authentication protocol given their requirements.

We now describe the evaluated protocols in detail.

A baseline approach is to use a *trusted server* to authenticate users and indicate message authors. Another baseline approach is for users to sign messages with their *long-term signature keys*. However, this precludes deniability.

Protocol	**Properties**								
	Authenticates authors	*Unforgeable*	*Deniability preserving*	*Message unlinkability preserving*	*Malicious user tolerant keys*	*Partition tolerant*	*Asynchronous*	*Cost*	*Restrictions*
No author authentication	-	-	●	●		●	●	None	None
Trusted server	●	●	●	-		-	●	None	Messaging: Trusted server(s)
Long-term signature keys	●	●	-	-	●	●	●	Constant metadata per message	None
Ephemeral signature keys	●	●	●	If refreshed	-	●	●	One constant-sized pairwise broadcast per user per refresh; constant metadata per message	None
Ephemeral signature keys for consistency checks	●	-	●	●	-	●	●	One constant-sized pairwise broadcast per user	Transcript consistency: non-signature-free with no signatures on messages
Pairwise channels	●	●	●	●		●	●	None	Encryption key type: Pairwise channels
Pairwise digest comparison	●	-	●	Table 3.4		-	-	None	Transcript consistency: Pairwise digest comparison

Table 3.2: Properties of protocols for the author authentication part of a group messaging protocol. "●": has the property; "-": does not have the property; "◐": partially has the property.

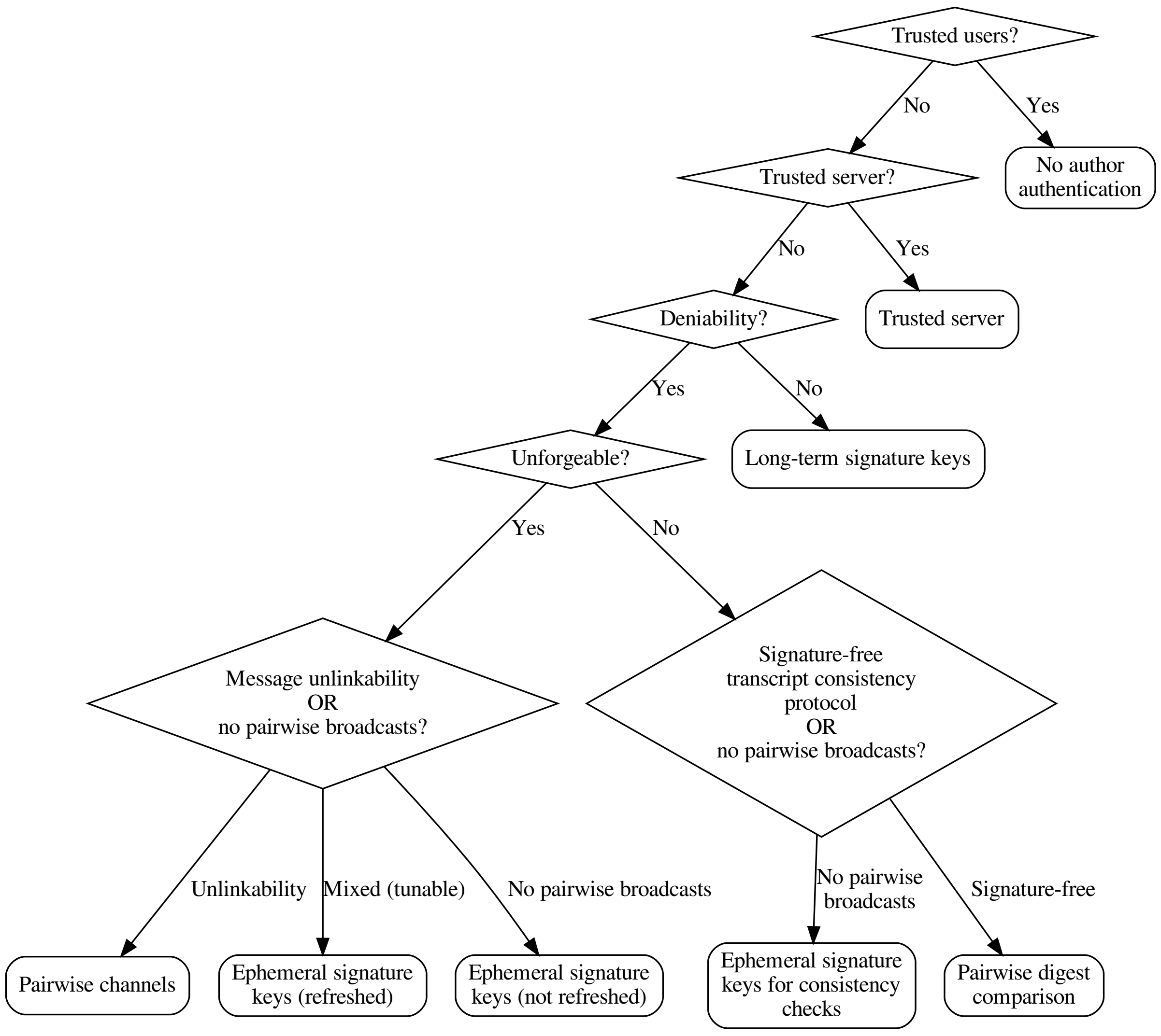

Trusted users?
No
Yes
Trusted server?
No author authentication
No
Yes
Deniability?
Trusted server
Yes
No
Unforgeable?
Long-term signature keys
Yes
No
Message unlinkability OR no pairwise broadcasts?
Signature-free transcript consistency protocol OR no pairwise broadcasts?
Unlinkability
Mixed (tunable)
No pairwise broadcasts
No pairwise broadcasts
Signature-free
Pairwise channels
Ephemeral signature keys (refreshed)
Ephemeral signature keys (not refreshed)
Ephemeral signature keys for consistency checks
Pairwise digest comparison

Figure 3.3: A flowchart of author authentication protocols, leading from desired properties to a protocol meeting those properties.

One way to gain deniability is to use *ephemeral signature keys*: each user generates an ephemeral signature key during group setup and broadcasts it to the group over pairwise Signal protocol channels, then uses that key to sign messages. This protocol was introduced by mpOTR [28] and is used by WhatsApp [22]. In the *ephemeral signature keys for consistency checks* protocol, the ephemeral keys are only used during transcript consistency checking after messages are sent, not to sign individual messages. This is used by SYM-GOTR [21].

Another approach is to use pairwise Signal protocol channels for authentication, since they are deniable. In the *pairwise channels* protocol, users send all messages other these channels. This is used by Signal [15] and Mobile CoWPI [24]. In the *pairwise digest comparison* protocol, the pairwise channels are used for transcript consistency checking only; we give a full description in Section 3.5. This is used by GOTR [20].

3.5 Transcript Consistency

In this section, we discuss *transcript consistency*, which means that all honest users agree on the set of messages sent so far and their ordering.[1] This is another security property that is important in the group setting but not the two-party setting.

Existing deployed group messaging protocols tend not to address transcript consistency. Many applications, such as WhatsApp, use the trusted server protocol described below, while Signal's group chats over pairwise channels do not check transcript consistency at all [15]. However, several protocols for ensuring transcript consistency appear in the academic literature.

We distinguish two types of adversary who may try to violate transcript consistency:

Malicious delivery service The delivery service, e.g., a network adversary or a malicious server, may drop, resend, or delay messages arbitrarily. However, we assume that cryptographic integrity checks (e.g., a MAC) prevent a malicious delivery service from altering the contents of messages.

Malicious users Malicious users in a group may arbitrarily violate the protocol in an attempt to violate transcript consistency among the other group members, or to prevent them from checking their own consistency.

[1]Our terminology differs slightly from Unger et al.[8], who define two different properties corresponding to transcript consistency, *speaker consistency* and *global transcript*, which reference the per-sender and linear orderings, respectively.

3.5.1 Properties

We evaluate protocols with respect to the following properties.

Malicious delivery service tolerant A malicious delivery service cannot cause any two honest users to believe that their message sets are consistent when they are not. A protocol partially fulfils this property if the delivery service consists of multiple servers and transcript consistency holds so long as at least one server is honest.

Malicious user tolerant (safety) Malicious users cannot cause any two honest users to believe that their message sets are consistent when they are not. If the protocol calls for users to discard messages found to be inconsistent, we additionally require that users have a consistent view of which messages should be discarded, at least eventually; otherwise the protocol only partially fulfils this property.

Malicious user tolerant (liveness) Any subset of honest users can check that their message sets are consistent and check the consistency of new messages, even if the remaining users act maliciously when sending messages or during consistency checking.

Assigns blame In case of an inconsistency, all honest users will agree on which users are at fault, even if the remaining users act maliciously during consistency checking.

Compatible orderings Message orderings that this protocol is compatible with.

Signature-free The protocol does not require signatures to accompany messages, so that some other form of author authentication may be used.

Message unlinkability preserving Same as for author authentication protocols.

Partition tolerant Any subset of users in a reliable network can check their internal transcript consistency.

Asynchronous Users do not need to communicate with other users before accepting a message's consistency.

Cost The communication cost of the protocol compared to no transcript consistency checking.

3.5.2 Protocols

Tables 3.3–3.4 evaluate protocols with respect to the above properties. Figure 3.4 presents the results in flowchart form, giving an algorithm that a group messaging protocol designer can use to select a transcript consistency protocol given their requirements.

Protocol	Properties									
	Malicious delivery service tolerant	*Malicious user tolerant (safety)*	*Malicious user tolerant (liveness)*	*Assigns blame*	*Compatible orderings*	*Signature-free*	*Message unlinkability preserving*	*Partition tolerant*	*Asynchronous*	*Cost*
No checking	-	-		-	Any	●	●	●	●	None
Trusted server	-	●	●	N/A	Any	●	●	-	●	ACKs/NACKs with server
Federated server agreement	◐	●	●	N/A	Linear	●	●	-	●	ACKs/NACKs with every server
ID chaining	●	-		-	Per-sender, causal	-	-	●	●	Constant metadata (ID) per message per immediate predecessor
Hash chaining	●	●	●	●	Per-sender, causal	-	-	●	●	Constant metadata (ID, hash) per message per immediate predecessor
Broadcast digest comparison	●	◐	-	-	Table 3.4	-	Table 3.4	-	-	One constant-sized broadcast per user per digest
Pairwise digest comparison	●	◐	-	-	Table 3.4	●	Table 3.4	-	-	One constant-sized pairwise broadcast per user per digest
Double digest comparison	●	●	-	◐	Table 3.4	-	Table 3.4	-	-	One constant-sized pairwise broadcast per user per digest, then one (# disagreements)-sized pairwise broadcast per user per digest

Table 3.3: Properties of protocols for the transcript consistency part of a group messaging protocol. "●": has the property; "-": does not have the property; "◐": partially has the property.

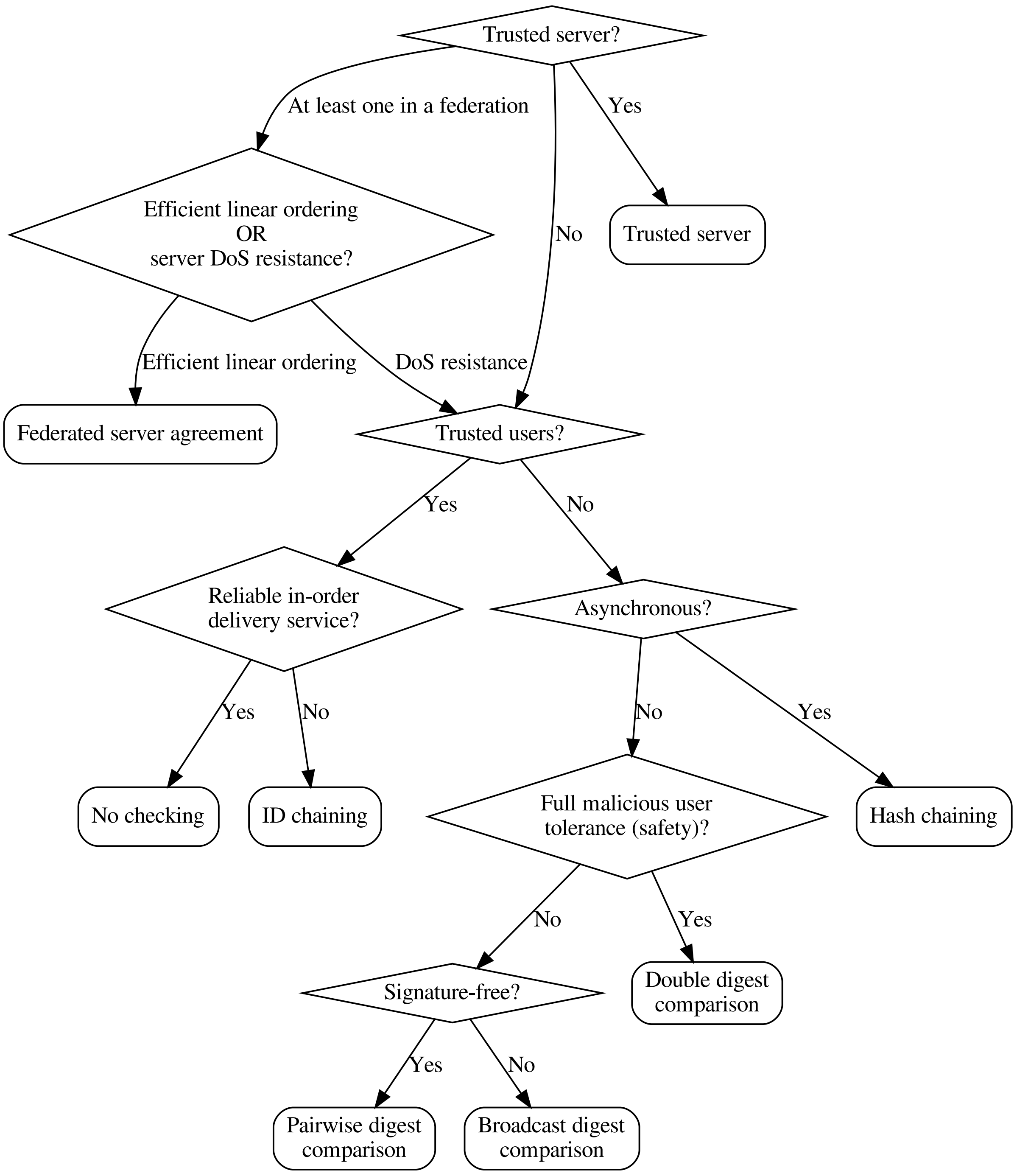

Trusted server?
At least one in a federation
No
Yes
Efficient linear ordering OR server DoS resistance?
Trusted server
Efficient linear ordering
DoS resistance
Federated server agreement
Trusted users?
Yes
No
Reliable in-order delivery service?
Asynchronous?
Yes
No
No
Yes
No checking
ID chaining
Full malicious user tolerance (safety)?
Hash chaining
No
Yes
Signature-free?
Double digest comparison
Yes
No
Pairwise digest comparison
Broadcast digest comparison

Figure 3.4: A flowchart of transcript consistency protocols, leading from desired properties to a protocol meeting those properties.

Digest	**Example**	**Properties**		
		Compatible orderings	*Message unlinkability preserving*	*Comparison frequency*
Message	SYM-GOTR [21]	Linear	●	Per-message
Transcript so far	GOTR [20]	Linear	-	Per-message
Transcript at end	mpOTR [28]	Any	-	Once at end
"Sealed block"	KleeQ [18, §5]	Per-sender, causal	◐	Per-block (about once each time every user sends a message)

Table 3.4: Properties of digest comparison protocols when used with particular digests. "●": has the property; "-": does not have the property; "◐": partially has the property.

We now describe the evaluated protocols in detail.

A baseline protocol is to use a *trusted server* to broadcast the same ciphertexts to all users, as in WhatsApp. A variant is *federated server agreement*, in which users send messages to a federation of servers, and users accept a received message only if they receive the same ciphertext from every server. This protocol was introduced by Mobile CoWPI [24].

In *ID chaining*, users assign unique numbers to each message that they author, such as a counter, and attach to each message the (author, number) pairs of its predecessors. Users request forwarding of missing predecessors from other users, in case they have been dropped by the network; this requires signatures on messages to validate their original authors. ID chaining can be used to implement *waiting causal broadcast* in the face of a malicious delivery service: messages are delivered in causal order by delaying them until their causal predecessors have been delivered [29].

Hash chaining is a variant of ID chaining in which message IDs additionally include a hash of the message itself, so that they uniquely identify messages even if a user maliciously reuses message numbers. By including a message's predecessor list when computing its hash, each message implicitly includes a structure similar to a hash chain or Merkle tree [30] of all prior messages. Various systems use hash chains to detect consistency violations, including Git [31] and fork-consistent distributed systems (e.g., SUNDR [32]).

Kleppmann et al. [7] mention hash chains in connection to transcript consistency of group messaging. Our policy of forwarding missing messages, instead of allowing permanent forks, appears in Summary Hash History [33] and Depot [17].

Digest comparison protocols instead check consistency after sending. Users can either wait to display messages until they are confirmed consistent, as in SYM-GOTR [21], or display messages immediately and report an error if an inconsistency is discovered later, as in mpOTR [28]. Note that in the former case, the protocol is neither asynchronous nor partition tolerant. In the *broadcast digest comparison* protocol, users periodically compute a hash digest of recent messages, sign it with their author authentication keys, and broadcast the signed digest to the group. Users compare all received digests for consistency. Variants of this protocol are used by mpOTR [28] and KleeQ [18]. Different choices of digest are detailed in Table 3.4. *Pairwise digest comparison* is the same, but digests are sent over pairwise Signal protocol channels to every other user instead of being signed and broadcast. This protocol is used in GOTR [20].

Double digest comparison is a variant of pairwise digest comparison that adds signatures to the digests, plus an extra round after sending digests in which users broadcast their received signed digests over pairwise channels. If a user notices that another user signed conflicting digests, they share the conflicting digests with the group and warn the application layer that not all users may have accepted the original message. This was introduced by SYM-GOTR [21].

3.6 Summary and Related Work

This chapter presented a novel taxonomy of secure group messaging protocols. The taxonomy divides protocols into largely interchangeable parts: messaging, encryption key management, author authentication, and transcript consistency. One can build a complete protocol by choosing the desired implementation of each part, using the above tables of properties and flowcharts as a guide.

To demonstrate its relevance, Table 3.5 classifies existing protocols according to the taxonomy.

Our general style of listing properties and then assessing protocols with respect to those properties follows Unger et al.'s survey [8]. We improve upon Unger et al.'s discussion of group messaging protocols by breaking protocols into parts, instead of evaluating existing works as whole protocols, and by generally discussing group messaging protocols in greater detail. In the encryption key management section, the cost evaluations in Table 3.1 partially match a similar table in the ART paper [27, Table 1], but no prior works

Protocol	**Messaging**	**Encryption key management**	**Author authentication**	**Transcript consistency**	**Notes**
WhatsApp	Server broadcast	Sender keys	Ephemeral signature keys	Trusted server	
Signal	Server pairwise sending	Pairwise channels	Pairwise channels	None	
mpOTR [28]	Direct broadcast	Plain shared key	Ephemeral signature keys	Broadcast digest comparison (transcript at end)	Ephemeral signature key exchange is consistent but synchronous
GOTR [20]	Direct broadcast	Sender keys	Pairwise digest comparison	Pairwise digest comparison (transcript so far)	Each sender key includes contributions from all users
SYM-GOTR [21]	Direct broadcast	Sender keys	Ephemeral signature keys for consistency checks	Double digest comparison	Each sender key includes contributions from all users
Mobile CoWPI [24]	Federated server pairwise sending	Pairwise channels	Pairwise channels	Federated server agreement	
KleeQ [18]	Gossip	Plain shared key	None	Broadcast digest comparison (sealed block)	Key negotiation uses implicit Diffie-Hellman tree

Table 3.5: A classification of existing secure group messaging protocols using the taxonomy.

survey author authentication or transcript consistency as we do.

Chapter 4

Causal TreeKEM: Post-Compromise Secure Group Key Agreement for Causal Broadcast

The classification of encryption key management protocols in Table 3.1 shows that TreeKEM is the most efficient protocol for a group that uses frequent state changes to add or remove users or to achieve post-compromise security. It also forms the core of the Messaging Layer Security (MLS) draft IETF standard [23], hence may be widely deployed in the future.

However, TreeKEM requires a consistent linear order on state changes. That is, there is a linear sequence of group states agreed upon by all group members, and the next state change message can only be generated by a user who knows the most recent state. When all communication happens through a trusted central server, the server can enforce an order without much difficulty, but as discussed in Section 3.2 below, not all applications have a central server. Additionally, we would like to minimize our trust in servers, for instance for the group messaging protocol in Chapter 5.

To solve this problem, I designed *Causal TreeKEM*, a novel modification of TreeKEM that requires only a consistent *causal* order on messages. A causal order is much more natural than a linear order in the asynchronous setting, especially when there is not a trusted central server. Causal TreeKEM supports arbitrary concurrent state changes while achieving strong and intuitive forward secrecy and post-compromise security guarantees.

This chapter presents Causal TreeKEM. Section 4.1 describes the necessary background on (ordinary) TreeKEM. Section 4.2 discusses the motivation for Causal TreeKEM in detail, explaining why techniques for enforcing a consistent linear order do not work well without a trusted central server. Section 4.3 introduces the main idea of Causal TreeKEM.

For space reasons, we further discuss Causal TreeKEM in Appendix A, including dynamic groups and security proofs, although the proofs are incomplete because there are currently no published security proofs for ordinary TreeKEM.

4.1 Background on TreeKEM

In this section, we describe (ordinary) TreeKEM, following the presentation in the MLS draft standard version 4 [23]. We focus only on the abstract group key agreement protocol, ignoring issues like authentication or precise message formats.

First, we fix some asymmetric cryptosystem with a *Derive-Key-Pair function* DKP mapping binary strings to asymmetric key pairs (private key, public key). As a convenience, we let priv(x) and pub(x) denote the private and public keys of DKP(x), respectively. We also fix a *Key Derivation Function* KDF mapping pairs of binary strings to binary strings, which we treat as a collision-resistant two-input hash function. We will use KDF to deterministically derive new secret values from existing ones, with a purpose string as the second argument to prevent collisions, e.g., (secret 2) = KDF((secret 1), “next”).

4.1.1 Ratchet Trees

A group of n users who wish to derive a shared group secret arrange themselves in a *ratchet tree*. This is a left-balanced binary tree with users at the leaves. Each node is labelled with three values: a secret string called its *node secret*, an asymmetric private key, and an asymmetric public key, satisfying

$$(\text{private key}, \text{public key}) = \text{DKP}(\text{node secret}).$$

See Figure 4.1 for an example. Optionally, a node's node secret may be blank, in which case its private and public keys are not defined.

When a group is first formed, its ratchet tree contains a root and a single leaf node corresponding to the group initiator. The initiator then constructs the group by adding users one at a time, as described in Section 4.1.4, creating a leaf for each user.

Assuming users follow the protocol correctly, the following invariant always holds: every user knows all node secrets, and hence private keys, on their leaf's path to the root, as well as all public keys in the tree. Users cannot compute other node secrets or private keys in the tree. Shared group secrets will be derived from root node secrets, which every user knows.

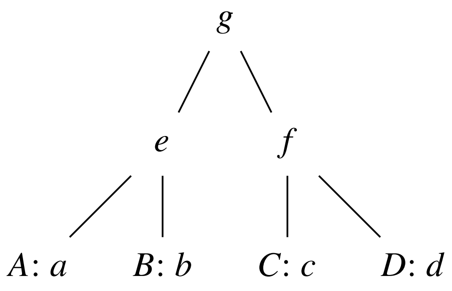


Figure 4.1: A ratchet tree with four users A, B, C, D. Here we show the node secrets at each node.

4.1.2 Key Updates

At any time, any user may perform a *key update* by sending an *Update message*. This replaces some of the node secrets in the tree, including the root node secret. Key updates are crucial to attaining forward secrecy and post-compromise security.

We define the *direct path* of a node to be the list of nodes on the path from that node to the root, including the node itself but excluding the root. The *copath* of a node is the list of siblings of nodes on its direct path. The *resolution* of a node is defined recursively by:

- The resolution of a node whose node secret is not blank is the singleton list containing the node.
- The resolution of a node with a blank node secret is the concatenation of its children's resolutions.

Examples are given in Figures 4.2–4.3.

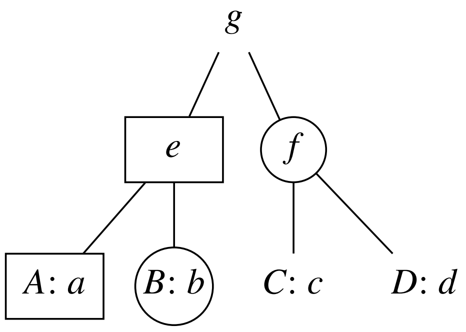


Figure 4.2: The direct path (boxed) and copath (circled) of A's leaf node.

To perform a key update, a user U first generates a fresh secret x. U then generates *path*

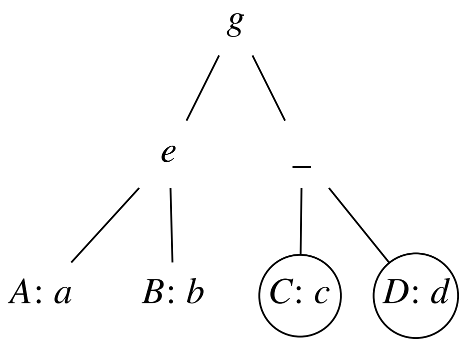


Figure 4.3: An example of a blank node's resolution. The circled nodes form the resolution of the blank node.

secrets for every node on its direct path plus the root by recursively setting

$$\begin{aligned}&\text{path secret of } U\text{'s leaf} := x\\ &\text{path secret of parent of a node } L := \text{KDF}(\text{path secret of } L, \text{“path”}).\end{aligned}$$

Then for each of these nodes, U updates its own copy of the ratchet tree by setting

$$\text{node secret of a node } L := \text{KDF}(\text{path secret of } L, \text{“node”}).$$

Note that the path secrets are temporary values used during the key update only; only the node secrets become part of the stored state. Figure 4.4 shows the secret derivation process in diagrammatic form, while Figure 4.5 shows the effect of an example update with $U = A$ in a tree with four users.

$$\begin{tikzcd}
x_1^P = x \arrow[r, "\text{KDF}(-, \text{“path”})"] \arrow[d, "\text{KDF}(-, \text{“node”})"] & x_2^P \arrow[r, "\text{KDF}(-, \text{“path”})"] \arrow[d, "\text{KDF}(-, \text{“node”})"] & x_3^P \arrow[r] \arrow[d, "\text{KDF}(-, \text{“node”})"] & \cdots \\
x_1^N & x_2^N & x_3^N &
\end{tikzcd}$$

Figure 4.4: Derivation of node secrets x_1^N, x_2^N, x_3^N and path secrets x_1^P, x_2^P, x_3^P from leaf to root, using starting secret x.

After updating the node secrets in its own state as above, U sends an Update message to the group to inform them of these changes, so that the group maintains a shared ratchet tree. The Update message contains:

(1) For each node L on U's direct path, the new public key for L, i.e., pub(new node secret for L). In the example of Figure 4.5, these public keys are $\text{pub}(x_1^N)$ for A's leaf and $\text{pub}(x_2^N)$ for the parent of A and B.

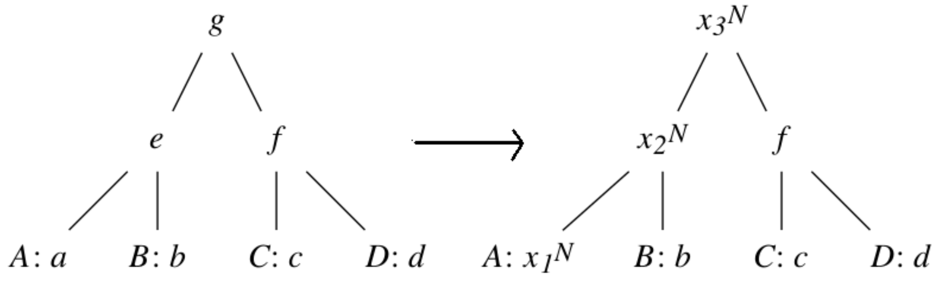


Figure 4.5: The change in a group's ratchet tree resulting from a key update by A with node secrets x_1^N, x_2^N, x_3^N from leaf to root. Here we show the tree of node secrets. Recall that users only know the node secrets on their own leaf's path to the root, but they know the public keys for all nodes in the tree.

(2) For each node M on U's copath, the new path secret for M's parent, encrypted asymmetrically under the public keys of every node in M's resolution. In the example of Figure 4.5, these encryptions are $Enc_{\text{pub}(b)}(x_2^P)$ for M equal to B's leaf and $Enc_{\text{pub}(f)}(x_3^P)$ for M equal to the parent of C and D. Here $Enc_K(x)$ denotes the encryption of x under the public key K.

See Figure 4.6 for an example. Note that when there are no blank nodes in the tree, an Update message contains one encrypted key for each node on the copath of the updating user's leaf, hence has size $O(\log(n))$.

$$\boxed{\text{pub}(x_1^N), \text{pub}(x_2^N); Enc_{\text{pub}(b)}(x_2^P), Enc_{\text{pub}(f)}(x_3^P)}$$

Figure 4.6: The Update message corresponding to the update in Figure 4.5.

When a user V receives this Update message, V uses it to update its own copy of the ratchet tree as follows:

- V sets the public keys for the nodes on U's direct path to those given in part (1) of the Update message.
- V decrypts the update's path secret for its nearest common ancestor M with U, i.e., the closest node that is an ancestor of both U's and V's leaves, from one of the encrypted values given in part (2) of the Update message. This is possible because by definition of a node's resolution, U has sent M's path secret encrypted under the public key K of some node on the path between V's leaf and M. By the invariant from Section 4.1.1, V knows the private key corresponding to K, hence can decrypt the path secret of M. In the example of Figure 4.5, B's nearest common ancestor with A is their parent, and B can decrypt x_2^P using priv(b). The nearest common ancestor of each of C and D with A is the root, and C and D can decrypt x_3^P using priv(f).

- V computes the update's path secrets for all ancestors of M, using KDF, and then computes the corresponding node secrets using KDF.

Figure 4.7 gives an example illustrating this process.

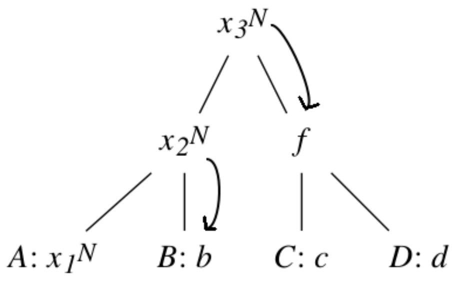


Figure 4.7: An illustration of how the TreeKEM Update message in Figure 4.6 works, i.e., how it informs other users of the new secrets. It sends (the predecessor x_2^P of) x_2^N to B by encrypting it under pub(b), while it sends (the predecessor x_3^P of) x_3^N to C and D by encrypting it under pub(f).

Observe that the key update protocol preserves the invariant from Section 4.1.1: each user knows all node secrets on their leaf's path to the root, as well as all public keys in the tree, but they cannot compute other node secrets in the tree.

4.1.3 Init and Application Secrets

The purpose of TreeKEM is to generate secrets shared by all group members, which they can then use to derive message encryption keys. To accomplish this, TreeKEM uses root node secrets to generate a chain of *init secrets*, which implicitly incorporate the root node secrets from all prior updates. Init secrets are then combined with group state information to generate *application secrets*, which are used to generate actual message encryption keys. Figure 4.8 shows this process diagrammatically. Incorporating old root secrets and group state information into the keys is necessary to provide post-compromise security and participant consistency.

4.1.4 Adding and Removing Users

Suppose a user U wishes to remove a user V from the group. To do this, they must share a new secret with all users except V, which every user except V then uses to derive a new application secret. That way, V cannot decrypt future message sent to the group.

TreeKEM implements this as follows. For a user U to remove a user V, they generate a *Remove message*, containing V's identifier and an Update message sent as if U were

$$\begin{tikzcd}
 & \text{application secret} & \\
\text{previous init secret} \arrow[r] & \cdot \arrow[u, "\text{KDF}(-,\text{“app”})"] \arrow[r, "\text{KDF}(-,\text{“init”})"] & \text{new init secret} \\
\text{Update's root secret} \arrow[ur] & & \text{Group info} \arrow[ul]
\end{tikzcd}$$

Figure 4.8: Deriving the new application secret after a TreeKEM Update message, using a chain of init secrets to incorporate the root node secrets from all prior state changes.

in V's position in the tree. Every user except V derives the new root node secret from this update and uses it to derive new init and application secrets. Then every user sets all node secrets on V's direct path plus the root to blank. Blank nodes are used here because otherwise U would know all of the new node secrets on V's direct path, violating the usual invariant. See Figure 4.9 for an example.

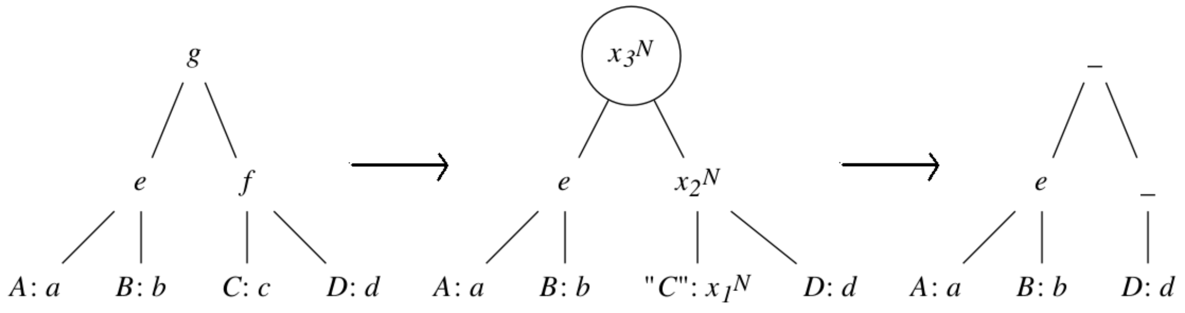


Figure 4.9: Removing a user in TreeKEM. Here A removes C with update node secrets x_1^N, x_2^N, x_3^N. The circled root node secret, x_3^N, is used to generate new init and application secrets unknown to C.

For a user U to add a user V, they send a *Welcome message* to V containing the current group state and tree of public keys and the most recent init secret. U then sends an *Add message* to the whole group, including V, which contains a public key for V. This public key could be a signed prekey that V previously uploaded to an untrusted server. Every user then:

- Adjusts the tree structure to accommodate a new leaf for V while keeping it left-balanced.
- Sets all node secrets on V's direct path plus the root to blank, and sets V's leaf's public key to the public key in the Add message.
- Derives new init and application secrets from the current init secret as in Figure

4.8, using a string of zeroes in place of the update root node secret.

Finally, V sets its own leaf node secret to the secret that was used to derive the public key in its Welcome message. It is presumed that V alone knows this node secret. See Figure 4.10 for an example.

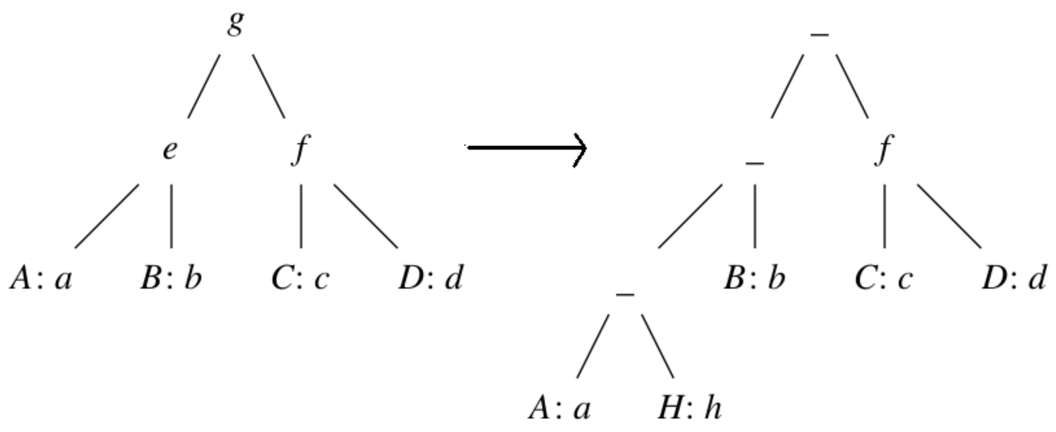


Figure 4.10: Adding a user in TreeKEM. Here A adds H with initial leaf node secret h.

Note that after processing the Welcome and Add messages, V knows all public keys, all node secrets on its direct path (namely, its leaf node secret and some blanks), and the new init and application secrets. Thus V can function as a group member.

4.1.5 Security

Current public documentation about TreeKEM does not address its security in detail. However, the intent is that TreeKEM has forward secrecy and post-compromise security, at the granularity of Update messages [34, §3.2.2.1], following the security model for ART [27]. That is, if an adversary compromises a subset of users S at a given point in time (i.e., makes a copy of their current states but then loses access to their devices), the adversary cannot compute later application secrets once all users in S send an Update message, and they cannot compute earlier application secrets from before the last time all users in S sent Update messages. In addition, removed users should not be able to compute application secrets after they are removed, and added users should not be able to compute application secrets from before they were added.

Intuitively, TreeKEM achieves the above post-compromise security guarantee as follows. An adversary who compromises some subset S of users, making a copy of their current states but then losing access to their devices, obtains the node secrets of every node that is an ancestor of a compromised user's leaf. Whenever a user makes a key update, all nodes on their direct path which do not have a separate unhealed descendant become healed, i.e., no longer known to the adversary. Hence after a subset S' of users has sent Update messages, the adversary only knows the node secrets of nodes that are an ancestor of some leaf in $S \setminus S'$. Thus once $S' \supset S$, the adversary does not know any node secrets. In particular, the root node secret is unknown to the adversary, so that its resulting application secret is unknown as well. Message encryption keys, which are derived from this application secret, are then secure.

4.2 Motivation

As mentioned in the introduction, TreeKEM implicitly assumes a consistent total order on state changes (key updates, adds, and removes). That is, there is a linear sequence of group states agreed upon by all group members, and the next state change message can only be generated by a user who knows the most recent state.

This is not just a matter of convention, and indeed, concurrent state changes can cause problems. For example, if two users concurrently send Update messages and then each add a user, the two added users share no init or application secrets in common and so cannot securely communicate.

We now discuss proposals from the MLS draft standard [23, §8] for maintaining a total order on state change messages and explain why they are inadequate.

4.2.1 Server-Enforced Ordering

In group communication platforms with a trusted central server, the server can decide on the order of state change messages and reject those that do not properly depend on their predecessor. The server can also coordinate between users; for example, if multiple users try to add new group members concurrently, the server can instruct them to act in a particular sequence, allowing all additions to succeed after a short delay.

However, this technique only works if all state change messages pass through a trusted central server, which not all group messaging protocols assume (see Section 4.2.4 below). Trust in the central server is required because a malicious server can reject all Update messages by a particular user. This allows the server to violate confidentiality forever if they compromise a user once, so that post-compromise security against the server does not hold.

4.2.2 Client-Enforced Ordering

Clients can enforce a total order on state change messages by using a protocol for *total order broadcast* (see [35] for a survey). Such a protocol would work regardless of the choice of messaging protocol, and it would not require trust in a central server, so long as users have alternative communication channels to prevent a server denial-of-service.

However, using a total order broadcast protocol adds considerable complexity. Such a protocol can also lead to long delays before state changes are committed in the total order. In particular, if the network is partitioned into two subsets of users, at most one of these subsets will be able to continue performing total order broadcast; the other will not be able to commit any state changes until the partition is healed, preventing them from achieving post-compromise security or adding and removing users. We expect long network partitions to happen frequently in the asynchronous setting, since users often go offline, making these delays unacceptable.

4.2.3 Partially Concurrent Key Updates

TreeKEM does have some support for concurrent Update messages [36, §5]. Update messages can be merged at nodes where they do not conflict, with an arbitrary winner at conflicting nodes [36, pg. 9], as shown in Figure 4.11.

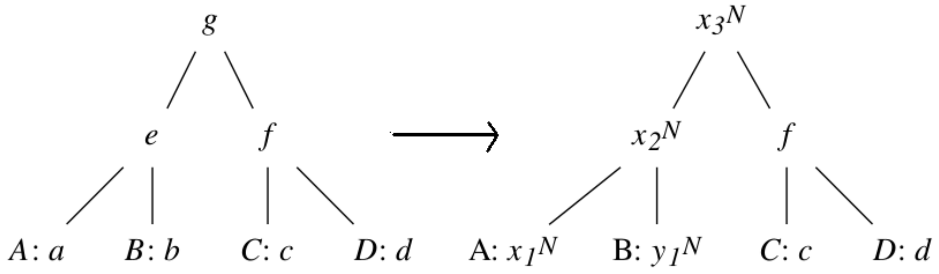


Figure 4.11: Merging updates in ordinary TreeKEM. Here A updates with node secrets x_1^N, x_2^N, x_3^N while B updates with node secrets y_1^N, y_2^N, y_3^N concurrently. Note that B learns x_2^N and x_3^N from A's Update message, so the usual invariant still holds.

If we use all of the root node secrets from multiple concurrent Update messages to generate init and application secrets in Figure 4.8, as suggested by Bhargavan et al. [36, Figure 6], then the application secret resulting from Figure 4.11 is unknown to an adversary who had previously compromised either A or B.

However, if the adversary later compromises another user, merging updates in this way reduces the post-compromise security guarantee relative to no concurrency. An example is as follows.

In the group shown in Figure 4.11, suppose the adversary compromises B, B sends an Update message, the adversary compromises C, and then C sends an Update message. Assuming TreeKEM has post-compromise security, the application secret resulting from C's update is unknown to the adversary.

Now suppose the same scenario occurs, except that A also sends an Update message concurrently to B's Update message, as in Figure 4.11. At no point is A compromised. However, now the application secret resulting from C's update is known to the adversary, as follows. From compromising B, the adversary knows b. Using this, the adversary learns x_2^N from A's Update message. Using x_2^N, the adversary learns the root node secret from C's Update message. Together with the init secret that the adversary learned by compromising C, this allows the adversary to compute the init and application secrets resulting from C's Update message.

In summary, TreeKEM's support for concurrent Update messages is not ideal, providing a weaker and less intuitive post-compromise security guarantee than for totally ordered Update messages.

4.2.4 Server-Free Use Cases

As discussed above, TreeKEM's linear ordering requirement only works well if there is a trusted central server to enforce the order. However, many potential use cases do not have a central server, trusted or otherwise. We now discuss these.

One use case is federated group communication platforms, in which there are multiple interconnected servers, typically run by different organizations. Examples include IRC and Matrix[1]. Using federated servers to enforce a total order would require the servers to coordinate total order broadcast, which adds complexity, especially if clients do not trust all servers.

Another use case is group communication systems that let clients communicate even if they are not connected to the global Internet. These include mobile ad-hoc networks, opportunistic networks, and delay-tolerant networks. FireChat[2] is one real-world example, with use cases including disaster response and censorship resistance.

A further potential use case is group applications that can work over a LAN but also connect to a central server when available. One popular application along these lines is Apple Airdrop[3], which lets users share files over a LAN. This avoids sending files to and

[1]`https://matrix.org/blog/index`
[2]`https://www.opengarden.com/firechat/`
[3]`https://support.apple.com/en-gb/HT203106`

from a cloud service and works without a connection to the global Internet. Chapter 5 targets applications of this form, and it was the original motivation for Causal TreeKEM.

4.3 Causal TreeKEM

We now present *Causal TreeKEM*, a novel modification to TreeKEM developed for this dissertation that supports arbitrary concurrent state changes. This section describes the core of the protocol, restricted to the case of static groups. We have also defined protocols for adding and removing users and written security proofs; for space reasons, these appear in Appendix A.

As in TreeKEM, we fix functions DKP, pub, priv, and KDF (see Section 4.1.1). Causal TreeKEM uses a ratchet tree, i.e., a binary tree with key pairs at each node, precisely as in TreeKEM, except that there are no node secrets in the stored state, just node key pairs. Node secrets will still play a role in Update messages.

We additionally fix binary *key combination operators* $\star_1, \star_2$, mapping a pair of asymmetric private keys (resp. public keys) to an asymmetric private key (resp. public key). We let $\star = (\star_1, \star_2)$ denote their combination acting on asymmetric key pairs. We require the following properties:

(1) If (x, X) and (y, Y) are valid key pairs, then so is

$$(x, X) \star (y, Y) = (x \star_1 y, X \star_2 Y).$$

(2) $\star$ is associative and commutative.

(3) $\star_1$ is *cancellative*: if $x \star_1 z = y \star_1 z$ for some z, then $x = y$.

Example 4.3.1. Let g be the generator of a Diffie-Hellman group with order $|g|$. Suppose we use a Diffie-Hellman based asymmetric cryptosystem, in which key pairs have the form (x, g^x). Then we can define $\star$ on key pairs by

$$(x, g^x) \star (y, g^y) := (x + y \pmod{|g|}, g^x g^y).$$

4.3.1 Key Updates

Users generate Update messages precisely as in ordinary TreeKEM. However, users process Update messages differently. Instead of treating an Update message as an instruction to *overwrite* keys with new key material, users treat it as an instruction to *combine* the new key material with the previous keys using $\star$.

Specifically, a user V receiving an Update message from a user U uses it to update its own view of the ratchet tree as follows:

- V updates the public keys for the nodes on U's direct path as

$$\text{(new public key)} := \text{(current public key)} \star_2 \text{(public key in Update message)}.$$

- V computes the update's path secret for its nearest common ancestor with U, as in ordinary TreeKEM.
- V computes the update's node secrets for all ancestors of its nearest common ancestor with U using KDF, as in ordinary TreeKEM. Then for each node L on the path from the nearest common ancestor to the root, inclusive, V sets

$$\begin{aligned}\text{(new private key for } L) := \text{(current private key for } L) \ \star_1 \\ \text{priv(update's node secret for } L).\end{aligned}$$

Figure 4.12 gives an example of a lone update (compare to Figure 4.5), while Figure 4.13 gives an example of merging two concurrent updates (compare to Figure 4.11).

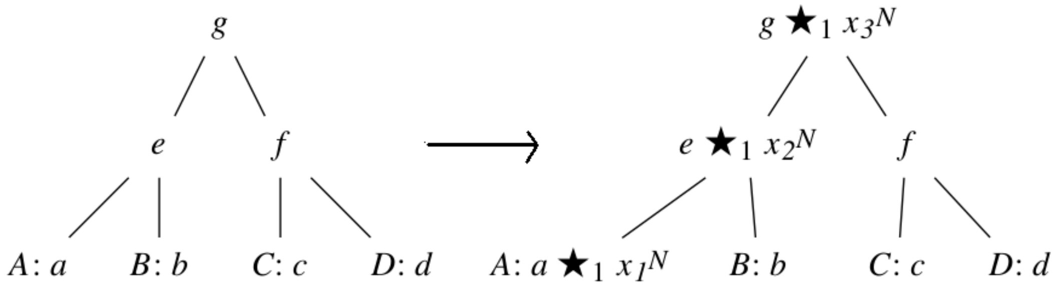


Figure 4.12: The result of a Causal TreeKEM update by A with node secrets x_1^N, x_2^N, x_3^N from leaf to root.

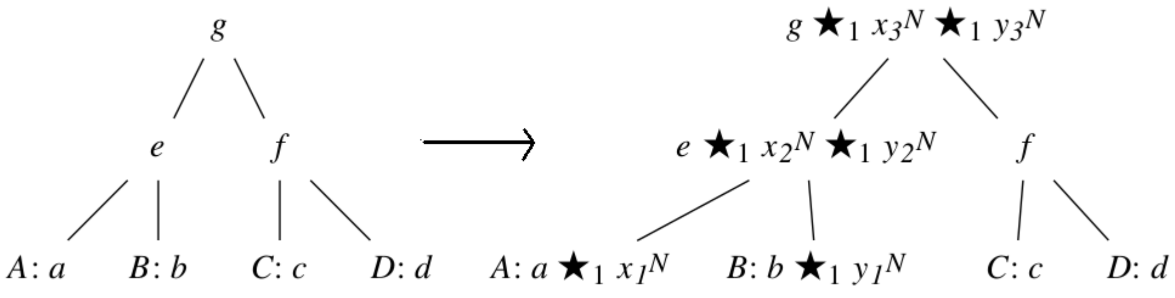


Figure 4.13: Merging updates in Causal TreeKEM. Here A updates with node secrets x_1^N, x_2^N, x_3^N while B updates with node secrets y_1^N, y_2^N, y_3^N concurrently.

Observe that the key update protocol preserves the usual invariant, except with private keys in place of node secrets: each user knows all private keys on their leaf's path to the root, as well as all public keys in the tree, but they cannot compute other private keys in the tree.

Additionally, concurrent key updates are handled in a natural way, with no need to discriminate between concurrent Update messages.

Furthermore, because $\star$ is associative and commutative, the order in which users process Update messages does not affect their final state. Any sequence of Update messages, applied in any order, yields the same tree of private and public keys.

4.3.2 Causal Ordering

Despite the associativity and commutativity of $\star$, users must still process Update messages in *causal* order, so that they have the private key needed to decrypt each Update message's nearest common ancestor path secret.

To describe this requirement precisely, we borrow some definitions from the concurrency literature. Let $<$ denote the causal order on state change messages.
Definition 4.3.2. Two messages m, m' are *concurrent* if they are incomparable under $<$, i.e., neither $m < m'$ nor $m' < m$. If $m < m'$, then m is a *causal predecessor* of m', and m' is a *causal successor* of m. The *direct causal predecessors* of m are the maximal elements of $\{l \mid l < m\}$.
Definition 4.3.3. A *configuration* is a set S of state change messages that is *downwards-closed*: if $m' \in S$ and $m < m'$, then $m \in S$. We define a *user configuration* to be a pair (U, S), where U is a user and S is a configuration. The *ratchet tree of* (U, S) is U's view of the ratchet tree after processing exactly the messages in S. The *root private key of S* is the root's private key in the ratchet tree of (U, S) for any user U; note that this does not depend on our choice of U, since all users share the root private key.

We now mandate that when a user U sends a state change message m, such as an Update message, they include a description of the set S of state change messages that they have already processed. This description could be a list of the hashes of the direct causal predecessors of m. Note that S is necessarily a configuration, and it is precisely the set of causal predecessors of m. When a user V receives this message, they compute the ratchet tree of (V, S) and use its private keys to decrypt the nearest common ancestor path secret in m.

Now all that we require of the underlying messaging protocol is *causal broadcast*, in which messages are delivered in causal order to all users. This is much easier to implement than totally ordered broadcast [29]. For example, the ID chaining and hash chaining transcript consistency protocols described in Section 3.5 enforce causal broadcast in an asynchronous setting. Hash chaining does so even in the face of malicious users.

4.4 Application Secrets

In contrast to TreeKEM, in Causal TreeKEM, the root private keys in a given state depend on all causally prior key updates. Thus we propose to use root private keys directly instead of generating separate init secrets.

Now suppose a user U wishes to encrypt an application message (not a state change message). Let S be the set of state change messages that U has processed so far. Then U generates the application secret for S by applying some key derivation function to the root private key for S and group state information, as shown in Figure 4.14. U then generates message encryption keys from the application secret, as in ordinary TreeKEM. Note that U must attach some description of S to its message, necessarily as plaintext, so that recipients know which version of the root private key and group state to use.

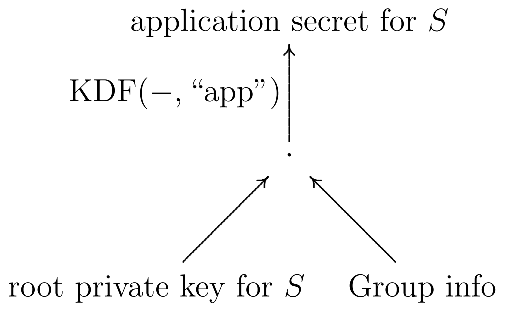


Figure 4.14: Deriving application secrets in Causal TreeKEM.

A user V receiving a message tagged with S computes the root private key of S and uses this to generate the application secret for S and then the message encryption key.

4.5 Summary and Related Work

This chapter presented Causal TreeKEM, a modification of the TreeKEM encryption key management protocol that supports arbitrary concurrent state changes. Causal TreeKEM has efficient support for adding and removing users and post-compromise security in a causal broadcast setting. This is necessary for asynchronous group messaging without a trusted central server.

TreeKEM and its predecessor, ART [27], are the only existing protocols that consider post-compromise security in a broadcast group setting. Both require a consistent linear order on state change messages. Even omitting post-compromise security, relatively few works have considered the problem of key agreement in a dynamic group. Key agreement protocols that do allow dynamic groups, such as GDH.2–3 [37] and TGDH [38], all require a consistent linear order on group membership changes, and they are synchronous because they require specific users to participate in group membership changes.

Chapter 5

From Secure Group Messaging to Secure Collaboration

5.1 Scenario

The TRVE Data project [7] seeks to develop an end-to-end encrypted collaborative document editing application, like Google Docs but without a trusted server. Existing work shows how to do this given a secure group messaging protocol with certain properties [39, 6]. Specifically, CRDTs allow a group of users to maintain a shared mutable document state by broadcasting CRDT update messages to the group, with updates resolved in a natural and communication-free way. As long as the update messages are end-to-end encrypted, the resulting document state is end-to-end encrypted as well. Thus it remains to design an appropriate secure group messaging protocol to broadcast the update messages.

Design goals for such a protocol, both for TRVE Data and for secure asynchronous collaboration in general, include:

Consistent causal broadcast Messages must be delivered in a consistent causal order to all users, but not necessarily a consistent linear order. Using CRDTs, this is necessary and sufficient for users to see a consistent view of their collaborative work [5].

Large, dynamic groups Groups can be large, and users can be added and removed.

Minimal trust Networks, servers, and even other users are not trusted to behave honestly or to follow the protocol specification. Specifically, the protocol should have confidentiality and integrity against powerful malicious networks and servers, it should maintain author authentication and transcript consistency in the face of malicious group members and outsiders, and it should resist denial-of-service attacks by malicious group members and outsiders.

Not just a central server Users can communicate using channels besides a single central server, such as alternative servers or direct peer-to-peer connections. In particular, a subgroup of users connected to each other but not the global Internet can continue collaborating and synchronize their state with the rest of the group later.

Mobile support The protocol is asynchronous and both communication- and computation-efficient, so that users can collaborate using mobile devices.

In this chapter, we use the taxonomy from Chapter 3 to design a secure group messaging protocol fulfilling the above goals. We do this both to construct a practical protocol and to illustrate the taxonomy's utility.

5.2 Design Choices

We choose protocol parts by following the flowcharts in Chapter 3 (Figures 3.2–3.4), with the exception of the messaging part.

Messaging To support mobile devices, we want to use a store-and-forward server that buffers messages. However, we also want a backup communication method in case the server performs a denial-of-service attack or users want to collaborate offline. To combine these two approaches without redundancy, we choose a gossip protocol, in which users or servers occasionally form peer-to-peer connections and synchronize their sets of messages. Typically users will only gossip with a central server, but they may also gossip with alternative servers or with other users, for example using WebRTC[1] to make direct peer-to-peer connections.

Encryption key management For efficiency in large groups, we want to *allow message broadcast.* To achieve strong security, we want post-compromise security, and we need to be able to add and remove users; thus we choose *efficient state changes.* However, because users can communicate using channels besides a single central server, we also need *no linear ordering requirement.* Thus we break from the flowchart in Figure 3.2 and instead choose Causal TreeKEM from Chapter 4, which achieves these properties simultaneously.

Author authentication As part of our goal of minimal trust, we do not assume *trusted users* or a *trusted server. Deniability* would be hard to achieve socially even if it

[1]`https://webrtc.org/`

was achieved technically, since groups may be large, and collaborative applications often keep edit histories. Thus we choose long-term signature keys.

Transcript consistency As part of our goal of minimal trust, we do not assume a *trusted server* or *trusted users*. Next, the protocol must be *asynchronous*. Thus we choose hash chaining.

5.3 A Unified Protocol

At a high level, we form a unified protocol using the above protocol parts as follows. All message exchanges occur through a peer-to-peer protocol we call *patching with hash chaining*, borrowing the term *patching* from KleeQ [18, §4]. Two peers, who can be users or servers, synchronize their message sets by performing this protocol. That is, after completing the protocol successfully, they both possess the union of their original message sets. The messages exchanged through this protocol are ciphertexts encrypted using Causal TreeKEM, plus Causal TreeKEM state change messages. Users in the group use the state change messages to compute encryption and decryption keys for the ciphertexts.

As in hash chaining, each message includes a per-author message number and a list of the (author, number, hash) triples of its direct causal predecessors. Message authors are enforced using signatures under the long-term signature keys.

Using this information, a patching with hash chaining interaction between two peers works as follows. The peers first exchange the maximum message number they have received from each user. Next, they exchange any messages which they know the other user does not have, based on the message numbers. They then check that they have the correct predecessors for each message by comparing hashes. If one peer is missing a message with a particular hash, then it must be the case that its author forked their sequence of messages, by sending two messages with the same number but different hashes. The peers then perform a binary search to identify the first forked message number, after which they forward messages that the other needs. Note that this resolves the entire fork after a number of rounds proportional to the log of the number of forked messages. Each peer that is a user then processes the messages in causal order, applying Causal TreeKEM state changes and delivering decrypted plaintexts to the application layer.

A detailed specification, including precise message formats and pseudocode for the patching with hash chaining protocol, can be found in Appendix B.

5.4 Properties

Our unified protocol broadly achieves the design goals listed in Section 5.1. We now describe some of its specific properties.

Functionally, the protocol delivers application plaintexts to the application layer in causal order. It allows for multiple forms of communication, including multiple servers and direct peer-to-peer connections. It is also asynchronous, and it is efficient: encrypted messages have small overhead, state change messages have size logarithmic in the group size, and gossiping is round- and communication-efficient.

For security, the protocol achieves strong confidentiality and integrity properties, including post-compromise security. It also guarantees author authentication, participant consistency, and consistent causal broadcast, even in the face of malicious servers or users.

However, there are some desirable properties that the protocol does not yet achieve. These are good areas for future work. One issue is that malicious users can still launch a denial-of-service attack due to limitations of Causal TreeKEM, which are inherited from ordinary TreeKEM. Specifically, they can send a bad Update message that does not allow all users to learn the new shared keys. Preventing this is an area of active research [40]. For now, it does hold that a user affected by such an attack can expose the malicious user.

Another missing property is forward secrecy. Users cannot delete their old Causal TreeKEM keys until they are sure that they have received all concurrent state change messages, which may take an arbitrary amount of time. These old keys can be used, together with a past compromise, to read old messages. Even without a past compromise, we cannot achieve forward secrecy at the granularity of individual messages by using deterministic key ratcheting and deleting message decryption keys, as described in Section 3.3. This is because a malicious user may send multiple messages with the same number, hence the same encryption key. A lack of forward secrecy may be acceptable for secure collaboration, since users will likely store a full plaintext history anyway.

Finally, the protocol does not have metadata confidentiality. Currently, the message ordering and authors are visible to non-users, such as the server. However, fixing this seems technically difficult; in particular, to ensure transcript consistency, newly added users need to be able to see this metadata for recent messages that they should not be able to decrypt, precluding metadata encryption.

5.5 Summary and Related Work

In this chapter, we presented a secure group messaging protocol meeting the requirements of CRDT-based collaboration. The protocol allows for flexible, asynchronous, and efficient communication, and it achieves strong security properties even in the face of malicious servers or users.

Ignoring the cryptographic properties, our protocol is an example of a *gossip protocol for optimistic replication*. See Saito and Shapiro [41] for a survey on this topic. Patching with hash chaining solves a special case of the *set reconciliation problem* introduced by Minsky et al. [42], in which two agents wish to merge a set with minimal communication. In our case, we have extra structure given by the partial order on the message set, which allows our protocol to be communication-efficient while avoiding the high computational complexity of existing set reconciliation protocols.

A few works on distributed systems use hashes to ensure consistent causality tracking in a gossip protocol, starting with Summary Hash History (SHH) [33, 43]. SHH does not include a practical set reconciliation mechanism. Patching with hash chaining is instead closely based on the Depot distributed key-value store [17], including in its use of a binary search to resolve forks. We make some optimizations for the messaging setting, such as using a list of direct causal predecessors instead of full version vectors in every message, since version vectors have size linear in the group size.

Chapter 6

Conclusion

End-to-end encryption improves users' privacy by reducing the trusted computing base, eliminating the need for users to trust servers' confidentiality. However, it has not yet spread to collaborative applications, such as Google Docs. CRDTs make it possible to extend end-to-end encryption to these collaborative applications by building on top of a secure group messaging protocol. However, existing secure group messaging protocols lack necessary properties for this task, and they do not minimize the trusted computing base as much as they could.

This dissertation investigated secure group messaging for the purpose of supporting end-to-end encrypted asynchronous collaborative applications. First, a taxonomy of existing protocols was developed by dividing them into largely interchangeable parts. Based on a gap identified from this taxonomy, Causal TreeKEM was created as a new protocol for managing encryption keys in a dynamic group without a central server. Finally, by combining Causal TreeKEM with protocols identified in the taxonomy, a secure group messaging protocol was designed meeting the requirements of CRDT-based collaboration. In particular, the protocol delivers messages in a consistent causal order to all users even in the face of malicious servers or users, supports dynamic groups, and does not require a single central server. Thus this dissertation is an important step towards developing end-to-end encrypted asynchronous collaborative applications with a minimal trusted computing base.

For future work, it would be interesting to improve the final protocol so that it is denial-of-service resistant even in the face of malicious group members. Additionally, it is desirable to add forward secrecy and metadata confidentiality, although these appear technically difficult to achieve.

The next major step towards end-to-end encrypted applications is to implement an appropriate secure group messaging protocol, such as this dissertation's final protocol, and integrate it with a CRDT library, such as Automerge[1]. However, converting secure protocol specifications into secure implementations is a difficult software engineering task.

[1]`https://github.com/automerge/automerge`

Bibliography

[1] Glenn Greenwald and Ewen MacAskill. NSA Prism program taps in to user data of Apple, Google and others. *The Guardian*, 2016. `https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data`.

[2] Joseph Cox and Max Hoppenstedt. Sources: Facebook has fired multiple employees for snooping on users. *Motherboard: Tech by Vice*, May 2018. `https://www.vice.com/en_us/article/bjp9zv/facebook-employees-look-at-user-data`.

[3] Joseph Cox. Snapchat employees abused data access to spy on users. *Motherboard: Tech by Vice*, May 2019. `https://www.vice.com/en_us/article/xwnva7/snapchat-employees-abused-data-access-spy-on-users-snaplion`.

[4] Frederic Lardinois. Google Drive will hit a billion users this week. *TechCrunch*, 2018. `https://techcrunch.com/2018/07/25/google-drive-will-hit-a-billion-users-this-week/`.

[5] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[6] Martin Kleppmann and Alastair R. Beresford. A conflict-free replicated JSON datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, Oct 2017.

[7] University of Cambridge Digital Technology Group. TRVE Data: Placing a bit less trust in the cloud. `https://www.cl.cam.ac.uk/research/dtg/trve/`. Retrieved 5 November 2018.

[8] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: Secure messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249, May 2015.

[9] David McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116, January 2008.

[10] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[11] Vinnie Moscaritolo, Gary Belvin, and Phil Zimmermann. Silent circle instant messaging protocol protocol specification. Technical report, 2012.

[12] Nikita Borisov, Ian Goldberg, and Eric Brewer. 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, pages 77–84, New York, NY, USA, 2004. ACM.

[13] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. Technical Report Revision 1, November 2016. `https://signal.org/docs/specifications/doubleratchet/`.

[14] Moxie Marlinspike and Trevor Perrin. The X3DH key agreement protocol. Technical Report Revision 1, November 2016. `https://signal.org/docs/specifications/x3dh/`.

[15] Paul Rösler, Christian Mainka, and Jörg Schwenk. More is less: On the end-to-end security of group chats in Signal, WhatsApp, and Threema. In *2018 IEEE European Symposium on Security and Privacy (EuroSP)*, pages 415–429, April 2018.

[16] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM.

[17] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM Trans. Comput. Syst.*, 29(4):12:1–12:38, December 2011.

[18] Joel Reardon, Alan Kligman, Brian Agala, Ian Goldberg, and David R. Cheriton. KleeQ : Asynchronous key management for dynamic ad-hoc networks. Tech report, 2007.

[19] Colin Boyd and Anish Mathuria. *Protocols for Authentication and Key Establishment*. Springer, 1st edition, 2010.

[20] Hong Liu, Eugene Y. Vasserman, and Nicholas Hopper. Improved group off-the-record messaging. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, pages 249–254, New York, NY, USA, 2013. ACM.

[21] Michael Schliep, Eugene Y. Vasserman, and Nicholas Hopper. Consistent synchronous group off-the-record messaging with SYM-GOTR. *PoPETs*, 2018(3):181–202, 2018.

[22] WhatsApp. WhatsApp encryption overview, 2017. `https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf`.

[23] Richard Barnes, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-04, IETF Secretariat, March 2019. `https://www.ietf.org/id/draft-ietf-mls-protocol-04.txt`.

[24] Michael Schliep and Nicholas Hopper. End-to-end secure mobile group messaging with conversation integrity and deniability. Cryptology ePrint Archive, Report 2018/1097, 2018. `https://eprint.iacr.org/2018/1097`.

[25] David G. Steer, L. Strawczynski, Whitfield Diffie, and Michael J. Wiener. A secure audio teleconference system. In *Proceedings of the 8th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '88, pages 520–528, London, UK, 1990. Springer-Verlag.

[26] Yongdae Kim, Adrian Perrig, and Gene Tsudik. Communication-efficient group key agreement. In *Proceedings of the 16th International Conference on Information Security: Trusted Information: The New Decade Challenge*, Sec '01, pages 229–244, Norwell, MA, USA, 2001. Kluwer Academic Publishers.

[27] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1802–1819, New York, NY, USA, 2018. ACM.

[28] Ian Goldberg, Berkant Ustaoğlu, Matthew D. Van Gundy, and Hao Chen. Multi-party off-the-record messaging. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 358–368, New York, NY, USA, 2009. ACM.

[29] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer, 2nd edition, 2011.

[30] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[31] Scott Chacon and Ben Straub. *Pro Git.* Apress, 2nd edition, 2014. `https://git-scm.com/book/en/v2`.

[32] Jinyuan Li, Maxwell Krohn, David Mazières, Dennis Shasha, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 121–136, Berkeley, CA, USA, 2004. USENIX Association.

[33] Brent B. Kang, Robert Wilensky, and John Kubiatowicz. SHH: A light-weight mechanism for decentralized ordering correctness. In *Proc. of Berkeley EECS Annual Research Symposium*, February 2004.

[34] 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-01, IETF Secretariat, October 2018. `https://www.ietf.org/id/draft-ietf-mls-architecture-01.txt`.

[35] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

[36] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous decentralized key management for large dynamic groups. Messaging Layer Security mailing list, 2018. `https://mailarchive.ietf.org/arch/msg/mls/v1CY0jFAOVOHokB4DtNqS__tX1o`.

[37] Michael Steiner, Gene Tsudik, and Michael Waidner. Diffie-hellman key distribution extended to group communication. In *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, CCS '96, pages 31–37, New York, NY, USA, 1996. ACM.

[38] Yongdae Kim, Adrian Perrig, and Gene Tsudik. Simple and fault-tolerant key agreement for dynamic collaborative groups. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 235–244, New York, NY, USA, 2000. ACM.

[39] Martin Kleppmann, Stephan A. Kollmann, Diana A. Vasile, and Alastair R. Beresford. From secure messaging to secure collaboration. Presented at Twenty-sixth International Workshop on Security Protocols, 2018.

[40] Katriel Cohn-Gordon. Trivial DoS by a malicious client. mls-protocol GitHub repository, 2018. `https://github.com/mlswg/mls-protocol/issues/21`.

[41] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.

[42] Yaron Minsky, Ari Trachtenberg, and Richard Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49(9):2213–2218, Sep. 2003.

[43] Brent ByungHoon Kang. *S2D2: A Framework for Scalable and Secure Optimistic Replication.* PhD thesis, EECS Department, University of California, Berkeley, Oct 2004.

[44] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997.

Appendix A

Dynamic Groups and Security Proofs for Causal TreeKEM

We now continue our discussion from Chapter 4 by elaborating on Causal TreeKEM.

A.1 Dynamic Groups

A.1.1 Blank Nodes

As in TreeKEM, we allow blank nodes. The private and public key of a blank node are undefined.

We adopt the following rules for blank nodes:

- A state change message that sets a node to blank overrides all concurrent updates to that node's key pair, i.e., any state change message that is not a causal predecessor or successor.
- When computing the ratchet tree of a configuration that includes a state change message involving blanks, we treat the total contribution, of that message and all of its causal predecessors, towards the key pair at any node where it is blank, to be some fixed constant value. In other words, a blank overwrites prior keys with a constant value instead of combining with them. A natural choice for the constant value is the identity of $\star$, if it exists.

These rules have essentially the same effect as the rule, in the totally ordered update case, to always process concurrent Update messages *before* messages that introduce blank nodes, namely, Adds and Removes. Such a rule is mentioned in the Messaging Layer Security working draft [23, §8].

A.1.2 Removing Users

We define a Remove message to contain:

- A list of the removed users.
- A new secret value s, encrypted under various node public keys such that it can be decrypted by the remaining users only.

For example, if a single user is being removed, then s is sent encrypted under the public keys in (the resolutions of) the removed user's copath nodes, similar to ordinary TreeKEM's approach but without KDF applications. At the opposite extreme, if users at alternating leaves are being removed, then s is sent encrypted under the leaf public keys of the remaining users.

Each user processes a Remove message as follows:

- Decrypt s and combine DKP(s) with the root key pair using $\star$.
- Delete the removed users' leaves and set all of their direct path key pairs to blank.

See Figures A.1–A.2 for an example.

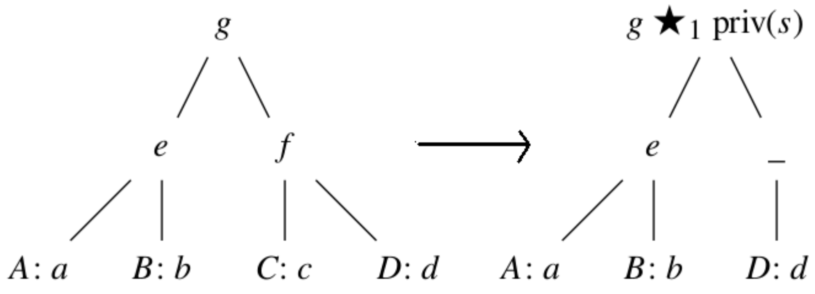


Figure A.1: Removing a user in Causal TreeKEM. Here A removes C with secret s.

$$\boxed{Enc_{\text{pub}(b)}(s), Enc_{\text{pub}(d)}(s)}$$

Figure A.2: The Remove message corresponding to the removal in Figure A.1. Here $Enc_K(x)$ denotes the encryption of x under the public key K.

Note that after concurrent Remove messages, the resulting root private key is unknown to any removed user, but two users removed in different messages can work together to determine the root private key. This issue is noted by Bhargavan et al. but not addressed by them [36, §5].

To resolve this issue, we require: if a user wishes to use their current group state, either to encrypt a communication or to generate a state change, after processing concurrent Remove messages, then they must first send a Remove message that removes the union of the concurrent messages' removed sets. This has the effect of sending a new secret to the remaining users only. (The same approach could be used with ordinary TreeKEM.)

Update messages concurrent to a Remove message are processed as usual, noting that their non-root private key contributions might be overwritten by blanks. Because they still contribute to the root private key, we intuitively expect to get the usual post-compromise security guarantee.

To prevent a removed user from altering the group state, we adopt the rule that no user will process an Update message from a removed user, unless another user generates a state change message that causally depends on it, presumably because they had not yet received the Remove message. Note that any Update messages that do get processed for this reason are harmless, since they cannot undo the Remove message's contribution to the group state.

Remark A.1.1. There remain issues with adjudicating concurrent membership changes, e.g., handling two users who remove each other, allegedly concurrently (but possibly actually one in response to the other). These seem orthogonal to the topic at hand.

A.1.3 Adding Users

A user U adds a user V as follows. First, U generates an Update message u but does not send it. Next, U sends a Welcome message to V containing:

(i) The current set of group members.

(ii) The root private key and public key tree for the ratchet tree resulting after U processes u.

(iii) U's direct path private keys in the ratchet tree of (U, S), for any configuration S such that there may exist a state change message, concurrent to the Add message (below), whose set of causal predecessors is S. Note that the direct path private keys exclude the root private key. (This information can be compressed by including the keys from some fixed configuration contained in all such S, plus the keys from all subsequent individual state changes, from which V can compute the keys for all S using $\star$.)

This Welcome message is sent encrypted under a public key that V has designated for use in encrypting Welcome messages to them. This public key could be a signed prekey that V previously uploaded to an untrusted server.

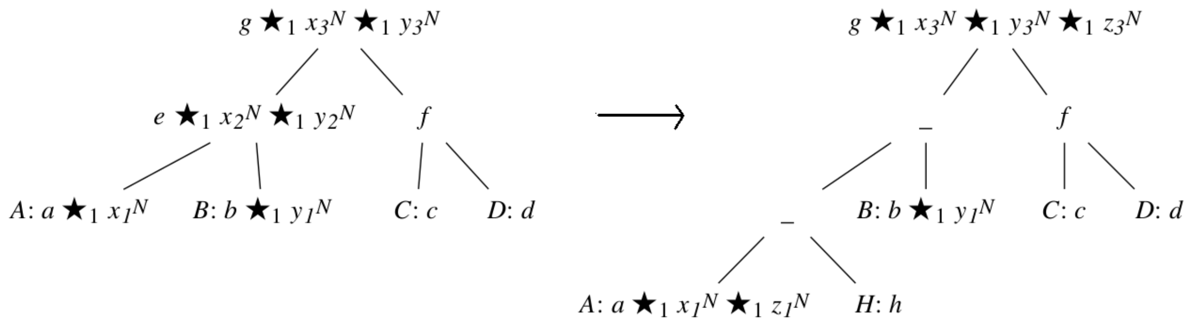
To process the Welcome message, V uses (i) and (ii) to set its initial state. V also stores (iii) so that V can decrypt the root private key contribution of any state change messages concurrent to its Add message (below), acting as if it is U, and combine those contributions with its root private key using $\star_1$. This is necessary because concurrent root private key contributions will be combined into future root private keys using $\star_1$ in the usual way, but they do not contribute to the root private key in (ii).

U then sends an Add message to the group, including V, containing u and a public key that V has designated for use as an initial leaf public key. This public key could be a second signed prekey. Every user except V processes u as a normal update. Every user, including V, then:

- Adjusts the tree structure to accommodate a new leaf for V while keeping it left-balanced.
- Sets all key pairs on V's direct path to blank, then sets V's leaf's public key to that in the Add message. Note that the root is not set to blank.

See Figures A.3–A.4 for an example.

By our rules for blank nodes, the effect of any concurrent Update message is to adjust nodes outside of V's direct path as usual, including the root node, while changes to



$g \bigstar_1 x_3^N \bigstar_1 y_3^N$
$e \bigstar_1 x_2^N \bigstar_1 y_2^N$
f
A: $a \bigstar_1 x_1^N$
B: $b \bigstar_1 y_1^N$
C: c
D: d
$g \bigstar_1 x_3^N \bigstar_1 y_3^N \bigstar_1 z_3^N$
_
f
_
B: $b \bigstar_1 y_1^N$
C: c
D: d
A: $a \bigstar_1 x_1^N \bigstar_1 z_1^N$
H: h

Figure A.3: Adding a user in Causal TreeKEM. Here A adds H with initial leaf private key h and update node secrets z_1^N, z_2^N, z_3^N, starting from the end state of Figure 4.13.

(i) A, B, C, D.
(ii) $g \star x_3^N \star y_3^N \star z_3^N$; public keys from end state of Figure A.3.
(iii)
- $S = \emptyset$: a, e
- A's update from Figure 4.13: x_1^N, x_2^N
- B's update from Figure 4.13: y_2^N

Figure A.4: The Welcome message corresponding to the addition in Figure A.3. Here $Enc_K(x)$ denotes the encryption of x under the public key K.

nodes on V's direct path will be overridden by the Add message's blank nodes. Thus we intuitively expect to get the usual post-compromise security guarantee for such an Update message. Note that V can decrypt the Update message's contribution to the root key pair using (iii) from the Welcome message.

An exceptional case is when V's Add message changes a concurrently updating user's location in the tree, either because they convert the updating user's old leaf into its new parent or because of concurrent Add messages (see below). In this case, we adopt the rule that the updating user's originally intended contribution to the root stands. That is, if the updating user generated a new secret s and they were originally at depth n in the tree, then the root key pair contribution is

$$\text{DKP}(\text{KDF}^{n+1}(s, \text{“path”}, \ldots, \text{“path”}, \text{“node”}),$$

where we have listed the second inputs to all $n + 1$ KDF applications in their order of application. We do this because all users can compute that contribution in the usual way, while they may not be able to compute

$$\text{DKP}(\text{KDF}^{n}(s, \text{“path”}, \ldots, \text{“path”}, \text{“node”})$$

if the updating user's depth decreases to $n - 1$.

Concurrent Add messages work essentially the same as an Update message concurrent to an Add message. The exception is if concurrent Add messages attempt to insert multiple users at the same leaf node. We adopt the rule: users process concurrent Add messages in some fixed order, e.g., lexicographically by name.

This works because neither the Add nor Welcome messages for a new user include any information that is specific to the leaf where they are added. Update messages which depend on an Add message may become nonsensical due to concurrent Add messages, in that they do not encrypt path secrets to the correct users, but any non-root key pairs affected by this will be overridden by blanks from the concurrent Add messages. Note that every user can still compute the root private key contribution from such an update.

A.2 Security Properties

In this section, we present security proofs for Causal TreeKEM. However, these proofs are not complete because there are currently no published security proofs for ordinary TreeKEM.

We consider passive network adversaries who also have the ability to compromise users, but who cannot actively interfere except to delay message receipt. We assume that users process state change messages in causal order regardless of the adversary's actions.

A.2.1 Post-Compromise Security

The post-compromise guarantee that we want is as follows. Let U be the set of users who have been compromised before a given point in time. Suppose there is a series of Update messages $m_1 < m_2 < \cdots < m_{|U|}$ such that:

- Each m_i was generated after the moment in which its author's most recent compromise occurred.
- The set of authors of $m_1, \ldots, m_{|U|}$ equals U.

There may be any number of intervening or concurrent updates. Then the private keys in the ratchet tree of any user who has processed $m_{|U|}$ are unknown to the adversary.

We now define security games to make our guarantee precise, inspired by the proof of ART's security [27]. In these games, we assume that the root public key is never sent on the network, so that we can reason about the root private key's secrecy.

Let Adv_i denote the maximum over all polynomial-time adversaries $\mathcal{A}$ of the *advantage* of $\mathcal{A}$ in Game i, i.e., the probability that $\mathcal{A}$ wins Game i minus 1/2.

Definition A.2.1. Let $\mathcal{C}$ be a set of user configurations, and let T be a configuration. A set of users U is *fresh at* T *w.r.t.* $\mathcal{C}$ if there is a sequence of Update messages $m_1 < m_2 < \cdots < m_k$ in T such that for every $A \in U$ and every $(A, S) \in \mathcal{C}$, there is an m_i authored by A with $m_i \notin S$. A ratchet tree node N is *fresh at* T *w.r.t.* $\mathcal{C}$ if the set of users with a leaf descended from N is fresh w.r.t. $\mathcal{C}$.

Game 1 (Security Game for Causal TreeKEM with Static Groups)**.** Fix a number of users n and a maximum number of Update messages M. The challenger initializes a group of size n with random key pairs at each node in the ratchet tree. The adversary makes a series of queries of the form:

- *Send*(A), where A is a user: Instruct A to generate an Update message, apply it to its own state, and give it to the adversary.

- *Recv*(A, m), where A is a user, m is the output of some call to *Send*, A has processed all of m's causal dependencies, and A has not yet processed m: Instruct A to process the Update message m.

- *Reveal*(A, S), where A is a user, S is a set of outputs from calls to *Send* that form a configuration, and A has previously processed all messages in S: Reveal the ratchet tree for the user configuration (A, S).

- *Test*(T), where T is a set of outputs from calls to *Send* that form a configuration: the challenger sets k_0 to be the root private key of T, sets k_1 to be a uniformly randomly sampled private key, chooses $b \in \{0, 1\}$ uniformly at random, and returns k_b.

- *Guess*(b') where $b' \in \{0, 1\}$: This terminates the game.

The adversary wins if they call *Test* exactly once, say with input T, they guess $b' = b$, the root node is fresh at T w.r.t. the set of all user configurations revealed by the adversary, and they call *Send* at most M times. Otherwise, they lose.
Remark A.2.2. The adversary should also have the ability to compromise the secrets contained in state change messages. From Causal TreeKEM's perspective, compromising such a secret is at most as bad as compromising some user's state immediately before and after sending the message, so we will be satisfied with just *Reveal* queries.
Game 2. Same as Game 1, but the adversary must specify at the start of the game the order in which they intend to make queries to *Send* and *Recv*, the user configurations on which they will call *Reveal*, and the configuration on which they will call *Test*. The inputs to *Recv*, *Reveal*, and *Test* are expressed in terms of the calls to *Send* that will generate them. If the adversary violates this specification, they lose.

In other words, the adversary must specify at the start of the game the causal dependency graph of Update messages that they will construct, plus the user configurations in the graph that they will reveal and the configuration in the graph that they will guess. We refer to the messages in this specification as *abstract messages*, since they represent Update messages but have not yet had their random values filled in.
Lemma A.2.3. *There is a non-tight reduction from Game 1 to Game 2, proving that* Adv_1 *is at most* Adv_2 *multiplied by a (large) function of* n *and* M*.*

Proof. Let us be given an adversary $\mathcal{A}$ for Game 1 with parameters n and M. We define an adversary $\mathcal{A}'$ for Game 2 which first guesses the required information, then plays as $\mathcal{A}$. If its guess turns out be wrong, $\mathcal{A}'$ plays a trivial strategy that wins with probability 1/2. The advantage of $\mathcal{A}'$ is that of $\mathcal{A}$ times the probability that its guess is correct, which is the inverse of some (large) function of n and M. □

Definition A.2.4. Define a *partial ratchet tree* to be a tree in which every node is labelled with a key pair, a public key, or null. We denote the label of a partial ratchet tree $\mathcal{T}$ at a node N by $\mathcal{T}_N$. Given two partial ratchet trees $\mathcal{T}$,$\mathcal{U}$ with the same shape, define their *product* $\mathcal{T} \star \mathcal{U}$ to be the partial ratchet tree whose label at a node N is given by

$$(\mathcal{T} \star \mathcal{U})_N := \begin{cases} (k_T, K_T) \star (k_U, K_U) & \text{if } (k_t, K_T) = \mathcal{T}_N,\ (k_U, K_U) = \mathcal{U}_N \text{ are both key pairs} \\ K_T \star_2 K_U & \text{if one of } \mathcal{T}_N, \mathcal{U}_N \text{ is a public key,} \\ & \quad \text{the other is a public key or key pair,} \\ & \quad \text{and } K_T, K_U \text{ are their public keys} \\ \mathcal{T}_N & \text{if } \mathcal{U}_N \text{ is null } (\mathcal{T}_N \text{ may also be null}) \\ \mathcal{U}_N & \text{if } \mathcal{T}_N \text{ is null.} \end{cases}$$

Given an Update message m, define the *partial ratchet tree of* m to be the partial ratchet tree that has every key pair that m contributes (i.e., the key pairs on its author's direct path plus the root) and is null elsewhere in the tree.
Game 3. Game 2 with two modifications:

- The order that the adversary commits to must be a total order on Update messages that all users agree on. That is, whenever the adversary calls *Send*(A) and receives an Update message m, they must call *Recv*(B, m) for all other users B before they call *Send* again.
- The adversary has an additional oracle:
  - *StarIn*$(\mathcal{T})$, where $\mathcal{T}$ is a partial ratchet tree with the same shape as the users' ratchet trees and every node in $\mathcal{T}$ is labelled with a key pair or null (not a public key): Instruct every user to update their ratchet tree $\mathcal{U}$ by setting $\mathcal{U} \leftarrow \mathcal{U} \star \mathcal{T}$.

For any choice of starting information I for Game 2, let *Game 2.I* denote a modified version of Game 2 in which the adversary must specify starting information I or lose, and let $Adv_{2.I}$ denote its advantage. We similarly define *Game 3.J* and $Adv_{3.J}$.

Note that any maximum-advantage adversary for Games 2 or 3 can be converted into an adversary who deterministically chooses its starting information, with the same advantage. Thus it suffices to reason about the advantages of Games 2.I and 3.J as I and J vary.
Lemma A.2.5. *For any I with n users and $\leq M$ Update messages, there is a J with n users and $\leq M$ Update messages such that there is a tight reduction from Game 2.I to Game 3.J, proving that $Adv_{2.I} = Adv_{3.J}$.*

Proof. Let us be given an adversary $\mathcal{A}$ for Game 2.I. We assume $\mathcal{A}$ and I are such that $\mathcal{A}$ never trivially fails by violating rules such as freshness.

Let $(P, <)$ be the partial order on abstract messages determined by I, i.e., the causal dependency graph that $\mathcal{A}$ plans to construct. From I, we can find a sequence of abstract messages $a_1 < a_2 < \cdots < a_k$ in P that witnesses the freshness of the configuration T on which $\mathcal{A}$ will call $Test$. Let $C = \{a_1, \dots, a_k\} \subset T$. From the total order C, we can derive a sequence of *Send* and *Recv* queries meeting the requirements of Game 3. Let J be given by: the order of *Send* and *Recv* queries is that given by C; the *Reveal* queries are those of I but with their configurations intersected with C; and the $Test$ configuration is $C = C \cap T$.

We now define an adversary $\mathcal{A}'$ for Game 3.J. $\mathcal{A}'$ poses as a challenger for Game 2.I and plays against $\mathcal{A}$, processing the queries of $\mathcal{A}'$ as follows.

- *Send*(A): Let P' be the set of (abstract) messages generated by previous calls to *Send*. Let a be the unique abstract message authored by A such that $a \notin P'$ but all of a's causal predecessors are in P. (Uniqueness holds because A always includes all of its prior messages as causal predecessors of its next message.)

  If $a \notin C$, compute the tree $\mathcal{T}$ of public keys in the ratchet tree of the configuration $\{m' \mid a' \in P, a' < a,\ a' \text{ is the abstract message for } m'\}$. Then generate an Update message m with author A using public keys from $\mathcal{T}$, using a random secret, and give this message to the adversary.

  If $a \in C$, let $R \subset P'$ be the set of messages m' for which $\mathcal{A}$ has called *Recv*(A, m'). Let a' be the predecessor of a in C, and let $Q = \{a'' \in R \mid a'' \notin C, a'' \not< a'\}$. (In case a is the minimum element of C, we set $Q = R \setminus C$.) Note that by the previous case, $\mathcal{A}'$ knows the private keys in every key pair that appears in an Update message in Q, since $Q \cap C = \emptyset$. Let $\mathcal{T}_2$ be the product of all partial ratchet trees for Update messages in Q. In Game 3.J, query *StarIn*$(\mathcal{T}_2)$, then query *Send*(A) and give its result to $\mathcal{A}$.

- *Recv*(A, m), where A is a user, m is the output of some call to *Send*, A has processed all of m's causal dependencies, and A has not yet processed m: If m was generated by a call to *Send* that reached the second case above, query *Recv*(A, m) in Game 3.J.

- *Reveal*(A, S), where A is a user, S is a set of outputs from calls to *Send*, and A has previously processed all messages in S: Query *Reveal*$(A, S \cap C)$ in Game 3.J. Interpret the answer as a partial ratchet tree $\mathcal{T}$. Let $\mathcal{T}'$ be the $\star$-product of $\mathcal{T}$ with every partial ratchet tree of an Update message in $S \setminus C$. Give $\mathcal{T}'$ to $\mathcal{A}$.

- *Test*(T), where T is a configuration: Query *Test*(C) in Game 3.J; let k be the result. Let $\mathcal{T}$ be the product of every partial ratchet tree of an Update message in $T \setminus C$, and let l be its root private key. Give $k \star_1 l$ to $\mathcal{A}$.

- *Guess*(b') where $b' \in \{0, 1\}$: Query *Guess*(b') in Game 3.J.

Essentially, we are using *StarIn* to simulate the effects of the messages in $P \setminus C$. Note that because $\star_1$ is cancellative, the guess by $\mathcal{A}$ will be correct if and only if the guess by $\mathcal{A}'$ is correct. To complete the proof, one checks that C, the input to *Test*, is fresh in Game 3.J because T is fresh in Game 2.I. □

Corollary A.2.6. *There is a tight reduction from Game 2 to Game 3, proving that* $Adv_2 \leq Adv_3$.

Game 4. Game 2 with two modifications:

- The order that the adversary commits to must be a total order on Update messages that all users agree on. That is, whenever the adversary calls *Send*(A) and receives an Update message m, they must call *Recv*(B, m) for all other users B before they call *Send* again.

- We use ordinary TreeKEM in place of Causal TreeKEM.

Game 4 should be essentially the security game for ordinary TreeKEM, but with the adversary constrained to choose its Update message order, *Reveal* queries, and $Test$ configuration at the beginning of the game. We expect that any reasonable proof of TreeKEM's security would imply that Adv_4 is negligible.

The only difference between Games 3 and 4 is that at various points in time, each user replaces their ratchet tree with the $\star$-product of that ratchet tree and some other partial ratchet tree. Sometimes this partial ratchet tree is provided by an adversary through $StarIn$; sometimes it comes from past state, since Causal TreeKEM processes Update messages using $\star$ instead of overwriting. Either way, because $\star_1$ is cancellative, these products should not affect which private keys are secret to an adversary. Thus we expect that we could modify a proof that Adv_4 is negligible to prove that Adv_3 is negligible, so that Causal TreeKEM is secure if ordinary TreeKEM is secure.
Remark A.2.7. Since internal nodes have their public keys revealed, we cannot reason about indistinguishability of their private keys from random. We may need to instead make some computational hardness assumption about $\star_1$, e.g., if it is hard for an adversary to compute y, then it is also hard to compute x given $x \star_1 y$. This follows from the cancellative property of $\star_1$ if it is easy to invert, since one can use x, $x \star_1 y$, and an inversion algorithm to compute y.

A.2.2 Forward Secrecy

In the discussion so far, we have implicitly assumed that users keep around all of the state change messages they have received forever, so that they can process arbitrarily delayed state change messages. Doing so should not harm the forward secrecy of application secrets: users need only store the node secrets of non-root nodes, and application secrets cannot be derived from these so long as users delete the corresponding path secrets. However, we have not rigorously evaluated forward secrecy.

A.2.3 Adding Users

The procedure for adding users is rather suspicious because it requires the adder to give away a good deal of private key material to the new user. In this section, we discuss why this is not a serious concern.

First, we address forward secrecy with respect to the added user. This means that the added user should not be able to read any communications that do not causally depend on their Add message.
Proposition A.2.8. *Suppose a user A adds a user B. Let S be a configuration that does not include the Add message, and let s be the root private key of S. Assume that the root private key contributions of the messages in S and the group's initial root private key are all independent and uniform random. Then as a random variable, s is independent of the private keys in parts (ii) and (iii) of B's Welcome message.*

Proof. First, write $s = v \star_1 w$, where v is the group's initial root private key and w is the combined contribution of all messages in S. By our independence assumption, v is independent of w and of part (iii) of B's Welcome message, since (iii) does not include any root private keys. Since v is uniform random and $\star_1$ is cancellative, regardless of w, s is independent of (iii).

Next, let x be the root private key in part (ii) of the Welcome message. Write $x = y \star_1 z$, where y is the root private key contribution of the Update message u used to generate the Welcome message, and z is the root private key of the preceding configuration. By our independence assumption and the assumption that S does not include the Add message, y is independent of z and s. Then x is independent of s, i.e., s is independent of the private key in (ii).

Finally, by the same reasoning as in the first paragraph, the private key x in part (ii) is independent of part (iii). Hence s, (ii), and (iii) are mutually independent, so s is independent of the private keys in (ii) and (iii) considered together. $\square$

Thus B can only determine s by breaking Causal TreeKEM, so long as KDF generates outputs that are indistinguishable from random. Note that this proposition does not help us if B compromised the group's initial root private key at some point; in principle, we should be able to prove a similar security result even if B has previously compromised some private keys.

A second concern with our protocol for adding users is its interaction with forward secrecy and post-compromise security: what happens if an adversary who already possesses some secret information gets hold of a Welcome message, either by compromising B or colluding with them? Note that such a compromise is at most as bad as if the adversary compromised A. In most update schedules, we expect that all users will store roughly the same amount of old state, for the purpose of decrypting late messages. Thus an adversary who compromises B learns at most as much as they would from compromising any other user. This seems acceptable.

In addition, compromising B does not give the adversary any private keys from the Add message's key update u, except the resulting root private key. Hence it appears that if B is compromised, revealing the Welcome message, then only B has to do a key update, not A. Thus we still get intuitive post-compromise security guarantees.

Finally, there may be issues if prekeys are reused in different conversations. Prekey reuse should be avoided if possible, but we cannot assure it asynchronously. If A gives away its initial leaf private key in an Add message in one conversation, and A is still using that leaf private key in another conversation, then A should send an Update message in the second conversation. Also, it is important that users designate which prekeys are for initial leaves and which are for Welcome messages, as we have described. If A gives away an initial leaf private key in one conversation that doubles as a Welcome encryption key in a second conversation, then the recipient automatically compromises A's initial state in that second conversation, allowing them to read early messages.

Appendix B

Group Messaging for Secure Collaboration: A Unified Protocol

We now give a detailed description of the protocol described in Chapter 5.

B.1 Terminology

We use *message* to mean a piece of data to be broadcast to the group members. Messages already use any necessary encryption and include any necessary metadata, so that they can be sent directly on the network. In contrast, we use *application plaintext* to mean a piece of data at the highest layer, i.e., a piece of data broadcast by a user's application layer to the other users' application layers. There are two types of messages:

- An *application message* encapsulates an encrypted application plaintext.
- A *state change message* encapsulates a Causal TreeKEM state change, i.e., a group initialization, key update, add, or remove.

We give precise formats for these message types in Sections B.3 and B.4.

A user assigns each message that they author a *number*, starting at zero and counting up. The *description* of a message consists of its author, number, and hash, where the *hash* is a cryptographic hash of the entire message. Barring hash collisions, descriptions uniquely identify messages, even if users act maliciously.

When defining message formats below, we include in each message a field called `predecessor info`, which consists of a list of descriptions. This defines a relation $<'$ on messages in which $m <' m'$ if the description of m appears in the `predecessor info` of m' We take $<$ to be the transitive closure of $<'$, and we say m is a *predecessor* of m' if $m < m'$. We say m is an *immediate predecessor* of m' if m is a maximal element of $\{m'' \mid m'' < m'\}$. When all users are honest, $<$ coincides with the causal order; in general, users may maliciously omit direct causal predecessors in a message's `predecessor info`, so that $<$ is a suborder of the causal order.

We also include in each message a field called `previous hash`, which consists of a hash, unless the message has number zero. We define the *previous message* of a message m

to be the message with the same author as m, number one lower, and hash given by `previous hash`.

A *peer* is either a user in the group, or a non-user who participates in the protocol to transmit messages between users, such as a server.

B.2 State

Each peer maintains an *accepted set* of messages. Informally, these are the messages that the peer has already processed, e.g., by delivering their application plaintext to the application layer. The accepted set starts out as the singleton set containing the state change message corresponding to Causal TreeKEM's group initialization message.

We assume any honest peer's accepted set satisfies the following invariants.

(i) Let m be in the peer's accepted set. Then for every description in m's `predecessor info`, there is a message in the accepted set with that description. In addition, if m has number greater than zero, then the previous message of m is in the accepted set, i.e., there is a message with the same author as m, number one lower, and hash given by `previous hash`.

(ii) The relation $<$ restricted to the accepted set is a partial order, i.e., there are no cycles.

(iii) In the relation $<$ restricted to the accepted set, no message is greater than a message with the same author and with greater or equal message number.

(iv) Every message is correctly formatted.

(v) If the peer is a user, then every message's `plaintext signature` (defined below) is a valid signature by the message's author, and any state change message can be processed without error, i.e., its `confirmation` is correct. Note that we do not require that application messages have a valid AEAD field, so that recently added users can check this property even for messages that they cannot decrypt.

Each user additionally maintains the state needed for Causal TreeKEM. This state includes long-term public signature keys for each user; how users learn these public keys is out of scope.

B.3 Application Messages

Application plaintexts are converted into application messages using a scheme similar to the one described in the MLS protocol draft [23, Section 9.2], but with additional causal predecessor information and a plaintext signature.

Specifically, we define the *application message* corresponding to `application plaintext` to be

```
 group ID, author, number, previous hash, predecessor info,
AEAD(key, nonce, application plaintext | encrypted signature, group
ID | author | number | predecessor info), plaintext signature,
```

where:

- `group ID` is a public value that uniquely identifies the group to each user.
- `author` is an identifier of the author within the group.
- `number` is the message's number.
- `previous hash` is the hash of the message by this author with number one lower, i.e., this message's previous message. If `number` is 0, this is omitted.
- `predecessor info` contains a description of each maximal message in the author's accepted set under $<$, i.e., this message's direct causal predecessors.
- `encrypted signature` is a signature of `group ID | author | number | previous hash | predecessor info | application plaintext` under the author's long-term signature key.
- `AEAD` refers to an Authenticated Encryption with Associated Data scheme [9]. We treat this as a function which inputs a symmetric key, a nonce (a unique value used alongside the key to prevent cryptanalysis), plaintext, and associated data, and outputs a ciphertext that encrypts and authenticates the plaintext under the key, and authenticates the associated data under the key without encrypting it. By "authenticate under the key", we mean that anyone possessing the key can verify that the ciphertext's author used that key to generate the ciphertext, and the ciphertext has not been tampered with since then. Here we write the arguments to `AEAD` in the order: symmetric key, nonce, plaintext, associated data.
- `key` and `nonce` are the symmetric Causal TreeKEM key and nonce corresponding to the Causal TreeKEM state induced by the accepted set, as described in Section 4.4.
- `plaintext signature` is a signature of the rest of the message under the author's long-term signature key.

The above format differs from MLS's `ApplicationMessage` in that we use `number` and `predecessor info` in place of MLS's `epoch` and `generation` numbers. Those numbers rely on MLS's linear ordering, and they do not check transcript consistency. Also, we add `plaintext signature`, while MLS includes only `encrypted signature`. Because there are no published security proofs for MLS, it is unclear whether we can omit `encrypted signature`.

B.4 State Change Messages

As with application messages, we convert Causal TreeKEM's state changes into messages using a scheme similar to the one described in the MLS protocol draft [23, §7], but with additional causal predecessor information.

Specifically, we define the *state change message* corresponding to a Causal TreeKEM state change `sc` to be

```
 sc, author, number, previous hash, predecessor info,
plaintext signature, confirmation,
```

where:

- `author`, `number`, `previous hash`, and `predecessor info` are the same as for application messages.
- `plaintext signature` is a signature, under the author's long-term signature key, of the *transcript hash* of all state changes up through and including `sc`. We define the transcript hash analogously to the `transcript_hash` in MLS [23, §5.7]. Specifically, the transcript hash of `sc` is a hash of

  `transcript hash of` $\texttt{sc}_1$`, ..., transcript hash of` $\texttt{sc}_k$`, sc,`

  where $sc_1, \ldots, sc_k$ are the maximal elements under $<$ of the accepted set's subset of state change messages, taken in lexicographic order.
- `confirmation` is

  ```
  HMAC(key, transcript hash | plaintext signature),
  ```

  where `key` is the Causal TreeKEM confirmation key[1] resulting from `sc`, and `HMAC` is a message authentication function [44]. `confirmation` authenticates this state change message, proving that its author is a group member and that the message has not been tampered with by a non-group member.

Note that `sc` includes the group ID, so we do not include it explicitly, unlike for application messages.

B.5 Adding Users

When adding a user, the adder must send some extra information to the added user along with Causal TreeKEM's Welcome message (Section A.1.3). Specifically, they must send the description of any message that may appear in the predecessor info of a state change message concurrent to the added user's Add message. This allows the added user to ignore messages that contain non-existent messages in their predecessor info, like any other user. Otherwise, the added user may accept application messages that other users ignore, violating transcript consistency of the application plaintexts. Additionally, the adder must send the confirmation key contributions for previous state change messages, so that the added user can compute the confirmation key of future state change messages.

[1]This is a secret derived from each configuration in addition to the application secret. Specifically, from each state change's root secret contribution, we derive a *confirmation contribution*; the *confirmation key* of a configuration S is the $\star_1$-combination of the confirmation contribution from all state changes in S.

B.6 Patching with Hash Chaining

In this section, we introduce the protocol *patching with hash chaining.* This is the peer-to-peer message exchange through which all group communication occurs.

Overview The goal of a patching with hash chaining interaction is for two peers to merge their accepted sets. That is, after a patching with hash chaining interaction between two honest peers, both peers' accepted sets become the union of their original accepted sets, except that if one peer is a user and the other is not, the user's set of accepted messages will omit invalid messages. Additionally, whenever an honest peer interacts with any peer, even a malicious one, patching with hash chaining preserves the invariants from Section B.2.

The core of the protocol is described in Section 5.3. It remains to give a more precise description of how users resolve forks.

Recursively define the *chain* of a message m to be m itself if m has number 0; else it is the chain of m's previous message followed by m. Thus the chain of m is a linearly ordered suborder of the predecessor order consisting of messages by the same author as m, with number going from 0 to that of m. Furthermore, barring hash collisions, the chain of a (valid) message is unambiguously defined, even in the presence of malicious users.

Two peers who encounter a fork, by identifying two messages with the same author and number but different hashes, merge the chains of the two messages, then exchange all messages that causally depend on previously missing messages. This resolves the fork. The peers merge their chains by doing a binary search on each others' chains to identify precisely which messages they are missing, taking advantage of the linear orderings. Letting M be the number of missing messages caused by a fork, resolving the fork in this way requires at most $4\log_2(M) + O(1)$ rounds, and only $O(\log(M))$ communication in excess of the communication used to send the missing messages.

The Protocol Two peers A and B perform patching with hash chaining by following the protocol in Figure B.1, using the protocol in Figure B.2 as a subroutine.

In an implementation, it is important to prevent one patching with hash chaining instance from blocking another. This can be done by making a copy of the accepted set for each protocol run and merging the results into the real accepted set when it finishes. Otherwise, a malicious peer can perform a denial-of-service attack on another peer by leaving a protocol run open for a long time.

Protocol `PatchWithHashChaining`(A: peer (initiator), B: peer):

1. A sends B a vector $\vec{a}$ containing, for each user in the group, the maximum number of a message by that author in A's accepted set. As an optimization, A may omit numbers that have not changed since the last time they patched with B.
2. B likewise sends A a vector $\vec{b}$. B also sends any messages in its accepted set whose number exceeds A's number for the corresponding author. Additionally, B sends the description of any maximal message in its accepted set that it has not just sent. As an optimization, B may omit numbers that have not changed since the last time it patched with A or that are at least A's number.

3a. A likewise sends B messages and descriptions.

3b. A first checks that every received message satisfies invariant (iv), and if not, it fails. A then arranges its received messages and its accepted set into a directed graph, with an edge to a message from each of the messages referenced in its `predecessor info` plus the message referenced in `previous hash`. If this graph has a cycle, or if invariant (iii) is violated, A fails. If some description does not have a corresponding message in A's received messages or accepted set, A applies procedure *Fix* below to the description. If *Fix* fails, A fails, else it adds the messages received during *Fix* to its received set and goes to the beginning of this step.

3c. Once all descriptions reference messages that A has, A processes the received messages in predecessor order. To process a message m, A does the following. If A is a non-user, A merely adds m to its accepted set. If A is a user, it checks invariant (v) for m; if the invariant is false, A deletes m and all of its successors from the set of received messages. (As an optimization, if A is a non-user but knows the public signature key of m's author, they may check `plaintext signature`.) Otherwise, A adds m to its accepted set. If m is a state change message, A applies it to the encryption key state. If m is an application message for which it has the decryption key, and the AEAD authenticator and `encrypted signature` are valid, then A decrypts m and delivers its application plaintext to the application layer.

4. B likewise performs steps 3b–c. This can be done in parallel to A's step 3b–c, once B receives the data from step 3a.

Figure B.1: Protocol `PatchWithHashChaining`. Numbers indicate rounds. We allow a peer to fail, in which case it terminates the protocol.

Protocol `Fix`(A: peer (initiator), B: peer, (C, n, h): description):

If $n > \vec{a}_C$ (where $\vec{a}$ is from round 1 of `PatchWithHashChaining`), A fails. Else A has some message m' in its accepted set with description (C, n, h'), where $h' \neq h$. Assuming B is honest, its accepted set likewise contains a message with description (C, n, h) for some $h \neq h'$.

A now attempts to identify the first message in the chain of m' that B lacks. It does this using a top-down binary search followed by a bounded binary search. Specifically, A first sends the hash of message number n in the chain of m' (i.e., the hash of m' itself), to which B responds "have" or "not have". If B responds "not have", A does the same for message numbers $n - 2^0$, $n - 2^1$, etc., until B responds "have" or A reaches message number 0. If A reaches message number 0 and B still replies "not have", then the first message in the chain of m' that B lacks is number 0. If B responds "have" to some $n - 2^{k+1}$ but not to $n - 2^k$, then A and B do a similar binary search in the range $(n - 2^{k+1}, n - 2^k]$, eventually finding the first message that B lacks. A then sends that message and all greater messages in the predecessor order.

The above paragraph also runs in parallel with A and B switched.

Figure B.2: Protocol `Fix`. We allow a peer to fail, in which case it terminates the protocol. There is an analogous protocol with A and B switched, using $\vec{b}$ in place of $\vec{a}$.