Key Agreement for Decentralized Secure Group Messaging with Strong Security Guarantees

Matthew Weidner[1], Martin Kleppmann[2], Daniel Hugenroth[2], and Alastair R. Beresford[2]

[1]Computer Science Department, Carnegie Mellon University
`maweidne@andrew.cmu.edu`
[2]Department of Computer Science and Technology, University of Cambridge
`{mk428,dh623,arb33}@cst.cam.ac.uk`

October 7, 2020

Abstract

Secure group messaging protocols provide end-to-end encryption for group communication. Practical protocols face many challenges, including mobile devices frequently being offline, group members being added or removed, and the possibility of device compromises during long-lived chat sessions. Existing work targets a centralized network model in which all messages are routed through a single server, which is trusted to provide a consistent total order on updates to the the group state. In this paper we adapt secure group messaging for *decentralized* networks that have no central authority. Servers may still optionally be used, but their trust requirements are reduced.

We define *decentralized continuous group key agreement* (DCGKA), a new cryptographic primitive encompassing the core of a decentralized secure group messaging protocol; we give a practical construction of a DCGKA protocol and prove its security; and we describe how to construct a full messaging protocol from DCGKA. In the face of device compromise our protocol achieves forward secrecy and post-compromise security. We evaluate the performance of a prototype implementation, and demonstrate that our protocol has practical efficiency.

1 Introduction

WhatsApp, Signal, and similar messaging apps have brought end-to-end encryption to billions of users globally, demonstrating that the benefits of such privacy-enhancing technologies can be enjoyed by users who are not technical experts. Modern secure messaging protocols used by these apps have several important characteristics:

Asynchronous: A user can send messages to other users regardless of whether the recipients are currently online. Offline recipients receive their messages when they are next online again (even if the sender is now offline). This property is important for mobile devices, which are frequently offline.

Resilient to device compromise: If a user's device is compromised, and all of that device's secret key material is revealed to the adversary, the protocol nevertheless provides *forward secrecy* (FS): any messages received before the compromise cannot be decrypted by the adversary. Moreover, protocols can provide *post-compromise security* (PCS) [12]: users periodically update their keys so the adversary eventually loses the ability to decrypt further communication. As secure messaging sessions may last for years, these properties are important for limiting the impact of a compromise.

Dynamic groups: Group members can be added and removed at any time.

In the case when only two users are communicating, the Signal protocol [36] is widely used. However, generalizations of this two-party protocol to groups of more than two users are not straightforward. For example, WhatsApp's group messaging protocol does not provide PCS [40,47]. Signal implements group messaging by sending each message individually to each group member via a two-party secure channel, which is inefficient for large groups.

Secure group messaging protocols have been the subject of much recent cryptographic work, which we summarize in Section 3. A notable example is the Messaging Layer Security (MLS) protocol, a standard under development by an IETF working group [5,35], which provides FS/PCS and is designed to scale to large groups. However, MLS assumes that all messages modifying the group state (i.e. adding/removing members or performing key updates for PCS) are delivered to all members in the same order. If two group members concurrently modify the group state, one of the requests must be rejected and retried. This total order is typically enforced by routing all messages through a centralized, semi-trusted delivery service, although in principle a blockchain with consensus protocol could also be used.

There are many systems in which such centralization is undesirable. Anonymity networks such as Tor [16] or Loopix [37] rely crucially on the assumption that no single node is able to observe all network traffic. Protesters use mesh networks, in which mobile devices exchange messages without any servers, to avoid censorship [3, 42]. If the company operating a central server goes out of business, a potentially difficult migration to another server is required [26]. Peer-to-peer content distribution networks like IPFS[1] and federated messaging systems such as Matrix[2] avoid central nodes to improve resilience. Blockchains reduce the degree of centralization, but instead have performance and scalability problems [4].

In this paper, we present the first *decentralized*, asynchronous secure group messaging protocol supporting dynamic groups. Our protocol works with any underlying network without requirements on message ordering or latency: it can be deployed in peer-to-peer or anonymity networks, and it does not require any servers or consensus protocol. If servers are optionally used, there is no need to trust them to order messages correctly, and users can switch from one server to another (or use multiple servers at the same time) without worrying about preserving message ordering during the switch.

Our protocol provides end-to-end encryption with forward secrecy and PCS, even when multiple users concurrently modify the group state. It is practical, using only efficient and widely deployed cryptographic primitives. It provides key agreement: messages to the group need only be encrypted and sent once with small constant overhead, regardless of the group size. Group membership changes and key updates (for PCS) require effort proportional to the group size.

The primary challenge in designing such a protocol is the fact that multiple group changes (adding or removing users) may happen concurrently. As the protocol is asynchronous, group changes must take effect immediately: we cannot wait for a quorum of group members or a set of servers to acknowledge a change. In the case of a network partition, the group may be split into two or more subsets of users; even if one subset cannot communicate with another, communication within each subset must still be possible [20]. Thus, multiple group membership changes may accumulate within each subset, which need to be reconciled when the partition heals. Our protocol ensures that all users converge to the same group membership and the same keys in all of these cases.

In this paper we make the following contributions:

- We address the challenge of handling multiple concurrent group membership changes in security protocols; no previous work on secure group messaging addresses this issue. We do this by introducing the concept of *decentralized group membership* (DGM) in Section 6.
- We define *decentralized continuous group key agreement* (DCGKA), a new security notion describing a protocol for key agreement in which symmetric keys change over time in response to

[1]`https://ipfs.io/`
[2]`https://matrix.org/`

group membership changes and PCS key updates. Our definition generalizes continuous group key agreement (CGKA) [2] to the decentralized setting.

- We construct an efficient protocol that implements DCGKA (Section 7), prove its correctness and security (Appendix E), and show how to use it to implement secure group messaging (Section 7.6).
- We evaluate the performance of a prototype implementation of our protocol (Section 8). For groups of 128 members, a key update operation takes 95 ms of CPU time per client and transmits 40 kB of network traffic.

2 Goals and Assumptions

In this section we summarize the goals of our protocol and the threat model for which it is designed.

A secure group messaging protocol allows a group with a given set of users to be created, allows group members to be added and removed, and allows messages to be sent to the current set of members. We distinguish between *application messages* (messages that a user wishes to send to the group) and *control messages* (sent by the protocol to update group state). The protocol must meet the following security goals:

Confidentiality: An application message sent by a group member can only be decrypted by users who are also members of the group at the time the message is sent.

Integrity: Messages cannot be undetectably modified by anyone but the member who sent them.

Authentication: The sender of a message cannot be forged, and only members can send messages to the group.

Forward secrecy: After a group member decrypts an application message, an adversary who corrupts that member cannot decrypt that message.

Post-compromise security (PCS): An adversary who corrupts a group member, learning their current state, can only decrypt messages until that group member sends a *PCS update* message that "heals" the compromise.

Eventual consistency: All group members receive the same set of messages (possibly in different orders), and all group members converge to the same view of the group state as they receive the same set of messages.

Our protocol ensures these security properties in the face of an adversary with the following capabilities:

Active network attacks and malicious servers: The adversary can read, modify, inject, or drop any messages sent over the network. If servers are used to relay messages, the adversary also fully controls those servers.

Temporary device compromise: From time to time, the adversary can compromise one or more group members' devices and obtain a copy of their private state (including all secret keys) at that moment.

While we can guarantee all of the security goals against network attacks at all time, a device compromise inevitably temporarily impacts confidentiality, integrity, and authentication. For PCS we assume that following a device compromise, the adversary does not use the private state it has acquired to impersonate the compromised user until that user sends a PCS update message. This assumption is standard in PCS protocols [12]. After the PCS update message is sent, the security properties are restored. In

Table 1: Summary of related work as discussed in this paper.

Protocol	Needs central server	Broadcast messages	Update Costs	PCS & FS	PCS in face of concurrent updates
MLS (TreeKEM)	✓	✓	$\mathcal{O}(\log n)$	$\frac{1}{2}$ (FS issues)	Only one sequence heals
Re-randomized TreeKEM	✓	✓	$\mathcal{O}(\log n)$	✓	
Causal TreeKEM	✗[1]	✓	$\mathcal{O}(\log n)$	$\frac{1}{2}$ (severe FS issues)	Any sequence heals
Signal groups	✗[1]	✗	$\mathcal{O}(n)$	$\frac{1}{2}$ (PCS issues)[2]	[Optimal][2]
Sender Keys	✗[1]	✓	$\mathcal{O}(n^2)$	FS only[3]	[Optimal][3]
Megolm	✗[1,4]	✓	$\mathcal{O}(n)$	PCS only	
Our DCGKA protocol	✗	✓	$\mathcal{O}(n)$	✓	All but last can be concurrent

[1] Does not specify how to determine group membership in the face of concurrent additions and removals.
[2] Optimal PCS in the face of concurrent updates is possible by using a 2-party protocol with optimal PCS+FS in place of pairwise Signal.
[3] Optimal PCS in the face of concurrent updates is possible at $\mathcal{O}(n^2)$ cost, but not used in practice.
[4] Not asynchronous in the face of concurrent operations.

the case where multiple group members are corrupted, the group is healed once all corrupted members send PCS update messages (possibly concurrently), and one PCS update is sent following all of those messages.

In general, we assume that group members correctly follow the protocol. Our protocol offers limited protection against malicious members (see Section 6); confidentiality is impossible if any group members are controlled by the adversary.

We assume the existence of a Public Key Infrastructure (PKI) that allows group members to obtain a correct public key for other users. Finally, we assume the standard security properties of the cryptographic primitives we use: authenticated symmetric encryption, public-key encryption (PKE), hash-based key derivation functions, and digital signatures.

3 Related Work

There are many existing secure messaging protocols [43]. Schemes for two-party communication, incorporating varying levels of post-compromise security and forward secrecy, have been studied extensively over the past few years, starting with the Signal protocol [36] and its analysis [11], followed by several new protocols and their analyses [6, 17, 23, 24, 38] as well as a modular analysis and generalization of Signal [1].

Among group messaging protocols for more than two parties, relatively few are both asynchronous and support dynamic groups, which we consider critical requirements for practical group messaging on mobile devices. We focus on such protocols in our discussion below. See Table 1 for a high-level comparison.

The MLS protocol, mentioned in Section 1, uses the *TreeKEM* key agreement protocol [5, 35]. TreeKEM is itself based on *Asynchronous Ratcheting Tree* [13], which was the first post-compromise secure group messaging protocol, but which does not support dynamic groups. Update messages in TreeKEM (i.e. adding or removing a group member, or performing a PCS key update) are $\mathcal{O}(\log(n))$ in size, where n is the number of group members. The result is a single key that is shared by all group members. This is achieved by arranging group members into a binary tree, with one leaf per group member, and each member knowing the secret keys on their leaf node's path to the root. However, MLS is inherently centralized, requiring a strict total order on messages modifying the group state. This has long been recognized as an issue with MLS [18, 22, 46], yet existing proposals for modifying MLS so that it does not require a server-enforced ordering are inadequate. One proposal [5, §12.2] is for clients to agree on the total order using a total order broadcast protocol [15]. However, total order broadcast is equivalent to consensus [9], so this would be slow, and a group would be unable to process any messages if too many members were offline. Another proposal uses a technique for combining concurrent updates [8, §5], but this approach does not apply to adding and removing group members, and it provides weak PCS guarantees for concurrent updates.

Alwen et al. [2] introduce *Re-randomized TreeKEM* to strengthen TreeKEM's forward secrecy. This protocol is even harder to decentralize: group members update each other's secret keys so that each secret key is only used once, allowing them to be deleted for forward secrecy, but this approach breaks if multiple concurrent messages are encrypted under the same public key.

In the other direction, *Causal TreeKEM* modifies TreeKEM to require only causally ordered message delivery (see Section 5), at the cost of even weaker forward secrecy [45, §4]. Like our work, Causal TreeKEM describes how to handle dynamic groups in the decentralized setting, although the protocol description is largely informal. Also, its post-compromise security is weaker than for our DCGKA protocol: after multiple corruptions, all corrupted group members must send PCS updates *in sequence*, while our protocol allows all but the last update to be concurrent.

In contrast to MLS, *Signal groups* use a simple protocol: the sender of each application message sends the message individually to each other group member using the two-party Signal protocol [40]. This approach quickly becomes inefficient in large groups, as every application message requires $n-1$ two-party messages in a group of size n. Also, care is needed to achieve PCS: using the ordinary Signal protocol, a group member effectively performs a PCS update only after receiving a message from every other group member, which would never happen if one member is always offline.

Sender Keys is another simple protocol, used by WhatsApp [40, 47]. Each group member generates a symmetric key for messages they send, and then sends this key individually to each other group member using the two-party Signal protocol. For each message sent by this member to the group, a new key is derived pseudorandomly from the previous key, providing a ratchet for forward secrecy. Whenever a user is removed, each remaining group member generates a new key and sends it to the other remaining members over the same two-party channels. This protocol could provide PCS by updating keys periodically, but WhatsApp does not do this.

The Sender Keys protocol has the advantage of not requiring a central server, but the disadvantage is that PCS updates are expensive. Note that if one user is compromised, all of the sender keys become known to the adversary. Since each user updates only their own sender key, in order to recover from the compromise, *each* of the n group members needs to generate a new key and send it to each of the $n-1$ other members, resulting in $\mathcal{O}(n^2)$ messages over the two-party channels. Our DCGKA protocol builds on Sender Keys but reduces the cost of PCS updates to $\mathcal{O}(n)$ (see Section 4).

An important detail that is not discussed in Causal TreeKEM, Signal groups or Sender Keys is how to determine the group membership in the case of multiple concurrent additions and removals. For example, in a group with members $\{A, B, C, D\}$, say A removes B from the group, while concurrently B removes A. User C sees A's removal first, while D sees B's removal first. Who are the final remaining group members? We discuss this and related problems in Section 6.

The Matrix *Megolm* protocol [33, 44] attempts to improve on Sender Keys. However, it handles concurrency issues by allowing group members to request old encryption keys from each other, violating both forward secrecy and asynchrony. Also, Matrix's approach to group membership is centralized, relying on a "homeserver" for each group that totally orders group membership operations [32].

DCGKA is based on CGKA, introduced by Alwen et al. [2]. Our construction for two-party secure messaging is based on work by Jost et al. [24] and Durak and Vaudenay [17].

4 Protocol Overview

We now introduce our protocol for decentralized secure group messaging. We begin in this section by presenting a high-level overview of the system architecture, before diving into the details in the following sections.

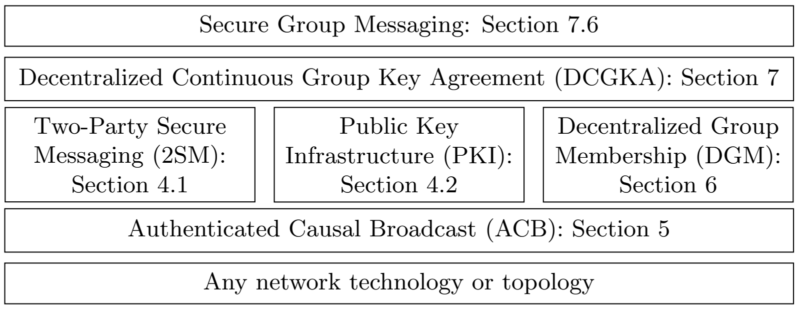
Secure Group Messaging: Section 7.6
Decentralized Continuous Group Key Agreement (DCGKA): Section 7
Two-Party Secure Messaging (2SM): Section 4.1
Public Key Infrastructure (PKI): Section 4.2
Decentralized Group Membership (DGM): Section 6
Authenticated Causal Broadcast (ACB): Section 5
Any network technology or topology

Figure 1: Modules that constitute our secure group messaging protocol.

At its core, our protocol builds on a simple idea: we establish two-party secure messaging channels between every pair of group members, and periodically exchange fresh random secrets over those channels. The result is a key ratchet for each group member, which is used to encrypt application messages sent by that member. Compared to the Sender Keys approach we reduce the cost of PCS updates and define an explicit strategy for handling concurrent group membership changes.

We decompose the protocol into a set of modules as illustrated in Figure 1. Our abstraction for network communication is *Authenticated Causal Broadcast* (ACB), which we discuss in Section 5. ACB provides authenticated message delivery to a whole group and to individual members, but provides no confidentiality. ACB makes no assumptions about the underlying network: it may optionally use any number of untrusted servers to store and forward messages, and use broadcast features (e.g. IP multicast) where available.

On top of this communication layer we use three modules: a two-party secure messaging protocol (Section 4.1); a PKI for obtaining group members' public keys (Section 4.2); and Decentralized Group Membership (DGM), a scheme for unambiguously determining the current set of group members in the face of concurrent additions and removals (Section 6).

These three modules are used by our DCGKA protocol (Section 7), which forms the core of our secure group messaging protocol: a method for deriving fresh keys for each group member in response to membership changes and messages received. Finally, we use these keys to encrypt application messages, yielding a full secure group messaging protocol (Section 7.6). Each application message is only encrypted once, regardless of the group size, and then distributed to group members through Authenticated Causal Broadcast.

In our protocol, the cost of an application message is constant, while the cost of adding or removing a user, and the cost of performing a PCS update, scale linearly with the number of group members. This design is well suited for situations in which application messages are small and frequent, e.g. in multiplayer games or in collaboration software such as multi-user document editors [25]. In contrast, Signal groups incur linear cost for every application message sent.

The application can determine how frequently to perform PCS updates independently from the rate at which application messages are sent. This is a trade-off between strength of PCS and network overhead: more frequent PCS updates shorten the time window of vulnerability following a device compromise, while increasing the network traffic and the number of cryptographic operations performed. For example, a typical application might perform PCS updates once per day.

4.1 Two-Party Secure Messaging Scheme

We start by discussing two-party secure messaging, which our protocol uses to transport secrets from one group member to another. We define the API of such a scheme as follows:

Definition 1. A bidirectional *two-party secure messaging* scheme $\mathsf{2SM} = (\mathsf{2SM\text{-}Init}, \mathsf{2SM\text{-}Send}, \mathsf{2SM\text{-}Receive})$ consists of the following algorithms:

```
2SM-Init($sk, pk$)

$\sigma$.mySks[0] $\leftarrow sk$;  $\sigma$.nextIndex $\leftarrow 1$;  $\sigma$.receivedSk $\leftarrow \varepsilon$
($\sigma$.otherPk, $\sigma$.otherPkSender, $\sigma$.otherPkIndex) $\leftarrow$ ($pk$, "other", $0$)

2SM-Send($m$)

($\sigma$.mySks[$\sigma$.nextIndex], myNewPk) $\leftarrow\$$ PKE-Gen()
(otherNewSk, otherNewPk) $\leftarrow\$$ PKE-Gen()
plaintext $\leftarrow$ ($m$, otherNewSk, $\sigma$.nextIndex, myNewPk)
msg $\leftarrow$ (PKE-Enc($\sigma$.otherPk, plaintext), $\sigma$.otherPkSender, $\sigma$.otherPkIndex)
$\sigma$.nextIndex++
($\sigma$.otherPk, $\sigma$.otherPkSender, $\sigma$.otherPkIndex) $\leftarrow$ (otherNewPk, "me", $\varepsilon$)
return msg

2SM-Receive($c$)

(ciphertext, keySender, keyIndex) $\leftarrow c$
if keySender = "me" then
  $sk \leftarrow \sigma$.mySks[keyIndex];  $\sigma$.mySks[$\leq$ keyIndex] $\leftarrow \varepsilon$
else $sk \leftarrow \sigma$.receivedSk
($m$, $\sigma$.receivedSk, $\sigma$.otherPkIndex, $\sigma$.otherPk) $\leftarrow$ PKE-Dec($sk$, ciphertext)
$\sigma$.otherPkSender $\leftarrow$ "other"
return $m$
```

Figure 2: Our two-party secure messaging protocol.

Initialization: 2SM-Init takes a public-key encryption (PKE) secret key sk for the local user and a PKE public key pk for the other party, and outputs an initial state σ.

Send: 2SM-Send takes a state σ and a plaintext message m, and outputs a new state σ' and a ciphertext c.

Receive: 2SM-Receive takes a state σ and a ciphertext c, and outputs a new state σ' and a plaintext message m.

We assume that 2SM runs on top of the ACB layer, which retransmits any messages that were dropped or tampered with, and authenticates message senders (see Section 5). The 2SM module therefore assumes reliable delivery, and focuses on providing confidentiality with forward secrecy and PCS. The two parties communicating via 2SM can send each other messages in any order, or even concurrently.

The Signal protocol is a popular implementation of 2SM, but it does not suffice for our purposes because it heals from a corruption only after several rounds of communication, not with each message sent. On the other hand, several other 2SM protocols proposed in the literature [17, 23, 24, 38] sacrifice efficiency by not assuming authentic message delivery. We instead use a simplified variant of the protocols by Jost et al. [24] and Durak and Vaudenay [17].

The specification of our 2SM protocol is given in Figure 2. Our starting point is an IND-CPA-secure public-key encryption scheme PKE = (PKE-Gen, PKE-Enc, PKE-Dec) as defined in Appendix B. Each party updates their public key every time they send a message, which easily provides PCS. To ensure forward secrecy, when one party sends a message, they also generate a new PKE key pair *for the other party*, include its secret key in the ciphertext, delete their own copy of the secret key, and keep the public key for the next time they send a message. Upon receiving this message, the other party deletes their old secret key and replaces it with the given one.

Each of the functions in Figure 2 implicitly takes the current state σ as argument, and returns the updated state σ'.

$\mathsf{2SM\text{-}Send}$ first generates new PKE key pairs for both parties, and stores the secret key for itself at the next unused index in the array $\sigma.\mathsf{mySks}$. The message m is then encrypted using $\sigma.\mathsf{otherPk}$, which is either the last public key we generated for the other party, or the last public key we received from the other party, whichever was more recent. $\sigma.\mathsf{otherPkSender}$ is the constant string `"me"` in the former case, and `"other"` in the latter case. $\mathsf{otherPkSender}$ and $\mathsf{otherPkIndex}$ are included in the message as metadata to indicate which of the recipient's public keys is being used.

$\mathsf{2SM\text{-}Receive}$ uses the message's metadata to determine which secret key to use for decryption, assigning it to sk. In case it is a secret key generated by ourselves, that secret key and all keys with lower index are deleted, indicated by $\sigma.\mathsf{mySks}[\leq \mathsf{keyIndex}] \leftarrow \varepsilon$, because we know these keys will never be needed again. The algorithm then stores the new public and secret keys contained in the message.

In Appendix C we formalize and prove the security properties of our 2SM protocol.

4.2 Public Key Infrastructure

Like many other protocols, DCGKA requires a means for one user to obtain correct public keys for the other group members. These public keys are then used to authenticate messages at the ACB layer, and to initialize the 2SM protocol. We assume that each user is identified by an ID, and we model this module as follows:

Definition 2. A *Public Key Infrastructure* scheme $\mathsf{PKI} = (\mathsf{PKI\text{-}PublicKey}, \mathsf{PKI\text{-}SecretKey})$ consists of two algorithms: $\mathsf{PKI\text{-}PublicKey}(\mathsf{ID}, \mathsf{ID}')$ returns the public key of user ID for use by ID', and $\mathsf{PKI\text{-}SecretKey}(\mathsf{ID}, \mathsf{ID}')$ returns the corresponding secret key. The latter function is only accessible to ID, while the former is public.

Note that we do not allow the adversary to query $\mathsf{PKI\text{-}SecretKey}(\mathsf{ID}, \mathsf{ID}')$ even after corrupting ID (although it may learn the secret indirectly if ID stores it in their state). This is reasonable because in our protocol, each secret key is only queried once, and hence it can be deleted from the PKI module after it is queried. Moreover, group members can update their PKI key pairs each time they send a PCS update message, thus healing from past corruptions.

In practice, instead of having strictly separate public keys for each ID', public keys should be a *prekey bundle*, containing both a long-term public key and an ephemeral *prekey*, and users should make a best effort to ensure each of their ephemeral prekeys is used only once and is updated after a compromise. Prekeys were introduced by Signal [31] and are standard for modern messaging protocols as they provide asynchrony and forward secrecy for the first message, albeit typically with help from a semi-trusted prekey distribution server. Trustworthy public key distribution is a longstanding problem [43, §IV], and decentralization and ephemeral prekeys introduce more issues, but addressing them is outside of the scope of this paper.

5 Authenticated Causal Broadcast (ACB)

In centralized group communication, users can rely on a server to provide a total ordering of messages, but this is not the case when moving to a decentralized setting. For example, Figure 3 shows a scenario that a decentralized secure messaging protocol needs to handle: in a group initially consisting of $\{A, B\}$, a network interruption makes A temporarily unable to communicate with B. During this interruption, A adds a new group member C, and B adds a new group member D. Moreover, A and C send a sequence of group messages, while B and D independently exchange a different set of messages. Eventually, connectivity is restored, and all of the group members learn about each others' messages.

In such a scenario, some messages are ordered relative to each other (for example, C can only send messages to the group after it has been added), while others are not ordered (A's addition of C is

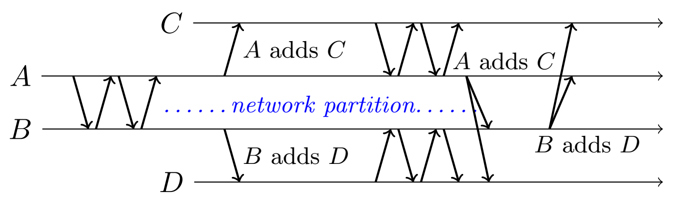

Figure 3: Concurrent group member additions during a network partition.

concurrent with B's addition of D). We can formalize this partial ordering as the *causal order* on messages [19, 29]:

Definition 3. The *causal order* is the partial order $\prec$ on messages sent to a group. We have $m_1 \prec m_2$ iff at least one of the following holds:

- m_1 and m_2 were sent by the same group member, and that member sent m_1 before sending m_2;
- m_2 was sent by group member x, and m_1 was received and processed by x before sending m_2;
- there exists m_3 such that $m_1 \prec m_3$ and $m_3 \prec m_2$.

We say m_1 and m_2 are *concurrent* if $m_1 \not\prec m_2$ and $m_2 \not\prec m_1$.

A group member processes messages *in causal order* if for every message m, before processing m, the group member processes all preceding messages $\{m' \mid m' \prec m\}$.

In a system that ensures processing in causal order, if one group member sends a sequence of successive messages, then all group members will process those messages in the order they were sent. On the other hand, if two messages are sent concurrently by two different group members, then other users may process those messages in either order.

We can ensure causally ordered processing of group messages by attaching metadata to each message. For instance, this metadata could be a vector clock [19, 34], or the hashes of the immediate causal predecessors of the message. If a group member receives a message whose causal predecessors have not yet arrived, it buffers that message until all causal predecessors have been processed. If necessary, it requests retransmission of any dropped or corrupted messages. Assuming every group member can eventually communicate with other members, every member will eventually receive all messages. The metadata that enables causal ordering can also be used to prevent replay attacks by an active network adversary.

Observe that, in contrast to totally-ordered broadcast, causal broadcast can be ensured using only local checks by each group member, while totally-ordered broadcast requires coordination equivalent to consensus [9]. Thus, causal broadcast is well suited to decentralized and asynchronous networks.

5.1 Broadcast and Direct Messages in Dynamic Groups

Our DCGKA protocol makes use of two types of message: *broadcast messages* are sent to all members of the group, while *direct messages* are sent to one specified recipient. We make this distinction only for reasons of efficiency; our security properties hold regardless of who receives which message.

How these types of message are implemented depends on the underlying network: if the network only supports broadcast, then all direct messages can be bundled into a single broadcast message and sent to the entire group. Each group member can then pick out the direct message intended for them from the broadcast message. The DCGKA protocol ensures that users can only decrypt direct messages intended for them. On the other hand, if the underlying network supports only unicast, any broadcast messages can be sent to each group member by separate unicast.

When group members are added and removed, it is unclear which users should receive a particular broadcast message. For example, in Figure 3, DCGKA messages sent by C during the network partition should eventually be delivered to B and D, even though at the time those messages were sent, C did not know that D was a group member, and vice versa.

The DCGKA protocol has a precise way of determining group membership, which we discuss in Section 6, and it ensures that the set of users who obtain the key to decrypt a particular application message is exactly the set of group members at the time that particular message was sent. For this reason, the Causal Broadcast layer may over-approximate the set of group members by potentially delivering a message to more users than strictly necessary. DCGKA ensures that any excess recipients of a message will not be able decrypt it.

The security game for DCGKA, presented in Appendix A, formalizes the assumptions on Causal Broadcast. This game allows the adversary to read all messages sent via the network.

5.2 Authentication

Another requirement of our Causal Broadcast layer is that it authenticates the sender of every message, preventing the adversary from modifying or injecting messages, or forging message senders. The simplest way of doing this is by attaching a signature under the sender's long-term identity key to each message. However, this approach does not provide post-compromise security, since an adversary who corrupts a group member once learns their long-term secret key forever.

No protocol is able to prevent an active attack by an adversary who corrupts the private state of group member A and then injects a message posing as A before A sends an update message [12]. Some two-party messaging protocols [17,23,24,38] provide limited security after such an attack, known as *post-impersonation security*, but such properties are expensive to achieve and seem to have limited practical use. Instead, our requirement is that such a corruption is healed by a PCS update: after A sends an update message and a group member B receives that update, B will reject any messages by the adversary impersonating A.

We use a *post-compromise secure signature scheme*, as defined by Cremers et al. [14, §5], which augments an ordinary signature with the ability to update the public key. A simple and practical protocol is as follows: when a group member performs a PCS update, they generate a new key pair, sign the new public key using the old secret key, and include this information in the update message. They also store the new secret key for use in signing future messages [14, §5.2].

Since causal order broadcast ensures that all messages from a given sender are processed in the order they were sent, recipients of update messages can store the most recent public key for each group member to check these chains of signatures.

6 Decentralized Group Membership (DGM)

We assume that every member of a group has the permission to add or remove other group members. A more restrictive policy (e.g. allowing only designated administrators to change the membership) can easily be implemented by ignoring changes made by unauthorized users.

Determining the "current" set of group members is challenging in a decentralized setting because different users may process additions and removals of group members in different orders. Nevertheless, when all of the membership changes have been processed, everyone must agree on who the group members are. The difficulty of achieving this can be illustrated by three examples:

1. Two group members A and B concurrently remove each other. Do the removals both take effect, cancel out, or something else? Note that among the remaining group members, some may see A's removal first, while others see B's removal first. Ordinarily users would ignore messages sent by

non-members, but in this case, users must take both removals into account, otherwise they will end up with inconsistent views of the group membership.

2. While group member A removes B, B concurrently adds a new group member C. Should C be in the group?

3. A group member A is concurrently added and removed. This can happen if A is initially in the group, one member removes and then re-adds A, while concurrently another member only removes A. Should A be in the group?

We define a *Decentralized Group Membership* (DGM) scheme to be an algorithm that determines the members of a group, given a set of all membership changes that have occurred so far in the group's lifetime. Designers of a DGM algorithm have the freedom to answer policy questions like those above.

Definition 4. A *group membership operation* is a tuple of one of the following forms:

- $(\texttt{"create"}, \mathsf{sender}, \mathsf{seq}, \mathsf{IDs})$, representing the creation of a group whose members are the set of users IDs.
- $(\texttt{"add"}, \mathsf{sender}, \mathsf{seq}, \mathsf{ID})$, representing the addition of user ID to the group.
- $(\texttt{"remove"}, \mathsf{sender}, \mathsf{seq}, \mathsf{ID})$, representing the removal of user ID from the group.
- $(\texttt{"ack"}, \mathsf{sender}, \mathsf{seq}, \mathsf{ackID}, \mathsf{ackSeq})$, representing an acknowledgment of another operation as described below.

Each operation contains sender, the ID of the user performing this operation, and seq, an integer sequence number (successive operations by the same user are given incrementing sequence numbers). An acknowledgment also contains $(\mathsf{ackID}, \mathsf{ackSeq})$, the sender and sequence number of the prior operation being acknowledged.

We define a causal order $\prec$ over these operations: $\mathsf{op}_1 \prec \mathsf{op}_2$ if op_1 and op_2 have the same sender and op_1 has a lower sequence number than op_2; or if op_2 is an acknowledgment of op_1; or if there exists op_3 such that $\mathsf{op}_1 \prec \mathsf{op}_3$ and $\mathsf{op}_3 \prec \mathsf{op}_2$.

Definition 5. A *decentralized group membership* (DGM) scheme is a function DGM that takes a set of group membership operations and associated partial order $\prec$, and returns the set of IDs of current group members.

6.1 Strong-Remove Decentralized Group Membership Scheme

We now define a particular DGM scheme that we call the *strong-remove decentralized group membership scheme* ($\mathsf{SR\text{-}DGM}$). This scheme errs on the side of removal in most situations where a user's membership in the group is ambiguous. We believe that this policy is desirable in practice, as it is easy to re-add a group member who has been inadvertently removed, but it is impossible to reverse a leak of confidential information that has occurred because a user believed to be removed was, in fact, still a group member.

In the design of our DGM scheme we are guided by one observation: a user who is being removed from a group should not be able to circumvent their removal. For example, if B is being removed by A, then B should not be able to evade removal by concurrently removing A. Moreover, if B is being removed, B should not be able to evade removal by adding a new device C controlled by B. For these reasons, our policy in example 1 of the previous section is that both removals should take effect, and C should not be in the group in example 2.

Definition 6. We define the function $\mathsf{SR\text{-}DGM}$ as follows. Let H be an input to $\mathsf{SR\text{-}DGM}$, a set of group membership operations with partial order $\prec$ as in Definition 4, with a unique $(\texttt{"create"}, \mathsf{ID}_0, \mathsf{seq}, \{\mathsf{ID}_1, \dots, \mathsf{ID}_n\})$ operation as the minimum element. Let H' be the set obtained by replacing the unique $\texttt{"create"}$ operation with operations $m_{\text{init}} = (\texttt{"add"}, \mathsf{ID}_0, \mathsf{seq}, \mathsf{ID}_0) \prec (\texttt{"add"}, \mathsf{ID}_0, \mathsf{seq}, \mathsf{ID}_1) \prec \cdots \prec (\texttt{"add"}, \mathsf{ID}_0, \mathsf{seq}, \mathsf{ID}_n)$, which are less than all other operations. Define a directed acyclic graph G by:

- The vertices of G are all $\texttt{"add"}$ operations in H'.
- The edges of G are all pairs (m_1, m_2) such that $m_1 = (\texttt{"add"}, \mathsf{ID}_1, \_, \mathsf{ID}_2)$, $m_2 = (\texttt{"add"}, \mathsf{ID}_2, \_, \mathsf{ID}_3)$, and $m_1 \prec m_2$. Here $\_$ stands for an arbitrary value.

Obtain G_{cancel} by deleting from G all edges (m_1, m_2) with $m_1 = (\texttt{"add"}, \mathsf{ID}_1, \_, \mathsf{ID}_2)$ and $m_2 = (\texttt{"add"}, \mathsf{ID}_2, \_, \mathsf{ID}_3)$ such that $\exists\, m_{\text{rem}} = (\texttt{"remove"}, \_, \_, \mathsf{ID}_2) \in H.\ m_1 \prec m_{\text{rem}} \wedge m_2 \not\prec m_{\text{rem}}$. Then

$$\begin{aligned}\mathsf{SR\text{-}DGM}(H) := \{\mathsf{ID} \mid \exists m = (\texttt{"add"}, \_, \_, \mathsf{ID}) \in H'.\\ (\exists \text{ a path from } m_{\text{init}} \text{ to } m \text{ in } G_{\text{cancel}}) \wedge\\ (\nexists m' = (\texttt{"remove"}, \_, \_, \mathsf{ID}) \in H.\ m \prec m')\}.\end{aligned}$$

In this definition, the paths in G show all of the ways that a given group member was added to the group, tracing each of their additions back to the operation m_{init} that created the group. G_{cancel} then deletes the edges that were "canceled" by a remove operation, which happens if some operation m_{rem} canceled the effect of m_1 either before or concurrently to m_2. If m_{rem} is causally greater than m_2, then the edge is not deleted, since then the sender of m_{rem} was already aware of ID_3 as a group member. The group then contains all users whose add operation still connects back to m_{init}, and which have not been removed by a causally greater operation.

7 Decentralized Continuous Group Key Agreement (DCGKA)

We now turn to Decentralized Continuous Group Key Agreement, the core of our secure group messaging protocol. DCGKA generates a sequence of *update secrets* for each group member, which can then be used to encrypt/decrypt application messages sent by that member. Only group members learn these update secrets, and fresh secrets are generated every time a user is added or removed, or a PCS update is requested. The key agreement protocol ensures that all users observe the same sequence of update secrets for each group member, regardless of the order in which messages are received.

7.1 The DCGKA Abstraction

Definition 7. A *decentralized continuous group key agreement* scheme consists of the algorithms DCGKA = (init, create, add, remove, update, $\mathsf{process}$). Except for init, all of the algorithms take a state γ and further arguments as specified below, and return a 4-tuple $(\gamma', \mathsf{control}, \mathsf{dmsgs}, I)$. Here, γ' is a new state, $\mathsf{control}$ is a control message that should be broadcast to the group (or ε if no message needs to be sent), dmsgs is a set of direct messages that should be sent to specified users, and I is a new update secret for the current user. Algorithm $\mathsf{process}$ returns two update secrets: one for the sender of the message being processed, and one for the current user.

Initialization: init takes the ID of the current user, and outputs an initial state γ.

Group creation: create takes a state γ and a set of IDs $\{\mathsf{ID}_1, \dots, \mathsf{ID}_n\}$, and creates a new group with the specified users and itself as members.

Member addition: add takes a state γ and a user ID, and adds that user to the group.

Member removal: remove takes a state γ and a user ID, and removes that user from the group.

PCS update: update takes a state γ and performs a key update for PCS.

Message processing: process is called when a control message is received. It takes a state γ, a user ID (the message sender), a control message control, and a direct message dmsg (or ε if there is no associated direct message).

All of the algorithms except init return a control message (to be broadcast) and a set of direct messages. These messages must be distributed to the other group members through Authenticated Causal Broadcast as discussed in Section 5, passing the control message to the process algorithm when it is delivered. If direct messages are sent along with a control message, we assume that the direct message for the appropriate recipient is delivered in the same call to process. Our algorithm never sends a direct message without an associated broadcast control message.

Besides the control and direct messages, create, add, remove, and update each return a new update secret for the current user performing this operation. When the control message from that operation is received by another user, the process algorithm returns a control message and direct messages to acknowledge the operation, and two update secrets: one for the user who initiated the create/add/remove/update, and one for the current user processing the operation. Finally, when the acknowledgment message is processed by another user, the process algorithm returns a new update secret for the sender of the acknowledgment.

We capture the security properties of DCGKA and the adversary's capabilities formally in the security game in Appendix A. In summary, the adversary is given access to oracles to cause group members to call the protocol algorithms, deliver messages in causal order, and corrupt group members (revealing their current state). The adversary is also given a transcript of all messages sent. However, due to the underlying Authenticated Causal Broadcast, the adversary is not allowed to modify messages, deliver them out of causal order, or forge the sender at the DCGKA level. We show that an adversary, given these capabilities, cannot distinguish update secrets from random with better than negligible probability.

7.2 Our DCGKA Protocol

Figure 4 contains the full specification of our protocol. The variable γ denotes the state, which consists of the variables initialized in init. This state is implicitly input to and returned from each function. The notation $\mathsf{2sm}[\cdot] \leftarrow \varepsilon$ means that 2sm is a dictionary (hash table) where every key is initially mapped to the default value ε, representing the empty string.

We start by discussing the algorithms create, update, and remove. They first generate a control message describing the group membership operation, to be broadcast to the group. They then determine the set of group members and pass that set to generate-seed, which generates a fresh random *seed secret* (with bit length given by an implicit security parameter) and encrypts it individually for each group member using the 2SM protocol from Section 4.1, returning a set of direct messages as (recipient, ciphertext) pairs. If needed, encrypt-to initializes a 2SM protocol instance using the PKI. In the case of update and remove, the set of group members is determined by the DGM algorithm discussed in Section 6. We also call process-create, process-update, or process-remove as appropriate, which use the seed secret just generated to compute an update secret I for the current user. The control message, set of direct messages, and update secret I are then returned.

Control messages have the form ("type", seq, content), where seq sequentially numbers control messages from the same sender, and content depends on the type. When the control message and the appropriate direct message are delivered, process is called, which in turn calls process-create, process-ack, process-update, process-remove, process-add, or process-add-ack depending on the type indicated in the control message. In the case of create/update/remove, after decrypting the seed secret in the direct

```
init(ID)

$\gamma$.myId $\leftarrow$ ID
$\gamma$.mySeq $\leftarrow 0$
$\gamma$.history $\leftarrow \emptyset$
$\gamma$.nextSeed $\leftarrow \varepsilon$
$\gamma$.2sm$[\cdot] \leftarrow \varepsilon$
$\gamma$.memberSecret$[\cdot,\cdot,\cdot] \leftarrow \varepsilon$
$\gamma$.ratchet$[\cdot] \leftarrow \varepsilon$
```

```
process(sender, controlMsg, dmsg)

(type, seq, info) $\leftarrow$ controlMsg
if type = "create" then
    return process-create(sender, seq, info, dmsg)
else if type = "ack" then etc ...
```

```
create(ID$_1, \dots,$ ID$_n$)

ids $\leftarrow$ {ID$_1, \dots,$ ID$_n$}
control $\leftarrow$ ("create", $++\gamma$.mySeq, ids)
dmsgs $\leftarrow$ generate-seed(ids)
$(\_, \_, I, \_) \leftarrow$ process-create($\gamma$.myId, $\gamma$.mySeq, ids, $\varepsilon$)
return (control, dmsgs, $I$)
```

```
process-create(sender, seq, IDs, dmsg)

op $\leftarrow$ ("create", sender, seq, IDs)
$\gamma$.history $\leftarrow \gamma$.history $\cup$ {op}
return process-seed(sender, seq, dmsg)
```

```
process-ack(sender, seq, (ackID, ackSeq), dmsg)

if (ackID, ackSeq) was a create/add/remove then
    op $\leftarrow$ ("ack", sender, seq, ackID, ackSeq)
    $\gamma$.history $\leftarrow \gamma$.history $\cup$ {op}
$s \leftarrow \gamma$.memberSecret[ackID, ackSeq, sender]
if $s \neq \varepsilon$ then
    $\gamma$.memberSecret[ackID, ackSeq, sender] $\leftarrow \varepsilon$
    return ($\varepsilon, \varepsilon$, update-ratchet(sender, $s$), $\varepsilon$)
else if dmsg $\neq \varepsilon$ then
    $s \leftarrow$ decrypt-from(sender, dmsg)
    return ($\varepsilon, \varepsilon$, update-ratchet(sender, $s$), $\varepsilon$)
else return ($\varepsilon, \varepsilon, \varepsilon, \varepsilon$)
```

```
update()

control $\leftarrow$ ("update", $++\gamma$.mySeq, $\varepsilon$)
recipients $\leftarrow$ member-view($\gamma$.myId) $\setminus$ {$\gamma$.myId}
dmsgs $\leftarrow$ generate-seed(recipients)
$(\_, \_, I, \_) \leftarrow$ process-update($\gamma$.myId, $\gamma$.mySeq, $\varepsilon$, $\varepsilon$)
return (control, dmsgs, $I$)
```

```
process-update(sender, seq, _, dmsg)

return process-seed(sender, seq, dmsg)
```

```
remove(ID)

control $\leftarrow$ ("remove", $++\gamma$.mySeq, ID)
recipients $\leftarrow$ member-view($\gamma$.myId) $\setminus$ {ID, $\gamma$.myId}
dmsgs $\leftarrow$ generate-seed(recipients)
$(\_, \_, I, \_) \leftarrow$ process-remove($\gamma$.myId, $\gamma$.mySeq, ID, $\varepsilon$)
return (control, dmsgs, $I$)
```

```
process-remove(sender, seq, removed, dmsg)

op $\leftarrow$ ("remove", sender, seq, removed)
$\gamma$.history $\leftarrow \gamma$.history $\cup$ {op}
return process-seed(sender, seq, dmsg)
```

```
add(ID)

control $\leftarrow$ ("add", $++\gamma$.mySeq, ID)
$c \leftarrow$ encrypt-to(ID, $\gamma$.ratchet[$\gamma$.myId])
op $\leftarrow$ ("add", $\gamma$.myId, $\gamma$.mySeq, ID)
welcome $\leftarrow$ ($\gamma$.history $\cup$ {op}, $c$)
$(\_, \_, I, \_) \leftarrow$ process-add($\gamma$.myId, $\gamma$.mySeq, ID, $\varepsilon$)
return (control, {(ID, welcome)}, $I$)
```

```
process-add(sender, seq, added, dmsg)

if added = $\gamma$.myId then
    return process-welcome(sender, seq, dmsg)
op $\leftarrow$ ("add", sender, seq, added)
$\gamma$.history $\leftarrow \gamma$.history $\cup$ {op}
if $\gamma$.myId $\in$ member-view(sender) then
    $\gamma$.memberSecret[sender, seq, added] $\leftarrow$
        update-ratchet(sender, "welcome")
    $I_{\text{sender}} \leftarrow$ update-ratchet(sender, "add")
else $I_{\text{sender}} \leftarrow \varepsilon$
if sender = $\gamma$.myId then return ($\varepsilon, \varepsilon, I_{\text{sender}}, \varepsilon$)
control $\leftarrow$ ("add-ack", $++\gamma$.mySeq, (sender, seq))
$c \leftarrow$ encrypt-to(added, ratchet[$\gamma$.myId])
$(\_, \_, I_{\text{me}}, \_) \leftarrow$ process-add-ack($\gamma$.myId,
        $\gamma$.mySeq, (sender, seq), $\varepsilon$)
return (control, {(added, $c$)}, $I_{\text{sender}}, I_{\text{me}}$)
```

```
process-add-ack(sender, seq, (ackID, ackSeq), dmsg)

op $\leftarrow$ ("ack", sender, seq, ackID, ackSeq)
$\gamma$.history $\leftarrow \gamma$.history $\cup$ {op}
if dmsg $\neq \varepsilon$ then
    $\gamma$.ratchet[sender] $\leftarrow$ decrypt-from(sender, dmsg)
if $\gamma$.myId $\in$ member-view(sender) then
    return ($\varepsilon, \varepsilon$, update-ratchet(sender, "add"), $\varepsilon$)
else return ($\varepsilon, \varepsilon, \varepsilon, \varepsilon$)
```

```
process-welcome(sender, seq, (adderHistory, c))

$\gamma$.history $\leftarrow$ adderHistory
$\gamma$.ratchet[sender] $\leftarrow$ decrypt-from(sender, $c$)
$\gamma$.memberSecret[sender, seq, $\gamma$.myId] $\leftarrow$
        update-ratchet(sender, "welcome")
$I_{\text{sender}} \leftarrow$ update-ratchet(sender, "add")
control $\leftarrow$ ("ack", $++\gamma$.mySeq, (sender, seq))
$(\_, \_, I_{\text{me}}, \_) \leftarrow$ process-ack($\gamma$.myId, $\gamma$.mySeq,
        (sender, seq), $\epsilon$)
return (control, $\epsilon$, $I_{\text{sender}}, I_{\text{me}}$)
```

```
generate-seed(recipients)

$\gamma$.nextSeed $\leftarrow\$$ KGen
return {(ID, encrypt-to(ID, $\gamma$.nextSeed)) |
        ID $\in$ recipients}
```

```
process-seed(sender, seq, dmsg)

recipients $\leftarrow$ member-view(sender) $\setminus$ {sender}
if sender = $\gamma$.myId then
    seed $\leftarrow \gamma$.nextSeed;  $\gamma$.nextSeed $\leftarrow \varepsilon$
else if $\gamma$.myId $\in$ recipients then
    seed $\leftarrow$ decrypt-from(sender, dmsg)
else
    control $\leftarrow$ ("ack", $++\gamma$.mySeq, (sender, seq))
    return (control, $\varepsilon, \varepsilon, \varepsilon$)
for ID $\in$ recipients do
    $s \leftarrow$ HKDF(seed, ID)
    $\gamma$.memberSecret[sender, seq, ID] $\leftarrow s$
$s \leftarrow$ HKDF(seed, sender)
$I_{\text{sender}} \leftarrow$ update-ratchet(sender, $s$)
if sender = $\gamma$.myId then return ($\varepsilon, \varepsilon, I_{\text{sender}}, \varepsilon$)
control $\leftarrow$ ("ack", $++\gamma$.mySeq, (sender, seq))
allMembers $\leftarrow$ member-view($\gamma$.myId)
forward $\leftarrow$ {(ID, encrypt-to(ID,
        $\gamma$.memberSecret[sender, seq, $\gamma$.myId])) |
        ID $\in$ allMembers $\setminus$ (recipients $\cup$ {sender})}
$(\_, \_, I_{\text{me}}, \_) \leftarrow$ process-ack($\gamma$.myId, $\gamma$.mySeq,
        (sender, seq), $\varepsilon$)
return (control, forward, $I_{\text{sender}}, I_{\text{me}}$)
```

```
encrypt-to(recipient, plaintext)

if $\gamma$.2sm[recipient] = $\varepsilon$ then
    $sk \leftarrow$ PKI-SecretKey($\gamma$.myId, recipient)
    $pk \leftarrow$ PKI-PublicKey(recipient, $\gamma$.myId)
    $\gamma$.2sm[recipient] $\leftarrow$ 2SM-Init($sk, pk$)
($\gamma$.2sm[recipient], ciphertext) $\leftarrow$
        2SM-Send($\gamma$.2sm[recipient], plaintext)
return ciphertext
```

```
decrypt-from(sender, ciphertext)

if $\gamma$.2sm[sender] = $\varepsilon$ then
    $sk \leftarrow$ PKI-SecretKey($\gamma$.myId, sender)
    $pk \leftarrow$ PKI-PublicKey(sender, $\gamma$.myId)
    $\gamma$.2sm[sender] $\leftarrow$ 2SM-Init($sk, pk$)
($\gamma$.2sm[sender], plaintext) $\leftarrow$
        2SM-Receive($\gamma$.2sm[sender], ciphertext)
return plaintext
```

```
update-ratchet(ID, input)

(updateSecret, $\gamma$.ratchet[ID]) $\leftarrow$
        HKDF($\gamma$.ratchet[ID], input)
return updateSecret
```

```
member-view(ID)

ops $\leftarrow$ {$m \in \gamma$.history | $m$ was sent or acked by ID
    (or the user who added ID, if $m$ precedes the add)}
return SR-DGM(ops)
```

Figure 4: Our DCGKA Protocol.

message, process-seed computes new update secrets for the sender of the control message (I_{sender}) and for the recipient (I_{me}).

7.3 Computing Update Secrets

Update secrets are computed as follows. First, the seed secret and a user ID are passed to a key derivation function (KDF) to produce a unique secret for each group member. For a seed secret sent by user sender, where seq is the sequence number of the appropriate control message, the state variable γ.memberSecret[sender, seq, ID] stores the per-member secret derived from that seed for member ID. We propose using a HMAC-based KDF or HKDF [27,28], like the Signal protocol [36], but other constructions with similar properties could be used instead.

Next, the per-member secret is passed to the update-ratchet function, which again uses the HKDF to implement a separate key ratchet for each member. The ratchet state for user ID is stored in γ.ratchet[ID]; each invocation of update-ratchet produces a new ratchet state and a new update secret. Formally, this ratchet can be modeled as a PRF-PRNG, as proposed by Alwen et al. [1, §4.3]. A PRF-PRNG combines a sequence of input secrets into a output secret, such that the output secret is indistinguishable from random provided that either the last input secret or the last state was uncompromised. In addition, multiple distinct inputs on the same state produce outputs that look random and independent, like a PRF.

Key agreement requires that any group member who computes such a ratchet obtains the same sequence of update secrets. This in turn requires that everyone uses the same sequence of per-member secrets as input to the ratchet, even though different group members may receive messages containing seed secrets in a different order. In order to achieve this, we rely on the fact that Causal Broadcast preserves the order of messages per sender. When a group member updates their own ratchet as a result of receiving a control message, they also broadcast an *acknowledgment* control message to the group. Other group members update the ratchet for the sender of the acknowledgment on receipt of that "ack" (this is done in the process-ack function). Because every group member sees the control messages from a sender in the same order, the ratchet updates for that sender are also performed in the same order.

In order to ensure forward secrecy, the per-member secret and previous ratchet state are deleted as soon as the new ratchet state and update secret have been computed. Seed secrets are likewise deleted as soon as they have been used. As the ratchet is a one-way function, once an update secret has been returned, it and any prior secrets cannot be computed again. Additionally, we require that the 2SM channels also ensure forward secrecy and PCS (see Appendix C).

7.4 Adding Group Members

An existing group member who wishes to add a new member calls the add algorithm, passing in the ID of the user to be added. This information is broadcast to the group in a control message, and additionally we construct a *welcome* message that is sent to the new member as a direct message. The welcome message contains the current ratchet state of the sender of the add operation, encrypted using a 2SM channel, and the history of group membership operations (necessary so that the new member can evaluate the DGM function).

Unlike create, update, and remove, no seed secret is generated in the add algorithm; instead, every group member moves their ratchet forward deterministically by calling update-ratchet with the constant value "add". This is sufficient because all existing group members are allowed to know the next update secrets following the add operation. Since the welcome message included the ratchet state of the sender just before the add, the added user can compute the update secret for the sender in process-welcome by also calling update-ratchet(sender, "add"). The added user's own ratchet is initialized by starting in the empty state ε and calling update-ratchet with a per-member secret obtained from update-ratchet(sender, "welcome"). The other existing group members can similarly initialize their instances of the ratchet state for the new user since they already have the ratchet state for sender.

Next, the newly added user needs to initialize their ratchet state for all of the other group members besides the user who added them. These are not sent by the adding user, but by the respective owners of those ratchets. When a group member processes an "add" control message, it replies with an "add-ack" control message, and additionally sends a direct message to the added user, containing their ratchet state encrypted using a 2SM channel. As the new member receives these acknowledgments, it builds up its collection of ratchet states and obtains an update secret for each existing group member.

Care is required when an add operation occurs concurrently with update or remove operations. For example, in a group with members $\{A, B, C\}$, say A performs an update while concurrently C adds D to the group. When A distributes a new seed secret through 2SM-encrypted direct messages, D will not be a recipient of one of those direct messages, since A did not know about D's addition at the time of sending.

In this example, B may receive the add and the update in either order. If B processes A's update first and then adds D, the seed secret from A is already incorporated into B's ratchet state at time time of adding D, and B sends this ratchet state to D along with the aforementioned "add-ack" message. On the other hand, if B processes the addition of D first, when B subsequently processes A's update, B updates its ratchet with a per-member secret that D does not know, and so D cannot perform the same update on its copy of B's ratchet.

Since key agreement requires all group members to use the same sequence of inputs to a given ratchet, this missing per-member secret needs to be forwarded to the new user D. This is done in the process-seed function, which computes the set:

$$\mathsf{member\text{-}view}(\gamma.\mathsf{myId}) \setminus \big(\mathsf{recipients} \cup \{\mathsf{sender}\}\big),$$

where sender is the user who initiated the update/remove (A in our example) and recipients are the recipients of the update/remove. This expression computes the set of users whose additions have been processed by the current user γ.myId, but who were not yet known to sender when they sent the message. Along with the "ack", each of these users is sent a 2SM-encrypted direct message containing the per-member secret that the current user is incorporating into their ratchet. This forwarded secret allows the recipient, in process-ack, to update the ratchet for the ack's sender accordingly.

Note that this forwarding does not violate forward secrecy: an application message sent to the group can still only be decrypted by those users who were group members at the time of sending. The forwarding process only ensures that all current group members obtain the same ratchet states and update secrets, even though different members may process concurrent member additions and PCS key updates in different orders. In the example above, if B first processes the addition of D, the forwarding process allows D to decrypt messages sent by B after B has processed A's update.

To simplify the presentation in Figure 4, we assume that if the same user is added more than once to the same group, then each user addition results in a separate protocol instance. It is also possible to extend the protocol with explicit handling of the case where the same user is added multiple times, possibly concurrently, as we show in Appendix D.

7.5 Security Analysis

Our DCGKA protocol achieves almost optimal DCGKA security. The exception occurs in the face of concurrent updates or removals: if the adversary corrupts two group members A and B, who then both concurrently send updates (or are removed), the adversary can use the two corrupted states together to learn the update secrets for messages that are causal successors to both updates. This is contrary to the definition of DCGKA security with optimal PCS, defined in Appendix A, which states that the group state should heal from these corruptions once each corruption is causally succeeded by an update message. In principle this could be addressed by performing additional updates if concurrent updates or removals are detected, but a better solution in practice is to simply perform PCS updates frequently in any case.

Thus we prove DCGKA security with respect to a weaker security notion, formalized by the predicate **dom-safe** in Appendix E. If the adversary corrupts multiple group members, this security notion requires that those corruptions are healed only once all corrupted group members have sent updates and one of these update messages is causally greater than all of the others. Concurrent update messages can then heal a corruption of any one of the updating group members, but cannot heal a corruption of multiple group members until another update message is sent that is causally greater than all of them.

Theorem 8. *Model* HKDF *as a random oracle, and model each pair* $(\mathsf{PKI\text{-}PublicKey}(\mathsf{ID}, \mathsf{ID}'), \mathsf{PKI\text{-}SecretKey}(\mathsf{ID}, \mathsf{ID}'))$ *as a random public-key encryption key pair, with* $\mathsf{PKI\text{-}PublicKey}(\mathsf{ID}, \mathsf{ID}')$ *public but* $\mathsf{PKI\text{-}SecretKey}(\mathsf{ID}, \mathsf{ID}')$ *callable only by* ID*. Let* λ *be the bit length of random values output by* KGen *and* HKDF*, and let the* $\mathsf{2SM}$ *protocol use a* (t', ϵ_{pke})*-CPA-secure PKE scheme. Then the protocol in Figure 4 is non-adaptively* $(t, c, n, \textbf{dom-safe}, \mathsf{SR\text{-}DGM}, \epsilon)$ *secure in the sense of Definition 9 in Appendix A, for* $t \approx t'$ *and*

$$\epsilon = 2c\left(2qn^2\epsilon_{pke} + qnt2^{-\lambda} + \binom{qn}{2}2^{-\lambda}\right).$$

The proof appears in Appendix E.

Another security consideration of our protocol is that all group membership operations (including the IDs of users being added or removed) are sent in plaintext. This simplifies some edge cases: for example, it allows concurrently added group members to add these operations to their $\gamma.\mathsf{history}$, enabling them to correctly compute the DGM function. We leave metadata privacy for these group membership operations for future work.

7.6 From DCGKA to Decentralized Secure Group Messaging

Once DCGKA has established a key ratchet for each group member that is shared among the group members, it is straightforward to use this to implement decentralized secure group messaging. Every application message is encrypted using symmetric keys derived from the sender's key ratchet, like in the two-party Signal protocol and MLS. Here decentralization is no issue because Signal and MLS already allow application messages to be interleaved, or even delivered out-of-order within fixed "epochs".

More formally, when one of the algorithms in Figure 4 returns an update secret I for group member A, the secret I should be used to initialize a protocol for *forward-secure authenticated encryption with associated data* (FS-AEAD), as defined by Alwen, Coretti, and Dodis [1, §4.2]. FS-AEAD allows for symmetric-key message encryption and decryption, like AEAD, but using different deterministically-derived keys for each message, deleting each key after it is used for forward secrecy. The FS-AEAD protocol should be used for application message encryption by A, and for decryption by the other group members, until the next time the DCGKA protocol outputs an update secret for A. The sequence number of A's last ratchet update should be included in the associated data of these messages, so that recipients know which FS-AEAD protocol state to use for decryption.

8 Performance

In this section we examine the performance of our protocol as a function of n, the number of group members.

8.1 Asymptotic Performance Analysis

In general, causal broadcast requires additional metadata in every message to establish the causal ordering. The size of this metadata is proportional to the number of concurrently sent messages by different group members, $\mathcal{O}(n)$ in the worst case [10]. However, we are able to reduce this overhead to zero because our DCGKA protocol does not require full causally ordered delivery. Instead, DCGKA only requires that:

- each group member's messages are delivered in order;
- acknowledgment messages are delivered after the message they acknowledge.

The acknowledgment messages and the sequence numbers that are already contained in DCGKA messages in plaintext are sufficient to ensure this. Additionally, Authenticated Causal Broadcast requires a PCS signature to authenticate the sender of each message (Section 5.2), adding a constant-size overhead to each message. The encryption that is applied to application messages (Section 7.6) and direct messages ($\mathsf{2SM}$) also adds a constant overhead. Each direct message also requires a constant number of public-key operations on both the sender and the recipient side.

Each create, update, or remove DCGKA operation broadcasts one constant-size control message and sends $\mathcal{O}(n)$ constant-size direct messages. Each other group member replies by broadcasting a constant-size acknowledgment, resulting in $\mathcal{O}(n)$ network traffic overall. The operation requires $\mathcal{O}(n)$ public key operations at the sender, and $\mathcal{O}(1)$ public key operations for each other group member.

Add operations send one constant-size control message and one direct message (the welcome message to the new user), and require $\mathcal{O}(1)$ public key operations at the sender. Each other group member broadcasts a constant-size acknowledgment and sends one constant-size direct message to the new member, resulting in $\mathcal{O}(n)$ network traffic overall. The acknowledgments require in total $\mathcal{O}(n)$ public key operations by the added user and $\mathcal{O}(1)$ public key operations by each other group member. The welcome message contains the history of group membership operations; in principle, its size is proportional to the number of previous membership operations and their acknowledgments—typically $\mathcal{O}(n^2)$. In practice, the welcome message only needs to include enough data to be able to evaluate the DGM function. Techniques from *Conflict-free Replicated Data Types* (CRDTs) [39, 41] can be used to design efficient representations of group membership history.

As an optimization, a group member can choose to delay sending acknowledgments until the next time it performs a PCS update, membership operation, or wants to send an application message. This may allow the protocol to coalesce multiple acknowledgments into a single message. Such a delay does not affect the security properties of the protocol.

This analysis assumes that the underlying network supports broadcast messaging. If it does not, each broadcast message must become $\mathcal{O}(n)$ unicast messages. However, these $\mathcal{O}(n)$ messages need not all be sent by the same sender: many group members (and, optionally, some number of untrusted servers) can be involved in disseminating a broadcast message by using a suitable network topology, e.g. a mesh network or gossip protocol. In many such scenarios, the cost of a broadcast message to each group member remains $\mathcal{O}(1)$.

8.2 Implementation and Empirical Measurements

We have implemented a prototype of our DCGKA algorithm in around 3500 lines of Java. The implementation is available as an open-source project on GitHub.[3] We use a Java implementation of Curve25519 [7];[4] all other cryptographic primitives use the built-in cryptography providers of the JVM. We ran the evaluation using OpenJDK 8 on a single machine with 16 GiB memory and an 8-core Intel i7 processor.

Our implementation demonstrates that the performance of DCGKA is good enough for practical use in medium-sized groups of up to 128 members, even with an implementation that is not highly optimized. In our experiments we execute multiple test scenarios consisting of an initial group setup followed by a

[3]TODO

[4]`https://github.com/trevorbernard/curve25519-java`

single group membership, PCS update, or message send operation. We measure the network traffic and CPU time resulting from that operation (including the processing of messages at all group members, and including any acknowledgments). We run all clients as separate threads in a single process and pass messages as serialized byte arrays. Hash functions and symmetric encryption use a 128-bit security level.

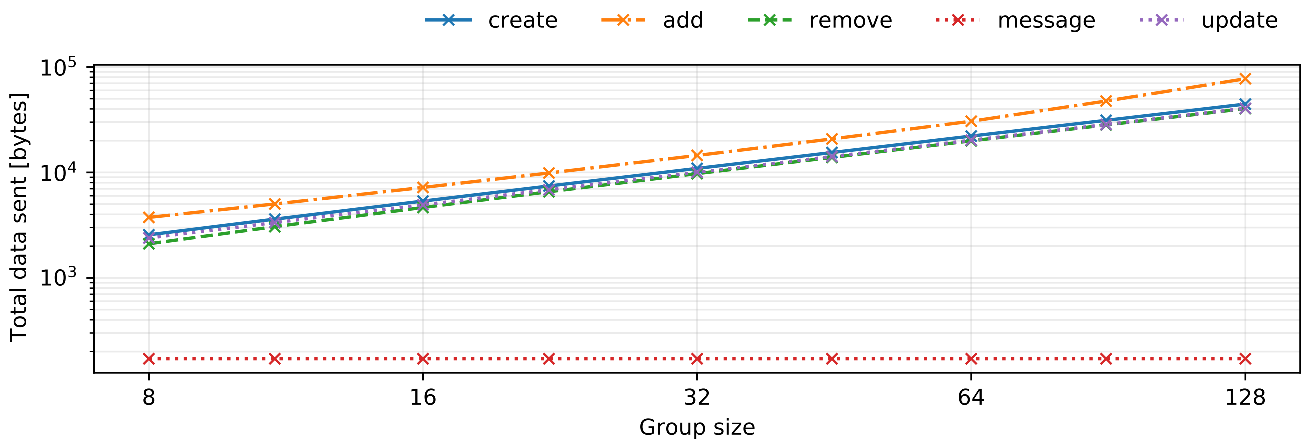

Figure 5: The total data volume sent by all clients while executing each type of operation, for groups ranging from 8 to 128 members. Broadcast messages are counted as a single outgoing message.

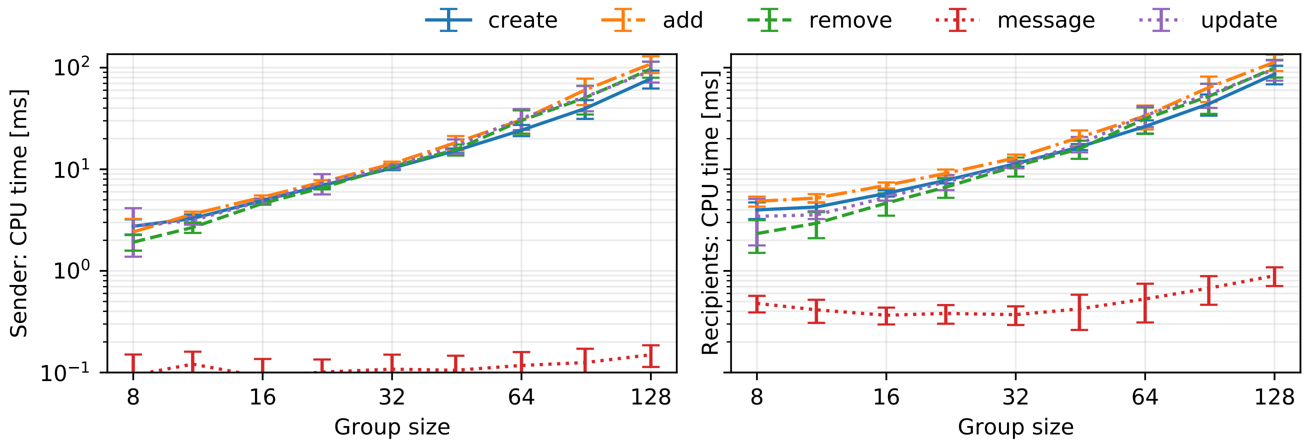

Figure 6: The CPU time (on a single core) to execute an operation, per sender or recipient, for groups ranging from 8 to 128 members. The error bars show the standard deviation over 25 independent executions.

Figure 5 shows that the total traffic for creating a group, adding a group member, or removing a group member grows linearly with the group size, as expected. Creating a new group of 128 members results in 43.4 kB being sent, and PCS updates (39.6 kB) and the group membership operations add (75.5 kB) and remove (39.3 kB) are in the same order of magnitude. Sending an application message incurs a constant overhead of 139 bytes regardless of group size. For our evaluation we send a 32 byte payload.

Figure 6 shows that the average computational effort on the sender or recipient side does not exceed 150 ms for group creation, PCS update, and membership operations on groups up to 128 members. For groups up to 64 members, the CPU times are less than 50 ms. Sending and receiving messages is very fast, taking less than 2 ms regardless of group size. Comparing these results with an average mobile network latency of around 50 ms, these results support our conclusion that DCGKA is practicable for real-world applications with medium-sized groups.

9 Conclusions and Future Work

In this paper we have shown, for the first time, how to enable secure group messaging with strong security guarantees (end-to-end encryption with forward secrecy and post-compromise security) in a decentralized and asynchronous setting. While the basic idea of sending secrets over two-party secure channels is simple, many details require careful design in order to meet our objectives: in particular, defining the set of current group members in the face of concurrent additions and removals, and ensuring that all group members obtain the same keys when users are added concurrently with other group members performing PCS updates.

Centralized protocols avoid such challenges by sequencing all updates through a semi-trusted server or consensus protocol. However, such centralization is undesirable in many settings, such as anonymous communication (mix networks), mesh networks, and peer-to-peer settings. By avoiding such centralization, our protocol allows secure group messaging to be deployed on any type of network, regardless of its topology. Even during a network partition, any subset of group members who are able to physically exchange messages can continue to communicate, update keys, and add or remove group members as usual. This gives our protocol much better robustness and censorship resistance than approaches based on a server that can become a single point of failure.

The downside of our protocol is that group membership operations and PCS key updates have $\mathcal{O}(n)$ cost in computation and network traffic for a group with n members, whereas the centralized MLS protocol requires only one broadcast message of size $\mathcal{O}(\log n)$ for the same operations [5]. We have shown in Section 8 that our $\mathcal{O}(n)$ cost is acceptable even for groups of over 100 members. For significantly larger groups, the MLS approach may be preferable, since MLS is designed to allow up to 50,000 members per group. However, it is debatable whether secure messaging for groups of thousands of people has a plausible threat model: large groups are more easily infiltrated by agents of the adversary, making the protocol's confidentiality properties irrelevant.

Beyond this paper, there are many open problems and interesting directions for future work:

Breaking the $\Omega(n)$ barrier. Is it possible to design a decentralized group messaging protocol that scales sub-linearly with the group size? It would be interesting to either break through the $\Omega(n)$ barrier (while maintaining FS and PCS), or to prove its impossibility. One approach might be to use constant-size puncturable public keys [21].

Metadata privacy. It would be desirable to protect metadata such as group membership operations from eavesdroppers. This is complicated by the fact that new group members must be able to learn about group membership operations that are concurrent to their addition.

Malicious user tolerance. Our protocol assumes that all group members follow the protocol correctly; a malicious group member can put the group into an inconsistent state by sending different encrypted seed secrets to different recipients as part of the same update. A more robust protocol would detect and prevent such protocol violations.

Tolerating bad randomness. Our 2SM protocol is vulnerable if a user's random number generator is compromised. We believe this can be addressed by combining new and old key pairs in a homomorphic way, as in [24].

Stronger security proofs. Our security proofs make some simplifying assumptions: we consider only non-adaptive adversaries; we ignore the effects of bad or leaked randomness, or users who fail to erase old secrets; and we treat hash functions as random oracles. It would be preferable to reduce these assumptions if possible.

Acknowledgments

We thank Giulia Fanti, Heather Miller, Justin Raizes, Katriel Cohn-Gordon, and Mansoor Ahmed-Rengers for providing feedback on various drafts of this work. Matthew Weidner was partially supported by a Churchill Scholarship from the Winston Churchill Foundation of the USA. Martin Kleppmann is supported by a Leverhulme Trust Early Career Fellowship and by the Isaac Newton Trust. Daniel Hugenroth is supported by a Nokia Bell Labs Scholarship and the Cambridge European Trust.

References

[1] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The Double Ratchet: Security notions, proofs, and modularization for the Signal protocol. In *Advances in Cryptology – EUROCRYPT 2019*, pages 129–158. Springer, 2019. Full version: `https://eprint.iacr.org/2018/1037`.

[2] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. Cryptology ePrint 2019/1189, 2019. `https://eprint.iacr.org/2019/1189`.

[3] Jacob Aron and Aviva Rutkin. Hong Kong protesters use a mesh network to organise. *New Scientist*, September 2017. archived at `https://perma.cc/VKH7-KE9K`.

[4] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. SoK: Consensus in the age of blockchains. In *1st ACM AFT*, page 183–198. ACM, 2019.

[5] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-09, Internet Engineering Task Force, March 2020. Work in Progress.

[6] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In *Advances in Cryptology – CRYPTO 2017*, pages 619–650. Springer, 2017. Full version: `https://eprint.iacr.org/2016/1028`.

[7] Daniel J Bernstein. Curve25519: New Diffie-Hellman speed records. In *9th International Conference on Theory and Practice in Public-Key Cryptography (PKC)*, pages 207–228. Springer, April 2006.

[8] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous decentralized key management for large dynamic groups. Messaging Layer Security mailing list, 2018.

[9] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[10] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Proc. Letters*, 39(1):11–16, July 1991.

[11] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila. A formal security analysis of the Signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 451–466, April 2017.

[12] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On post-compromise security. In *29th IEEE Computer Security Foundations Symposium (CSF)*, pages 164–178. IEEE, June 2016.

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

[14] Cas Cremers, Britta Hale, and Konrad Kohbrok. Efficient post-compromise security beyond one group. Cryptology ePrint Archive, Report 2019/477, 2019. `https://eprint.iacr.org/2019/477`.

[15] 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.

[16] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical Report ADA465464, Naval Research Laboratory, Washington DC, 2004.

[17] F. Betül Durak and Serge Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. In *Advances in Information and Computer Security*, pages 343–362. Springer, 2019. Full version: `https://eprint.iacr.org/2018/889`.

[18] Alexey Ermishkin. Subject: [MLS] Message ordering. MLS Mailing List, May 2018.

[19] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, 1988.

[20] Seth Gilbert and Nancy A Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, June 2002.

[21] Matthew D. Green and Ian Miers. Forward secure asynchronous messaging from puncturable encryption. In *2015 IEEE Symposium on Security and Privacy*, page 305–320. IEEE, 2015.

[22] Matthew Hodgson. Subject: [MLS] MLS in decentralised environments. MLS Mailing List, April 2018.

[23] Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In *CRYPTO 2018*, pages 33–62. Springer, 2018. Full version: `https://eprint.iacr.org/2018/553`.

[24] Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In *EUROCRYPT 2019*, pages 159–188, 2019. Full version: `https://eprint.iacr.org/2018/954.pdf`.

[25] Martin Kleppmann, Stephan A Kollmann, Diana A Vasile, and Alastair R Beresford. From secure messaging to secure collaboration. In *26th International Workshop on Security Protocols*, pages 179–185. Springer, March 2018.

[26] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: You own your data, in spite of the cloud. In *2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 154–178. ACM, October 2019.

[27] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *CRYPTO 2010*, pages 631–648. Springer, August 2010.

[28] Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010.

[29] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, July 1978.

[30] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic curves for security. RFC 7748, January 2016.

[31] Moxie Marlinspike and Trevor Perrin. The X3DH key agreement protocol. Technical Report Revision 1, 2016. archived at `https://perma.cc/633M-J2WM`.

[32] Matrix.org Foundation. Client-Server API: 10.4 room membership, 2019. archived at `https://perma.cc/7SZV-ZV9B`.

[33] Matrix.org Foundation. End-to-end encryption implementation guide, 2019. archived at `https://perma.cc/75RC-HS9B`.

[34] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel & Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[35] 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-04, Internet Engineering Task Force, January 2020. Work in Progress.

[36] Trevor Perrin and Moxie Marlinspike. The Double Ratchet algorithm. Technical Report Revision 1, November 2016. archived at `https://perma.cc/AJL9-MBSB`.

[37] Ania M Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The Loopix anonymity system. In *USENIX Security Symposium*, 2017.

[38] Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In *CRYPTO 2018*, pages 3–32. Springer, 2018. Full version: `https://eprint.iacr.org/2018/296`.

[39] Nuno M. Preguiça, Carlos Baquero, and Marc Shapiro. Conflict-free replicated data types (CRDTs). *CoRR*, abs/1805.06358, 2018.

[40] 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 EuroS&P*, pages 415–429, April 2018.

[41] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, October 2011.

[42] Lokman Tsui. The coming colonization of Hong Kong cyberspace: government responses to the use of new technologies by the umbrella movement. *Chinese J. Comm*, 8(4):1–9, 2015.

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

[44] Richard van der Hoff. Megolm group ratchet, 2019.

[45] Matthew Weidner. Group messaging for secure asynchronous collaboration. Master's thesis, University of Cambridge, Cambridge, UK, June 2019. archived at `https://perma.cc/XA8S-BHFN`.

[46] Matthew Weidner. Subject: [MLS] Proposals for handling concurrent messages. MLS Mailing List, March 2019.

[47] WhatsApp. WhatsApp encryption overview, 2017. archived at `https://perma.cc/QD7M-GPG5`.

A Security Game for DCGKA

The correctness and security of a DCGKA scheme are formally captured by the security game in Figure 7. The game definition and our description of it are based on the CGKA version introduced by Alwen et al. [2, §3.2]. DGM denotes the DGM scheme that we are using to determine group membership. The game is a key indistinguishability game, parameterized by the random bit b, which determines whether challenges return actual update secrets or random values. The adversary's goal is to guess b.

```
init
    $b \leftarrow\!\$\ \{0,1\}$
    $\forall \mathsf{ID} : \gamma[\mathsf{ID}] \leftarrow \mathsf{init}(\mathsf{ID})$
    $\mathsf{counter}[\cdot] \leftarrow 0$
    public $\mathsf{controlMsgs}[\cdot,\cdot] \leftarrow \varepsilon$
    public $\mathsf{directMsgs}[\cdot,\cdot,\cdot] \leftarrow \varepsilon$
    $\mathbf{I}[\cdot,\cdot] \leftarrow \varepsilon$
    $\mathsf{needsResponse}[\cdot,\cdot] \leftarrow \mathsf{false}$
    $\mathsf{challenged}[\cdot,\cdot] \leftarrow \mathsf{false}$
    $\mathsf{delivered}[\cdot,\cdot,\cdot] \leftarrow \mathsf{false}$
    $\mathsf{addTarget}[\cdot,\cdot] \leftarrow \varepsilon$

reveal($\mathsf{ID}, c$)
    require $\mathbf{I}[\mathsf{ID}, c] \neq \varepsilon$
    require $\neg\mathsf{challenged}[\mathsf{ID}, c]$
    $\mathsf{challenged}[\mathsf{ID}, c] \leftarrow \mathsf{true}$
    return $\mathbf{I}[\mathsf{ID}, c]$

challenge($\mathsf{ID}, c$)
    require $\mathbf{I}[\mathsf{ID}, c] \neq \varepsilon$
    require $\neg\mathsf{challenged}[\mathsf{ID}, c]$
    $I_0 \leftarrow \mathbf{I}[\mathsf{ID}, c]$
    $I_1 \leftarrow\!\$\ \mathsf{KGen}()$
    $\mathsf{challenged}[\mathsf{ID}, c] \leftarrow \mathsf{true}$
    return $I_b$

corrupt($\mathsf{ID}$)
    return $\gamma[\mathsf{ID}]$
```

```
create-group($\mathsf{ID}_0, \{\mathsf{ID}_1, \ldots, \mathsf{ID}_n\}$)
    require controlMsgs is empty
    require $\mathsf{ID}_0 \notin \{\mathsf{ID}_1, \ldots, \mathsf{ID}_n\}$
    $(\gamma[\mathsf{ID}_0], \mathsf{control}, \mathsf{dmsgs}, I) \leftarrow$
        $\mathsf{create}(\gamma[\mathsf{ID}_0], \mathsf{ID}_1, \ldots, \mathsf{ID}_n)$
    if $\mathsf{control} = \varepsilon \vee I = \varepsilon$ then win
    $\mathsf{controlMsgs}[\mathsf{ID}_0, 1] \leftarrow \mathsf{control}$
    $\mathsf{directMsgs}[\mathsf{ID}_0, 1] \leftarrow \mathsf{dmsgs}$
    $\mathbf{I}[\mathsf{ID}_0, 1] \leftarrow I$
    $\mathsf{needsResponse}[\mathsf{ID}_0, 1] \leftarrow \mathsf{true}$
    $\mathsf{counter}[\mathsf{ID}_0] \leftarrow 1$
    $\mathsf{delivered}[\mathsf{ID}_0, 1, \mathsf{ID}_0] \leftarrow \mathsf{true}$

add-user($\mathsf{ID}, \mathsf{ID}'$)
    require $\mathsf{valid\text{-}member}(\mathsf{ID}) \wedge \mathsf{ID} \neq \mathsf{ID}'$
    $c \leftarrow {+}{+}\mathsf{counter}[\mathsf{ID}]$
    $(\gamma[\mathsf{ID}], \mathsf{control}, \mathsf{dmsgs}, I) \leftarrow \mathsf{add}(\gamma[\mathsf{ID}], \mathsf{ID}')$
    if $\mathsf{control} = \varepsilon \vee I = \varepsilon$ then win
    $\mathsf{controlMsgs}[\mathsf{ID}, c] \leftarrow \mathsf{control}$
    $\mathsf{directMsgs}[\mathsf{ID}, c] \leftarrow \mathsf{dmsgs}$
    $\mathbf{I}[\mathsf{ID}, c] \leftarrow I$
    $\mathsf{needsResponse}[\mathsf{ID}, c] \leftarrow \mathsf{true}$
    $\mathsf{addTarget}[\mathsf{ID}, c] \leftarrow \mathsf{ID}'$

remove-user($\mathsf{ID}, \mathsf{ID}'$)
    require $\mathsf{valid\text{-}member}(\mathsf{ID}) \wedge \mathsf{ID} \neq \mathsf{ID}'$
    $c \leftarrow {+}{+}\mathsf{counter}[\mathsf{ID}]$
    $(\gamma[\mathsf{ID}], \mathsf{control}, \mathsf{dmsgs}, I) \leftarrow$
        $\mathsf{remove}(\gamma[\mathsf{ID}], \mathsf{ID}')$
    if $\mathsf{control} = \varepsilon \vee I = \varepsilon$ then win
    $\mathsf{controlMsgs}[\mathsf{ID}, c] \leftarrow \mathsf{control}$
    $\mathsf{directMsgs}[\mathsf{ID}, c] \leftarrow \mathsf{dmsgs}$
    $\mathbf{I}[\mathsf{ID}, c] \leftarrow I$
    $\mathsf{needsResponse}[\mathsf{ID}, c] \leftarrow \mathsf{true}$
```

```
send-update($\mathsf{ID}$)
    require $\mathsf{valid\text{-}member}(\mathsf{ID}) \wedge \mathsf{ID} \neq \mathsf{ID}'$
    $c \leftarrow {+}{+}\mathsf{counter}[\mathsf{ID}]$
    $(\gamma[\mathsf{ID}], \mathsf{control}, \mathsf{dmsgs}, I) \leftarrow \mathsf{update}(\gamma[\mathsf{ID}])$
    if $\mathsf{control} = \varepsilon \vee I = \varepsilon$ then win
    $\mathsf{controlMsgs}[\mathsf{ID}, c] \leftarrow \mathsf{control}$
    $\mathsf{directMsgs}[\mathsf{ID}, c] \leftarrow \mathsf{dmsgs}$
    $\mathbf{I}[\mathsf{ID}, c] \leftarrow I$
    $\mathsf{needsResponse}[\mathsf{ID}, c] \leftarrow \mathsf{true}$

deliver($\mathsf{ID}, c, \mathsf{ID}'$)
    require $\mathsf{controlMsgs}[\mathsf{ID}, c] \neq \varepsilon$
    require $\neg\mathsf{in\text{-}history}[\mathsf{ID}, c, \mathsf{ID}']$
    require $\mathsf{should\text{-}receive}(\mathsf{ID}, c, \mathsf{ID}')$
    require $\mathsf{causally\text{-}ready}(\mathsf{ID}, c, \mathsf{ID}') \vee$
        $\mathsf{add\text{-}ready}(\mathsf{ID}, c, \mathsf{ID}')$
    $(\gamma[\mathsf{ID}'], \mathsf{control}, \mathsf{dmsgs}, I, I') \leftarrow$
        $\mathsf{process}(\gamma[\mathsf{ID}'], \mathsf{ID}, \mathsf{controlMsgs}[\mathsf{ID}, c],$
            $\mathsf{directMsgs}[\mathsf{ID}, c, \mathsf{ID}'])$
    if $\mathsf{should\text{-}decrypt}(\mathsf{ID}, c, \mathsf{ID}')$ then
        if $I \neq \mathbf{I}[\mathsf{ID}, c]$ then win
    else if $I \neq \varepsilon$ then win
    $\mathsf{mustRespond} \leftarrow \big(\mathsf{needsResponse}[\mathsf{ID}, c] \wedge$
        $\mathsf{should\text{-}decrypt}(\mathsf{ID}, c, \mathsf{ID}')\big) \vee$
        $\mathsf{adds\text{-}member}(\mathsf{ID}, c, \mathsf{ID}')$
    if $\mathsf{mustRespond} \wedge (\mathsf{control} = \varepsilon \vee I' = \varepsilon)$
        then win
    if $\mathsf{control} \neq \varepsilon$ then
        $c' \leftarrow {+}{+}\mathsf{counter}[\mathsf{ID}']$
        $\mathsf{controlMsgs}[\mathsf{ID}', c'] \leftarrow \mathsf{control}$
        $\mathsf{directMsgs}[\mathsf{ID}', c'] \leftarrow \mathsf{dmsgs}$
        $\mathbf{I}[\mathsf{ID}', c'] \leftarrow I'$
        $\mathsf{needsResponse}[\mathsf{ID}', c'] \leftarrow \mathsf{false}$
    $\mathsf{delivered}[\mathsf{ID}, c, \mathsf{ID}'] \leftarrow \mathsf{true}$
```

Figure 7: Oracles of security game for DCGKA(DGM)

Initialization The **init** oracle sets up the game and all the variables needed to keep track of the execution. The random bit b is used for real-or-random challenges. The dictionary γ keeps track of all of the users' states, while counter[ID] stores the number of messages that have been sent so far

$$\mathsf{valid\text{-}member}(\mathsf{ID}) := \exists (T \in \mathsf{controlMsgs})\ (\mathsf{delivered}[T, \mathsf{ID}])$$

$$\mathsf{in\text{-}history}(\mathsf{ID}, c, \mathsf{ID}') := \exists (T \in \mathsf{controlMsgs})\ \big(\mathsf{controlMsgs}[\mathsf{ID}, c] \preceq T \wedge \mathsf{delivered}[T, \mathsf{ID}']\big)$$

$$\mathsf{causally\text{-}ready}(\mathsf{ID}, c, \mathsf{ID}') := \forall (T \in \mathsf{controlMsgs})\ \big(T \prec \mathsf{controlMsgs}[\mathsf{ID}, c] \Longrightarrow \mathsf{in\text{-}history}(T, \mathsf{ID}')\big)$$

$$\begin{aligned}\mathsf{add\text{-}ready}(\mathsf{ID}, c, \mathsf{ID}') := {} & (\mathsf{addTarget}[\mathsf{ID}, c] = \mathsf{ID}') \wedge \\ & \forall (T \in \mathsf{controlMsgs})\ \big(\mathsf{should\text{-}decrypt}(T, \mathsf{ID}') \Longrightarrow \mathsf{delivered}[T, \mathsf{ID}']\big)\end{aligned}$$

$$\mathsf{should\text{-}decrypt}(\mathsf{ID}, c, \mathsf{ID}') := \mathsf{ID}' \in \mathsf{DGM}\big(\{T \in \mathsf{controlMsgs} \mid T \preceq \mathsf{controlMsgs}[\mathsf{ID}, c]\}\big)$$

$$\mathsf{should\text{-}receive}(\mathsf{ID}, c, \mathsf{ID}') := \mathsf{ID}' \in \mathsf{DGM}\big(\{T \in \mathsf{controlMsgs} \mid T \preceq \mathsf{controlMsgs}[\mathsf{ID}, c] \vee \mathsf{in\text{-}history}(T, \mathsf{ID}')\}\big)$$

$$\begin{aligned}\mathsf{adds\text{-}member}(\mathsf{ID}, c, \mathsf{ID}') := {} & \big(\textbf{let } S = \{T \in \mathsf{controlMsgs} \mid \mathsf{in\text{-}history}(T, \mathsf{ID}')\} \textbf{ in} \\ & (\mathsf{DGM}(S \cup \{\mathsf{controlMsgs}[\mathsf{ID}, c]\}) \setminus \mathsf{DGM}(S) \neq \emptyset)\big)\end{aligned}$$

Figure 8: Predicates used in the security game. Here we use $\mathsf{delivered}[T, \mathsf{ID}]$ as an abbreviation for $\mathsf{delivered}[\mathsf{ID}', c', \mathsf{ID}]$ when $T = \mathsf{controlMsgs}(\mathsf{ID}', c')$, and likewise for inputs to the predicates.

by ID. Note that these counters are never reset, unlike the variable ctr in the CGKA security game, which is reset with each CGKA epoch. $\mathsf{controlMsgs}[\mathsf{ID}, c]$ stores the c-th control message generated by ID, while $\mathsf{directMsgs}[\mathsf{ID}, c, \mathsf{ID}']$ stores the corresponding direct message intended for ID'. Corresponding to $\mathsf{controlMsgs}[\mathsf{ID}, c]$, $\mathbf{I}[\mathsf{ID}, c]$ stores the update secret output by the sender, $\mathsf{needsResponse}[\mathsf{ID}, c]$ stores whether recipients are required to return an acknowledgment when processing $\mathsf{controlMsgs}[\mathsf{ID}, c]$ (i.e., it is an output of create add, remove, or update), and, $\mathsf{challenged}[\mathsf{ID}, c]$ stores whether $\mathbf{I}[\mathsf{ID}, c]$ has been challenged or revealed by the adversary. Additionally, for each user ID', $\mathsf{delivered}[\mathsf{ID}, c, \mathsf{ID}']$ indicates whether $\mathsf{controlMsgs}[\mathsf{ID}, c]$ has been delivered to ID'. Finally, $\mathsf{addTarget}[\mathsf{ID}, c]$ is ID' if $\mathsf{controlMsgs}[\mathsf{ID}, c]$ is an add message that added ID', or ε if it is not an add message.

Both controlMsgs and directMsgs are marked **public**, indicating that they are readable by the adversary.

Group Creation The **create-group** oracle causes ID_0 to create a group with members $\mathsf{ID}_0, \ldots, \mathsf{ID}_n$. It requires that no previous messages have been sent, i.e., the game has just started (if the **require** statement fails, the game aborts and the adversary loses). To avoid trivial protocols that do not output any update secrets, create must output a non-ε control message and update secret; if not, we reveal b to the adversary, indicated by the keyword **win**. We store the returned messages and update secret, increment the sender's counter, and mark the message as delivered to its sender. Here we use the notation $\mathsf{directMsgs}[\mathsf{ID}_0, 1] \leftarrow \mathsf{dmsgs}$ to mean $\mathsf{directMsgs}[\mathsf{ID}_0, 1, \mathsf{ID}'] \leftarrow \mathsf{dmsg}$ for each pair $(\mathsf{ID}', \mathsf{dmsg}) \in \mathsf{dmsgs}$. To avoid trivial protocols, we set $\mathsf{needsResponse}[\mathsf{ID}_0, 1] \leftarrow \mathsf{true}$, ensuring that other group members will output response messages and update secrets of their own when processing the create message.

Adding, removing, and performing updates The three oracles add-user, remove-user, and send-update allow the adversary to cause some user to call the corresponding algorithm. The predicate valid-member ensures that the sender has received a message before, so that their control messages are related to the existing group. In contrast to the CGKA security game, we do not make any check that the requested group membership operations are "reasonable", e.g., the removed user is already in the group. This is because the DGM scheme may assign some significance to seemingly redundant or unreasonable operations.

Delivering control messages The oracle $\mathsf{deliver}(\mathsf{ID}, c, \mathsf{ID}')$ delivers $\mathsf{controlMsgs}[\mathsf{ID}, c]$ (together with the direct message $\mathsf{directMsgs}[\mathsf{ID}, c, \mathsf{ID}'])$ to user ID'. It first makes several ordering-related checks, which formalize the precise delivery requirements for messages (discussed briefly in Section 5):

- The predicate $\mathsf{in\text{-}history}$ is used to ensure that ID' has not already been delivered this message or a causally later one. The latter case can occur if ID' was already delivered a message adding them to the group which is causally greater than this message, hence should have informed of ID' of any relevant metadata about this message.
- The predicate $\mathsf{should\text{-}receive}$ ensures that ID' is eligible to receive this message. We dictate that ID' is eligible to receive the message if they consider themselves group members after receiving the message, taking into account all messages they have received already. This rules out confusing situations in which group members consider themselves not in the group, but allows them to receive messages besides just the ones they should be able to decrypt—in particular, messages concurrent to their own addition. Allowing group members to receive messages concurrent to their addition allows us to simplify our DCGKA protocol, since then group members can learn the metadata (but not update secrets) of group membership changes and acknowledgments concurrent to their addition, which are impossible to include in the message adding them to the group.
- The predicate $\mathsf{causally\text{-}ready}$ ensures that ID' processes messages in causal order, by requiring them to have received all causally lesser messages.
- The exception to this rule is encoded by $\mathsf{add\text{-}ready}$: a group member may receive a message adding them to the group without receiving prior messages. In particular, the first message a group member receives will always be the message adding them to the group. However, this would get confusing if a group member was added to the group, removed, and then re-added later, and they were allowed to receive the second add message before receiving all messages in between the first add and remove, since they would later have to go back and process old messages. To avoid this, $\mathsf{add\text{-}ready}$ requires ID' to first receive any causally prior messages that they $\mathsf{should\text{-}decrypt}$.

These restrictions on the adversary's ability to deliver messages are reasonable because they can easily be enforced by the Authenticated Causal Broadcast layer, as discussed in Section 5.[5]

After checking these conditions, the oracle delivers the message to ID'. To ensure correctness, we mandate that the output secret I is correct (i.e., equal to the update secret returned when its sender processed the message) if ID' should be able to decrypt it. This occurs precisely if they were an intended recipient of the message.

Finally, delivering the message may cause ID' to return a response control message, which is handled like any other generated message. The variable $\mathsf{mustRespond}$ mandates that this response exists and has an associated update secret if it is required by $\mathsf{needsResponse}$ (i.e., the delivered message was output by create, add, remove, or update) and ID' should be able to decrypt the processed message. A response is also required if the message adds a group member, from the perspective of ID', since the added group member needs an update secret from ID' in order to decrypt their application messages; this is checked by $\mathsf{adds\text{-}member}$. If a response is not required, ID' may choose to output a response regardless, in which case we do not require their response to have an associated update secret.

[5]The Causal Broadcast layer can enforce the restriction in $\mathsf{add\text{-}ready}$ by including, along with each message adding ID', a vector clock describing the causally maximal prior messages T satisfying $\mathsf{should\text{-}decrypt}(T, \mathsf{ID}')$. The Causal Broadcast layer for ID' would then delay processing the add message until its own vector clock was at least as large.

$$\begin{aligned}
&\mathsf{corrupts}(\mathbf{q}_i = \mathbf{challenge}(\mathsf{ID}, c), \mathbf{q}_j = \mathbf{corrupt}(\mathsf{ID}')) := \\
&\qquad \big(\mathsf{ID}' \text{ was added to the group by a} \\
&\qquad\qquad \text{message} \preceq \mathsf{message}(\mathsf{ID}, c)\big) \land \\
&\qquad (\mathsf{delivered}[\mathsf{ID}, c, \mathsf{ID}'] \text{ was false at the time of } \mathbf{q}_j)
\end{aligned}$$

$$\begin{aligned}
&\mathbf{safe}(\mathbf{q}_1, \ldots, \mathbf{q}_q) := \\
&\qquad \forall \big((i, j) \text{ s.t. } \mathbf{q}_i = \mathbf{challenge}(\mathsf{ID}, c) \text{ for some } \mathsf{ID}, c \text{ and} \\
&\qquad\qquad \mathbf{q}_j = \mathbf{corrupt}(\mathsf{ID}') \text{ for some } \mathsf{ID}'\big) \\
&\qquad \Big(\mathsf{corrupts}(\mathbf{q}_i, \mathbf{q}_j) \implies \\
&\qquad\qquad \neg\mathsf{should\text{-}decrypt}[\mathsf{ID}, c, \mathsf{ID}'] \lor \\
&\qquad\qquad \exists k \ \big(\mathbf{q}_k = \mathbf{update}(\mathsf{ID}') \land k > j \ \land \\
&\qquad\qquad\qquad \mathsf{message}(\mathbf{q}_k) \preceq \mathsf{controlMsgs}[\mathsf{ID}, c]\big)\Big)
\end{aligned}$$

Figure 9: Safety predicate **safe** and its helper predicate corrupts. Here $\mathsf{message}(\mathbf{q}_k)$ denotes the control message generated during $\mathbf{q}_k$.

Challenges and corruptions Challenges and corruptions are handled as in the CGKA security game. In order to capture that update secrets must look random, the attacker is allowed to issue a challenge for any update secret, using **challenge**. The attacker may instead directly reveal the update secret using **reveal**. To model forward secrecy and post-compromise security, the attacker is allowed to learn the current state of any user by calling the oracle **corrupt**. Note that because of our requirements about when users output update secrets and response messages, we do not have to worry about trivial protocols that give the adversary nothing to challenge.

Avoiding trivial attacks At the end of the game, a *safety predicate* P (specified as a parameter to the security game) is evaluated on the sequence of queries $\mathbf{q}_1, \ldots, \mathbf{q}_q$ made by the adversary. If $\mathsf{P}(\mathbf{q}_q, \ldots, \mathbf{q}_q)$ is false, the adversary loses. The safety predicate is meant to prevent trivial attacks, in which the adversary corrupts a group member's state and then uses it directly to decrypt an update secret. Changing P changes the precise PCS and FS guarantees required by the security game.

The particular safety predicate **safe** in Figure 9 describes *optimal* PCS and FS, including optimal PCS in the face of concurrent updates: after multiple corruptions, update secrets must be safe once all corrupted group members update, even if they all update concurrently. To define **safe**, we first define a predicate $\mathsf{corrupts}(\mathbf{q}_{\text{corr}}, \mathbf{q}_{\text{chall}})$, which determines whether, in the absence of PCS and removals, the query $\mathbf{q}_{\text{corr}}$ allows the adversary to trivially learn the update secret challenged by $\mathbf{q}_{\text{chall}}$, by corrupting a group member who was added to the group causally prior to the challenged message, before that group member received the message. Observe that using $\forall(i, j)\ (\neg\mathsf{corrupts}(\mathbf{q}_i, \mathbf{q}_j))$ in place of our actual safety predicate would model forward secrecy: the adversary may only corrupt users who were added after a challenged message or had already received the challenged message. Instead, **safe** merely requires that each corruption was healed, either because the corrupted group member was no longer in the group at the time of the message, or because the corrupted member sent an update message after they were corrupted and causally prior to the challenged message.

Thus corruptions of a group member after they receive the challenged message are allowed, expressing FS, and corruptions healed by an update causally prior to the challenged message are allowed, expressing

PCS. Furthermore, group membership is enforced cryptographically, since the adversary is free to corrupt the states of any group members who are not intended recipients of the challenged message, modeling collusion between those members.

However, some protocols only achieve weaker forms of PCS or FS, including our own (cf. the last two columns in Table 1). Thus we instead prove security of our DCGKA protocol with respect to the security predicate **dom-safe** defined in Appendix E, which allows slightly sub-optimal PCS in the face of concurrent updates.

Advantage In the following, a *non-adaptive* (t, c, n)*-attacker* is an attacker $\mathcal{A}$ that runs in time at most t, makes at most c **challenge** queries, references at most n IDs, and must specify the sequence of queries it plans to make in advance, before seeing the result of any queries. The attacker wins the DCGKA security game if it correctly guesses the random bit b in the end and the safety predicate P evaluates to true on the queries made by the attacker. The *advantage* of $\mathcal{A}$ against a DCGKA scheme DCGKA with respect to the safety predicate P and DGM scheme DGM is defined by

$$\mathsf{Adv}^{\mathsf{DCKGA},\mathsf{P},\mathsf{DGM}}_{\text{dcgka-na}}(\mathcal{A}) := 2\left|\Pr[\mathcal{A} \text{ wins}] - \frac{1}{2}\right|.$$

Definition 9. (Non-adaptive DCGKA security) A DCGKA scheme DCGKA is *non-adaptively* $(t, c, n, \mathsf{P}, \mathsf{DGM}, \epsilon)$*-secure* if for all non-adaptive (t, c, n)-attackers $\mathcal{A}$,

$$\mathsf{Adv}^{\mathsf{DCKGA},\mathsf{P},\mathsf{DGM}}_{\text{dcgka-na}}(\mathcal{A}) \leq \epsilon.$$

B Public-Key Encryption

We reproduce a standard definition of public-key encryption for use in our two-party secure messaging protocol.

Definition 10. A *public-key encryption (DCGKA)* scheme $\mathsf{PKE} = (\mathsf{PKE\text{-}Gen}, \mathsf{PKE\text{-}Enc}, \mathsf{PKE\text{-}Dec})$ consists of the following algorithms:

Key generation $\mathsf{PKE\text{-}Gen}$ outputs a fresh key pair (pk, sk).

Encryption $\mathsf{PKE\text{-}Enc}$ takes a public key pk and a message m, and outputs a ciphertext c.

Decryption $\mathsf{PKE\text{-}Dec}$ takes a secret key sk and a ciphertext c, and outputs a message m.

Correctness of a PKE scheme is defined by

$$\Pr[(pk, sk) \leftarrow_{\$} \mathsf{PKE\text{-}Gen}, c \leftarrow_{\$} \mathsf{PKE\text{-}Enc}(pk, m),$$
$$m' \leftarrow \mathsf{PKE\text{-}Dec}(sk, c) : m' = m] = 1$$

for all messages m.

We require IND-CPA security. This means that for the security game, the challenger generates a key pair $(pk, sk) \leftarrow \mathsf{PKE\text{-}Gen}$ and a challenge bit b, gives pk to the adversary, and then inputs messages m_0, m_1 of the same length from the adversary, returning $\mathsf{PKE\text{-}Enc}(pk, m_b)$. The adversary wins if they correctly guess the bit b. The *advantage* of an adversary $\mathcal{A}$ against a PKE scheme PKE is

$$\mathsf{Adv}^{\mathsf{PKE}}_{\text{pke-ind-cpa}}(\mathcal{A}) := 2\left|\Pr[\mathcal{A} \text{ wins}] - \frac{1}{2}\right|.$$

Definition 11. A PKE scheme PKE is (t, ϵ)*-CPA-secure* if for all attackers $\mathcal{A}$ running in time t,

$$\mathsf{Adv}^{\mathsf{PKE}}_{\text{pke-ind-cpa}}(\mathcal{A}) \leq \epsilon.$$

```
init
    $b \leftarrow\!\$\ \{0,1\}$
    $(pk_A, sk_A) \leftarrow\!\$$ PKE-Gen()
    $(pk_B, sk_B) \leftarrow\!\$$ PKE-Gen()
    $\gamma_A \leftarrow$ 2SM-Init($sk_A, pk_B$)
    $\gamma_B \leftarrow$ 2SM-Init($sk_B, pk_A$)
    public ciphertexts$_A[\cdot]$, ciphertexts$_B[\cdot] \leftarrow \varepsilon$
    plaintexts$_A[\cdot]$, plaintexts$_B[\cdot] \leftarrow \varepsilon$
    nextSend$_A$, nextSend$_B \leftarrow 1$
    nextDelivered$_A$, nextDelivered$_B \leftarrow 1$
    return $(pk_A, pk_B)$

corrupt-A
    return $\gamma_A$

send-A($m$)
    plaintexts$_A$[nextSend$_A$] $\leftarrow m$
    ($\gamma_A$, ciphertexts$_A$[nextSend$_A$]) $\leftarrow$ 2SM-Send($\gamma_A, m$)
    nextSend$_A \leftarrow$ nextSend$_A + 1$

deliver-A
    require nextDelivered$_B <$ nextSend$_B$
    ($\gamma_A, m$) $\leftarrow$ 2SM-Receive($\gamma_A$, ciphertexts$_B$[nextDelivered$_B$])
    if $m \neq$ plaintexts$_B$[nextDelivered$_B$] then win
    nextDelivered$_B \leftarrow$ nextDelivered$_B + 1$

challenge-A($m_0, m_1$)
    require $|m_0| = |m_1|$
    send-A($m_b$)
```

Figure 10: Oracles of security game for 2SM (A only; oracles for B are analogous).

C Security of Our Two-Party Secure Messaging Scheme

In this section, we define and prove the security of our two-party secure messaging scheme 2SM from Section 4.1.

C.1 Security Game

Our 2SM security game is essentially the two-party restriction of our DCGKA security game, with every message counting as an update message, except that we use IND-CPA security instead of secret indistinguishability. The oracles of our security game appear in Figure 10.

The **init** oracle sets up the game and all the variables needed to keep track of the execution. The random bit b is used for IND-CPA challenges. The states of the two parties, A and B, are initialized using key pairs for a public-key encryption scheme PKE. ciphertexts$_A[c]$ and plaintexts$_A[c]$ store the ciphertext and plaintext, respectively, for the c-th message sent by A, and likewise for B. The ciphertexts arrays are marked **public**, indicating that they are readable by the adversary, and the parties' public keys are returned to the adversary.

After calling **init**, the adversary is free to call the remaining oracles to simulate message sending and receiving, corrupt the parties' states, and issue IND-CPA challenges. Note that **deliver-A** enforces authentic in-order delivery of messages from B (if the **require** statement fails, the game aborts and the adversary loses). However, messages can be arbitrarily delayed, and messages from the two parties may be interleaved. **deliver-A** checks that A correctly decrypts the original plaintext, enforcing correctness (otherwise **win** reveals b to the adversary).

At the end of the game, the predicate **2SM-safe** in Figure 11 is evaluated on the sequence of queries $\mathbf{q}_1, \ldots, \mathbf{q}_q$ made by the adversary, and if it is false, the adversary loses. This is to prevent trivial attacks, in which the adversary corrupts a group member's state and then uses it directly to decrypt a message. Specifically, **2SM-safe**($\mathbf{q}_1, \ldots, \mathbf{q}_q$) fails to hold if there are queries $\mathbf{q}_i =$ **corrupt-A**, $\mathbf{q}_j =$ **challenge-B**(m_0, m_1) such that A had not yet received the message corresponding to $\mathbf{q}_j$ at the time of $\mathbf{q}_i$, unless A healed this corruption by sending a message after $\mathbf{q}_i$ that was delivered to B before $\mathbf{q}_j$, and likewise with A and B swapped.

An attacker wins the 2SM security game if it correctly guesses the random bit b in the end and the safety predicate **2SM-safe** evaluates to true on the queries made by the attacker.

$$\begin{aligned}
&\textbf{2SM-safe}(\mathbf{q}_1, \ldots, \mathbf{q}_q) := \\
&\quad \forall\big((i,j) \text{ s.t. } \mathbf{q}_i = \textbf{challenge-C}(m_0, m_1) \text{ for some} \\
&\qquad C \in \{A, B\} \text{ and } \mathbf{q}_j = \textbf{corrupt-}\overline{\mathbf{C}} \text{ for } \overline{C} \neq C\big) \\
&\quad \Big((\mathbf{q}_i\text{'s ciphertext was not delivered by the time of } \mathbf{q}_j) \\
&\quad \implies \exists k\ \big((\mathbf{q}_k = \textbf{send-}\overline{\mathbf{C}}(m) \text{ or} \\
&\qquad \textbf{challenge-}\overline{\mathbf{C}}(m_0', m_1')) \wedge k > j \wedge (\mathbf{q}_k\text{'s ciphertext} \\
&\qquad \text{was delivered by the time of } \mathbf{q}_i)\big)\Big)
\end{aligned}$$

Figure 11: Safety predicate **2SM-safe**. Note the similarity to **safe** in Figure 9.

Definition 12. A *non-adaptive (t,q)-attacker* is an attacker $\mathcal{A}$ that runs in time t, makes at most q queries, and must specify the sequence of queries it plans to make in advance, before seeing the result of any queries. The *advantage* of $\mathcal{A}$ against a 2SM scheme 2SM is

$$\mathsf{Adv}^{\mathsf{2SM}}_{\text{2sm-na}}(\mathcal{A}) := 2\left|\Pr[\mathcal{A} \text{ wins}] - \frac{1}{2}\right|.$$

Definition 13. A 2SM scheme is *non-adaptively (t,q,ϵ)-secure* if for all (t,q)-attackers $\mathcal{A}$,

$$\mathsf{Adv}^{\mathsf{2SM}}_{\text{2sm-na}}(\mathcal{A}) \leq \epsilon.$$

C.2 Security Proof

Theorem 14. *The 2SM scheme defined in Figure 2 is non-adaptively $(t, q, 2q\epsilon_{pke})$-secure, where ϵ_{pke} is such that the PKE protocol is (t', ϵ_{pke})-secure for $t' \approx t$.*

Proof. Let $\mathbf{q}_1, \ldots, \mathbf{q}_q$ be the sequence of queries made by the adversary, which we can fix in advance because it is non-adaptive. Without loss of generality these queries satisfy **2SM-safe**. Similarly, we assume the adversary never fails a **require** clause. Then in terms of the bit b sampled at the beginning of the game,

$$\begin{aligned}
&\mathsf{Adv}^{\mathsf{2SM}}_{\text{2sm-na}} = \\
&\quad \Big|\Pr[\mathcal{A} \text{ outputs } 1 \mid b = 1] - \Pr[\mathcal{A} \text{ outputs } 1 \mid b = 0]\Big|.
\end{aligned}$$

We now proceed with a hybrid argument. Let H_0 denote the original security game. Let H_1 be the same game but without the **win** condition in **deliver-A** and **deliver-B**. It is not difficult to check that our 2SM protocol is correct, i.e., the parties always correctly decrypt messages assuming authentic in-order delivery to each party, so these games are equivalent.

Next, let S_A be the set of all queries $\mathbf{q}_i$ calling **send-A** or **challenge-A** that are *uncompromised*, which we define by: for all queries $\mathbf{q}_j = \textbf{corrupt-B}$, the implication in **2SM-safe** holds for $\mathbf{q}_i$ and $\mathbf{q}_j$. Let S_B be analogous and $S = S_A \cup S_B$. By safety, S includes all **challenge-A** and **challenge-B** queries.

Define hybrids $H_1 = H_2^0, H_2^1, \ldots, H_2^{|S|}$ by: H_2^i is the same as H_1, except for the first i queries in S, the outputs of **2SM-Send** in those queries use $\mathsf{PKE\text{-}Enc}(\mathsf{otherPk}, r)$ in place of

$$\mathsf{PKE\text{-}Enc}(\mathsf{otherPk}, (m, \mathsf{otherNewSk}, \mathsf{nextIndex}, \mathsf{myNewPk})),$$

where r is a random value of the same length as $(m, \mathsf{otherNewSk}, \mathsf{nextIndex}, \mathsf{myNewPk})$. (In the corresponding **receive** queries, we use the correct value $(m, \mathsf{otherNewSk}, \mathsf{nextIndex}, \mathsf{myNewPk})$ as the decrypted ciphertext.)

We claim that given an adversary distinguishing between H_2^{i-1} and H_2^i (more formally, distinguishing between the distributions resulting from H_2^{i-1} with a fixed value of b and H_2^i with the same value of b), we can construct an adversary of comparable efficiency winning the IND-CPA game for PKE. Let $\mathbf{q}_{\text{repl}}$ be the query replaced in H_2^i, and assume it is a **send-A** or **challenge-A** query. Let otherPk be the public key used for encryption in $\mathbf{q}_{\text{repl}}$. There are two cases to consider: either otherPk was received from B, possibly as the public key in 2SM-Init (if the last A query, excluding **corrupt-A**, was **receive-A** or **init**), or it comes from a key pair generated by A (if the last A query, excluding **corrupt-A**, was **send-A** or **challenge-A**).

In the first case, because $\mathbf{q}_{\text{repl}}$ is uncompromised and B deletes the secret key corresponding to otherPk as soon as it is used, no **corrupt-B** query reveals this secret key to the adversary. Hence we can reduce the PKE IND-CPA game to the H_2^{i-1}/H_2^i distinguishing game by simulating H_2^{i-1} (with the PKE game's public key substituted for otherPk at the time its key pair is generated, either in the 2SM game's **init** oracle or in algorithm 2SM-Send) until we get to the 2SM-Send call in $\mathbf{q}_{\text{repl}}$. In that algorithm, we use the PKE game's encryption oracle in place of

$$\mathsf{PKE\text{-}Enc}(\mathsf{otherPk}, (m, \mathsf{otherNewSk}, \mathsf{nextIndex}, \mathsf{myNewPk})),$$

passing $(m, \mathsf{otherNewSk}, \mathsf{nextIndex}, \mathsf{myNewPk})$ and a random string of the same length as arguments. We then continue simulating H_2^{i-1}, exploiting the fact that we can do this without knowing the secret key corresponding to otherPk. This simulates either H_2^{i-1} or H_2^i depending on the value of the bit b in the IND-CPA game for PKE.

In the second case, because $\mathbf{q}_{\text{repl}}$ is uncompromised, we also have that $\mathbf{q}_{\text{prev}}$ is uncompromised, where $\mathbf{q}_{\text{prev}}$ is the last **send-A** or **challenge-A** query issued before $\mathbf{q}_{\text{repl}}$. Since $\mathbf{q}_{\text{prev}}$ was replaced in a previous hybrid step, in both H_2^{i-1} and H_2^i, the encrypted portion of $\mathbf{q}_{\text{prev}}$'s message encrypts a random plaintext. In particular, it is independent of the value of otherNewSk used in $\mathbf{q}_{\text{prev}}$, which is the secret key corresponding to the otherPk used in $\mathbf{q}_{\text{repl}}$. Thus we can simulate $\mathbf{q}_{\text{prev}}$ without knowing otherNewSk. This lets us again reduce the PKE IND-CPA game to the H_2^{i-1}/H_2^i distinguishing game.

Finally, the adversary's advantage in $H_2^{|S|}$ is 0, since the game does not depend on the bit b. □

Remark 15. One threat not modeled by our security definition is bad or leaked randomness. Our protocol is vulnerable to this threat: if the adversary learns all random values used in a call to 2SM-Send, they partially compromise both parties. We believe this can easily be fixed by combining newly received key pairs with old ones in a homomorphic way, as in [24], at the expense of incompatibility with practical primitives like X25519 [30].

D Multiple additions and removals of the same user

A group member may be added multiple times concurrently, or added, removed, and then added back at a later time. The pseudocode in Figure 4 does not handle this case explicitly, but we can account for concurrent additions, by replacing the process-welcome function with the version shown in Figure 12.

This version takes the union of the added member's history with adderHistory, so it contains all messages causally prior to a message they have received. This may cause the added member to skip over other add messages without getting to send an add-ack; thus the added member forwards their current ratchet state to any users they do not recognize in the welcome-ack message. They also forward their per-member secret as needed, like in process-seed. Observe that add-remove-add sequences are handled naturally, with other group members forwarding their current ratchet states in response to the second addition, thus skipping over any ratchet states that the recipient missed while they were removed.

```
process-welcome(sender, seq, (adderHistory, c))

oldHistory ← $\gamma$.history
$\gamma$.history ← $\gamma$.history ∪ adderHistory
$\gamma$.ratchet[sender] ← decrypt-from(sender, $c$)
$\gamma$.memberSecret[sender, seq, $\gamma$.myId] ←
        update-ratchet(sender, "welcome")
$I_{\mathsf{sender}}$ ← update-ratchet(sender, "add")
ratchetFwd ← ∅;  perMemberForward ← ∅
if oldHistory ≠ ∅ then
    allMembers ← member-view($\gamma$.myId)
    ratchetFwd ← {(ID, ("ratchet",
        encrypt-to(ID, $\gamma$.ratchet[$\gamma$.myId]))) |
        ID ∈ allMembers \ SR-DGM(oldHistory)}
    memberFwd ← {(ID, ("member", encrypt-to(ID,
        $\gamma$.memberSecret[sender, seq, $\gamma$.myId)) |
        ID ∈ allMembers \ member-view(sender)}
control ← ("wel-ack", ++$\gamma$.mySeq, (sender, seq))
(_, _, $I_{\mathsf{me}}$, _) ← process-wel-ack($\gamma$.myId,
        $\gamma$.mySeq, (sender, seq), $\varepsilon$)
return (control, ratchetFwd ∪ memberFwd, $I_{\mathsf{sender}}$, $I_{\mathsf{me}}$)
```

```
process-wel-ack(sender, seq, (ackID, ackSeq), dmsg)

if dmsg ≠ $\varepsilon$ then
    (type, $s$) ← decrypt-from(sender, dmsg)
    if type = "ratchet" then
        $\gamma$.ratchet[sender] ← $s$
    else $\gamma$.memberSecret[ackID, ackSeq, sender] ← $s$
return process-ack(sender, seq, (ackID, ackSeq), $\varepsilon$)
```

Figure 12: Handling the case when the same user is added multiple times.

E DCGKA Security Proof

We now prove the security of our DCGKA protocol (Theorem 8).

As described in Section 7.5, we achieve slightly suboptimal PCS in the face of concurrent updates. We formalize the weakened security property that we do achieve by replacing the predicate **safe** in the DCGKA security game with the predicate **dom-safe** defined in Figure 13. It differs from **safe** in that it additionally checks that for each challenge $\mathbf{q}_i$, there is a fixed update, remove, or create query $\mathbf{q}_d$ whose message causally dominates all of the update messages $\mathsf{message}(\mathbf{q}_k)$ and removals used to heal $\mathbf{q}_i$ from corruptions.

For removals, more specifically, instead of allowing corruptions of any user who is not an intended recipient of the challenged message, we only allow corruptions of a user who is not an intended recipient of the dominating message $\mathsf{message}(\mathbf{q}_d)$ and who was not added back to the group between $\mathbf{q}_d$ and the challenged message. A special case of this is that if A and B are removed concurrently and then both corrupted, we do not claim security for subsequent messages until after an update or remove $\mathbf{q}_d$ which dominates both removals. This expresses the fact that A and B can collude to decrypt the intermediate messages. However, we do claim security if only A (resp., only B) is corrupted, as can seen by taking $\mathbf{q}_d$ to be A's removal query, so A cannot decrypt the intermediate messages without collusion.

Given this safety predicate, the basic idea of our security proof is that all 2SM instances used by $\mathsf{message}(\mathbf{q}_d)$ are uncompromised, hence $\mathbf{q}_d$'s seed secret is unknown to the adversary. The same then holds for the challenged update secret, since that secret factors in $\mathbf{q}_d$'s seed secret via the key ratchet. Later corruptions by the adversary are foiled by forward secrecy, which we guarantee by using the key ratchet and deleting secrets after use.

Lemma 16 (Correctness)**.** *Our DCGKA protocol is correct. I.e., for any adversary, none of the* **win** *clauses in the security game will ever be triggered. This holds even if we relax the requirement of causally ordered delivery to the weaker ordering requirement described in Section 8, by replacing the predicate*

$$
\begin{aligned}
&\textbf{dom-safe}(\mathbf{q}_1, \ldots, \mathbf{q}_q) := \\
&\quad \forall\big(i \text{ s.t. } \mathbf{q}_i = \textbf{challenge}(\mathsf{ID}, c) \text{ for some } \mathsf{ID}, c\big) \\
&\quad \exists d \Bigg(\big(\mathbf{q}_d = \textbf{update}(-), \textbf{remove}(-), \text{or } \textbf{create}(-)\big) \wedge \\
&\qquad \big(\mathsf{message}(\mathbf{q}_d) \preceq \mathsf{controlMsgs}[\mathsf{ID}, c]\big) \wedge \\
&\qquad \forall\big(j \text{ s.t. } \mathbf{q}_j = \textbf{corrupt}(\mathsf{ID}') \text{ for some } \mathsf{ID}'\big) \\
&\qquad \Big(\mathsf{corrupts}(\mathbf{q}_i, \mathbf{q}_j) \implies \\
&\qquad\quad \big(\neg\mathsf{should\text{-}decrypt}[\mathsf{message}(\mathbf{q}_d), \mathsf{ID}'] \wedge \\
&\qquad\qquad \nexists a\ (\mathbf{q}_a = \textbf{add}(-, \mathsf{ID}') \wedge \\
&\qquad\qquad\quad \mathsf{message}(\mathbf{q}_a) \not\preceq \mathsf{message}(\mathbf{q}_d) \wedge \\
&\qquad\qquad\quad \mathsf{controlMsgs}[\mathsf{ID}, c] \not\preceq \mathsf{message}(\mathbf{q}_a))\big) \\
&\qquad \bigvee\ \exists k\ \big(\mathbf{q}_k = \textbf{update}(\mathsf{ID}') \wedge k > j\ \wedge \\
&\qquad\qquad \mathsf{message}(\mathbf{q}_k) \preceq \mathsf{message}(\mathbf{q}_d))\big)\Big)\Bigg)
\end{aligned}
$$

Figure 13: Safety predicate **dom-safe** for our protocol. The predicate corrupts is defined in Figure 9, while message($\mathbf{q}$) denotes the control message generated during $\mathbf{q}$.

causally-ready(ID, c, ID$'$) *in oracle* **deliver** *with the predicate*

$$
\begin{aligned}
&\mathsf{ack\text{-}order\text{-}ready}(\mathsf{ID}, c, \mathsf{ID}') := \\
&\quad \big((\forall c' \leq c)\ (\mathsf{in\text{-}history}(\mathsf{ID}, c', \mathsf{ID}'))\big) \wedge \\
&\quad \big((\mathsf{controlMsgs}(\mathsf{ID}, c) \text{ is an ack, add-ack, or wel-ack of} \\
&\qquad T \in \mathsf{controlMsgs}) \implies \mathsf{in\text{-}history}(T, \mathsf{ID}')\big)
\end{aligned}
$$

Proof sketch. All 2SM protocol messages decrypt correctly because for each pair of users ID, ID$'$, all 2SM messages from ID to ID$'$ are delivered in order, and because we use the PKI correctly.

Next, all calls to member-view are correct, in the sense that for each pair of users ID, ID$'$, at any time, letting T be the most recent control message sent by ID$'$ that ID has received, a call to member-view(ID$'$) by ID returns the same set as a call to member-view(ID$'$) made by ID$'$ immediately after generating T. This holds by our ordering requirements, the fact that all group members acknowledge all add and remove messages they receive (including ones concurrent to their own addition, unless subsumed by a double-add of themselves), and the fact that all group members eventually put all remove and add messages (plus acks of those messages) in their own history sets. The latter fact holds noting that group members take the union with their adder's history each time they are added, and they receive all messages concurrent to their own addition (unless subsumed by a double-add of themselves).

Next, we have the invariant: for each pair of users ID, ID$'$ with states γ, γ', respectively, at any time, letting T be the most recent control message sent by ID$'$ that ID has received and such that ID was an intended recipient of T (i.e., should-decrypt(T, ID)), γ.ratchet[ID$'$] has the same value that γ'.ratchet[ID$'$] had immediately after generating T.

Initially, this is true as all users' ratchet states are all ε. Subsequently, we break into cases depending on T, noting that γ.ratchet[ID$'$] only changes when processing messages from ID$'$, and that such messages are delivered in order by assumption. If T is a create, update, remove, or ack message, then both ID and ID$'$ derive the same per-member secret for ID$'$, hence update the ratchet states identically. Note that for an ack message, we make use of the assumption that ID received the acknowledged message before the ack, hence ID has the per-member secret available. We also use the fact that if T is an ack and ID could not decrypt the acknowledged message, then ID' forwards their per-member secret to ID in the direct message. Add, add-ack, and welcome-ack messages are similar, noting that all recipients ratchet their states forward even if they were not intended recipients, but with some special cases concerning the added user:

- All group members in the added user's view of the group (after processing the addition) receive the added user's per-member secret: they can either compute it directly from the adder's ratchet, or are forwarded it by the added user in $\mathsf{memberFwd}$.
- All group members in the added user's view of the group know the added user's previous ratchet state when processing the welcome-ack: the first time the user is added, everyone knows this is just ε. On subsequent adds, any members who do not know the previous ratchet state, because the added user did not previously consider them group members, are forwarded it by the added user in $\mathsf{ratchetFwd}$. Note that this case and the previous case are disjoint, so $\mathsf{ratchetFwd} \cup \mathsf{memberFwd}$ contains at most one entry per user.
- The added user knows all other group member's previous ratchet states when processing their add/add-ack: these are forwarded to the added user in the add message (in the adder's case) or in the add-ack messages (all others).

Finally, going through the proof of the previous invariant, it is easy to see that at every point, ID also outputs the correct update secret when processing T. Furthermore, users always output non-ε update secrets and response messages when required by the security game. $\square$

Proof of Theorem 8. Fix a non-adaptive (t, c, n) attacker $\mathcal{A}$. Let $\mathbf{q}_1, \ldots, \mathbf{q}_q$ be the sequence of queries made by $\mathcal{A}$. Without loss of generality these queries satisfy **dom-safe**. Similarly, we assume the adversary never fails a **require** clause. Then in terms of the bit b sampled at the beginning of the game,

$$\mathsf{Adv}_{\text{dcgka-na}}^{\mathsf{DCGKA},\textbf{dom-safe},\mathsf{SR\text{-}DGM}} = \left| \Pr[\mathcal{A} \text{ outputs } 1 \mid b = 1] - \Pr[\mathcal{A} \text{ outputs } 1 \mid b = 0] \right|.$$

We now proceed with a hybrid argument. Let H^0 denote the original game. At the top level, we pass through c hybrid games $H^1, \ldots, H^c$, where in the i-th hybrid, the first i **challenge** queries return a random value regardless of b (among other changes). Then H^c does not depend on b, so the adversary's advantage is 0. Thus it remains to bound the adversary's ability to distinguish between the games $H^0, \ldots, H^c$ (i.e., between the distributions corresponding to these games with a fixed value of b—since there are two distributions for each game, our bound gets an extra factor of two).

We explain the step from H^0 to H^1; the remaining steps are analogous, giving ϵ a factor of c.

Let $\mathbf{q}_{\text{chall}}$ be a challenge query. By assumption, the conclusion of **dom-safe** holds for $\mathbf{q}_{\text{chall}}$. Let $\mathbf{q}_{\text{dom}}$ be the update, remove, or create query guaranteed by **dom-safe**. Define a hybrid $H^{0.1}$ which is the same as H^0, except that we replace the following 2SM messages with encryptions of random values of the same length as the original inputs:

- all direct messages sent along with $\mathsf{message}(\mathbf{q}_{\text{dom}})$ (where $\mathsf{message}(\mathbf{q}_{\text{dom}})$ denotes the control message generated by $\mathbf{q}_{\text{dom}}$)
- for each add message not causally less than $\mathsf{message}(\mathbf{q}_{\text{dom}})$ and not causally greater than $\mathsf{message}(\mathbf{q}_{\text{chall}})$, all direct messages sent along with the add message or its corresponding add-ack, or welcome-ack message.

By the security of our 2SM protocol, the adversary's advantage in distinguishing $H^{0.1}$ from H^0 is at most $n^2\epsilon_{2sm} \leq 2qn^2\epsilon_{pke}$, noting that n^2 upper bounds the number of 2SM channels and q upper bounds the number of queries to each 2SM channel. Indeed, if $\mathcal{A}$ could distinguish between one of these message replacements, then we could make an adversary for the corresponding pair of group members' 2SM channels, which makes the same corruptions as $\mathcal{A}$ restricted to that group member, and which successfully challenges the replaced message, by simulating the DCGKA game to $\mathcal{A}$. The resulting 2SM adversary satisfies the 2SM safety predicate by **dom-safe**, which ensures that all replaced messages (which are the challenged messages in the 2SM game) are uncorrupted. Note that in this reduction, we exploit our assumption that we model each PKI key pair $(\mathsf{PKI\text{-}PublicKey}(\mathsf{ID}, \mathsf{ID}'), \mathsf{PKI\text{-}SecretKey}(\mathsf{ID}, \mathsf{ID}'))$ as a fresh output of $\mathsf{PKE\text{-}Gen}$. This allows us to substitute the public keys returned by the 2SM game, which also come from fresh outputs of $\mathsf{PKE\text{-}Gen}$, in place of $\mathsf{PKI\text{-}PublicKey}(\mathsf{ID}, \mathsf{ID}')$ and $\mathsf{PKI\text{-}PublicKey}(\mathsf{ID}', \mathsf{ID})$ when we give those values to $\mathcal{A}$ in the simulated DCGKA game, where $\mathsf{ID}, \mathsf{ID}'$ are the users in the replaced 2SM channel. Also, because the PKI secret keys are independent for each pair $\mathsf{ID}, \mathsf{ID}'$, we do not need need to know $\mathsf{PKI\text{-}SecretKey}(\mathsf{ID}, \mathsf{ID}')$ or $\mathsf{PKI\text{-}SecretKey}(\mathsf{ID}', \mathsf{ID})$ to simulate the rest of the DCGKA game—a fact which would not hold if we only had a single PKI key pair for each user.

In $H^{0.1}$, all values available to the adversary can be written as functions of:

(a) secret values remaining in group members' states immediately after they processed $\mathsf{message}(\mathbf{q}_{\text{chall}})$ (the most that the adversary can learn through later corruptions)

(b) update secrets corresponding to messages causally greater than or equal to $\mathsf{message}(\mathbf{q}_{\text{dom}})$ but not causally greater than or equal to $\mathsf{message}(\mathbf{q}_{\text{chall}})$ (which the adversary can learn through **reveal** or **challenge** queries)

(c) a single value (encompassing all remaining secrets in the protocol) which is independent of the seed secret of $\mathbf{q}_{\text{dom}}$.

To see this, we consider all values available to the adversary. Note that when discussing various "independent" values, we mean that they are independent when all considered together in the sense of (c).

- Protocol messages: All protocol messages not causally greater than or equal to $\mathsf{message}(\mathbf{q}_{\text{dom}})$ are independent of the seed secret of $\mathsf{message}(\mathbf{q}_{\text{dom}})$, since it was generated at random during $\mathbf{q}_{\text{dom}}$. The replacements for the direct messages of $\mathsf{message}(\mathbf{q}_{\text{dom}})$ in $H^{0.1}$ are likewise independent, as are the replacements for the direct messages of the add, add-ack, and welcome-ack messages specified in the definition of $H^{0.1}$. All other messages are functions of independent random values (i.e., other seed secrets, for create, update, remove, and ack messages) and (a) (for add, add-ack, and welcome-ack messages causally greater than $\mathsf{message}(\mathbf{q}_{\text{chall}})$). Note that by the delivery ordering requirements, it is not possible to have an add message causally less than $\mathsf{message}(\mathbf{q}_{\text{dom}}$ whose add-ack or welcome-ack is causally greater than $\mathsf{message}(\mathbf{q}_{\text{dom}}$, so the above cases (not causally greater than or equal to $\mathsf{message}(\mathbf{q}_{\text{dom}})$; causally greater than $\mathsf{message}(\mathbf{q}_{\text{chall}})$; or replaced in $H^{0.1}$) do indeed include all add, add-ack, and welcome-ack messages.

- Return values of **corrupt** queries: Corruptions of group members with $\mathsf{message}(\mathbf{q}_{\text{chall}})$ in their history (at the time of the corruption) return functions of (a) and independent random values (e.g., other seed secrets). Corruptions of group members with $\mathsf{message}(\mathbf{q}_{\text{dom}})$ not in their history are independent since the seed secret was generated at random during $\mathbf{q}_{\text{dom}}$, and a group member's state can only be a function of $\mathsf{message}(\mathbf{q}_{\text{dom}})$ when it is in their history. Corruptions of group members who are not intended recipients of $\mathbf{q}_{\text{dom}}$ are independent since their states do not depend on the seed secret. No other corruptions are allowed by **dom-safe**.

- Return values of **reveal** queries and other **challenge** queries: The $\mathsf{challenged}$ variable in the security game ensures that $\mathsf{message}(\mathbf{q}_{\text{chall}})$ is never revealed. Queries for messages causally greater than or equal to $\mathsf{message}(\mathbf{q}_{\text{chall}})$ are functions of (a) and independent values, while queries causally less than $\mathsf{message}(\mathbf{q}_{\text{dom}})$ are easily independent. The remaining queries are included in (b).

The values included in (a) are functions of:

(a1) per-member secrets of $\mathbf{q}_{\text{dom}}$ for group members besides the sender of $\mathsf{message}(\mathbf{q}_{\text{chall}})$

(a2) per-member secrets of arbitrary other messages for group members besides the sender of $\mathsf{message}(\mathbf{q}_{\text{chall}})$

(a3) ratchet states resulting from $\mathbf{q}_{\text{chall}}$

(a4) group members' 2SM states after processing $\mathsf{message}(\mathbf{q}_{\text{chall}})$.

Here we use the fact that group members delete nextSeed and entries in memberSecrets from their states as soon as those values are used.

Part (a2) is obviously independent of the seed secret of $\mathbf{q}_{\text{dom}}$ in the same sense as (c), so it is not a concern. The same holds for (a4) because the 2SM protocol does not store any information related to message plaintexts. The values included in (a1), (a3), and (b), together with the challenged update secret, are all derived from the seed secret of $\mathbf{q}_{\text{dom}}$ via a tree of HKDF applications. Specifically, at the root of this tree is the seed secret seed of $\mathbf{q}_{\text{dom}}$. The root's children are the corresponding per-member secrets $\mathsf{HKDF}(\mathsf{seed}, \mathsf{ID}_1), \ldots, \mathsf{HKDF}(\mathsf{seed}, \mathsf{ID}_m)$, where $\{\mathsf{ID}_1, \ldots, \mathsf{ID}_m\}$ are the intended recipients of $\mathsf{message}(\mathbf{q}_{\text{dom}})$. Each of these per-member secrets then combines with a previous ratchet state (or ε, if $\mathbf{q}_{\text{dom}}$ is a create message) to continue a key ratchet. At points where an add occurred, the adder's key ratchet splits in two, with the second branch corresponding to the key ratchet of the added user. Note that this is the case regardless of whether or not the add message is the first received by the added user, since the added user always updates their ratchet state with their per-member secret derived from the adder's key ratchet.

Observe that after truncating this tree to the minimum subtree T containing all of the values in (a1), (a3), (b), and the challenged update secret, these values are all at separate leaves of T, i.e., no value is a descendant of another.

Let $H^{0.2}$ be the game which is the same as $H^{0.1}$, except we replace all values appearing in T with random and independent values. Since seed is already random and independent of all other information available to the adversary (i.e., (a2), (a4), and (c)), we model HKDF as a random oracle, and the only values in T available to the adversary are at separate leaves, there are only two ways the adversary can distinguish these two games: either due to an input collision in $H^{0.1}$ (i.e., distinct nodes generated from the same HKDF inputs), or by inverting one of the HKDF applications. The latter occurs with probability at most $qnt2^{-\lambda}$, since there are at most qn nodes in the tree (n per query). The former occurs with probability at most $\binom{qn}{2}2^{-\lambda}$, since each pair of HKDF inputs either includes distinct constant values (e.g., IDs) or include a value of length λ that is random and independent of other HKDF inputs (by induction from the root of the tree upwards). Hence the adversary's advantage in distinguishing $H^{0.2}$ from $H^{0.1}$ is at most $qnt2^{-\lambda} + \binom{qn}{2}2^{-\lambda}$.

Finally, in $H^{0.2}$, the challenged update secret has been replaced with a random value independent of all information known by the adversary. Hence we may take $H^1 = H^{0.2}$. $\square$