BeeKEM: Decentralized, Secure and Efficient Group Key Agreement

Derek Yen*,[1], Andrés Fábrega*,[2], Liangrun Da[3], Martin Kleppmann[4],
John Mumm[3], Sunoo Park[1], Brooklyn Zelenka[3]

[1]*New York University,* {dwy218, sunoo.park}@nyu.edu
[2]*Cornell University,* andresfg@cs.cornell.edu
[3]*Ink & Switch,* me@liangrunda.com, {john.mumm, brooklyn}@inkandswitch.com
[4]*University of Cambridge,* martin.kleppmann@cst.cam.ac.uk

***Abstract*—Group key agreement protocols are essential for modern secure messaging. Most existing group key agreement protocols assume a centralized model with a semi-trusted service that mediates the communication. This is efficient, but problematic for some important applications, since a central service can be a choke point for surveillance and censorship. There is a nascent literature on *decentralized* group key agreement that avoids such reliance, but existing proposals either do not scale, with update costs linear or quadratic in the group size, or lack proofs of security. Centralized protocols can offer much lower (logarithmic) cost.**

We present BeeKEM, the first decentralized group key agreement protocol with logarithmic update cost in the common case (degrading to linear in the worst case) and proofs of security. We provide an open-source implementation and demonstrate that it is competitive with OpenMLS. BeeKEM opens the door for a range of communication and collaboration applications offering not only end-to-end encryption, but also metadata privacy and censorship resistance.

1. Introduction

Continuous group key agreement (CGKA) is the cryptographic backbone of modern secure group messaging protocols, which are used by billions of people for their private communications. For example, Messaging Layer Security (MLS) [10] is an IETF standard adopted by RCS on iOS and Android, Cisco Webex, Wire, and others [14], [22], [36]. A CGKA protocol allows the members of a group to establish shared keys for message encryption, to add or remove group members, and to periodically update their keys to achieve forward secrecy and post-compromise security [7].

Most popular messaging protocols, like MLS, rely on a centralized delivery service through which all communication is routed. The central service maintains consistent state between users by imposing a total order on past operations, such as member addition, removal, or key update. This is possible due to the service's global view of network communications. Conversely, an adversary who compromises the service can observe all communication patterns, selectively delay or drop messages, and deny service. Although such an adversary may not be able to read message contents if end-to-end encrypted (as in MLS), the service is nonetheless a potent choke point for censorship and metadata surveillance.

The potential for censorship and surveillance is lower in peer-to-peer communication, but CGKA is designed for the centralized setting. An emerging literature on *decentralized* CGKA (DCGKA) removes this dependence on a central authority, allowing users' devices to communicate via any channel, including peer-to-peer or via anonymity networks. Weidner et al. [34] recently introduced security definitions and a state-of-the-art protocol construction for DCGKA. Unfortunately, their approach is not scalable: key updates incur a communication cost of $O(n)$ where n is the number of group members, while for MLS the cost is $O(\log n)$. Prior work has shown that group chats with thousands of members are used in practice, in contexts such as political protests where censorship resistance can be essential [4], yet $O(n)$ overhead would be prohibitive in such contexts.

We introduce BeeKEM, the first DCGKA scheme with proofs of security that matches MLS's $O(\log n)$ update cost in the common case. In the case of concurrent key updates, BeeKEM's performance degrades gracefully, in the worst case reaching $O(n)$, which is on par with Weidner et al.'s protocol [34]. We give a formal specification of BeeKEM and prove it secure under the existing DCGKA security definitions [34]. We also provide an open-source, production-ready implementation whose performance we evaluate against OpenMLS (a popular MLS implementation) [30] and the original DCGKA implementation [34].

In more detail, our contributions are as follows.

- CONSTRUCTION: We introduce **BeeKEM, a new DCGKA scheme** where the communication and computation cost of key updates is $O(\log n)$ when updates are sequential, degrading gradually to $O(n)$ as the degree of concurrency increases (§4).
- MODELING: We define a **new security property** for DCGKA (§3) called *cross-fork security*, which defends against a type of attack unique to the decentralized setting. We also introduce **new extensions**

*. These authors contributed equally to this work.

of existing DCGKA security definitions (§3).

- PROVABLE SECURITY: We give **rigorous security proofs** for BeeKEM (§5 and Appendix B).
- IMPLEMENTATION: We provide an **open-source, production-ready implementation** of BeeKEM[1] that can support a wide range of decentralized end-to-end encrypted collaboration applications.
- EVALUATION: We evaluate our performance with respect to Weidner et al. [34] and OpenMLS [30] (§6). **BeeKEM performs competitively with OpenMLS** when connectivity is good. It also handles partitions that would take MLS offline, at a cost in efficiency.
- EXTENSIONS: We sketch constructions called $\text{BeeKEM}^{\text{PQ}}$ and $\text{BeeKEM}^{\text{FS}}$ which respectively achieve post-quantum security and stronger forward secrecy (§7).

Broader Impacts. BeeKEM can be used in any application that requires end-to-end encryption (E2EE) for efficient group communication. It is suitable for messaging, but it can also support other group collaboration applications such as collaborative calendars, text editors, spreadsheets, and more.

A core impact of building BeeKEM is to bring efficient E2EE to a wide range of decentralized *local-first* [26] collaboration software. Applications such as an E2EE collaborative document editor Patchwork[2] have already been built upon our work which would not be viable with $O(n)$ communication overhead in a group or organization with, say, thousands of members.

2. Why Decentralize?

MLS [12] and other CGKA-based protocols assume a centralized delivery service, as noted in Section 1. The service may consist of one or more servers. This assumption is suitable when all members of the communicating group are able to reliably connect to a central service. If some users are offline, CGKA protocols allow the users who are online to continue communicating. The design of TreeKEM, the CGKA underlying MLS [13], heavily relies on the assumption of this ordering enforced by the central service.

The Problem. In centralized protocols, users may be online, but unable to communicate with the delivery service, and thus cut off from essential communication. For example, what if a user can communicate with a subset of other users, perhaps via a local network, but not with the service?

This type of scenario is of growing practical importance.

In early 2026, Iran enacted a months-long nationwide Internet shutdown that allowed people inside the country to access domestic Internet services, but limited access to the international Internet to a few people with special privileges [32]. Since 2026, Russia has imposed intermittent Internet restrictions with the aim of pushing domestic users towards a national messaging app that supports state surveillance [33]. Some users circumvent such measures by use of Starlink and similar satellite services, at great risk—being caught with a Starlink terminal in Iran can mean death [32].

Protesters in Hong Kong have attempted to use mesh networking apps to organize in spite of Internet shutdowns in 2014 [9] and 2019 [4], but these apps were found to have weak security [3]. Nevertheless, reliance on local networking technology, if implemented well, has the potential to enable users to circumvent government censorship, and secure communication based on mesh networking is an important subject of ongoing research [23].

A *network partition* occurs when a set of users is split into two or more subsets such that communication within each subset is possible, but communication across subsets is not. In the example of Iran, the Internet shutdown creates a network partition where one subset of users is inside the country and the other subset is outside. With a centralized protocol like MLS, the delivery service will sit on one side of the partition or the other (or be rendered ineffective by being split across the partition). Consider the impact of a partition on a chat group including both international journalists and sources within Iran: one of those two subsets loses the ability to add/remove members, update keys, or to communicate, while the Internet shutdown is in effect.

Beyond these specific examples, watchdog organisations and the United Nations have expressed concern over the recent global rise in Internet shutdowns, reporting a record 313 shutdowns across 52 countries just in 2025 (e.g., [1], [29]). Moreover, even absent shutdowns, network unreliability remains a significant problem at a global scale [19].

The Solution. A decentralized protocol like BeeKEM overcomes these limitations. During a network partition, each subset of users can continue using the features of the app as normal. Only users within the same subset will be able to see each others' messages while the partition lasts, but once connectivity is restored, they are able to sync and catch up on any activity that occurred in the other subsets.

The decentralized approach does not assume any particular network topology, so any communication channel can be used. For example, in the case of a national Internet shutdown, one option would be to have two delivery services, one inside the country and one outside, enabling communication among domestic and foreign users, respectively. Communication between the two could occasionally be bridged, for example, by those with privileged access to both networks, or by volunteers willing to temporarily put a satellite Internet terminal on their roof for an hour at night.

Another advantage of a decentralized architecture is metadata privacy. Anonymity networks such as Tor [18] or Loopix/Nym [31] allow one-to-one communication without revealing to a network observer who is communicating, and generalizations to group communication exist [21]. These systems take great care to avoid any single node knowing who is communicating, lest it be compromised. A centralized delivery service, even if accessed over an anonymity network, can be a potent target for adversaries as it can see

1. https://github.com/inkandswitch/keyhive/tree/main/beekem
2. https://www.inkandswitch.com/project/patchwork/

all communication. A decentralized protocol can run over an anonymity network without creating such a weakness.

3. Model and Definitions for DCGKA

Section 3.1 recalls syntax introduced by Weidner et al. [34]. Section 3.2 introduces our communication model, and Section 3.3 overviews security properties for DCGKAs, including both prior definitions and our novel contributions.

3.1. DCGKA Syntax

A DCGKA scheme consists of a tuple of algorithms that provide support for the basic functioning of a messaging group, including initialization, membership changes, and rotating secrets. The DCGKA abstraction models *local* operations run by group members, which compose into a multi-party protocol. Each user is identified by some unique identifier id that belongs to a set of application-specific IDs $\mathcal{I}$ (e.g., public keys, usernames, or phone numbers), and each holds a local state, denoted by st. A DCGKA scheme Π consists of the algorithms listed below, which are run locally by a user id with state st (whom we call the *calling user*). All algorithms return a *control message*, which the calling user broadcasts upon algorithm completion.[3]

- ***Initialization:*** $st_0 \leftarrow \Pi.\mathsf{Init}(id)$ is invoked when the calling user has just joined a group; it takes as input id and returns an initial state st_0 for user id.
- ***Group Creation:*** $(st', R, \mathcal{M}, S) \leftarrow \Pi.\mathsf{Create}(st, I)$ takes as input st and a set of user IDs $id \in I \subseteq \mathcal{I}$, and returns an updated state st' for the calling user, a *control message* R to be broadcast to the other group members, a *direct message set* $\mathcal{M} = \{(id', M)\}_{id' \in I}$ (indicating message M to be sent to user id' from id), and a new *group secret*[4] $S \in \mathcal{S}$.
- ***Member Addition:*** $(st', R, \mathcal{M}, S) \leftarrow \Pi.\mathsf{Add}(st, id')$ takes as input the calling user's state st and the ID id' of the group member to be added, and returns $(st', R, \mathcal{M}, S)$ as defined under Create.
- ***Member Removal:*** $(st', R, \mathcal{M}, S) \leftarrow \Pi.\mathsf{Remove}(st, id)$ takes as input st and the ID id' of the group member to be removed, and returns $(st', R, \mathcal{M}, S)$ as defined under Create.
- ***Update:*** $(st', R, \mathcal{M}, S) \leftarrow \Pi.\mathsf{Update}(st)$ rotates the group secret; it takes as input st and returns $(st', R, \mathcal{M}, S)$ as defined under Create.
- ***Message Processing:*** $(st', R, \mathcal{M}, S_s, S_r) \leftarrow \Pi.\mathsf{Process}(st, id, R, M)$ is invoked when a new control message is received; it takes as input st, the (authenticated) ID id' of the sender of the message, the control message R, and the direct message M from id' to the calling user (or ε if there is no direct message); and it returns $st, \mathcal{M}, M$ as defined under Create, and new secrets S_s, S_r for the sender and recipient of the control message, respectively.

Without loss of generality, our exposition considers a single group. Multiple groups can be realized by multiple single-group protocols run independently in parallel.

3.2. Authenticated Causal Broadcast (ACB)

We work in the communication model of *authenticated causal broadcast* (ACB) [34]. In the decentralized setting, there is no central authority to impose a global ordering on messages; messages can arrive in different orders for different users, e.g., due to a network partition. We therefore do not assume that messages arrive in the same order for all users, only that they arrive in ***causal order***.

Definition 1 (Causal order)**.** *The* causal order *is a partial order over operations, denoted $\prec$. We have that $a \prec b$ ("a* causally precedes *b" or "b* causally depends upon *a") if:*

- *a and b were each performed by the same group member, who performed a before b; or*
- *the group member who performed b did so after having received the message corresponding to a; or*
- *(transitivity) there exists c such that $a \prec c$ and $c \prec b$.*

If neither $a \prec b$ nor $b \prec a$, then we say a and b are concurrent*, denoted $a \sim b$. We call the causal ordering between operations their* causal relationship*.*

An ACB has three properties. First, it delivers messages to users in a causal order: if $a \prec b$, then a always arrives before b. (There is no guarantee about concurrent operations, which may arrive in any order, and may be ordered differently for different group members.) Second, an ACB is *reliable*: messages eventually arrive. Third, it provides *authentication*: a message's receiver is assured of who sent it and that the message was not modified in transit.

3.3. DCGKA Security Goals & Threat Model

The high-level objective of a DCGKA scheme is to ensure the confidentiality of update secrets S. We target the standard CGKA security properties of (1) *Correctness*, (2) *Forward Secrecy*, and (3) *Post-Compromise Security*. We additionally introduce and achieve a novel security property unique to the decentralized setting, which we call (4) *Cross-Fork Security* (CFS).

Intuitively, CFS considers attackers who can view group state on both sides of a partition; such attackers can use state from users compromised on one side of the partition to attack the group secret on the other side of the partition. This type of attack was first noted informally by Weidner et al. [34], but they do not formally define it and their scheme is not secure against it. Alwen et al. [8] develop a scheme with security against this type of attack, which they call a *cross-fork attack*, albeit in a different (centralized) setting from DCGKA. BeeKEM is the first DCGKA scheme with security against this type of attack; we formally define CFS and prove that BeeKEM achieves a form of it.

3. *Direct messages*, in contrast, are "chat messages" sent between users.

4. This is also often referred to as the *update secret* in the (D)CGKA literature; we use the term *group secret* as we find it more intuitively contrasts with *personal secrets* held by individual users.

Concurrency vs. Security in BeeKEM. BeeKEM's robustness to concurrency and network unreliability across long sequences of operations requires users to retain secrets longer than is optimal for security, thus weakening Forward Secrecy. Section 7 characterizes this tradeoff, explains why it may be inherent for DCGKA schemes with sublinear update costs, and describes an alternative system design BeeKEM$^{\mathrm{FS}}$ that strengthens BeeKEM's security properties to full Forward Secrecy (and Cross-Fork Security) at the expense of robustness to concurrency.

The rest of this section informally defines our security goals, later formalized in Figure 3 in §5. Forward Secrecy and Post-Compromise Security as discussed below are generalizations of the respective standard definitions (e.g. [7]) to the decentralized setting, as presented in [34].

Terminology. A user's ***view*** comprises i) the history of operations; ii) group membership; and iii) the group secret S (possibly $S = \perp$). Users' views evolve as they process operations. A ***fork*** occurs when users diverge in their views of operation history; users with the same view during a fork belong to the same ***branch***.

1. Correctness. We define correctness for each DCGKA operation. The below definitions state requirements following an honest execution of each operation with calling user id, assuming sequential operations in isolation. For concurrent operations, behavior is specified by a ***materialization***, a function which dictates the effects of concurrent operations.

- Create: All users in the initial set of members I can access all subsequent group secrets (until removed).
- Add, Init: An added member, upon running Init, can access all subsequent group secrets (until removed).
- Remove: A removed user can no longer access group secrets subsequent to their removal.
- Update: The group secret is well defined ($S \neq \perp$) and accessible to all group members in id's view.
- Process: After id processes the message of an operation op by id', id has a view of the group in which group membership and the group secret S are identical to id' immediately after performing op.

We define an additional correctness property which we call ***Correctness Under Concurrency (CUC)***, which states that if the group is partitioned and updates are performed on separate branches, after the partition heals, each user id can access every group secret S resulting from an Update performed by a user whose view includes id as a member. CUC can be in tension with the security properties below.

BeeKEM is parameterized by a ***key retention parameter*** κ, where users retain their κ most recent personal secrets, in order to access group secrets from other branches. CUC is only possible when users retain all personal secrets ($\kappa = \infty$). We define a weaker notion ***κ-CUC***, which states that a user id may recover updates from branches that fork at a point causally subsequent to the κ-th most recent Update performed by id. Then, ∞-CUC $\equiv$ CUC.

2. Forward Secrecy (FS). *Forward Secrecy* in the decentralized setting requires that: a given group secret S cannot be accessed by an attacker who has compromised a set of users where each user has processed an update causally subsequent to the update that established S.

BeeKEM achieves a parametrized form of FS adapted for decentralized settings with high concurrency. It does not achieve full FS for two reasons, both of which relate to the fact that to support CUC, users retain personal secrets for longer than is ideal for security. The first reason also arises in centralized TreeKEM [7], and relates to users who update infrequently. The second reason seems inherent to the decentralized setting, and can be partially addressed by tuning a tradeoff between CUC and FS with a choice of parameter. Our alternative system design BeeKEM$^{\mathrm{FS}}$ (§7) fully addresses both issues at the cost of CUC.

The First Reason. Correctness requires a group member who has not performed an update in a long time[5] to retain access to all group secrets during that time. If such a user is compromised, all those secrets may be exposed. The weaker notion of FS achieved by TreeKEM is called ***Forward Secrecy with Updates (FSU)*** [7], which weakens FS by stipulating that, if a user id is compromised, the adversary cannot access secrets prior to the most recent update *by user* id. By contrast, full FS protects secrets prior to the most recent update by *any* user processed by id.[6]

We adapt FSU to the decentralized setting: a group secret S established by Update op cannot be accessed by an attacker who has compromised a set of users where each user has performed an update causally subsequent to op.

The Second Reason. Consider a group where users perform Update operations during a two-way partition. Users in one half of the partition will produce group secrets which are accessible to Alice. In the other half, updates will be made to Alice's last known secret $\mathsf{sk}_{\mathrm{old}}$ from before the fork. When the partition heals, to catch up on the other partition's messages, Alice needs those group secrets, which she can access only if she retained $\mathsf{sk}_{\mathrm{old}}$. If Alice did an Update during the partition and deleted her old secret $\mathsf{sk}_{\mathrm{old}}$ for the sake of FS, she would lose her ability to "catch up."

Alwen et al. propose a change to TreeKEM in [7] which strengthens its FSU to FS. Section 7 discusses an analogous decentralized construction, BeeKEM$^{\mathrm{FS}}$, which has the drawback of restricting users' ability to access group secrets on forks. (Forks are not a concern in TreeKEM's centralized context.) We conjecture the tradeoff between CUC and FS may be inherent in decentralized settings.

3. Post-Compromise Security (PCS). PCS requires: if an attacker compromises the state of one or more users, if each compromised user performs an Update, the attacker cannot access any group secret causally after all such Updates.

5. By "long time," we mean "over many Update operations."

6. Note that FSU does not mention "processing" updates because a user id instantaneously processes their own update.

4. Cross-Fork Security (CFS). CFS is a novel security property we introduce which is applicable in decentralized settings. It formalizes security against *cross-fork attacks*, where state from a compromised user is used to attack the group secret defined by a concurrent update [8]. We first state CFS for a single compromised user to build intuition before giving the full definition. CFS states that if the group forks into branches $b_1, \ldots, b_m$ and the attacker compromises a user id who has performed an update op on branch b', the attacker cannot access the group secrets of Update operations on any branch b^* which diverged from b' prior to op. For example, if Alice and Bob concurrently perform updates then an attacker obtains Alice's state, CFS prevents the attacker from obtaining Bob's new group secret (or any subsequent group secret established on Bob's branch). CFS only applies while a fork is active; compromises after a fork heals are covered by FS.

For multiple compromised users, CFS states that if the group forks into branches $b_1, \ldots, b_m$ and the attacker compromises one or more users id (across branches B), each with a most recent update operation op_{id}, the attacker cannot access any group secrets defined on a branch b^* which forked from B causally prior to each respective op_{id}.

5. κ-FSU and κ-CFS. As discussed earlier, key retention is necessary for CUC. However, such key retention is incompatible with FS and CFS. We define weaker security notions for FS and CFS which we call ***κ-FSU*** and ***κ-CFS***. Both are parameterized by κ and are equivalent to FSU and CFS when $\kappa = 1$; we define them with respect to a single compromise for brevity, but the multi-compromise definition extends the definitions of FSU and CFS in the natural way.

κ-FSU requires that: if an attacker compromises the state of a user id, they cannot access update secrets from update operations prior to the κ-th most recent update by id. Likewise, κ-CFS requires: when the group state forks, if an attacker compromises the state of a user id, the attacker cannot access any group secrets defined on forks which occurred before the κ-th most recent update by id.

When used with key retention parameter κ, BeeKEM achieves κ-CUC, κ-FSU, and κ-CFS. Higher values of κ degrade the FS and CFS properties but offer greater access to group secrets defined under forking (stronger CUC). Our variant construction BeeKEM$^{\text{FS}}$ (§7) achieves full FS and CFS, but has no support for accessing group secrets defined under forking (i.e., $\kappa = 1$). To summarize symbolically:

$$\textbf{FSU:}\quad \kappa\text{-FSU} \Leftarrow \cdots \Leftarrow 1\text{-FSU} \quad \Leftrightarrow \quad \text{FSU} \Leftarrow \text{FS}$$
$$\textbf{CFS:}\quad \kappa\text{-CFS} \Leftarrow \cdots \Leftarrow 1\text{-CFS} \quad \Leftrightarrow \quad \text{CFS}$$

Example. Figure 1 illustrates the three security properties.

Threat Model. We follow the standard threat model from prior DCGKA literature [34], [35]. We assume users execute the protocol honestly. We consider an adversary who may compromise users of the group to obtain a snapshot of their local state, and who may choose when the ACB delivers messages (without interfering with ACB guarantees; see §3.2). Control over the ACB implies the adversary can cause arbitrary network partitions.

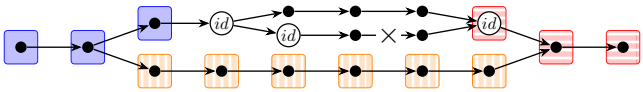

Figure 1. An example operation graph for BeeKEM with $k = 2$, showing which group secrets are protected under which security properties when a user id has their state compromised at $\times$. Nodes denote Update operations; nodes labeled id represent updates performed by id; edges denote the causal order (written left to right to follow time). Blue (solid) boxes are protected under 2-FSU; red (horizontal stripe) boxes under PCS; and orange (vertical stripe) boxes under 2-CFS. The group secrets of unboxed nodes are accessible to the attacker.

4. Our Construction: BeeKEM

Section 4.1 provides an initial overview of the BeeKEM protocol and its basic operations. Section 4.2 discusses cryptographic primitives and requirements of BeeKEM. Section 4.3 describes the BeeKEM construction in detail.

4.1. Design Overview

The BeeKEM Tree. BeeKEM belongs to the Asynchronous Ratcheting Trees (ART) family of protocols, which also include the TreeKEM protocol used in MLS [7], [15]. As such, our scheme is underpinned by one primary data structure: a binary tree that encodes group membership and the current group secret, which we call the *BeeKEM tree*.

The BeeKEM tree B is a perfect binary tree that stores the group secret S encrypted at the root. Each user keeps a local copy of the tree, which they update as they perform and receive operations. Each leaf is either empty or corresponds to a group member, containing their unique user identifier id and public key pk_{id}. The associated secret key sk_{id} is also stored in the user's local state. User IDs are fixed throughout the protocol, but user keys can evolve. The core property of the BeeKEM tree is given below.

Invariant 1. *Access to the private key associated with a node v (leaf or inner node) is sufficient to decrypt the encrypted private key of any node u on the direct path of v.*

In particular, Invariant 1 implies that possession of any personal secret sk_{id} suffices to access the group secret S at the root. Thus, any group member can (locally, without interaction) recover S by starting at their leaf and moving up the tree, iteratively decrypting inner-node secrets. We refer to secret keys of inner nodes as ***subgroup secrets***.

Next, we describe how BeeKEM implements the DCGKA operations defined in Section 3.

Update & Process. For post-compromise security, group members must periodically rotate their keys using the Update operation (§3). An Update operation in BeeKEM consists of *rotating all secrets in the path from a leaf to the root*. When a user id performs an update, they first generate $d = \lceil \log_2 n \rceil$ new public-private key pairs, one

per tree level; note that this leads to a new $(\mathsf{pk}_{id}, \mathsf{sk}_{id})$ and a new S. They then overwrite each node in the path from their leaf to the root by storing the ℓ-th new key pair at the node on the ℓ-th level[7]; as before, with public keys in the clear, and private keys encrypted. Crucially, each secret key is encrypted in a way that preserves Invariant 1. As such, all group members retain access to S after any update. The updating user then broadcasts all new public keys and ciphertexts (i.e. encrypted secret keys) to other members of the group, who modify their local version of the tree on these nodes accordingly (using the Process operation (§3)).

"Blanking" Nodes. When the tree is initialized or expanded, its new nodes are "blank," without any associated data. Add and Remove operations can "blank" nodes by removing their associated data. When a *leaf* node is blanked, we call it a ***tombstone***, and it is given a special value indicating its former association with a certain user, distinguishing it from an "empty" node that never had associated data. Blanked inner nodes look like empty nodes.

Adding & Removing Members. The Add and Remove operations work similarly, by editing all of a leaf's ancestors up to the root every time a leaf is edited. For an Add operation, the calling user id associates the new user id' with an empty leaf of the tree,[8] labeling the leaf with id' and their initial public key $\mathsf{pk}_{id'}$ (obtained through our assumed PKI). Then, the calling user *blanks* that leaf's ancestor nodes, up to and including the root. This is done to preserve Invariant 1: at the start of the Add operation, the ancestors of the leaf of id' contain secret keys which id' cannot access. By blanking the leaf's ancestors, the group secret becomes undefined ($S = \bot$). The calling user then broadcasts these changes to other members. A Remove operation works similarly: the calling user blanks the leaf of the removed member and all its ancestors, and broadcasts this. Immediately after an Add or Remove operation, the group secret is undefined; it will be reestablished by the next Update operation.[9]

Replaying Operations & Handling Concurrency. The central challenge of the decentralized setting is that concurrent operations may be delivered in any order. After a user performs an operation, they broadcast a message containing the data necessary for other users to implement corresponding changes to their copy of the BeeKEM tree. In general, if different users receive operations in different orders, they may end up with different trees. In contrast, our construction carefully ensures all users who have received the same set of operations arrive at the same local state.

BeeKEM handles concurrency by detecting concurrent operations and *replaying* them in an order independent of the order in which they were received. BeeKEM records the entire history of operations in a second data structure, a directed acyclic graph (DAG) which encodes the causal relationships between operations and enables efficient detection of concurrency. We call this data structure the ***operation graph*** and denote it by G. Each user maintains a local copy of G, which is extended whenever they perform a local operation or process an operation message from another user. Users traverse G to replay concurrent operations, such that users observe a consistent ordering; §4.3 provides details.

7. Leaves are at level 0 and the root is at level $d - 1$.

8. If there are no more empty leaves, the tree size is expanded. See §4.3.

9. In some CGKA schemes, the group secret is also defined after Create, Add and Remove; it is possible to simulate this behavior in BeeKEM by having the calling user immediately perform an Update after these other operations. BeeKEM is simpler to understand and analyze with group secrets only defined by Update operations.

4.2. Building Blocks

Notation. $\lambda \in \mathbb{N}$ is a security parameter and negl denotes a negligible function (in λ). For brevity, we generally omit the security parameter from algorithm inputs.

Assumptions. We assume a public key infrastructure (PKI) which allows a user id to provide a public key pk_{id} which can later be obtained by another user id'. The PKI will be used when calling Add to include the targeted user's initial key in the group. We further assume a reliable broadcast protocol (RBP) [27], such as a gossip network that all users are part of.

Non-Interactive Key Exchange (NIKE). A *non-interactive key exchange* scheme allows two parties, given knowledge of each other's public keys, to establish a shared symmetric key. It is "non-interactive" in the sense that, once the public keys have been published, no further communication is necessary for the two users to compute their shared key. Diffie-Hellman key exchange [17] is a notable example of a NIKE scheme. In general, a NIKE is a pair of algorithms $\mathsf{NIKE} \coloneqq (\mathsf{Kg}, \mathsf{SharedKey})$ as follows:

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\!\$\ \mathsf{NIKE.Kg}$: A randomized algorithm that takes λ as implicit input and outputs a key pair $(\mathsf{pk}, \mathsf{sk})$.
- $\mathsf{dk} \leftarrow \mathsf{NIKE.SharedKey}(\mathsf{pk}, \mathsf{sk})$: A deterministic algorithm that takes as input a public key from one key pair and a secret key from another, and returns a shared secret key which we denote dk (for "derived key").

The NIKE.SharedKey algorithm has a ***symmetry property***:

$$\Pr_{\substack{(\mathsf{pk}_1,\mathsf{sk}_1)\\(\mathsf{pk}_2,\mathsf{sk}_2)}}[\mathsf{SharedKey}(\mathsf{pk}_1, \mathsf{sk}_2) = \mathsf{SharedKey}(\mathsf{pk}_2, \mathsf{sk}_1)] = 1,$$

for $(\mathsf{pk}_1, \mathsf{sk}_1), (\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow\!\!\$\ \mathsf{NIKE.Kg}$. We will rely on HKR-CKS security of NIKE schemes, which requires that an attacker given a set of public keys cannot distinguish between actual shared keys and keys randomly sampled from the shared key keyspace. We refer readers to [20] for a formal definition. At a high level, HKR-CKS adversary $\mathcal{A}$ has a user registration oracle (which returns the generated public key), and a challenge/test oracle that takes as input a pair of registered users and returns their shared key (in the real world) or a freshly generated key (in the ideal world). An adversary wins if they can tell which of the two worlds they are in; we denote $\mathcal{A}$'s advantage in doing so by $\mathrm{Adv}^{\text{hkr-cks}}_{\mathsf{NIKE}}(\mathcal{B})$.

Symmetric Encryption (SE). A symmetric encryption scheme is a tuple of algorithms $\mathsf{SE} := (\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec})$:

- $\mathsf{sk} \leftarrow\!\!\$\ \mathsf{SE.Kg}$: A randomized algorithm that takes λ as implicit input and outputs a (symmetric) secret key sk.
- $ct \leftarrow\!\!\$\ \mathsf{SE.Enc}(\mathsf{sk}, m)$: A randomized algorithm that takes as input a secret key sk and a message m, and returns a ciphertext ct.
- $m \leftarrow \mathsf{SE.Dec}(\mathsf{sk}, ct)$: A deterministic algorithm that takes as input a secret key sk and a ciphertext ct.

We require standard correctness of SE: for any message m, $\Pr_{\mathsf{sk} \leftarrow\!\!\$\ \mathsf{SE.Kg}}[\mathsf{SE.Dec}(\mathsf{sk}, \mathsf{SE.Enc}(\mathsf{sk}, m)) = m] > 1 - \mathsf{negl}$.

BeeKEM requires that the shared keys obtained from the NIKE are compatible with the key space used in the symmetric encryption scheme. That is, if one uses the NIKE $\mathsf{SharedKey}$ to furnish derived keys dk and uses them as secret keys sk for encryption and decryption under SE, the encryption and decryption algorithms of the scheme still function correctly and securely.[10]

We require standard multi-user IND-CPA [11] security of SE, which requires that an attacker that can obtain encryptions of chosen messages for any user cannot distinguish the encryptions of any two challenge messages of the same length for any user. $\mathrm{Adv}^{\text{mu-cpa}}_{\mathsf{SE}}(\mathcal{C})$ denotes the advantage of adversary $\mathcal{A}$ in the IND-CPA game.

Hash DAG. A *hash DAG* is a directed acyclic graph (DAG) in which nodes are (formatted) strings containing hashes of other nodes, which are treated as their predecessors in the graph. Hash DAGs may be used to represent a history of events and the causal relationships between them, where the event with node a has its hash contained in the node of event b if $a \prec b$. Intuitively, being able to write the hash of a in the description of b requires that a existed prior to b, thus implying $a \prec b$ by the definition of causal order (Definition 1 (Bullet 2)). More generally, there is a path from b to a in a hash DAG if and only if $a \prec b$.

BeeKEM constructs its operation graph G as a hash DAG. When a user performs an operation, they construct a string representation of that operation, including its author, the payload of data associated with that operation, and the hashes of its predecessor operations. Users also track the ***head operations*** of their copy of G, where the *heads* of a hash DAG are the nodes which have no successors; a (nonempty) hash DAG always has one or more heads. Head operations are the most recent operations a user is aware of.

ACB from Hash DAG over RBP. Our BeeKEM implementation uses a standard construction of an ACB from an RBP (a weaker communication model) using a digital signature scheme and a hash DAG (the operation graph) [24], [25]. Hence, while our proofs assume an ACB, our implementation only requires an RBP.

10. In our implementation of BeeKEM (see §6), we use Elliptic-Curve Diffie-Hellman as the NIKE and ChaCha20-Poly1305 as the symmetric encryption scheme.

Materialization Function. Recall again that the primary challenge of the decentralized setting is managing concurrency. Concurrent messages may be received and processed by users in any order, but users must converge upon the same group state (i.e., BeeKEM tree) regardless of the order messages arrive.

Formally, a user's BeeKEM tree should be a function only of their operation graph. This deterministic mapping from sets of operation messages to BeeKEM trees is called the ***materialization*** function. Materialization is invoked during $\mathsf{Process}$; we call the process of reconciling a batch of concurrent operations a ***merge***. BeeKEM's materialization function is designed to have the following properties:

- *Strong convergence*: Two users who have observed the same set of operations compute identical tree state, regardless of operations' order of receipt. (Note that this implies agreement not just on the current group members but also their placement among the leaves.)
- *Remove liveness*: If a user is targeted for removal in a Remove operation among a batch of concurrent operations, that user is not present in the group after the merge of those operations.
- *Add liveness under Remove-wins*: If a user is targeted for addition in an Add operation among a batch of concurrent operations, that user is present in the group after merge—unless that same user is targeted by a concurrent Remove, which takes precedence.
- *No secret after merge*: There is no new group secret defined after the merge of concurrent operations. (Only an Update can define a group secret.)

Readers familiar with conflict-free replicated data types (CRDTs) may recognize concepts and vocabulary from that literature here. BeeKEM is in fact a CRDT.

4.3. Detailed Description of BeeKEM

First, we introduce the two data structures BeeKEM relies upon (§4.3.1); second, we describe how BeeKEM implements the operations of a DCGKA scheme (§4.3.2); and third, we explain how BeeKEM handles concurrency (§4.3.3). BeeKEM's complete specification in pseudocode is in Fig. 6 and 7 in App. A.

4.3.1. BeeKEM's Data Structures. BeeKEM is supported by two primary data structures (introduced in §4.1): the BeeKEM tree B, which encodes group membership and the update secret; and the operation graph G, which stores operation history and causal relationships.

BeeKEM Tree. The BeeKEM tree is a perfect binary tree B that stores an encryption of S at the root when $S \neq \perp$. Each group member maintains a local copy of B.

We discuss the structure of a tree of depth $d = \lceil \log_2 n \rceil$. A leaf stores (id, pk_{id}) if it is associated with a group member id; or $(id, \perp)$ if id has been removed from the group; or $\varnothing$ if it is empty. An inner node either stores $\varnothing$, if it is blank; or one or more ***versions***. A version contains i) a public

key, and ii) one or more encryptions of the corresponding private key. In the simplest case there is one version with one ciphertext; we explain the simple case here then address the general case when discussing Update in §4.3.2. Let v denote an inner node with associated public-private key pair $(\mathsf{pk}_v, \mathsf{sk}_v)$ and children l and r. Invariant 1 requires that sk_v can be computed given access to either child's secret key, sk_l or sk_r. BeeKEM achieves this by encrypting sk_v *under a symmetric key shared between its children*: using a NIKE, we derive a shared key dk_{lr} from the keys of both children, which is used to encrypt $c_v \leftarrow\!\!\$\ \mathsf{SE.Enc}(\mathsf{dk}_{lr}, \mathsf{sk}_v)$; then (pk_v, c_v) is stored at v. Given a child secret key sk_l, one can re-compute the shared key as

$$\mathsf{dk}_{lr} \leftarrow \mathsf{NIKE.SharedKey}(\mathsf{pk}_r, \mathsf{sk}_l), \tag{1}$$

using the sibling r's public key pk_r, which is stored in the clear (and vice-versa for the other child's secret). Having thus computed dk_{lr}, one can decrypt c_v and recover sk_v. With this structure holding throughout the tree, any group member can access all subgroup secrets on their leaf's path to the root, recursively: for each level $\ell \in \{0, \ldots, d-2\}$, they can use their secret key from level ℓ and its sibling public key to derive a shared key dk as in (1) and use dk to decrypt the level-$(\ell+1)$ ancestor's subgroup secret.

Direct Paths & Co-Paths. A leaf's *direct path* is the path of nodes from that leaf to the root. A leaf's *co-path* is the ordered set of nodes which are the siblings of each node in that leaf's direct path. By a *user's direct path*, we mean the user's leaf's direct path; likewise for the *user's co-path*.

Operation Graph. As aforementioned, the operation graph G is a hash DAG which records the history of DCGKA operations performed by the group. Each user stores a local copy of G, which encodes the operations processed by that user so far and allows detection of causal order. When a user performs a new operation, they broadcast a message containing their current head operations as its predecessors.

Every node in G stores the type of operation (Create Add, Remove, or Update) that it corresponds to, the ID of the calling user thereof, the hashes of its predecessors in G, and other operation-specific fields (e.g., for an Add or Remove, the id' of the added or removed user).

4.3.2. BeeKEM's Operations.

Group Creation. In BeeKEM, the Create operation initializes a group with only the calling user as a member.[11] This results in the calling user creating initial versions of B and G in their local state as follows. The initial version of B is a two-level tree, containing only (id, pk_{id}) for the calling user in the first leaf, with the other two nodes (i.e., the second leaf and root nodes) empty. As such, there is no group secret defined upon group creation. The initial version of G contains a unique sink of the graph (similar to an "initial commit" in Git), which is the only node with no predecessor. Looking ahead, this initial operation is always a valid point for beginning a replay of the operation graph. Finally, the user stores sk_{id} separately in their local state.

Adding a Member. When a user id performs an Add operation to add a user id', they first identify a fresh (empty) leaf for the new user and annotate it with id' as its owner and their public key $\mathsf{pk}_{id'}$. We assume this initial public key $\mathsf{pk}_{id'}$ is obtained through a PKI external to BeeKEM, which we treat as a black box. The fresh leaf is chosen as the rightmost empty leaf in our implementation.[12] If there are no empty leaves, then we expand the tree to the next power of two to create more empty leaves (the old root becomes the left child of the new root). Then, every node on the direct path of the leaf for id' is blanked, including the root. This is done to preserve Invariant 1, as those nodes' subgroup secrets (including the group secret) are inaccessible to a member of their subtree after the tree-expansion and before the blanking. Notice that there is no defined group secret after an Add; it is only established again after a user performs an update. To conclude, id broadcasts $(id', \mathsf{pk}_{id'})$ as a control message. Finally, Add runs in time $(\log n)$, where n is the group size.

Removing a Member. When user id performs a Remove operation to remove a user id', they identify id''s leaf in the BeeKEM tree[13] and blank that leaf and all nodes in its direct path to the root. This is again to maintain Invariant 1: before the blanking, the nodes of the direct path have a subgroup secret accessible to a member no longer part of the group. Remove does not free up leaves formerly assigned to a group member for use in future Add operations. These leaves are instead kept as *tombstones* to help with concurrent group changes; strategies to free up tombstones are an optimization.[14] To conclude member removal, id broadcasts the removed id' as a control message to all the other members of the group. As with additions, there is no defined group secret immediately following a Remove operation; a subsequent update operation is required to re-establish the update secret. Remove also runs in time $O(\log n)$.

Group Secret Updates. Before discussing the Update algorithm, we conclude our description of the general structure of the BeeKEM tree. Recall that an inner node may have more than one versions; and that a single version may have more than one ciphertext. We first explain the cause of multiple versions. Under concurrent updates, different forks ascribe different public-private key pairs to the same node, thus defining different versions. When partition heals, members keep all versions of a node established by concurrent operations; such a node is called a ***conflict node***. A causally subsequent update supersedes the conflict and establishes a

11. While the DCGKA syntax in §3 allows Create to take as input a set of user IDs to initialize the group with, BeeKEM restricts this set to be empty. This is without loss of generality: group members can be added immediately after via calls to Add.

12. This is arbitrary; it does not matter which empty leaf is used, as long as there is a deterministic way of finding the next fresh leaf to use.

13. We assume for simplicity of exposition that Add only ever targets id' not in the group and that Remove only ever targets id' in the group. Our implementation aborts when these conditions do not hold.

14. Our implementation of BeeKEM frees up tombstones (turning them into empty nodes) whenever they form a contiguous right edge.

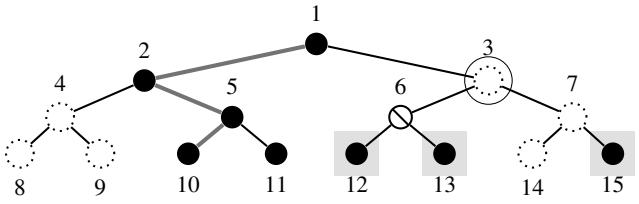

Figure 2. A BeeKEM tree showing different types of nodes. Dotted nodes denote blank nodes, and the slashed node at 6 is a conflict node. (This tree could arise if the users at leaves 12 and 13 concurrently performed updates, then the partition healed and the user at leaf 14 was removed.) The user at node 10 performs an update. When encrypting the subgroup secret for node 2, the user will create a throwaway key pair (as node 4 has an empty resolution). When encrypting the group secret at node 1, the user finds the resolution of node 3, which is the greyed nodes 12, 13, and 15.

single version. If the root is a conflict node, we consider the group secret undefined, i.e. $S = \bot$.

The cause of multiple ciphertexts explains something readers may wonder: for a given inner node, why use dk to encrypt sk rather than use dk itself as the subgroup secret? This is because situations can arise where the sibling node either is blank (and has no public key) or is a conflict node, with no canonical public key. In such situations, BeeKEM collects the nodes which "ought" to have had access to the sibling's subgroup secret, if it had one, and makes sk available to them by collecting their pk, computing corresponding dk, and encrypting sk under each. This creates agreement on the subgroup secret sk even when there is no agreement on dk. Formally, the ***resolution*** of a non-blank, non-conflict node is defined as just the node itself; the resolution of a blank or conflict node is the set of its highest descendants which are not blank or conflict nodes. Figure 2 gives an example of a node resolution.

We now present the update algorithm. Let id be the updating user, and let v_ℓ denote the level-ℓ node on their direct path, for $\ell \in \{0, \ldots, d-1\}$. Then v_0 is id's leaf and v_{d-1} is the root. First, user id generates d fresh public-private key pairs $(\mathsf{pk}_\ell, \mathsf{sk}_\ell) \leftarrow\!\!\$\ \mathsf{NIKE.Kg}()$. The ℓ-th key pair $(\mathsf{pk}_\ell, \mathsf{sk}_\ell)$ replaces the keys associated with v_ℓ. In particular, $(\mathsf{pk}_0, \mathsf{sk}_0)$ are the new keys for id. The user updates v_0 with pk_0, and stores sk_0 locally. User id then travels up their direct path iteratively, processing the new key pairs via an analogous procedure to that given in (1) for accessing shared secrets. For every inner node v_ℓ for $\ell \in \{1, \ldots, d-1\}$, the updating user enshrines the subgroup secret of v_ℓ as follows. First, id takes the resolution of the co-path sibling of $v_{\ell-1}$ (said differently: the other child of v_ℓ). Then, for every node u in the sibling's resolution,[15] id takes its public key $\mathsf{pk}^u_{\ell-1}$ and computes

$$\mathsf{dk}^u_{\ell-1} \leftarrow \mathsf{NIKE.SharedKey}(\mathsf{pk}^u_{\ell-1}, \mathsf{sk}_{\ell-1}),$$

where $\mathsf{sk}_{\ell-1}$ denotes, as above, the secret key of $v_{\ell-1}$ (which id just sampled). Then, id encrypts the new subgroup secret sk_ℓ under each $\mathsf{dk}^u_{\ell-1}$ as $c^u_\ell \leftarrow\!\!\$\ \mathsf{SE.Enc}(\mathsf{dk}^u_{\ell-1}, \mathsf{sk}_\ell)$. This creates a new version with pk_ℓ and all c^u_ℓ, which id publishes at v_ℓ. Ultimately, $S = \mathsf{sk}_{d-1}$ is encrypted and stored at the root v_{d-1}, becoming the new group secret. (Accordingly, when a user decrypts the group secret in the general case, they skip any blank or conflict nodes on their direct path, going up their ancestors: some higher node contains a ciphertext they can decrypt.)

The control message that results from an update contains a list of the new versions for each node on the direct path; and the set of public keys rendered obsolete by this operation. Update runs in time $O(\log n)$ in the best case, where each node of the co-path is non-blank and non-conflict. Performance degrades with concurrency, which creates conflict nodes; in the worst case it decays to $O(n)$, if a resolution set contains $O(n)$ nodes.

Processing Operations. As described, BeeKEM's Update, Add, and Remove operations only result in control messages, which are broadcast from the calling user to all other group members via an ACB (§3); BeeKEM does not rely on direct messages. All three of BeeKEM's control-message types have the same structure: (1) the operation that was performed, and (2) the new content for the nodes modified by the operation (which updates, additions, and blankings). Then, $\mathsf{Process}$ simply involves updating the user's local copies of G and B according to (1) and (2), by replaying the operation that was received and appending the operation to the operation graph G.

4.3.3. Handling Concurrency. Recall that BeeKEM can detect concurrency using the operation graph G (§4.2). When concurrent operations are processed, BeeKEM *replays* the entire history of operations from the initial Create operation,[16] using the materialization function to determine the effects of concurrent operations and guaranteeing convergence between users.

Definition 2 (Topological sort)**.** *Let G be a DAG. A* topological sort *of G is an ordering of its nodes such that, if directed edge (u, v) is in G, then u precedes v in the ordering.*

To compute the materialization, the user first (deterministically) topologically sorts[17] the operation graph G with respect to the reverse of the hash edges; the resulting topological sort respects the causal order over operations by construction (see §4.2). In doing this, the user also identifies where G forks into separate branches; these periods of concurrency are called ***batches***, and they are also ordered with respect to each other by the topological sort.

The user then applies each sequential operation and each batch in the topological sort order. Sequential operations are applied one-by-one in the expected way; but when encountering a batch, the operations of the batch are applied in a special way to handle their concurrency. Within a batch, the user iterates over the sorted operations collecting two

15. In the edge case where the resolution set is empty, id generates a "throwaway" key pair and publishes its pk at the sibling.

16. An optimization is to periodically checkpoint to replay fewer operations. In practice we find replaying the full history does not cause performance issues.

17. A topological sort can be computed efficiently for a DAG [16].

sets: a set $\mathfrak{A}$ of added members; and a set $\mathfrak{R}$ of removed members. The user applies each operation in the batch in the topological sort order. When performing concurrent Update operations, the different versions of a node are all kept (unlike sequential updates, which supersede each other). If the operation is Add or Remove, they add the targeted user to the relevant set. At the end, the members in $\mathfrak{R}$ have their leaves' direct paths (re-)blanked (as concurrent Updates could have populated their direct paths). Then, the members in $\mathfrak{A} \setminus \mathfrak{R}$ are sorted in some deterministic way (e.g. lexicographic ordering over IDs), removed from the tree, then reinserted in that order. This reinsertion is done because the same leaf may have been assigned to distinct members on different branches.

BeeKEM Pseudocode. In App. A, we present the complete BeeKEM protocol as pseudocode in Fig. 6 and 7.

5. Security Analysis of BeeKEM

We overview our security analysis of BeeKEM, which rigorously demonstrates it achieves security properties from §3.3. Full proofs are in App. B.

Security Definition. Our security and correctness notions are captured in the key indistinguishability game introduced by Weidner at al. [34], which formalizes the security goals for DCGKA schemes (given in §3). Figure 8 in App. B provides a full formal definition of the game.

Key Indistinguishability Game. The ki-DCGKA game starts by initializing a CGKA scheme Π and variables γ to track the flow of control and direct messages to be delivered.

After setup, the adversary $\mathcal{A}$ runs with oracle access to the operations of Π and other oracles described below. $\mathcal{A}$ has perpetual access to the list of (control and direct) messages exchanged. Using oracles to interface with Π, $\mathcal{A}$ may create a group, modify group membership (i.e., add or remove users), trigger updates, and process messages. These oracles call the corresponding algorithm in Π, and add the resulting control and direct messages to local variables maintained by the game. These messages are delivered when the adversary calls a dedicated delivery oracle, which results in each receiving user processing the message. Due to the ACB, the adversary cannot modify messages, deliver messages out of casual order, or forge sender identities.

The adversary may compromise users of its choice via a dedicated Compromise oracle, which reveals the chosen users' local state. In addition, the adversary may obtain any user's group secrets, either via the Reveal oracle (which simply reveals the secret), or the Challenge oracle, which returns either the real secret or a random string (of the same length) depending on the value of a random challenge bit.

ki-DCGKA is parametrized by a ***safety predicate*** P, which rules out sets of adversary queries that would lead to "trivial wins" that do not violate any of our desired security properties (e.g., compromising a member's state and immediately recovering their group secret). P encodes the precise PCS and FS guarantees that a scheme must have if secure under the ki-DCGKA game with respect to P.

$\mathsf{bee\text{-}safe}_\kappa(q_1, \ldots, q_n)$:

$\forall\big(i, j$ s.t. $q_i = \textsc{Challenge}(id, c)$ for some id, c and
$q_j = \textsc{Compromise}(id')$ for some $id'\big)$,
$\big(\exists(u_1, \ldots, u_\kappa)$ s.t. $\forall l, q_{u_l} = \textsc{SendUpdate}(id')$, // k-FSU
$\mathsf{q2op}(q_i) \prec \mathsf{q2op}(q_{u_\kappa}) \prec \cdots \prec \mathsf{q2op}(q_{u_1}) \preceq \mathsf{q2op}(q_j)\big)$
$\vee\big(\exists u$ s.t. $q_u = \textsc{SendUpdate}(id')$, // PCS
$\mathsf{q2op}(q_j) \prec \mathsf{q2op}(q_u) \preceq \mathsf{q2op}(q_i)\big)$
$\vee\big(\exists(u_1, \ldots, u_\kappa)$ s.t. $\forall l, q_{u_l} = \textsc{SendUpdate}(id')$, // k-CFS
$(\mathsf{q2op}(q_{u_\kappa}) \prec \cdots \prec \mathsf{q2op}(q_{u_1}) \preceq \mathsf{q2op}(q_j))$
$\wedge(\mathsf{q2op}(q_i) \sim \mathsf{q2op}(q_j))\big)$

Figure 3. bee-safe predicate, where $q_1, \ldots, q_n$ denote the n oracle queries performed by the adversary and κ is the parameter controlling the tradeoff between fork tolerance and forward secrecy. $a \preceq b$ denotes $a \prec b \vee a = b$.

We define the advantage of adversary $\mathcal{A}$ over DCGKA scheme Π with respect to safety predicate P as

$$\mathrm{Adv}^{\text{ki-dcgka}}_{\Pi,\mathsf{P}}(\mathcal{A}) = \Pr\left[\,\text{ki-DCGKA}^{\mathcal{A}}_{\Pi,\mathsf{P}} \Rightarrow 1\,\right].$$

BeeKEM Safety Predicate. Figure 3 formally defines the safety predicate bee-safe that BeeKEM satisfies. The three clauses of the disjunction correspond to the three security properties achieved by BeeKEM: κ-FSU, PCS, and κ-CFS (see §3.3). bee-safe is parameterized by κ; smaller values of κ weaken the predicate, excluding more query sets.

Theorem 1. *Let* SE *be a symmetric encryption scheme,* NIKE *be a non-interactive key exchange scheme compatible with* SE*, and* $\Pi = \mathsf{BeeKEM}[\mathsf{SE}, \mathsf{NIKE}]$*. Let* $\mathcal{A}$ *be a* ki-DCGKA *adversary that makes at most* c *challenge oracle queries and at most* n *member additions. Then, we give adversaries* $\mathcal{B}$*, and* $\mathcal{C}$ *such that, for any integer* $k \geq 1$*,*

$$\mathrm{Adv}^{\text{ki-dcgka}}_{\Pi,\mathsf{bee\text{-}safe}_k}(\mathcal{A}) \leq c \cdot \lceil \log_2 n \rceil \cdot \big(\mathrm{Adv}^{\text{hkr-cks}}_{\mathsf{NIKE}}(\mathcal{B}) + \mathrm{Adv}^{\text{mu-cpa}}_{\mathsf{SE}}(\mathcal{C})\big).$$

We present the full proof of this theorem in Appendix B.

Proof Sketch. We first use a standard hybrid argument to bound the advantage of an adversary that performs multiple challenge queries by the advantage of an adversary that performs a single challenge query. This allows us to focus on one challenge query thereafter, which simplifies the proof.

The core of the proof is then computing the advantage of the single-query adversary. We do so via a hybrid argument over the nodes that are modified during the update that preceded the challenge, which established the current group secret at that time. For each node in the update path, from the bottom up, we define a pair of hybrids: the first replaces computations of NIKE shared keys with random keys, invoking NIKE security; and the second replaces encryptions of subgroup secrets with encryptions of all zeros, invoking encryption security. Arguing that the NIKE and IND-CPA adversaries can simulate the DCGKA adversary's environment, including DCGKA operation oracles, is non-trivial; App. B gives details. □

6. Implementation and Evaluation

We evaluate our implementation and compare it to two existing systems: OpenMLS [30], a popular implementation of MLS and its centralized TreeKEM protocol [7], and Weidner et al.'s DCGKA scheme [34], which we call WKHB based on the authors' initials. Our implementation is open source;

Our experiments explore two scenarios. In Section 6.1 we compare the three implementations in a setting where the group members perform Add, Remove, and Update operations *sequentially*, i.e., where one user's operation is received by the other users before the next operation is generated. We show that in this setting, the cost of MLS and BeeKEM grow logarithmically with the group size as expected, whereas the cost of WKHB grows linearly.

In Section 6.2 we measure how BeeKEM's performanceis affected by operations performed during a network partition, resulting in an operation graph containing concurrent operations. This experiment shows that the cost of resolving conflict nodes in BeeKEM is approximately proportional to the fraction of users who called Update at least once during the network partition, resulting in a worst-case conflict resolution cost that is linear in the group size.

We perform our experiments by simulating all users, and the network between them, on a single machine with a 16-core AMD CPU and 128 GB of RAM. We run each experiment 5 times and report the median. Run-to-run timing is stable: the standard deviation across the 5 repetitions stays below 1% of the median for slower (multi-millisecond) operations. Greater variance (up to 16%) was observed only on sub-millisecond operations where the absolute variance is negligible. We therefore omit error bars from our graphs.

6.1. Best-Case Protocol Comparison

Sequential execution of operations with no concurrency is the best case for BeeKEM, since it avoids introducing conflict nodes that would require resolution. For MLS, this sequential execution is the only possible mode of operation.

Fig. 4 shows the CPU time used by each of the three systems for different operations at group sizes varying from 8 to 512 members. "Sender" refers to the cost for the user who invokes Add, Remove, or Update. "Recipient" refers to the cost for an existing group member calling Process after receiving the operation from another user. "New recipient" refers to the cost of Process for the new member being added by an Add operation.

For the sender, the $O(n)$ cost of Update in WKHB is clearly visible in Fig. 4. BeeKEM has the lowest CPU time for both senders and recipients of the three systems we compared. We were surprised to see linear growth in the OpenMLS CPU time at larger group sizes. On further investigation, we found that although the core TreeKEM construction in MLS has $O(\log n)$ cost, OpenMLS additionally recomputes tree and parent hashes over the whole ratchet tree, which has $O(n)$ cost and becomes a significant contributor to CPU time at large group sizes.

All three systems have a higher cost for the new member being added by an Add operation, since this user needs to download a welcome message and compute the current group state. In OpenMLS this requires $O(n)$ work to verify the leaf signatures of the whole ratchet tree; in WKHB the cost is building the ratchet state for each of the n members; and for BeeKEM the cost is replaying the $O(n)$ operation history. BeeKEM is the most expensive of the three on this metric, though they are asymptotically similar. Each Update grows the welcome message by 2.5 kB and increases the Process time for that message by 40 µs.

In order to separate the effects of implementation language (BeeKEM and OpenMLS are written in Rust, WKHB in Java) and other implementation details (such as OpenMLS's ratchet tree recomputation) from the core CGKA cost, we additionally perform an asymptotic analysis whose results are reported in Table 1. We instrument the three implementations to count the number of invocations of public-key cryptographic primitives (Diffie-Hellman, HPKE, signatures, and asymmetric keypair generation), since these dominate the runtime, whereas symmetric primitives (hashes, KDFs, and AEADs) are comparatively fast. For recipients we compute the average number of invocations per group member who calls Process. We then report which of $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, or $O(n^2)$ best fits the observed invocation counts.We also measure the size of the network messages and the size of data in persistent storage (in bytes), and likewise map these to a big-O class.

6.2. BeeKEM Under Network Partitions

Next, we evaluate BeeKEM's performance under network partitions. We do not compare to other implementations: MLS cannot operate under partition, and WKHB's behavior under partition is largely the same as in §6.1.

The scenario we test is a group of $n = 64$ members that is initially connected. A network partition then divides it into four equal-sized subsets such that members within each subset can communicate, but members across subsets cannot. During the network partition, Update is called by U group members, sampled uniformly without replacement. If $U > n$, once all n group members have called Update, a new round begins and all members are again eligible to be sampled. We then end the partition and allow all members to communicate again.

Fig. 5 shows the result of varying $U \in [0, 2n]$ and measuring two aspects of post-partition recovery: (1) the CPU time for the first group member (chosen randomly) to call Update after the end of the partition, and the size of the message it generates; and (2) the cumulative CPU time until all conflict nodes in the tree have been updated through successive Updates, and the cumulative network traffic this generates. The first metric captures the immediate cost of resuming encrypted application traffic; the second captures the recovery window before BeeKEM returns to the sequential behavior examined in Section 6.1.

If no member updates during the partition, there are no conflicts to resolve. As the fraction of members that have

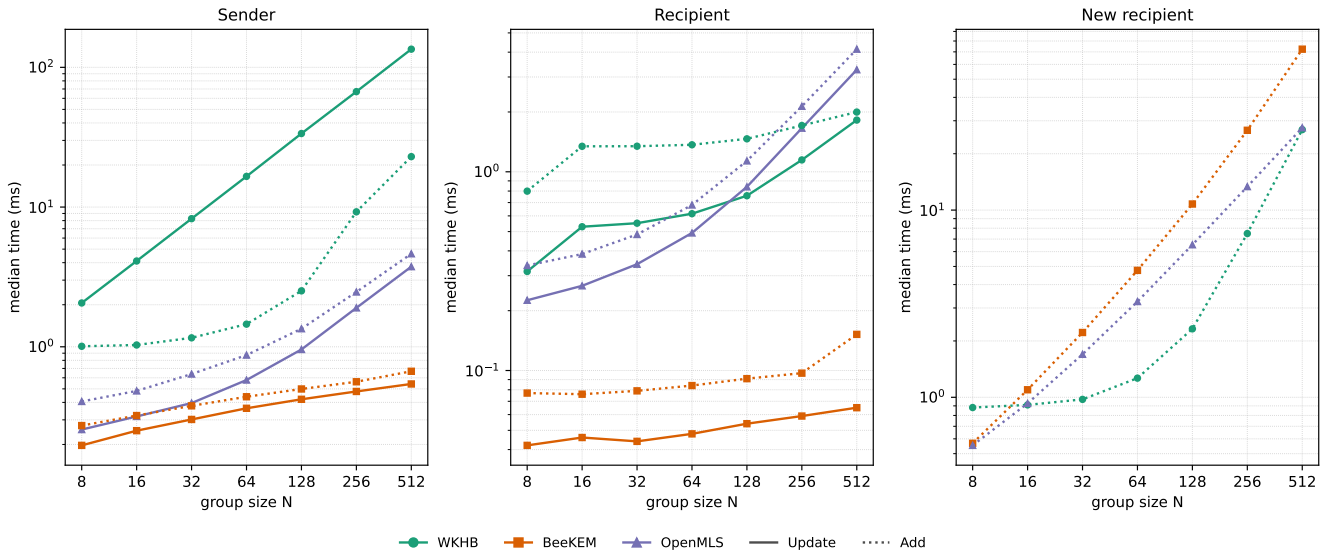

Figure 4. Per-operation, per-user CPU cost as the group size grows. Solid lines are for Update operations; dotted lines for Add; and Remove operations are not shown since their cost is very similar to Update.

TABLE 1. ASYMPTOTIC COST OF OPERATIONS PERFORMED SEQUENTIALLY. "SENDER PRIMITIVES" IS THE NUMBER OF PUBLIC-KEY PRIMITIVES INVOKED BY THE USER CALLING Add/Remove/Update; "RECIPIENT PRIMITIVES" IS FOR A USER CALLING Process AFTER RECEIVING THE OPERATION. HERE n IS THE GROUP SIZE, h_B THE SIZE OF THE OPERATION HISTORY, AND h_D THE SIZE OF WKHB'S GROUP-MEMBERSHIP HISTORY.

System	Operation	Sender primitives	Recipient primitives	Message size	Persistent storage
OpenMLS	Update / Remove	$O(\log n)$	$O(1)$	$O(\log n)$ broadcast	$O(n)$
	Add	$O(\log n)$	$O(1)$	$O(\log n)$ commit plus $O(n)$ welcome	
BeeKEM	Update / Remove followed by Update	$O(\log n)$	$O(1)$ existing recipient	$O(\log n)$ broadcast	$O(n + h_B)$
	Add followed by Update	$O(\log n)$	$O(1)$ existing recipient; $O(h_B)$ new recipient	$O(\log n + h_B)$ with welcome history	
WKHB	Update / Remove	$O(n)$	$O(1)$	$O(n)$	$O(n^2 + h_D)$
	Add	$O(1)$	$O(1)$	$O(h_D)$ welcome history	

updated during the partition increases, Fig. 5 shows that the total post-partition recovery cost increases linearly, but it plateaus once each member has updated once, and further updates add little further cost. The cost of the first post-message update grows more slowly.

7. BeeKEM Extension Sketches

We sketch BeeKEM$^{\text{FS}}$ and BeeKEM$^{\text{PQ}}$, variant protocols that respectively achieve FS and CFS; and post-quantum security. We defer a full treatment to future work.

BeeKEM$^{\text{FS}}$: a fully FS and CFS extension. The BeeKEM protocol was designed for a setting where users can define group secrets during partition, but re-encrypting the data generated and encrypted during the partition may be infeasible. Other contexts may feature constraints where correctness under concurrency is not as important as more robust security properties. We describe BeeKEM$^{\text{FS}}$, a variant construction that achieves "full" (rather than κ-) FS and CFS at the expense of CUC: BeeKEM$^{\text{FS}}$ still tolerates and resolves concurrent operations, but if a user performs an Update under partition, they cannot later recover any group secrets which were defined by members in other branches.

The key idea is analogous to the fix to TreeKEM in [7]: we use *Updatable Public-Key Encryption* (UPKE), which (informally) ensures that a given public-private key pair is only ever used to encrypt a single message. UPKE extends standard PKE so that UPKE.Enc outputs both a ciphertext and a new public key pk'; and UPKE.Dec outputs both the message and a new corresponding secret key sk'.

The BeeKEM$^{\text{FS}}$ tree has a similar structure to BeeKEM's: filled leaf nodes have an associated id and pk_{id}; inner nodes have versions which contain a pk and several encryptions of the corresponding secret key sk under different public keys. To access the root secret: at level ℓ, the user identifies *their* ciphertext in the parent (encrypted under pk_ℓ), decrypts the ciphertext under sk_ℓ to obtain $\mathsf{sk}_{\ell+1}$, and also a new secret key sk'_ℓ, which they retain in their local state. The user *immediately* deletes the old corresponding secret key sk_ℓ from their local state.

BeeKEM$^{\text{FS}}$'s Update procedure generates a payload which contains changes to both the updating user's direct path *and to their co-path*. First, the updating user gener-

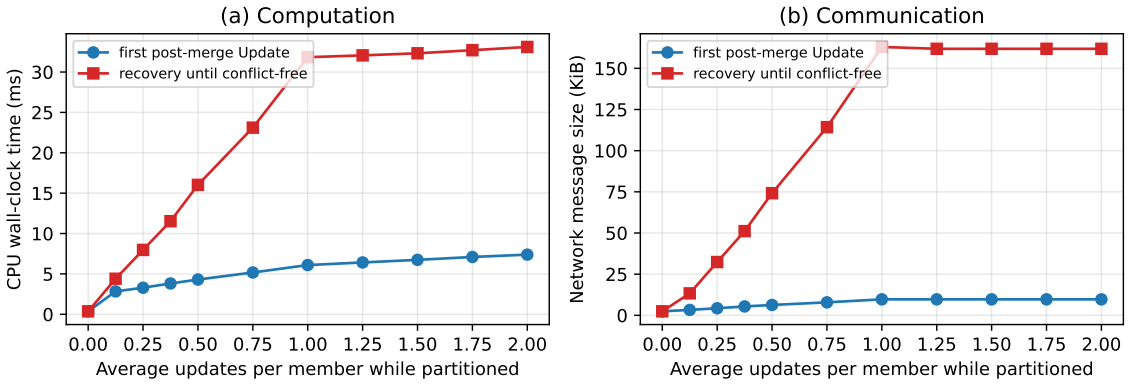

Figure 5. Cost of the first post-partition Update (blue) and cumulative cost of resolving all conflict nodes in the BeeKEM tree (red). Left: CPU time, right: network bandwidth.

ates a sequence of d public-private key pairs $(\mathsf{pk}_\ell, \mathsf{sk}_\ell)$ for $\ell \in \{0, \ldots, d-1\}$, as before. They take $(\mathsf{pk}_0, \mathsf{sk}_0)$ as their new personal public-private key pair. Then, going up the tree, at level ℓ they ascribe the key pair $(\mathsf{pk}_{\ell+1}, \mathsf{sk}_{\ell+1})$ to the parent as follows. First, they collect all relevant public keys from the sibling's resolution in a set R; and to R they add their own public key at level ℓ, pk_ℓ. Then, for each public key pk_u in the set R, they compute $ct, \mathsf{pk}'_u \leftarrow \mathsf{UPKE.Enc}(\mathsf{pk}_u; \mathsf{sk}_{\ell+1})$, and store ct under pk_u at the parent, and also set pk'_u as the new public key of u. Note that this includes the public key on the calling user's side, who then immediately updates the public key of their direct path node from pk_ℓ to some pk'_ℓ; this is necessary to preserve the symmetry of the tree. The root secret is thus updated to $S = \mathsf{sk}_{d-1}$. Group creation, state initialization, and membership changes are identical to BeeKEM. Lastly, when a user receives an Update message, they immediately decrypt the group secret and delete their old secret.

Intuitively, BeeKEM$^{\mathrm{FS}}$ achieves FS and CFS because users immediately delete their old personal secret upon processing an Update operation.

BeeKEM$^{\mathrm{PQ}}$: A Post-Quantum Extension. Building efficient messaging secure against quantum computers is an area of active work [28]. Next, we describe BeeKEM$^{\mathrm{PQ}}$, a variant DCGKA construction that can be built from cryptographic primitives with practical, secure PQ instantiations. BeeKEM$^{\mathrm{PQ}}$ is built using key encapsulation mechanisms (KEM), a PQ primitive [2]. We sketch its construction.

The basic idea is to change computations of shared sibling keys from using a NIKE to using a KEM. (All other cryptographic primitives remain unchanged, except that we instantiate them with PQ constructions.) A KEM is defined by three algorithms: Kg, Encap, and Decap [2]. $(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\!\$\ \mathsf{Kg}()$ generates a public-private key pair. $(c, \mathsf{dk}) \leftarrow\!\!\$\ \mathsf{Encap}(\mathsf{pk})$ "encapsulates" an input public key and returns a ciphertext c and *derived key* dk, which can be "decapsulated" via $\mathsf{dk} \leftarrow \mathsf{Decap}(\mathsf{sk}, c)$. Each node in the BeeKEM$^{\mathrm{PQ}}$ tree, instead of storing just a public key and encrypted subgroup secret, additionally stores ciphertext c generated by *encapsulating the sibling public key*, i.e., $(c, \mathsf{dk}) \leftarrow\!\!\$\ \mathsf{Encap}(\mathsf{pk}_s)$ where pk_s is the sibling public key. The sibling can recover dk by decapsulating c with its subgroup secret, i.e., $\mathsf{dk} \leftarrow \mathsf{Decap}(\mathsf{sk}, c)$. Key dk is then used as the shared key between siblings as in BeeKEM. BeeKEM$^{\mathrm{PQ}}$ is thus able to achieve PQ security.

8. Related Work

DCGKA. Our work contributes to the literature on *decentralized* continuous group key agreement. The state-of-the-art DCGKA constructions are Causal TreeKEM [35] and the WKHB scheme [34]. BeeKEM notably differs from both.

Our biggest innovation compared to WKHB is our $O(\log n)$ update communication cost vs $O(n)$; linear complexity is not practical for large groups, which are an important use case for group messaging (see also §1). WKHB achieves better FS than BeeKEM—akin to how TreeKEM has worse FS than the naive, linear-time CGKA construction via pairwise channels. On the other hand, BeeKEM achieves better PCS than WKHB.

Causal TreeKEM also achieves $O(\log n)$ update communication. Their protocol does not achieve FS for reasons analogous to BeeKEM (retaining old update secrets to handle concurrent updates which arrive later), and it lacks formal security proofs and has no implementation. Its PCS is also imperfect, as noted in [34]: if an adversary compromises multiple users, if those users concurrently perform an update, the adversary can compute the new group secret that results when all users processes the new updates. Causal TreeKEM also requires the additional assumption that there exists a commutative/associative operation over key pairs to combine updates, whereas BeeKEM relies on standard cryptographic primitives.

A key difficulty faced by both WKHB and Causal TreeKEM lies with maintaining PCS under concurrent updates: indeed, both schemes suffer from attacks in which the attacker can compute the new group secret if the compromised users perform updates concurrently. BeeKEM avoids these issues because of the *No secret after merge* property

(§4.2): there is no group secret defined via concurrent updates.

Other CGKA Variants. While our focus is on DCGKA, we briefly note other CGKA variants in prior work [5], [6], [8]. Some of these share similar motivation to ours, but in centralized settings. For example, concurrent CGKA schemes support concurrent updates [5], [6], but they do not support processing operations that occurred before the latest processed update (a narrow form of concurrency).

Fork-resilient CGKA (FR-CGKA) [8] is also motivated by the problem of key agreement over unreliable infrastructure. Unlike our serverless setting, which supports message delivery over any medium (including, e.g., sneakernets), FR-CGKA operates in the server-aided CGKA model, which relies on a central (but possibly untrusted or unreliable) infrastructure for message delivery. In FR-CGKA, users' states and group secrets may diverge; accordingly, FR-CGKA schemes also face the possibility of cross-fork attacks [8], related to our notion of cross-fork security for DCGKA. Like BeeKEM, there exist FR-CGKA schemes with logarithmic communication cost and security against cross-fork attacks [8], but these have not been accompanied by an implementation (as far as we are aware, there are no FR-CGKA schemes with available implementations). More broadly, FR-CGKA and DCGKA have not been closely connected in prior literature; an examination of the relationship between the models would be interesting future work (in particular, to what extent attacks and security techniques in one model are applicable in the other).

9. Conclusion

We introduce BeeKEM, the first DCGKA scheme to achieve logarithmic update costs, thus bringing DCGKA performance up to par with that of centralized CGKA. We prove the security of BeeKEM, and provide a production-ready implementation as an open-source artifact. Our evaluation shows that BeeKEM's performance is comparable to, and in some cases better than, state-of-the-art CGKA. Our BeeKEM implementation enables E2EE in a wide range of decentralized *local-first* collaboration software; such applications have already been built on our work.

10. Acknowledgments

The authors would like to thank Matthew Weidner and Zachary DeStefano for providing feedback on an earlier version of this paper. Brooklyn Zelenka, John Mumm, and Martin Kleppmann are funded by the Advanced Research + Invention Agency (ARIA) through project code MSAI-PR01-P043. Martin Kleppmann also gratefully acknowledges his crowdfunding supporters including SoftwareMill.

References

[1] Access Now. Rising and resisting in the darkness: internet shutdowns in 2025. Access Now, March 2026.

[2] Gorjan Alagic. Recommendations for key-encapsulation mechanisms. Technical Report NIST SP 800-227 ipd, National Institute of Standards and Technology, 2025.

[3] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Mesh messaging in large-scale protests: Breaking Bridgefy. In *Topics in Cryptology – CT-RSA 2021: Cryptographers' Track at the RSA Conference 2021, Virtual Event, May 17–20, 2021, Proceedings*, page 375–398, Berlin, Heidelberg, 2021. Springer-Verlag.

[4] Martin R Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Collective information security in large-scale urban protests: the case of Hong Kong. In *30th USENIX Security Symposium*. USENIX, 2021.

[5] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, and Krzyzstof Pietrzak. DeCAF: Decentralizable CGKA with fast healing. In *Security and Cryptography for Networks: 14th International Conference, SCN 2024, Amalfi, Italy, September 11–13, 2024, Proceedings, Part II*, page 294–313, Berlin, Heidelberg, 2024. Springer-Verlag.

[6] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzyzstof Pietrzak, and Michael Walter. CoCoA: Concurrent continuous group key agreement. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022*, pages 815–844, Cham, 2022. Springer International Publishing.

[7] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 248–277, Cham, 2020. Springer International Publishing.

[8] Joël Alwen, Marta Mularczyk, and Yiannis Tselekounis. Fork-resilient continuous group key agreement. In *Advances in Cryptology – CRYPTO 2023: 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20–24, 2023, Proceedings, Part IV*, page 396–429, Berlin, Heidelberg, 2023. Springer-Verlag.

[9] Jacob Aron and Aviva Rutkin. Hong Kong protesters use a mesh network to organise. *New Scientist*, September 2014.

[10] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) protocol. IETF RFC 9420, https://www.rfc-editor.org/rfc/rfc9420.html, July 2023.

[11] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of cryptology*, 2008.

[12] Benjamin Beurdouche, Eric Rescorla, Emad Omara, Srinivas Inguva, and Alan Duric. The Messaging Layer Security (MLS) architecture. IETF RFC 9750, https://www.rfc-editor.org/rfc/rfc9750.html, April 2025.

[13] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous decentralized key management for large dynamic groups. https://inria.hal.science/hal-02425247/, 2018.

[14] Cisco Webex. End-to-end encryption for Webex Meetings and Webex Calling. https://help.webex.com/en-us/article/5h5d8ab/End-to-end-encryption-for-Webex-Meetings-and-Webex-Calling#section_nbd_hmq_z5b, 2025.

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

[16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 20.4. MIT Press, April 2022.

[17] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

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

[19] Mat Ford. Internet (un)reliability around the world. Internet Society Pulse, Oct 2022.

[20] Eduarda S. V. Freire, Dennis Hofheinz, Eike Kiltz, and Kenneth G. Paterson. Non-interactive key exchange. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *Public-Key Cryptography – PKC 2013*, pages 254–271, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[21] Daniel Hugenroth, Martin Kleppmann, and Alastair R. Beresford. Rollercoaster: An efficient group-multicast scheme for mix networks. In *30th USENIX Security Symposium*, pages 3433–3450, 2021.

[22] IETF News. End-to-end encryption based on the IETF-developed Messaging Layer Security set be used on 100s of millions of mobile devices. https://www.ietf.org/blog/rcs-adopts-mls/, 2025.

[23] David Inyangson, Sarah Radway, Tushar M. Jois, Nelly Fazio, and James Mickens. Amigo: Secure group mesh messaging in realistic protest settings. In Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier, editors, *Proceedings of the 2025 ACM SIGSAC CCS 2025, Taipei, Taiwan, October 13-17, 2025*, pages 4244–4258. ACM, 2025.

[24] Martin Kleppmann. Making CRDTs Byzantine fault tolerant. In *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '22, pages 8–15. Association for Computing Machinery, 2022.

[25] Martin Kleppmann and Heidi Howard. Byzantine eventual consistency and the fundamental limits of peer-to-peer databases.

[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] João Leitão, José Pereira, and Luís Rodrigues. Gossip-based broadcast. In Xuemin Shen, Heather Yu, John Buford, and Mursalin Akon, editors, *Handbook of Peer-to-Peer Networking*, pages 831–860. Springer US, 2010.

[28] Rohan Mahy and Richard Barnes. ML-KEM and Hybrid Cipher Suites for Messaging Layer Security. Internet Draft draft-ietf-mls-pq-ciphersuites-02, Internet Engineering Task Force, 2026. Work in Progress.

[29] Vibhu Mishra. UN warns of rising internet shutdowns as digital blackouts spread worldwide, Jan 2026.

[30] Phoenix R&D and Cryspen. OpenMLS, 2026. OpenMLS is a Rust implementation of the Messaging Layer Security (MLS) protocol, as specified in RFC 9420.

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

[32] The Economist. Iran's regime walls off the internet. https://www.economist.com/middle-east-and-africa/2026/03/26/irans-regime-walls-off-the-internet, March 2026.

[33] The Economist. Russia wants to limit contact with the outside world. https://www.economist.com/europe/2026/03/26/russia-wants-to-limit-contact-with-the-outside-world, March 2026.

[34] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[35] Matthew A Weidner. Group messaging for secure asynchronous collaboration. Master's thesis, University of Cambridge, 2019.

[36] Wire. Demystifying MLS: How a new standard is redefining secure collaboration. https://wire.com/en/blog/messaging-layer-security-mls-explained, 2025.

Appendix A. BeeKEM pseudocode

We present BeeKEM as pseudocode in Figures 6 and 7.

Appendix B. Details of Security Analysis

Proof of Theorem 1. Similar to [7], our proof will consist of two steps: first, we bound the advantage of an adversary that makes n challenge queries by one that makes a single challenge query; and second, we bound the advantage of an adversary that performs a single challenge query.

Lemma 1. *Let* SE *be a symmetric encryption scheme,* NIKE *be a NIKE scheme compatible with* SE*, and* $\Pi = \mathsf{BeeKEM}[\mathsf{SE}, \mathsf{NIKE}]$*. Let* $\mathcal{A}$ *be a* ki-DCGKA *adversary that makes* c *queries to the challenge oracle. There exists a* ki-DCGKA *adversary* $\mathcal{A}'$ *that makes a single query to the challenge oracle such that, for any integer* $k \geq 1$*,*

$$\mathrm{Adv}^{\text{ki-dcgka}}_{\Pi,\mathsf{bee\text{-}safe}_k}(\mathcal{A}) \leq c \cdot \mathrm{Adv}^{\text{ki-dcgka}}_{\Pi,\mathsf{bee\text{-}safe}_k}(\mathcal{A}').$$

We omit the proof of Lemma 1 as it follows from a standard hybrid argument on challenge queries.

Theorem 2. *Exactly the same as Theorem 1, except that* $\mathcal{A}$ *makes* ***only one*** *challenge oracle query.*

Proof (Theorem 2). Let $Q = (q_1, \ldots, q_e)$ be the queries made by $\mathcal{A}$, and let $q_{ch} \coloneqq \textsc{Challenge}(id, c)$ be the (unique) challenge query. Let B be the BeeKEM tree stored in $\mathsf{states}[id]$, the game variable that stores the local state of each user at the time q_i is called. In BeeKEM, only update operations generate new group secrets (§4.3). So, B was generated by the last query $q_{up} \in Q$ performed by the adversary before q_{ch} corresponding to an update processed by id. Let: id^* be the user that triggered the update (id^* may be id); $d \leq \lceil \log_2 n \rceil$ be tree depth at the time of update; and $v_0, \ldots, v_d$ be id's direct path. q_{up} generates $(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow\!\!\$\ \mathsf{NIKE.Kg}$ for id^* and stores pk_0 at v_0; for every $j \in \{1, \ldots, d-1\}$ thereafter, it generates $(\mathsf{pk}_j, \mathsf{sk}_j) \leftarrow\!\!\$\ \mathsf{NIKE.Kg}$, computes $c_{j,u} \leftarrow \mathsf{SE.Enc}(\mathsf{dk}^u_j, \mathsf{sk}_j)$ for all u in the sibling resolution, where $\mathsf{dk}^u_j \leftarrow \mathsf{NIKE.SharedKey}(\mathsf{sk}_{j-1}, \mathsf{pk}^u_{j-1})$, and stores $(\mathsf{pk}_j, \{c_{j,u}\})$ at v_j.

We proceed via a hybrid argument over nodes $v_0, \ldots, v_d$.

- *Real world*: Game G_0 corresponds to ki-DCGKA conditioned on challenge bit $b = 1$.
- For $j \in [1, d]$:
  - *NIKE hybrid*: Game G_{2j-1} is identical to game G_{2j-2}, except that all dk^u_j during query q_{up} are computed by sampling keys as random from NIKE's key space, as opposed to via $\mathsf{SharedKey}$.
  - *Encryption hybrid*: Game G_{2j} is identical to game G_{2j-1} except that all $\mathsf{SE.Enc}(dk^u_j, \mathsf{sk}_j)$ during query q_{up} are replaced by $\mathsf{SE.Enc}(dk^u_j, 0^{|\mathsf{sk}_j|})$.
- *Ideal world*: The last game from the previous step, G_{2d}, is equal to ki-DCGKA conditioned on $b = 0$, since the root secret key is independent of the rest of B.

Π.Create(st, I) :

// assumes user has called Π.Init(st, ε) first
assert $st.G = \varepsilon \wedge I = \varnothing$
pk $\leftarrow$ PKI.Access$(st.id)$
Π.create-helper$(st, id,$ pk$)$
op $\leftarrow$ "`create`"{*public-key* : pk}
$st.G$.record(op)
return $\{st : st, R :$ op$, M : \varnothing, S : \perp\}$

Π.create-helper$(st, id,$ pk$)$:

$st.root \leftarrow$ B.buildBalancedTree(2)
$st.leaf \leftarrow$ B.getFreshLeaf$(st.root)$
B.setMember$(st.leaf, id,$ pk$)$
$st.G \leftarrow$ G.create()
return

Π.Add(st, id') :

assert $id' \notin st.root$
if should-replay(st) : Π.replay(st)
pk $\leftarrow$ PKI.Access(id')
Π.add-helper$(st, id',$ pk$)$
op $\leftarrow$ "`add`"{*added-id* : id', *public-key* : pk}
$st.G$.record(op)
return $\{st, R :$ op$, M : \varnothing, S : \perp\}$

Π.add-helper$(st, id',$ pk$)$:

$f \leftarrow$ getFreshLeaf$(st.root)$
B.setMember$(f, id',$ pk$)$
if B.is-full$(st.root)$: $st.r \leftarrow$ doubleTree$(st.root)$
$p \leftarrow$ getParent$(st.leaf)$
blankPath$(p, st.root)$
return

Π.Remove(st, id') :

assert $id' \in st.root$
if should-replay(st) : Π.replay(st)
removed-keys $\leftarrow$ Π.remove-helper(st, id')
op $\leftarrow$ "`remove`"{*removed-id* : id',
removed-keys : removed-keys}
$st.G$.record(op)
return $\{st, R :$ op$, M : \varnothing, S : \perp\}$

Π.remove-helper(st, id') :

$l \leftarrow$ getMemberNode$(st.root, id')$
removed-keys $\leftarrow$ blankPath$(leaf, st.root)$
return removed-keys

Π.Update(st) :

if should-replay(st) : Π.replay(st)
version-path $\leftarrow \{\}$
removed-keys $\leftarrow \varnothing$
// update caller's personal keypair
$\mathsf{pk}'_{leaf}, \mathsf{sk}'_{leaf} \leftarrow \mathsf{Kg}(1^\lambda)$
st.sk $\leftarrow \mathsf{sk}'_{leaf}$
removed-keys $\leftarrow$ getPks$(st.leaf)$
new-version $\leftarrow (\mathsf{pk}'_{leaf}, \varepsilon)$
$st.leaf$.setVersions({new-version})
version-path$[st.leaf] \leftarrow$ {new-version}
// iteratively update direct path nodes
$v \leftarrow st.leaf$
$\mathsf{sk}_v \leftarrow st$.sk
while $v \neq st.root$:
  $p \leftarrow$ getParent(v)
  $\mathsf{pk}'_p, \mathsf{sk}'_p \leftarrow \mathsf{Kg}(1^\lambda)$
  $sib \leftarrow$ getSibling(v)
  newCts $\leftarrow \{\}$
  for $\mathsf{pk}_u \in$ getResolutionPks(sib) :
    $\mathsf{dk}'_u \leftarrow$ NIKE.SharedKey$(\mathsf{pk}_u, \mathsf{sk}_v)$
    $\mathsf{ct}_u \leftarrow$ SE.Enc$(\mathsf{dk}'_u; \mathsf{sk}'_p)$
    newCts$[\mathsf{pk}_u] \leftarrow \mathsf{ct}_u$
  removed-keys $\leftarrow$ removed-keys $\cup\, p$.getPks
  new-version $\leftarrow (\mathsf{pk}'_p,$ newCts$)$
  p.setVersions({new-version})
  version-path$[p] \leftarrow$ {new-version}
  $v \leftarrow p$
  $\mathsf{sk}_v \leftarrow \mathsf{sk}'_p$
op $\leftarrow$ "`update`"{*leaf* : $st.leaf$,
version-path : version-path,
removed-keys : removed-keys}
$st.G$.record(o)
return $\{st : st, R :$ op$, M : \varnothing, S : \mathsf{sk}_v\}$

Π.update-helper$(st,$ version-path$)$:

assert path-is-valid$(st.root,$ version-path$)$
for $(v,$ new-version$) \in$ version-path :
  updated-versions $\leftarrow \varnothing$
  // only keep versions whose public keys haven't been removed
  for $(\mathsf{pk}_v, \mathsf{cts}) \in v$.getVersions() :
    if $\mathsf{pk}_v \notin st$.removed-keys :
      updated-versions $\leftarrow$ updated-versions $\cup\, \{(\mathsf{pk}_v, \mathsf{cts})\}$
  updated-versions $\leftarrow$ updated-versions $\cup$ new-version
  setVersions$(v,$ updated-versions$)$
return

Figure 6. Full specification of the BeeKEM protocol, part 1.

We now analyze pairs of neighboring hybrids.

For NIKE Hybrids. We design $\mathcal{B}$ such that $\forall j \in [1, d]$,

$$\left| \Pr[\, \mathrm{G}_{2j-1} = 1 \,] - \Pr[\, \mathrm{G}_{2j-2} = 1 \,] \leq \mathrm{Adv}^{\text{hkr-cks}}_{\mathsf{NIKE}}(\mathcal{B}) \right|.$$

Adversary $\mathcal{B}$ runs $\mathcal{A}$ and simulates the ki-DCGKA game, but uses the challenge oracle in its own HKR-CKS [20] game to to compute every dk^u_j during query q_{up}. To do so, during q_{up}, $\mathcal{B}$ first computes $(\mathsf{pk}_j, \mathsf{sk}_j)$ as usual, and then calls the HKR-CKS registration oracle $u + 1$ times to obtain $(\mathsf{pk}^*_0, \mathsf{pk}^*_1, \ldots, \mathsf{pk}^*_u)$; key pk^*_0 is used as the public key for node v_{j-1}, and all other keys are used as the public keys of the nodes in the sibling resolution. Then, $\mathcal{B}$ calls its test oracle u times with inputs $(\mathsf{pk}^*_0, \mathsf{pk}^*_i)_{i \in \{1,\ldots,u\}}$ to obtain challenge keys $(\mathsf{ck}_1, \mathsf{ck}_2, \ldots, \mathsf{ck}_u)$; notice that all challenge keys are either fresh keys, or computed via SharedKey. Then $\mathcal{B}$ uses ck_i to compute $c^i_j \leftarrow\!\!\$\ \mathsf{SE.Enc}(\mathsf{ck}_i, \mathsf{sk}_j)$, and stores $(\mathsf{pk}_j, \{c^i_j\}_{i \in [u]})$ in v_j. Notice that $\mathcal{B}$ can compute nodes above level j and below $j - 1$ by following the BeeKEM protocol honestly.

Π.DecryptSecret(st) :

if should-replay(st) : Π.replay(st)
assert Π.has-root-key(st)
// iteratively decrypt parent subgroup secret
$v \leftarrow st.leaf$
$\mathsf{sk}_v \leftarrow st.\mathsf{sk}$
while $v \neq st.root$:
  $p \leftarrow$ getParent(v)
  if isBlank(p) $\vee$ isConflict(p) : // skip such parents
    $v \leftarrow p$
    continue
  $(\mathsf{pk}_p, \mathsf{ctMap}_p) \leftarrow$ getVersions(p)[0] // parent has one version
  $sib \leftarrow$ getSibling(v)
  for $\mathsf{pk}_u \in$ getResolutionPks(sib) :
    $\mathsf{dk}_u \leftarrow \mathsf{NIKE.SharedKey}(\mathsf{pk}_u, \mathsf{sk}_v)$
    $\mathsf{ct}_u \leftarrow \mathsf{ctMap}_p[\mathsf{pk}_u]$
    $\mathsf{sk}_p \leftarrow \mathsf{SE.Dec}(\mathsf{dk}_u; \mathsf{ct}_u)$
    if $\mathsf{sk}_p = \bot$: **continue**
  assert $\mathsf{sk}_p \neq \bot$
  $v \leftarrow p$
  $\mathsf{sk}_v \leftarrow \mathsf{sk}_p$
return sk_v

Π.Process(st, id, R, M) :

$\mathsf{op} \leftarrow R$
assert op.predecessors $\subseteq G$ // op is in causal order
if op.predecessors $\subseteq$ G.head(G) : // operation is not concurrent
  if should-replay(st) :
    Π.replay(st)
  Π.apply(st, op)
else : // operation is concurrent
  if st.pending-structural-ops :
    $st.G$.record(op)
  else if op.τ = "add" $\vee$ op.τ = "remove" :
    st.pending-structural-ops $\leftarrow$ true
    $st.G$.record(op)
  else :
    Π.apply(st, op)
$S \leftarrow \Pi$.DecryptSecret(st) // possibly $\bot$
return $\{st : st, R : \varepsilon, M : \varepsilon, S_s : S, S_r : S\}$

Π.Init(id, G) :

$st.id \leftarrow id$; $st.G \leftarrow G$
st.pending-structural-ops $\leftarrow$ false
$\mathsf{pk}, \mathsf{sk} \leftarrow \mathsf{NIKE.Kg}(1^\lambda)$
$\mathsf{PKI.Register}(id, \mathsf{pk})$
$st.\mathsf{sk} \leftarrow \mathsf{sk}$
st.retainedKeys.register(pk, sk)
Π.replay(st) // restores user state
return

Π.has-root-key(st) :

return $|st.root.\text{getVersions}()| = 1$

Π.should-replay(st) :

heads $\leftarrow$ G.head($st.G$)
return st.pending-structural-ops $\vee$ ($|$heads$| > 1$)

Π.replay(st) :

batches $\leftarrow$ topo-sort($st.G$)
for batch $\in$ batches :
  Π.apply-batch(st, batch)
st.pending-structural-ops $\leftarrow$ false
return

Π.apply-batch(st, ops) :

for op $\in$ ops :
  Π.apply(st, op)
removed-members $\leftarrow$ {op.id | op.τ = "remove", op $\in$ ops}
added-members $\leftarrow$ {op.id | op.τ = "add", op $\in$ ops}
added-members $\leftarrow$ added-members $\setminus$ removed-members
sort-added-leaves(st, added-members)
for $id \in$ removed-members :
  blankPath(st, op.id)
return

Π.apply(st, op) :

match op.τ :
"add" : Π.add-helper(st, op.id, op.pk)
"remove" : Π.remove-helper(st, op.id)
"update" : Π.update-helper(st, op.version-path)
"create" : Π.create-helper(st, op.id, op.*public-key*)
return

Figure 7. Full specification of the BeeKEM protocol, part 2.

Let us discuss how $\mathcal{B}$ simulates other queries for $\mathcal{A}$. ki-DCGKA queries that modify group membership modify tree structure, but do not require cryptographic operations. So $\mathcal{B}$ can execute the protocol honestly to simulate these. Similarly, updates that are independent of q_{up} (i.e., which did not establish or use any pk^u_{j-1}), do not rely on any dk^u_j, and so $\mathcal{B}$ can also execute the protocol honestly to simulate these. Future updates can be computed by following the protocol honestly, since $\mathcal{B}$ knows all relevant public keys in the update path of q_{up}.

Our remaining focus is thus on earlier updates from sibling resolutions. First, consider each update operation $\{q^k_{up}\}_{k\in[u]}$ that preceded q_{up} and established pk_k. A problem for our reduction approach above is that the pk^*_k that $\mathcal{B}$ obtained from its game needs to be consistent with the public key for the j-th node generated earlier during q^k_{up}. Fortunately, by definition, the sibling of the updating leaf in q^k_{up} is the *leaf in the challenged* q_{up}, so all keys in the co-path of the updating nodes of q^k_{up} are available to $\mathcal{B}$. Then, by NIKE symmetry, $\mathcal{B}$ can compute dk using the *secret key of the sibling*: in q^k_{up}, computing dk as $\mathsf{SharedKey}(\mathsf{sk}_{j-1}, \mathsf{pk}^*_k)$ instead of $\mathsf{SharedKey}(\mathsf{sk}^*_k, \mathsf{pk}_{j-1})$.

For Encryption Hybrids. We design $\mathcal{C}$ s.t. $\forall j \in [1, d]$,

$$\left| \Pr[\, \mathrm{G}_{2j} = 1 \,] - \Pr[\, \mathrm{G}_{2j-1} = 1 \,] \leq \mathrm{Adv}^{\text{mu-cpa}}_{\mathsf{SE}}(\mathcal{C}) \right|.$$

Adversary $\mathcal{C}$ runs $\mathcal{A}$ and simulates the ki-DCGKA game, but uses the challenge oracle in its own mu-CPA game to com-

ki-DCGKA$^{\mathcal{A}}_{\Pi[\mathsf{DGM}],P}$:

for $id \in I$: $\mathsf{states}[id] \leftarrow \Pi.\mathsf{Init}(id)$
$\mathsf{ctr}[\cdot] \leftarrow 0$
$\mathsf{ctrM}[\cdot,\cdot], \mathsf{dirM}[\cdot,\cdot], \mathsf{scrts}[\cdot,\cdot], \mathsf{addTrgt}[\cdot,\cdot] \leftarrow \varepsilon$
$\mathsf{needsRes}[\cdot,\cdot], \mathsf{chal}[\cdot,\cdot], \mathsf{dlvd}[\cdot,\cdot,\cdot] \leftarrow \mathsf{false}$
$b \leftarrow\$ \{0,1\}$

$\mathsf{win} \leftarrow 0$
$\mathsf{queries} \leftarrow ()$
$b' \leftarrow\$ \mathcal{A}^{\mathcal{O}}(\mathsf{ctrM}, \mathsf{dirM})$
if $b' = b$: $\mathsf{win} \leftarrow 1$
if $P(\mathsf{queries}) = 0$: $\mathsf{win} \leftarrow 0$
return win

REVEAL(id, c) :

if $\mathsf{scrts}[id,c] = \varepsilon$: **return** $\perp$
if $\mathsf{chal}[id,c] \neq \mathsf{false}$: **return** $\perp$
$\mathsf{chal}[id,c] \leftarrow \mathsf{true}$
return $\mathsf{scrts}[id,c]$

CHALLENGE(id, c) :

if $\mathsf{scrts}[id,c] = \varepsilon$: **return** $\perp$
if $\mathsf{chal}[id,c] \neq \mathsf{false}$: **return** $\perp$
$S_1 \leftarrow \mathsf{scrts}[id,c]$; $S_0 \leftarrow\$ \{0,1\}^{|S_0|}$
$\mathsf{chal}[id,c] \leftarrow \mathsf{true}$
return S_b

COMPROMISE(id) :

return $\mathsf{states}[id]$

CREATEGROUP(id_0, I) :

if $\mathsf{ctrM}[.,.] \neq \varepsilon$: **return** $\perp$
if $id_0 \in I$: **return** $\perp$
$(st, R, M, S) \leftarrow \mathsf{Create}(\mathsf{states}[id_0], I)$
if $R = \varepsilon \ \vee S = \varepsilon$: $\mathsf{win} \leftarrow 1$
$\mathsf{states}[id_0] \leftarrow st; \mathsf{scrts}[id_0, 1] \leftarrow S$
$\mathsf{ctrM}[id_0, 1] \leftarrow R; \mathsf{dirM}[id_0, 1] \leftarrow M$
$\mathsf{ctr}[id_0] \leftarrow 1$
$\mathsf{needsRes}[id_0, 1] \leftarrow \mathsf{true}$
$\mathsf{dlvd}[id_0, 1, id_0] \leftarrow \mathsf{true}$

MODUSER(id, id', Op) :

if $\mathsf{Op} \notin \{\mathsf{Add}, \mathsf{Remove}\}$: **return** $\perp$
if $\neg\mathsf{valid\text{-}member}(id) \vee id = id'$: **return** $\perp$
$\mathsf{ctr}[id] \leftarrow \mathsf{ctr}[id] + 1; c \leftarrow \mathsf{ctr}[id]$
$(st, R, M, S) \leftarrow \mathsf{Op}(\mathsf{states}[id], id')$
if $R = \varepsilon \ \vee S = \varepsilon$: $\mathsf{win} \leftarrow 1$
$\mathsf{states}[id] \leftarrow st; \mathsf{scrts}[id, c] \leftarrow S$
$\mathsf{ctrM}[id, c] \leftarrow R; \mathsf{dirM}[id, c] \leftarrow M$
if $\mathsf{Op} = \mathsf{Add}$: $\mathsf{addTrgt}[id, c] \leftarrow id'$
else : $\mathsf{needsRes}[id, c] \leftarrow \mathsf{true}$
$\mathsf{dlvd}[id, c, id] \leftarrow \mathsf{true}$

SENDUPDATE(id) :

if $\neg\mathsf{valid\text{-}member}(id)$: **return** $\perp$
$\mathsf{ctr}[id] \leftarrow \mathsf{ctr}[id] + 1; c \leftarrow \mathsf{ctr}[id]$
$(st', R, M, S) \leftarrow \mathsf{Update}(\mathsf{states}[id])$
if $R = \varepsilon \ \vee S = \varepsilon$: $\mathsf{win} \leftarrow 1$
$\mathsf{states}[id] \leftarrow st; \mathsf{scrts}[id, c] \leftarrow S$
$\mathsf{ctrM}[id, c] \leftarrow R; \mathsf{dirM}[id, c] \leftarrow M$
$\mathsf{needsRes}[id, c] \leftarrow \mathsf{true}$
$\mathsf{dlvd}[id, c, id] \leftarrow \mathsf{true}$

DELIVER(id, c, id') :

if $\mathsf{in\text{-}history}(id, c, id')$: **return** $\perp$
if $\neg\mathsf{should\text{-}receive}(id, c, id')$: **return** $\perp$
if $\neg\mathsf{casually\text{-}ready}(id, c, id')$
$\quad \wedge\mathsf{add\text{-}ready}(id, c, id')$: **return** $\perp$
if $\mathsf{ctrM}[id, c] = \varepsilon$: **return** $\perp$
$(st, R, M, S_s, S_r) \leftarrow$
$\quad \mathsf{Process}(\mathsf{states}[id'], id, \mathsf{ctrM}[id, c], \mathsf{ctrM}[id, c, id'])$
$\mathsf{states}[id'] \leftarrow st$
if $\mathsf{should\text{-}decrypt}(id, c, id')$:
$\quad$ **if** $S_s \neq \mathsf{scrts}[id, c]$: $\mathsf{win} \leftarrow 1$
else if $S_s \neq \varepsilon$: $\mathsf{win} \leftarrow 1$
$\mathsf{mustRes} \leftarrow \big(\mathsf{needsRes}[id, c] \wedge$
$\quad \mathsf{should\text{-}decrypt}(id, c, id')\big) \vee \mathsf{adds\text{-}member}(id, c, id')$
if $\mathsf{mustRes} \wedge (R = \varepsilon \vee S_r = \varepsilon)$: $\mathsf{win} \leftarrow 1$
if $R \neq \varepsilon$:
$\quad \mathsf{ctr}[id'] \leftarrow \mathsf{ctr}[id'] + 1; c' \leftarrow \mathsf{ctr}[id']$
$\quad \mathsf{ctrM}[id', c'] \leftarrow R; \mathsf{dirM}[id', c'] \leftarrow M$
$\quad \mathsf{scrts}[id', c'] \leftarrow S_r; \mathsf{dlvd}[id', c', id'] \leftarrow \mathsf{true}$
$\mathsf{dlvd}[id, c, id'] \leftarrow \mathsf{true}$

Figure 8. Key indistinguishability security for DCGKA schemes with respect to corrupted predicate P. Adversary $\mathcal{A}$ has access to oracles $\mathcal{O} = \{$REVEAL, CHALLENGE, COMPROMISE, CREATEGROUP, MODUSER, SENDUPDATE, DELIVER$\}$.

pute every $\mathsf{SE.Enc}(\mathsf{dk}_j^u, \mathsf{sk}_j)$ during query q_{up}. Concretely, $\mathcal{C}$ calls its key generation oracle u times, and then calls its challenge oracle (once per key index) with challenge messages sk_j and $0^{|\mathsf{sk}_j|}$. Finally, it embeds the resulting challenge ciphertext as the ciphertext in v_j. Since dk_j^u is now random from the previous hybrid transition, $\mathcal{B}$ simulates the correct distribution for $\mathcal{A}$.

The only place where node ciphertexts interact with other operations are future update operations that require decryption of $\mathsf{SE.Enc}(\mathsf{dk}_j^u, \mathsf{sk}_j)$. However, such decryptions will fail if sk_j is the all-zero string. So, to get around this, $\mathcal{C}$ simply maintains maintain a lookup table for subgroup secrets sk_j, returning them whenever a node decryption is performed, instead of running SE.Dec. All other operations can be simulated honestly, as they are independent of sk_j.

Combining all hybrid transitions, we thus get that

$$\mathrm{Adv}^{\text{ki-dcgka}}_{\Pi,\text{bee-safe}_k}(\mathcal{A}) \leq \lceil \log_2 n \rceil \cdot \big(\mathrm{Adv}^{\text{hkr-cks}}_{\mathsf{NIKE}}(\mathcal{B}) + \mathrm{Adv}^{\text{mu-cpa}}_{\mathsf{SE}}(\mathcal{C})\big)$$

Simulating Compromise Queries. We have not yet excluded the possibility that $\mathcal{A}$ performs a compromise query which $\mathcal{B}/\mathcal{C}$ cannot provide a convincing response for. We now argue that bee-safe (Figure 3) restricts $\mathcal{A}$ such that $\mathcal{B}$ and $\mathcal{C}$ can simulate responses to its compromises.

First consider the NIKE adversary $\mathcal{B}$. It answers some of its update oracle queries with public keys pk^* obtained from its HKR-CKS registration oracle (the corresponding

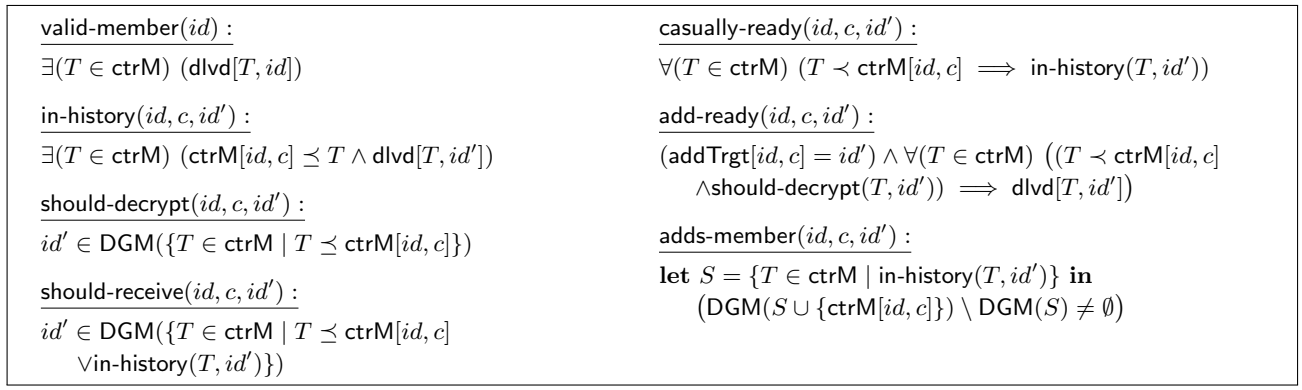

Figure 9. Helper predicates for DCGKA security game (Figure 8). Deterministic function DGM is a *decentralized group membership function* [34], which takes as input a set of control messages and their casual relationships, and outputs the set of group members resulting from the scheme.

secret keys are unknown to $\mathcal{B}$). $\mathcal{B}$ associates these pk^* with nodes which can access the group secret established by q_{up}. Suppose $\mathcal{A}$ performs a compromise query q_c which would obtain the personal secret of a leaf which is either (i) one of these HKR-CKS keys sk^* or (ii) which is able to obtain such an HKR-CKS key via Invariant 1. Such an adversary $\mathcal{A}$ would not satisfy bee-safe, as by construction, q_c obtains key material which can access the challenged secret q_{up}. Therefore, any valid $\mathcal{A}$ only performs compromise queries which $\mathcal{B}$ can faithfully simulate.

Now consider $\mathcal{C}$. $\mathcal{C}$ simulates the encryptions performed during the update q_{up} by replacing the encryptions of subgroup secrets $\mathsf{SE.Enc}(\mathsf{dk}_j, \mathsf{sk}_j)$ with encryptions from its CPA challenge oracle on challenge pairs $(\mathsf{sk}_j, 0^{|\mathsf{sk}_j|})$. Suppose that $\mathcal{A}$ performs a compromise q_c which obtains a personal secret which can access any dk_j on the updated path: the distribution of the ciphertexts, given dk_j, would be incorrect. Once again, such an $\mathcal{A}$ is excluded by bee-safe, by construction of q_c. Hence, both $\mathcal{B}$ and $\mathcal{C}$ succeed in simulating responses for any $\mathcal{A}$ valid under bee-safe. $\square$