
# Cryptographic Administration for Secure Group Messaging

David Balbás, *IMDEA Software Institute & Universidad Politécnica de Madrid;*
Daniel Collins and Serge Vaudenay, *EPFL*

https://www.usenix.org/conference/usenixsecurity23/presentation/balbas

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

# Cryptographic Administration for Secure Group Messaging

David Balbás
*IMDEA Software Institute, Spain*
*Universidad Politécnica de Madrid, Spain*
david.balbas@imdea.org

Daniel Collins
*EPFL, Lausanne, Switzerland*
daniel.collins@epfl.ch

Serge Vaudenay
*EPFL, Lausanne, Switzerland*
serge.vaudenay@epfl.ch

## Abstract

Many real-world group messaging systems delegate group administration to the application level, failing to provide formal guarantees related to group membership. Taking a cryptographic approach to group administration can prevent both implementation and protocol design pitfalls that result in a loss of confidentiality and consistency for group members.

In this work, we introduce a cryptographic framework for the design of group messaging protocols that offer strong security guarantees for group membership. To this end, we extend the continuous group key agreement (CGKA) paradigm used in the ongoing IETF MLS group messaging standardisation process and introduce the administrated CGKA (A-CGKA) primitive. Our primitive natively enables a subset of group members, the group admins, to control the addition and removal of parties and to update their own keying material in a secure manner. Notably, our security model prevents even corrupted (non-admin) members from forging messages that modify group membership. Moreover, we present two efficient and modular constructions of group administrators that are correct and secure with respect to our definitions. Finally, we propose, implement, and benchmark an efficient extension of MLS that integrates cryptographic administrators.

## 1 Introduction

In our current era of unprecedented digital communication, billions of people use instant messaging services daily. Building messaging protocols that provide security guarantees to users is a challenging task for many reasons. One of them is that protocol participants must be able to exchange messages *asynchronously* and should not be required to be online and available at all times. Besides this, they must always be ready to send or receive messages spontaneously (i.e. without additional interaction). Moreover sessions are long-lived, in contrast to protocols such as TLS, and the secrets are stored in potentially vulnerable mobile devices, and so providing security guarantees under state exposure has become standard.

Messaging protocols are designed either for two-party conversations, such as the Signal [33] and OTR [20] protocols, or for group conversations. In the group case ( [5–7, 17, 31]), modern protocols often achieve *forward security* (FS) and *post-compromise security* (PCS) [24] to protect past and future communications, respectively, upon state compromise. The most common approach for designing a group scheme with these features consists of group members running a protocol to derive a single, *common* group key that they can update on-demand. To capture the fundamental requirements of a group key agreement primitive for messaging, Alwen et al. [5] introduce the *continuous group key agreement* (CGKA) primitive, which includes support for asynchrony, dynamic groups and key ratcheting. This approach (and, at its core, the *TreeKEM* protocol [17]) has been adopted by the Messaging Layer Security (MLS) [13] work-group of the Internet Engineering Task Force (IETF). MLS and other CGKA protocols rely on a centralized *delivery service* (DS) that orders and distributes *control messages* to group members for group membership and key updates.

One of the major challenges in the design of group messaging protocols is the need to account for group *evolution* or *dynamics*: the list of group members may change at any point in time, requiring complex key agreement protocols. As a baseline for ensuring practical security guarantees, formal security proofs in a realistic adversarial model are essential, especially in a complex setting like group messaging.

### 1.1 Group Administration

Group messaging protocols require careful handling of group membership, particularly to prevent membership changes from reducing the confidentiality of previously sent messages. Overall, securing group membership involves three main aspects: (1) *key updates*, ensuring that new members cannot read past messages and removed members cannot read future messages; (2) *membership consistency*, ensuring that all members faithfully know the list of members at any time; and (3) *securing control messages* (i.e., notifications for member

addition and removal operations) from active adversaries and from the delivery service itself.

Many state-of-the-art protocols, including passively-secure CGKAs [5, 31], Sender Keys (WhatsApp, Signal Messenger), [40] and Matrix [29], include cryptographic mechanisms for securing key updates, but provide weaker and sometimes even no guarantees for securing membership consistency and control messages. We identify membership consistency as both a correctness and a security property that is critical for confidentiality (otherwise, the sender of a message may not know the receivers) but is often ignored in the literature. Failing to secure control messages can also result in catastrophic attacks. Practical examples include the *burgle into a group attack* [35], which exploits the lack of authentication of control messages to allow an adversary with partial control over the central server to enter arbitrary group chats in Signal and WhatsApp. Recent attacks on the Matrix protocol [1] make use of similar vulnerabilities, enabling the server to take over the control of a group.

In order to secure group membership, we observe that there is a strong trend in practice to distinguish between at least two types of group members: administrators (admins) and standard users. In groups with administrators, all group changes are either performed or approved by the admins. Therefore, we address the problem of secure group management by developing a cryptographic framework for *group administration*.

**Administration in messaging apps.** Generally, an admin has all the capabilities of a standard user plus a set of administrative rights. In practice, admins are implemented at the application level via policies enforced either by the central server or users. Examples are the popular messaging apps Signal, Telegram and WhatsApp (as of 2022).

- In WhatsApp, only admins can add and remove users, create a group invite link, and govern the admin subgroup. All groups must have at least one admin; when the last admin leaves, a user is selected randomly as the new admin.
- In Telegram, the group creator can designate other admins with diverse sets of capabilities. Besides adding and removing users, admins can impose partial bans on any user's capabilities, such as sending or receiving messages, and can even restrict the content that users can send [37].
- In Signal Messenger, admins can specify whether all members or only admins can add and remove users from a group (in the latter case non-admins can *request* to add users) and create a group invite link.

Despite administration mechanisms being widely deployed, there is little mention of admins in the literature. Existing CGKA and group messaging approaches make no formal distinction between admin and non-admin users, which results in giving admin capabilities to all users.

**Security goals.** There are four main security goals that our cryptographic administrators aim to achieve. In groups where no distinction is made between admins and standard members, our solutions can be extended to the whole group by treating all members as admins; these goals nonetheless still apply:

- Reduce trust on the delivery service, such that it has no control over group administration and membership.
- Mitigate the impact of insider attacks [9, 30] on protocol execution. Insider adversaries, or compromised members, cannot control a group unless they are administrators[1].
- Increase the resilience of implementations of messaging protocols, preventing pitfalls such as the *burgle into a group attack* [35] or the recent attacks on Matrix [1].
- Reduce concurrency issues, especially when the delivery service is not a central server [38], since only a reduced set of members are able to commit group changes.

**Admin capabilities.** Let $G = \{\mathsf{ID}_1, \ldots, \mathsf{ID}_n\}$ be a group of users participating in messaging or continuous key exchange and $G^* \subseteq G$ be a non-empty subset of group administrators. Unlike regular group members, the administrators $\mathsf{ID} \in G^*$ that we consider can: (1) add and remove members from the group, (2) approve/reject join and removal requests, (3) designate other administrators, (4) give up their admin status, (5) remove the admin status of other users. For performance and security, regular group members should be able to remove themselves and make key updates without admin approval.

These correspond to the common administration features among the solutions above. For Telegram, their "fine-grained" administration is practical as their central server decrypts all messages, which is incompatible with our schemes.

## 1.2 Contributions

In this work, we cast group administration as a formal *cryptographic* problem. The complexity of secure messaging requires modular constructions and proofs of security, which are our main goal. Our core contributions are as follows.

1. We introduce the *administrated* CGKA (A-CGKA) primitive in Section 3 by extending the continuous group key agreement (CGKA) primitive. We embed A-CGKA with a game-based correctness notion which emphasises the role of *group dynamics*.
2. Extending existing CGKA key indistinguishability security notions, we introduce a game-based security notion (Section 3.3) which further aims to prevent even fully corrupted non-admin users from modifying group membership.
3. We present two A-CGKA constructions, IAS and DGS, each built on top of a CGKA protocol. Each approach provides different security and efficiency properties. We

[1] Note that denial-of-service attacks from malicious non-admin insiders as in [7] are not necessarily prevented; we also remark that this family of attacks does not affect confidentiality. This issue is discussed in later sections.

formalise both protocols in detail in Section 4, analyze their performance in Section 6.1, and prove correctness and security (Section 5.1).

4. We propose an extension to MLS in Section 4.3 that provides efficient secure administration that we also implement and benchmark locally (Section 5.2).
5. We consider additional administration mechanisms in Section 6.3 and discuss their possible implementation.

### 1.3 Overview

**From CGKA to A-CGKA.** Inspired by newer versions of the MLS draft standard, CGKA has been increasingly formalised in the so-called *propose and commit* paradigm [6, 7, 9]. In CGKA, each user maintains a state which is input to and updated by local CGKA algorithms. Users in a given group can create proposal messages to propose to add or remove users, or to update their keying material for PCS reasons. Proposals are then combined by a member to form a *commit* message. This is then *processed* by users which make the committed changes effective.

We extend CGKA to A-CGKA to support administration on the primitive level. We support additional proposal types, namely for adding and removing admins, as well as for admin key updates. Users only process group changes that have been attested by an admin. Our correctness notion for both CGKA and A-CGKA enforces (A)-CGKA semantics and ensures that users who process the same control messages have consistent views of group and key evolution.

**Security.** Our security notion captures two core guarantees. Firstly, like previous work [5, 31], we consider a key indistinguishability game where the adversary drives CGKA execution via oracles and may compromise parties. We prevent the adversary from winning the game trivially in so-called cleanness predicates, which are protocol-dependent, similar to previous work [5, 28].

Secondly, differing from standalone CGKA, we require that the adversary is unable to forge an (admin) commit message that results in a change in group structure for the processing party, even if the adversary knows the group key. Security is ensured insofar as the adversary does not compromise an administrator then trivially forge a control message, i.e., they are permitted to compromise many non-admins. Our security notion allows for FS and PCS guarantees with respect to the admin keying material.

**Constructions.** We provide two modular constructions of A-CGKA from CGKA, as well as an extension of MLS that supports administration. We describe them in Section 4. In our first construction, individual admin signatures (IAS), admins keep track of their own signature key pair. Admin proposals and commits which change the group structure or admin structure or keys are signed using the committing admin's signature key. Admins update their signature keys via admin update proposals or by crafting commit messages.

Our second construction, dynamic group signature (DGS), relies on a secondary CGKA and authenticates a group of admins as a whole (possibly all group members). Instead of maintaining individual signatures, admins execute within this CGKA and use the common secret to derive a signature key pair for each epoch. Non-admins keep track of the signature public key and verify that commits are signed using it.

Finally, we propose an extension of MLS that admits secure administration. We embed the MLS protocol with A-CGKA functionality more organically by leveraging its credential infrastructure. Moreover, we implement and benchmark the efficiency of our MLS extension (and include a reference to the source code); we present our results in Section 5.

**Proofs.** We present the security theorems for IAS and DGS under our A-CGKA definitions in Section 5, where we only include proof sketches; we provide all proofs for correctness and security in the full version [11].

### 1.4 Additional Related Work

Many works in the two-party messaging literature laid the foundations for modern group messaging protocols, especially regarding FS and PCS. Initial work includes OTR [20] and Signal [33] (the latter being formalized in [4, 23]).

The TreeKEM protocol in MLS [17], was inspired by Asynchronous Ratchet Trees [25]. Later, variants arose like Tainted [31], Insider-Secure [9], Re-randomized [5], and Causal [39] TreeKEM. MLS is studied in [6, 21].

CGKAs have been recently used to formally build full group messaging protocols [6]. Besides TreeKEM, CGKA variants include [3, 7, 8, 38]. Side works deal with multi-group security [26], efficient key schedules for multiple groups [2], and concurrency [19]. [34] surveys group key exchange protocols. Group admins were considered in [35], although without a formal cryptographic approach. An alternative approach towards securing group membership was taken in the Signal Private Group System [22] which we discuss in Section 6.2.

The full version of this work is available in [11]. An earlier version containing some preliminary results appears in [10].

## 2 Notation

A *user*, *participant*, or *party* is an entity that takes part in a protocol. Users (resp. groups) are identified by a unique, public identifier $\mathsf{ID}$ (resp. gid). Users keep an internal *state* $\gamma$ with all information used for protocol execution. If $\gamma$ is leaked, we say $\mathsf{ID}$ suffers a *state compromise* or a *corruption*.

To assign the output of an algorithm $\mathsf{Alg}$ on input $x$ to variable $a$, we write $a \leftarrow \mathsf{Alg}(x)$, and $a \leftarrow\!\!\$\, \mathsf{Alg}(x)$ for randomized algorithms (to make the randomness $r$ explicit, we write $a \leftarrow \mathsf{Alg}(x;r)$). Blank values are denoted by $\perp$.

We let $\lambda \in \mathbb{N}$ be the security parameter. In our security games, the predicate '**require** $P$' enforces that a logical condition $P$ is satisfied; otherwise the oracle/algorithm aborts and returns $\perp$. The game predicate '**reward** $P$' is such that if $P$ holds, the adversary wins the game. The keyword '**public** var' indicates that the adversary has read access to variable var.

To store and retrieve values, we often use dictionaries: $\mathsf{A}[k] \leftarrow a$ adds the value $a$ to the dictionary $\mathsf{A}$ under key $k$, overwriting if necessary. $b \leftarrow \mathsf{A}[k]$ retrieves $\mathsf{A}[k]$ and assigns it to variable $b$. A dictionary $\mathsf{A}$ is initialized as $\mathsf{A}[\cdot] \leftarrow a$; where all values are set to $a$. The use of the prefix operator $\mathsf{Alg}(\texttt{++x})$, is equivalent to writing first $x \leftarrow x+1$ and then $\mathsf{Alg}(\mathsf{x})$.

# 3 (Administrated) Continuous Group Key Agreement

In this section, we introduce the CGKA and A-CGKA primitives, and the corresponding correctness and security notions.

## 3.1 Continuous Group Key Agreement

The aim of the Continuous Group Key Agreement (CGKA) primitive [5] is to provide shared secrets (denoted by $k$) to dynamic groups of users over time. In CGKA, each group, labelled with a group identifier gid, is subject to additions (add), removals (rem), and user state refreshes/key updates (upd). The evolution of a CGKA in time is captured by *epochs*; a member advances an epoch every time they successfully process a commit message, at which point there is a change in group structure and/or shared secret from their view.

We define CGKA below. Note that the primitive is *stateful*: each user keeps their own state $\gamma$ and calls each of the following algorithms locally which may update the state.

**Definition 1.** *A continuous group key agreement (CGKA) scheme is a tuple of algorithms* $\mathsf{CGKA} = (\mathsf{init}, \mathsf{create}, \mathsf{prop}, \mathsf{commit}, \mathsf{proc}, \mathsf{prop\text{-}info}, \mathsf{props})$ *such that:*

- $\gamma \leftarrow\!\!\$\ \mathsf{init}(1^\lambda, \mathsf{ID})$ *takes a security parameter* $1^\lambda$ *and an identity* ID *and outputs an initial state* $\gamma$*.*
- $(\gamma', T) \leftarrow\!\!\$\ \mathsf{create}(\gamma, \mathsf{gid}, G)$ *takes a state* $\gamma$*, a group identifier* gid*, and a list of group members* $G = \{\mathsf{ID}_1, \ldots, \mathsf{ID}_n\}$ *and outputs a new state* $\gamma'$ *and a control (welcome) message* $T$*, where* $T = \perp$ *indicates failure.*
- $(\gamma', P) \leftarrow\!\!\$\ \mathsf{prop}(\gamma, \mathsf{gid}, \mathsf{ID}, \mathsf{type})$ *takes a state, a group identifier, an* ID*, and a proposal type* $\mathsf{type} \in \mathsf{types} = \{\mathsf{add}, \mathsf{rem}, \mathsf{upd}\}$*, and outputs a new state* $\gamma'$ *and a proposal message* $P$*, where* $P = \perp$ *indicates failure.*
- $(\gamma', T, k) \leftarrow\!\!\$\ \mathsf{commit}(\gamma, \mathsf{gid}, \vec{P})$ *takes a state, a group identifier, and a vector of proposals* $\vec{P}$*, and outputs a new state* $\gamma'$*, a control message* $T$ *where* $T = \perp$ *indicates failure, and the new group secret* $k$*.*
- $(\gamma', \mathsf{acc}) \leftarrow \mathsf{proc}(\gamma, T)$ *takes a state and a control message* $T$*, and outputs a new state* $\gamma'$ *and an acceptance bit* acc*, where* $\mathsf{acc} = \mathsf{false}$ *indicates failure.*
- $(\mathsf{gid}, \mathsf{type}, \mathsf{ID}, \mathsf{ID}') \leftarrow \mathsf{prop\text{-}info}(\gamma, P)$ *takes a state and a proposal* $P$*, and outputs the group identifier of the proposal* gid*, its type* type*, the* ID *of the user* affected *by the proposal and the proposal creator* $\mathsf{ID}'$*.*
- $\vec{P} \leftarrow \mathsf{props}(\gamma, T)$ *takes a state and control message* $T$ *and outputs the vector of proposals* $\vec{P}$ *associated with* $T$*, where* $\vec{P} = \perp$ *indicates failure.*

*Finally, given* ID*'s state* $\gamma$ *and* gid*, the (possibly empty) set of group members in* gid *from* ID*'s perspective is stored as* $\gamma[\mathsf{gid}].G$*, and the group secret* $k$ *for* gid *is* $\gamma[\mathsf{gid}].k$*.*

**Protocol execution.** Once every user has initialized their state using init, a group is created when a party calls create on a list of IDs. Users can also authenticate and register keys on a PKI when appropriate in init, as in [5, 7, 31]. In Section 4.1, we expand on the use of PKI. The create algorithm outputs a control message $T$ that must be processed by prospective group members, including the creator, to join group gid.

In our formalism, any user can propose a member addition (add), member removal (rem) or key update (upd, only available for the caller) at any time. This is done via the prop method, which outputs a proposal message $P$. Proposals encode the information needed to make a change in the group structure or keying material, but the encoded changes are not immediately applied to the group. We emphasise that only the caller of prop can use argument $\mathsf{type} = \mathsf{upd}$ to propose to update (i.e., refresh) their keying material, in which case the input ID is ignored. Following [6], we define prop-info which outputs proposal attributes to support possibly encrypted proposals (e.g., as in MLSCiphertext [13]).

Proposed changes become effective once a user commits a (possibly empty) vector of proposals $\vec{P} = (P_1, \ldots, P_m)$ using commit, which outputs the new group key $k$ and a control message $T$ that contains the information needed by all current and incoming group members to process the changes. Typically, the commit algorithm also updates the keying material of the caller. Control messages are processed via proc, which updates the caller's state and outputs a success bit acc. Users can also parse the proposals encoded in a commit message $T$ as needed using props.

**Commit semantics.** We assume that proposals input to commit are processed in some deterministic, publicly-known, a priori determined order, that we call the *policy*. It is possible to define an explicit policy algorithm that defines this order.

**Alternative definitions.** In previous CGKA definitions [5, 31], including old versions of MLS, group changes are made effective directly by processing proposals, without committing them first, and multiple groups were not considered. Other minor differences include the speculative execution of

operations in [31] and the fact that groups are initially of size 1 in [6, 9].

## 3.2 Administrated CGKA

An administrated continuous group key agreement (A-CGKA) is a CGKA where only a group $G^*$ of ID's, the so-called *group administrators*, can commit (and therefore make effective) changes to the group structure, such as adding and removing users. As with the group of users $G$ in both CGKA and A-CGKA, the group of administrators $G^*$ is dynamic.

**Definition 2.** *An administrated continuous group key agreement (A-CGKA) scheme is a tuple of algorithms* $\mathsf{A\text{-}CGKA} = (\mathsf{init}, \mathsf{create}, \mathsf{prop}, \mathsf{commit}, \mathsf{proc}, \mathsf{prop\text{-}info}, \mathsf{props})$ *such that:*

- *Algorithms* $\mathsf{init}, \mathsf{proc}, \mathsf{prop\text{-}info}, \mathsf{props}$ *are defined as for a* CGKA *(Definition 1).*
- *In* prop *and* prop-info, types *is redefined as* $\mathsf{types} = \{\mathsf{add}, \mathsf{rem}, \mathsf{upd}, \mathsf{add\text{-}adm}, \mathsf{rem\text{-}adm}, \mathsf{upd\text{-}adm}\}$.
- $(\gamma', T) \leftarrow\!\!\$\ \mathsf{create}(\gamma, \mathsf{gid}, G, G^*)$ *additionally takes a group of admins* $G^*$.
- $(\gamma', T, k) \leftarrow\!\!\$\ \mathsf{commit}(\gamma, \mathsf{gid}, \vec{P}, \mathsf{com\text{-}type})$ *additionally takes a commit type* $\mathsf{com\text{-}type} \in \mathsf{com\text{-}types} = \{\mathsf{std}, \mathsf{adm}, \mathsf{both}\}$.

*Given* ID*'s state* $\gamma$ *and* gid, $\gamma[\mathsf{gid}].G$ *and* $\gamma[\mathsf{gid}].k$ *are defined as in Definition 1, and* $\gamma[\mathsf{gid}].G^*$ *stores the set of admins in* gid *from* ID*'s perspective.*

The execution of an A-CGKA is analogous to CGKA. Besides the introduction of the group of admins, we introduce three additional proposal types add-adm, rem-adm, and upd-adm for admin additions, removals and key updates, respectively. The commit type com-type specifies the scope of a commit operation, that is, whether it affects the general group (std), the administration of the group (adm), or both at the same time (both). For the latter, a simple example is when an admin is both adding a member (group modification) and refreshing its admin keys (admin modification).

We note that the create algorithm enforces the condition $\emptyset \subset G^* \subseteq G$; our correctness and security notions ensure this holds throughout execution. Thus, the group administrators are always a subset of the group members. We take this approach following previous CGKAs [5, 7, 31] and group messaging protocols [6, 13] where only group members can perform commits or make changes in the group.

**Real-world administrators.** A-CGKA captures the main admin features in commercial applications such as WhatsApp and Signal as mentioned in the introduction. We remark that the fact that non-admins are not allowed to make changes (except for leaving a group) is a desired consequence of our formulation of A-CGKA. A more fine-grained solution, at the expense of additional A-CGKA formalism, is to allow admins to send a policy change proposal, to e.g. modify the ability of all members to call commit to add new users. We briefly discuss formalising administration outside of the CGKA framework in Section 6.

**Correctness.** Due to their similarity, we define the correctness of CGKA and A-CGKA together. We relegate the game and the full description to the full version of this paper [11]. The main properties captured by our correctness notion are the following. First, *view consistency* ensures that all users which process the same sequence of commit messages have the same group view (i.e., $G$, $G^*$ and key $k$) in each group gid. Second, *message processing* enforces that the group structure ($G$/$G^*$) and $k$ can only be modified due to calls to proc. Third, *forking states* ensures that if the group is partitioned into subgroups that process different sequences of commit messages (thus leading to different group views), members in each partition keep consistent views.

Separately, we ensure that a user's state is not modified whenever a particular algorithm call fails, meaning that we require incorrect input not to affect the functionality of the protocol as observed for two-party messaging in [16].

## 3.3 Security

Any A-CGKA protocol must satisfy CGKA security (i.e. key indistinguishability) and, to capture the security of group evolution, prevent unauthorized (standard) users from deciding on changes to a group. Moreover, an A-CGKA where the adversary fully controls a standard group member is not key indistinguishable, but should nevertheless protect group membership. We define (A)-CGKA security in Definition 3.

**Definition 3** (Security of (A)-CGKA)**.** *A CGKA* CGKA *(resp. A-CGKA* A-CGKA*) is* $(t, q, \varepsilon)$*-secure w.r.t. the predicates* $\mathsf{C}_{\mathsf{cgka}}, (\mathsf{C}_{\mathsf{adm}}, \mathsf{C}_{\mathsf{forge}})$ *if, for any adversary* $\mathcal{A}$ *limited to* $q$ *oracle queries and running time* $t$*, the advantage of* $\mathcal{A}$ *in the* $\mathsf{KIND}_{\mathsf{(A)\text{-}CGKA}}$ *game (Figure 1) given by*

$$\left|\Pr[\mathsf{KIND}^{\mathcal{A}}_{\mathsf{(A)\text{-}CGKA},\mathsf{C}_{\mathsf{cgka}},(\mathsf{C}_{\mathsf{adm}},\mathsf{C}_{\mathsf{forge}})}(1^\lambda) = 1] - \frac{1}{2}\right|$$

*is bounded by* $\varepsilon$*, where the probability is taken over the choice of the challenger and adversary's random coins.*

**Overview** At its core, Figure 1 is a key indistinguishability game that captures the security of the common group secret, extending the game in [5]. It considers a partially active adversary $\mathcal{A}$ who can make forgery attempts and schedule messages but cannot totally control message delivery. Namely, the adversary can inject a control message to a specific party ID, but this message is not stored in the array T that keeps track of all honestly generated messages after proc is called. The main consequence of this is that injected proposals cannot be included into commits; nevertheless, the adversary can make commits on arbitrary proposals created via $O^{\mathsf{Prop}}$.

$\mathsf{KIND}^{\mathcal{A}}_{\text{(A)-CGKA},\mathsf{C_{cgka}},\mathsf{C_{adm}},\mathsf{C_{forge}}}(1^\lambda)$

$b \leftarrow\$ \{0,1\}$; $\mathsf{K}[\cdot],\mathsf{ST}[\cdot] \leftarrow \bot$
**public** $\mathsf{T}[\cdot],\mathsf{G}[\cdot],\mathsf{ADM}[\cdot] \leftarrow \bot$
$\mathsf{prop\text{-}ctr},\mathsf{com\text{-}ctr},\mathsf{exp\text{-}ctr} \leftarrow 0$
$\mathsf{ep}[\cdot],\mathsf{exp}[\cdot] \leftarrow (-1,-1)$; $\mathsf{C}[\cdot] \leftarrow -1$
$\mathsf{chall}[\cdot],\mathsf{forged} \leftarrow \mathsf{false}$
$\mathsf{ST}[\mathsf{ID}] \leftarrow\$ \mathsf{init}(1^\lambda,\mathsf{ID})\ \forall\ \mathsf{ID}$
$b' \leftarrow\$ \mathcal{A}^{O}(1^\lambda)$
**require** $\mathsf{C_{cgka}} \vee \mathsf{forged}$
**return** $1_{b=b'}$

$O^{\mathsf{Prop}}(\mathsf{ID},\mathsf{ID}',\mathsf{type})$

$(\gamma,P) \leftarrow\$ \mathsf{prop}(\mathsf{ST}[\mathsf{ID}],\mathsf{ID}',\mathsf{type})$
$\mathsf{T}[\mathsf{ep}[\mathsf{ID}],\texttt{'prop'},{+}{+}\mathsf{prop\text{-}ctr}] \leftarrow P$
$\mathsf{ST}[\mathsf{ID}] \leftarrow \gamma$

$O^{\mathsf{Commit}}(\mathsf{ID},(i_1,\ldots,i_k),\mathsf{com\text{-}type})$

$\vec{P} \leftarrow (\mathsf{T}[\mathsf{ep}[\mathsf{ID}],\texttt{'prop'},i])_{i=(i_1,\ldots,i_k)}$
$(\gamma,T,k) \leftarrow\$ \mathsf{commit}(\mathsf{ST}[\mathsf{ID}],\vec{P},\mathsf{com\text{-}type})$
**if** $T=\bot$ **return** / failure
$\mathsf{T}[\mathsf{ep}[\mathsf{ID}],\texttt{'com'},{+}{+}\mathsf{com\text{-}ctr}] \leftarrow (T,\mathsf{com\text{-}type})$
$\mathsf{T}[\mathsf{ep}[\mathsf{ID}],\texttt{'vec'},\mathsf{com\text{-}ctr}] \leftarrow \mathsf{props}(\mathsf{ST}[\mathsf{ID}],T)$
$(t_s,t_a) \leftarrow \mathsf{ep}[\mathsf{ID}]$
**if** $\mathsf{com\text{-}type}=\mathsf{adm}$ $t_s \leftarrow t_s-1$
$\mathsf{K}[t_s+1] \leftarrow k$; $\mathsf{ST}[\mathsf{ID}] \leftarrow \gamma$

$O^{\mathsf{Create}}(\mathsf{ID},G,G^*)$

$(\gamma,T) \leftarrow\$ \mathsf{create}(\mathsf{ST}[\mathsf{ID}],G,G^*)$
**if** $T=\bot$ **return** / failure
$\mathsf{T}[(-1,-1),\texttt{'com'},{+}{+}\mathsf{com\text{-}ctr}] \leftarrow (T,\mathsf{both})$
$\mathsf{ST}[\mathsf{ID}] \leftarrow \gamma$

$O^{\mathsf{Deliver}}(\mathsf{ID},(t_s,t_a),c)$

**require** $\mathsf{ep}[\mathsf{ID}] \in \{(t_s,t_a),(-1,-1)\}$
$(T,\mathsf{com\text{-}type}) \leftarrow \mathsf{T}[(t_s,t_a),\texttt{'com'},c]$
**if** $\mathsf{C}[(t_s,t_a)] \in \{c,-1\}$, $\mathsf{C}[(t_s,t_a)] \leftarrow c$
**else return** / bad commit for epoch
$(\gamma,\mathsf{acc}) \leftarrow \mathsf{proc}(\mathsf{ST}[\mathsf{ID}],T)$
**if** $\neg\mathsf{acc}$ **return** / failure
**if** $\mathsf{ID} \notin \gamma.G$ / ID removed
  $\mathsf{ep}[\mathsf{ID}] \leftarrow (-1,-1)$
**else** / ID in group, update dictionaries
  $\mathsf{ep}[\mathsf{ID}] \leftarrow (t_s,t_a)$
  **if** $\mathsf{com\text{-}type} \in \{\mathsf{std},\mathsf{both}\}$
    $\mathsf{K}[t_s+1] \leftarrow \gamma.\mathsf{k}$
    $\mathsf{G}[t_s+1] \leftarrow \gamma.G$
    $\mathsf{ep}[\mathsf{ID}] \leftarrow \mathsf{ep}[\mathsf{ID}]+(1,0)$
  **if** $\mathsf{com\text{-}type} \in \{\mathsf{adm},\mathsf{both}\}$
    $\mathsf{ADM}[t_a+1] \leftarrow \gamma.G^*$
    $\mathsf{ep}[\mathsf{ID}] \leftarrow \mathsf{ep}[\mathsf{ID}]+(0,1)$
$\mathsf{ST}[\mathsf{ID}] \leftarrow \gamma$

$O^{\mathsf{Reveal}}(t_s)$

**require** $(\mathsf{K}[t_s] \neq \bot) \wedge \neg\mathsf{chall}[t_s]$
$\mathsf{chall}[t_s] \leftarrow \mathsf{true}$
**return** $\mathsf{K}[t_s]$

$O^{\mathsf{Expose}}(\mathsf{ID})$

$\mathsf{exp}[\mathsf{ID},{+}{+}\mathsf{exp\text{-}ctr}] \leftarrow \mathsf{ep}[\mathsf{ID}]$
**return** $\mathsf{ST}[\mathsf{ID}]$

$O^{\mathsf{Challenge}}(t_s)$

**require** $(\mathsf{K}[t_s] \neq \bot) \wedge \neg\mathsf{chall}[t_s]$
$\mathsf{chall}[t_s] \leftarrow \mathsf{true}$
**if** $b=0$ **return** $\mathsf{K}[t_s]$
**if** $b=1$ **return** $r \leftarrow\$ \{0,1\}^\lambda$

$O^{\mathsf{Inject}}(\mathsf{ID},m,t_a)$

**require** $\mathsf{C_{adm}} \wedge (\mathsf{ep}[\mathsf{ID}]=(\cdot,t_a))$
**require** $(t_a \neq -1) \wedge (m,\cdot) \notin \mathsf{T}$
$(\gamma,\bot) \leftarrow \mathsf{proc}(\mathsf{ST}[\mathsf{ID}],m)$
**if** $\mathsf{C_{forgery}}$
  $\mathsf{forged} \leftarrow \mathsf{true}$ / successful forgery
  **return** $b$ / adversary wins
**else return** $\bot$

**Figure 1:** Key indistinguishability (KIND) security game for (single-group) (A)-CGKA, parametrized by the $\mathsf{C_{cgka}}$, $\mathsf{C_{adm}}$ and $\mathsf{C_{forge}}$ predicates. Highlighted code is executed only when considering an A-CGKA.

Informally, $\mathcal{A}$ wins if it plays a clean game where either it wins due to making key-indistinguishability challenges or manages to forge a message which, after being processed by a member, changes its view of $G, G^*$ except for some reduction in $G$ from users who elect to leave the group themselves.

**Epochs.** Messages output by successful create, commit, and prop calls are stored in T and uniquely labelled by the challenger using counters prop-ctr, com-ctr. We model group evolution using *epochs*, represented as a pair of integers $(t_s,t_a)$. The *standard epoch* $t_s$ represents the time period between two key evolutions; a different key should be derived in each $t_s$. The *administrative epoch* $t_a$ represents the time between two changes in the group administration.

Epochs advance every time a commit is processed; $t_s$ advances if the commit type com-type $\in \{\mathsf{std},\mathsf{both}\}$ and $t_a$ advances if com-type $\in \{\mathsf{adm},\mathsf{both}\}$. Group members can be in different epochs. If a participant ID is not in the group, then $\mathsf{ep}[\mathsf{ID}]=(-1,-1)$ holds.

**Challenges.** At any point in the game, the adversary can challenge with respect to a standard epoch $t_s$ by calling $O^{\mathsf{Challenge}}$. In a challenge, the adversary is given the group key $\mathsf{K}[t_s]$ if the challenger's bit is $b=0$, and a random string $r \leftarrow\$ \{0,1\}^\lambda$ if $b=1$. The adversary must try to determine the value of $b$ by outputting a guess $b'$ of $b$. A given execution is considered valid when either the *standard cleanness predicate* $\mathsf{C_{cgka}}$ (presented below) is true or the adversary makes a forgery and the *admin cleanness predicate* is true.

**Exposure mechanisms.** In order to capture group key ratcheting (for FS and PCS), the adversary can *expose* a user ID and *reveal* the group secret $k$. An exposure leaks the entire current state of ID stored in $\mathsf{ST}[\mathsf{ID}]$. We keep track of exposures in $\mathsf{exp}[\cdot]$. On the other hand, a reveal leaks the group key to the adversary on a specified epoch $t_s$ – in this case, $\mathsf{chall}[t_s]$ is set to true to prevent the adversary from challenging on $t_s$.

**Injections.** For A-CGKA, the adversary can also win the game by successfully *injecting* a forged commit. An injection can be attempted by calling $O^{\mathsf{Inject}}(\mathsf{ID},m,t_a)$, given that ID is in admin epoch $t_a$, where ID is the target group member and $m$ is the forged message. Note we require $t_a \neq -1$ since the adversary could otherwise trivially invite a new user into a new group that it controls. Forgeries can only be attempted if the *administrative predicate* $\mathsf{C_{adm}}$ is not violated. The adversary wins the game if the forgery is accepted by any group member $\mathsf{ID} \in G$ and if the *forgery predicate* $\mathsf{C_{forge}}$ that we

introduce below is true.

**Cleanness predicates.** The security game in Figure 1 is parametrized by three cleanness predicates, $\mathsf{C}_{\mathsf{cgka}}$, $\mathsf{C}_{\mathsf{adm}}$ and $\mathsf{C}_{\mathsf{forge}}$. The first predicate $\mathsf{C}_{\mathsf{cgka}}$ follows approaches like [5, 12] to parametrise the security of the common (A)-CGKA key. Namely, this predicate excludes trivial attacks on the protocol and captures its exact security (with respect to key indistinguishability), which in our case comprises forward security and post-compromise security after updates.

The predicate $\mathsf{C}_{\mathsf{adm}}$ models administration security. This predicate should be more permissive in some aspects than $\mathsf{C}_{\mathsf{cgka}}$, as injections should be permitted even if the adversary knows the state of a (standard) group member.

Finally, we characterize forgeries performed using the $O^{\mathsf{Inject}}$ oracle with respect to a predicate $\mathsf{C}_{\mathsf{forge}}$. The predicate we describe captures the fact that if admins have not been corrupted, then non-admins can only make group changes for parties that remove themselves from a group. We formalize some suitable predicates in Appendix A.

**Limitations.** Our security definition does not allow arbitrary message injections to participants. Thus, attacks on robustness are not captured by our security model. So long as non-admins are allowed to make commits, our A-CGKA schemes will only provide as much security as the underlying core CGKA: using MLS's TreeKEM, for example, a malicious non-admin can deny service by sending a malformed commit message that can be processed only by some of the users. This can be fixed at the expense of using NIZKs within TreeKEM [7, 27]. In any case, we note that confidentiality is not compromised under this family of attacks, as new users cannot be added.

If only admins are allowed to commit, then our schemes (to be introduced) are safe against this attack vector for non-strongly robust variants of TreeKEM, such as the one used in MLS [13] (up to version 16 at the time of writing). Standard users can still attain FS and in particular PCS when their update proposals are committed.

Furthermore, we do not model initial authentication (we implicitly assume an incorruptible PKI; see Section 6.2) and randomness manipulation. We also note that multi-group security can be captured rather easily. The main difference besides notation complexity is that exposing the state of a party implies a security loss in all groups that the party is a member of simultaneously.

# 4 Constructions

A first attempt at A-CGKA would simply require group members to keep a list of administrators over time. Whenever an admin wants to make a commit, it can check whether the admin-changing proposals were made by administrators, then commit them, and other users will verify admin conditions upon processing. This approach is functional but insecure in our model due to a lack of admin authentication. Namely, an adversary can easily forge a commit message and impersonate an admin. Many notions of CGKA security [5, 31] do not necessarily imply such a level of authentication.

One partial fix is to require admins to sign using a key derived from a long-term identity key. Then, security cannot be recovered if the admin is compromised once, resulting in the adversary winning the A-CGKA game too. Our constructions provide FS and PCS to admin authentication mechanisms in order to circumvent this problem.

## 4.1 Individual Admin Signatures

In our first construction, *individual admin signatures* (IAS), we build a generic and modular administration mechanism on top of an arbitrary CGKA protocol (denoted by CGKA). Each group administrator $\mathsf{ID} \in G^*$ maintains their own signature key pair $(\mathsf{ssk}, \mathsf{spk})$. Each key pair is independent from the keys used in CGKA, which is mostly used as a black-box. Group members keep track of the list of admins $G^*$ which is possibly updated upon processing a control message. Proposed changes to the group and to the administration are signed using an admin's keying material. The IAS construction is presented in Figure 2; helper functions are deferred to Appendix B.

**States.** We represent the state of a participant by $\gamma$, which is in part a dictionary of states, indexed by group identifiers i.e. $\gamma[\mathsf{gid}]$. Users further maintain a common state via $\gamma.\mathsf{s0}$ encoding the underlying CGKA state, security parameter $1^\lambda$ in $\gamma.1^\lambda$ and the user's ID in $\gamma.\mathsf{ME}$. For each group gid, users keep a separate state that encodes the list of group administrators $\gamma[\mathsf{gid}].\mathsf{adminList}$ and two administration-related signature key pairs. The state also keeps the group members as $\gamma[\mathsf{gid}].G = \gamma[\mathsf{gid}].\mathsf{s0}.G$, the admins as $\gamma[\mathsf{gid}].G^* = \gamma[\mathsf{gid}].\mathsf{adminList}[\cdot].\mathsf{ID}$, and the CGKA key as $\gamma[\mathsf{gid}].\mathsf{k} = \gamma[\mathsf{gid}].\mathsf{s0}.\mathsf{k}$.

All implemented A-CGKA algorithms, including init, are stateful as if executed by the same party and, as written, *do not explicitly return the updated local state*. Instead, they modify the state during runtime. In the event of algorithm failure, the state is not modified and appropriate failure values are output. We often omit the group identifier to simplify presentation.

**Randomness.** In our constructions, we make randomness used by protocol algorithms explicit, including sampled randomness $r_0 \in \{0,1\}^\lambda$ as input. Namely, for the input randomness $r_0$ used in any randomised method, we apply a PRF $(r_1, \ldots, r_k) \leftarrow H_k(r_0, \gamma)$ that combines the entropy of $r_0$ and the state $\gamma$. We do this to reduce the impact of randomness leakage and manipulation attacks [12], and does not interfere with the protocol otherwise.

**PKI.** IAS assumes a basic, incorruptible PKI functionality where all parties are authenticated with the PKI. The PKI provides a fresh signature public key spk for which only the party ID can retrieve the corresponding secret key ssk. This functionality is used in two different places: 1) When the

group of administrators expands; namely, when a party $\mathsf{ID}'$ crafts a group gid or makes an admin add proposal; and 2) When a non-admin user wishes to remove themselves from gid (a 'self-remove').

For these purposes, we define a getSpk algorithm, which on input $(\mathsf{ID}, \mathsf{ID}', \mathsf{gid})$ for subject ID and caller $\mathsf{ID}'$ outputs spk relevant to the context the call is made in. We also assume a method $\mathsf{getSsk}(\mathsf{spk}, \mathsf{ID}, \mathsf{gid})$ that returns the ssk associated to spk when called by ID given they uploaded it. During protocol execution, parties upload signature key pairs $(\mathsf{ssk}, \mathsf{spk})$ to the PKI via an abstract $\mathsf{registerKeys}(\mathsf{ID})$ method both in initialisation and during the two aforementioned scenarios.[2] Formally, the adversary can only call getSpk.

**Group creation.** Once the users have initialized their states, the create algorithm creates the group gid from the list of members $G$, the admin list from $G^*$, and outputs a (signed) control message $T$ for the new members in $G$. The adminList variable includes pairs of the form $(\mathsf{ID}, \mathsf{spk}_{\mathsf{ID}})$ for parties $\mathsf{ID} \in G^*$. The public signature keys are obtained via getSpk and each admin's private key can be retrieved from the PKI via getSsk while they are processing $T$, the control message that adds them to the group. The group creator directly stores such key pair as $(\gamma.\mathsf{ssk}', \gamma.\mathsf{spk}')$.

**Proposals.** Any group member can use prop to create a proposal of a non-admin type; the algorithm calls CGKA.prop in this case. Administrative proposals are restricted to admins and crafted by makeAdminProp, which includes an administrative signature in the proposal. The signature is included to prevent an (insider) adversary from forging the sender of the proposal in an attempt to impersonate an admin. Proposal creation does not have any effect on the state other than the storage of temporary keys for proposals with type $\mathsf{type} = \mathsf{upd\text{-}adm}$. In the case of an add-adm proposal to promote ID to admin status, the proposer $\gamma.\mathsf{ME}$ retrieves a public signature key spk of ID from the PKI using getSpk. In the case of an rem proposal where $\mathsf{ID} = \mathsf{ME}$ (i.e. a self-remove), the caller samples a new signature key pair, registers the public key spk with the PKI and signs their proposal. The prop-info method simply retrieves the main information of a proposal.

**Commits.** The commit algorithm, which can only be called by group administrators except when only key updates and self-removes are proposed, performs the following actions: (1) Clean the input vector of proposals $\vec{P}$, ensuring they are well-formed via propCleaner/enforcePolicy. For security reasons, we adopt the main features of the MLS policy (removing duplicates and prioritizing removals) in our construction [13], but variations can be used. In addition, we verify the legitimacy of admin proposals and that self-remove proposals are correctly signed via verifyPropSigs. For the former task, predicate $\mathsf{VALID}_P$ verifies the proposal data. Finally, we ensure users removed from $G$ are also removed from the adminList. (2) Carry out the administrative and the standard commits and produce an administrative commit message $C_A$ (which is the clean admin proposal vector), a standard CGKA commit $C_0$, and an updated adminList. (3) Generate a new (temporary) administrative signature key pair $(\gamma.\mathsf{ssk}', \gamma.\mathsf{spk}')$. (4) Produce the final control message $T$ which includes the new $\mathsf{spk}'$. The message $T$ is again split into two components: A first component $T_W$ (for welcome) includes all the required information for incoming A-CGKA members, including the new list of admins. A second component $T_C$ (for commit) contains the updating information for group members. Both components are signed together using the committer's current $\gamma.\mathsf{ssk}$.

The props method, given a commit, retrieves a corresponding list of proposals from CGKA.props and the admin commit list $C_A$.

**Processing control messages.** The proc method takes a control message $T$ as input and updates the state accordingly. The algorithm returns $\mathsf{acc} = \mathsf{true}$ if the processing succeeds, in which case the state is updated. Otherwise, the state remains the same. During an execution of proc, some checks must pass before the state is updated. For newly added users, p-Wel verifies the message signature on the adminList and attempts to process the message via the underlying CGKA.For group members, p-Com verifies the administrator signature and the signatures in the admin proposals. The state is updated if all verification succeeds; a removed user blanks their state, and temporary keys are updated if necessary. The case in which the $T$ is not signed is handled by proc directly by verifying that no changes to the group structure are made except possibly for signed (and verified) self-removals (only key updates).

**Commit and propose policies.** Our construction allows standard users to perform a commit if there are no changes in the group structure or in the administration. This is an optional design choice that does not affect security in our model (and could be reflected in a correctness predicate). Similarly, we choose to enforce that standard users cannot propose administrative changes (even if these could be later ignored by admins).

**Security mechanisms.** The security of the group administration is provided by the admin signatures; an adversary should not be able to commit changes to the group unless it compromises the state of one of the group administrators. The update mechanism provides optimal post-compromise security. On the other hand, administrative actions are undeniable and traceable both by group members and the message delivery service.

**On optimal forward security.** Note that, as defined, our construction does not satisfy forward security with respect to injection queries even if the underlying CGKA provides optimal forward security. Concretely, suppose that ID makes their last update in epoch 3, and then their state is exposed in epoch 5. Then ID can trivially forge commit messages for

[2] This abstraction is made to reduce notational complexity.

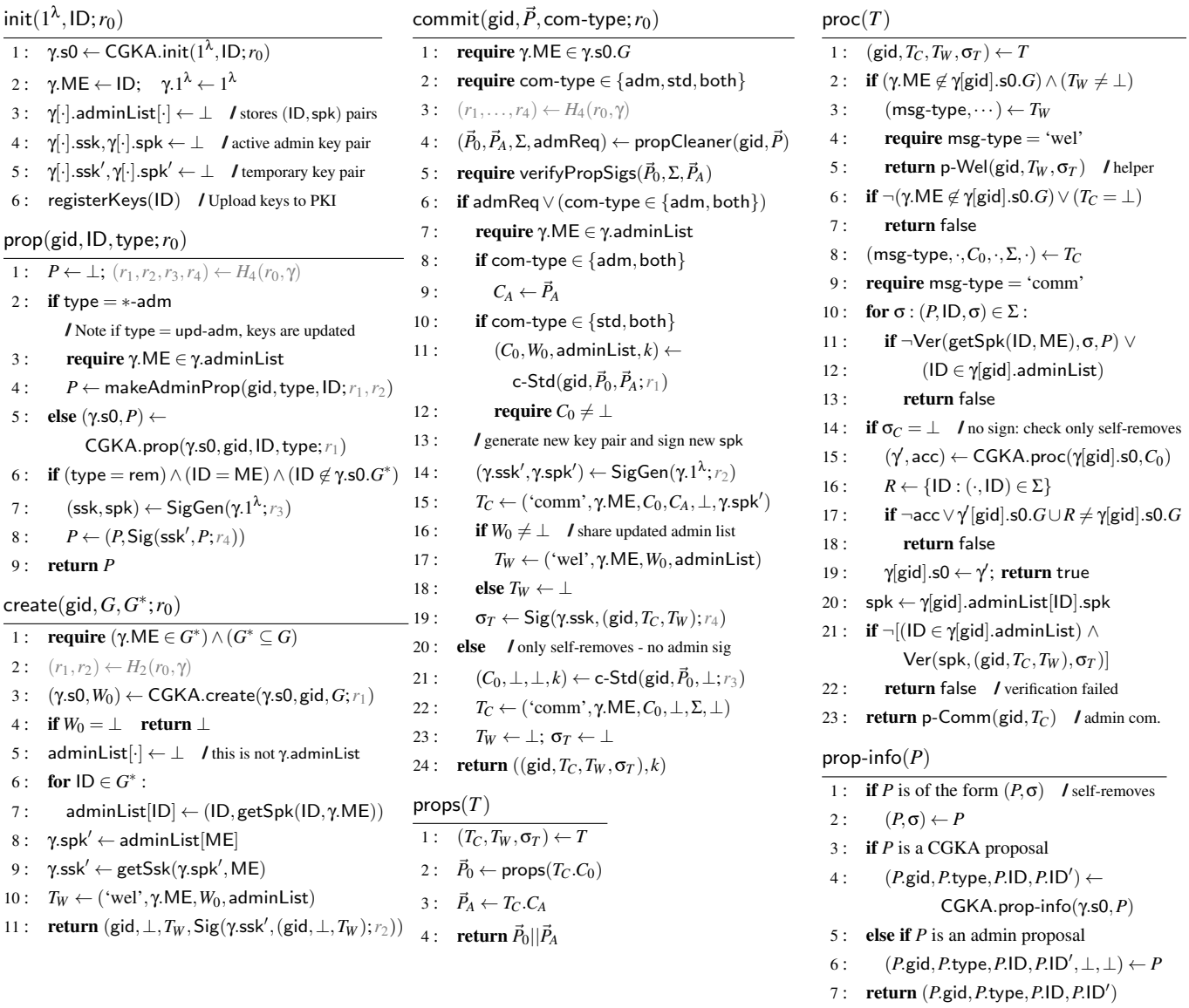


**Figure 2:** Individual admin signatures (IAS) construction of an A-CGKA, built from a CGKA, a signature scheme $\mathcal{S} = (\mathsf{SigGen}, \mathsf{Sig}, \mathsf{Ver})$, and $n$-PRFs $H_n : R \times \mathsf{ST} \to R^n$ for $n \leq 4$, randomness space $R$ and state space $\mathsf{ST}$. The state values representing the group, the admins, and the group key are assigned as: $\gamma[\mathsf{gid}].G = \gamma[\mathsf{gid}].\mathsf{s0}.G$, $\gamma[\mathsf{gid}].G^* = \gamma[\mathsf{gid}].\mathsf{adminList}[\mathsf{ID}].\mathsf{ID}$, and $\gamma[\mathsf{gid}].\mathsf{k} = \gamma[\mathsf{gid}].\mathsf{s0.k}$. We assume that $\gamma$ refers to $\gamma[\mathsf{gid}]$ whenever gid is a function parameter.

parties that are in epochs 3 and 4 since their keying material has not been updated. A similar forward security issue is present in the MLS standard affecting confidentiality [5].

Optimal security can be easily achieved by replacing regular signatures with *forward-secure signatures* [15]. Forward-secure signatures allow signers to non-interactively update their secret keys and provide forward security given state exposure. In IAS, it suffices to use forward-secure signatures such that whenever an epoch passes and an admin has not sampled a new signature key, they invoke the signature scheme's secret key update function, where new signature keys are otherwise derived as in the construction. We note that forward-secure signatures involve an overhead that may be undesirable in some cases, and also they are not used in current protocols (signatures are already used in MLS' CGKA, for instance). In Theorem 1, we characterize the exact security of IAS using standard primitives via our sub-optimal predicate. In this way, the security of both alternatives is fully characterised.

## 4.2 Dynamic Group Signature

In our second construction, *dynamic group signature* (DGS), the group administrators agree on a *common* signature key pair that they use for signing administrative messages on an underlying CGKA. To agree on a secret and generate a common key pair, they run a separate CGKA. As opposed to IAS, group administrators may now be opaque to group members if the concrete CGKA which is used allows it. Notably, group members do not need to keep track of an administrator list; admins implicitly track this via their CGKA.

**Protocol.** The DGS protocol is formally specified in the full version of this paper [11]; we summarise it below. In the algorithm, we refer to the primary (or standard) CGKA as CGKA, and to the administrative CGKA as CGKA$^*$. The first CGKA allows group members to agree on a common secret and group composition as in IAS, whereas the second exists only for administrative purposes (i.e., admins deriving a common signature key). Note that CGKA$^*$ is not necessarily implemented in the same way as the primary CGKA. This feature can be exploited by a protocol designer to provide different performance/security e.g. if stronger security is desired for CGKA$^*$. Note DGS can be extended to support self-signed removals exactly as in IAS.

**States.** Each party stores $\gamma$.s0, corresponding to the primary CGKA, as well as $\gamma$.sA, corresponding to CGKA$^*$, which are used for each group they consider. For a group gid, we assume that gid is used by the main CGKA and gid$^*$ by the admin CGKA, and we assume that $\mathsf{gid}_1$ and $\mathsf{gid}_2^*$ are distinct for all $\mathsf{gid}_1, \mathsf{gid}_2$. Besides these fields, the state includes the administrative public key $\gamma$[gid].spk known by all group members (and can be a public group parameter, known for instance by a central server) to enable verification. The state variables are now $\gamma[\mathsf{gid}].G = \gamma.\mathsf{s0}[\mathsf{gid}].G$, $\gamma[\mathsf{gid}].G^* = \gamma.\mathsf{sA}[\mathsf{gid}^*].G$, and $\gamma[\mathsf{gid}].\mathsf{k} = \gamma.\mathsf{s0}[\mathsf{gid}].\mathsf{k}$.

**PKI.** As in IAS, we assume a similarly incorruptible PKI functionality. Here, we assume that admins register their (admin) signature public keys whenever they are sampled (only upon group creation) or updated which can be obtained by users using the call $\mathsf{getSpk}(\mathsf{gid})$ (which is only required by DGS for incoming group members). As opposed to IAS, authentication could be implemented while ensuring $k$-anonymity such that a member authenticates his group membership but not his identity. The set of group administrators can be opaque to the central server and to the rest of the group (whenever the underlying CGKAs preserve the anonymity of group members with respect to external parties).

**Group creation.** The create algorithm creates a group for the two separate CGKAs by calling the corresponding two create methods. These calls output new states s0 and sA, which overwrite the stored states, as well as control messages $W_0$ and $W_A$, which are collated into a *create* control message $T = T_{CR}$. We assume that an initial group signature public key is sampled and uploaded to the PKI.

**Proposals.** The prop algorithm generates a proposal message $P$ by using CGKA.prop when the input type is standard and CGKA$^*$.prop when it is administrative ($*$-adm). As in IAS, a validity check is made using the $\mathsf{VALID}_P$ predicate. Administrative proposals are signed with $\gamma$.spk; this protects against insider adversaries that re-send old proposals or try to create new ones. The prop-info and props algorithms is fully based on the respective CGKA's prop-info algorithms.

**Commits.** For a given gid, administrative changes are committed via CGKA$^*$.commit (which outputs a $C_A$) and standard group changes via CGKA.commit (outputting $C_0$ as usual). Note that in $C_A$, we update the CGKA$^*$ secret $\mathsf{k}_{\mathsf{adm}}$, that is used to update the admin key pair $(\mathsf{ssk}', \mathsf{spk}')$.

The new admin key $\mathsf{spk}'$ is included in the final A-CGKA commit message, so that group members can process it. In order to prove the authenticity of the commit (and of $\mathsf{spk}'$), the committer signs the whole commit message including $C_A, C_0$ and $\mathsf{spk}'$ with the old admin key $\gamma$.ssk. In addition, the committer must verify all proposal signatures in advance. Commits in both CGKAs are independent: CGKA can be updated while CGKA$^*$ is not, and vice-versa.

**Processing control messages.** Newly added users verify the admin signature, process the welcome message using CGKA.proc and store the new public admin key ($\mathsf{spk}'$, provided in $T$) in $\gamma$.spk.

Group members verify the administrator signature (if the commit requires administrative rights) using $\gamma$.spk. Then, depending on the commit type at least one of CGKA and CGKA$^*$ are updated via the corresponding CGKA proc algorithm. Given CGKA$^*$ or both CGKAs are updated, the updated admin key is set as $\gamma.\mathsf{spk} \leftarrow \mathsf{spk}'$. In case $T$ contains a create message $T_{CR}$, both CGKAs process the respective welcome messages contained in $T_{CR}$ separately.

**Incoming users.** The administrative spk can be a public value that a server can store. Hence, an incoming member can verify the authenticity of an administrative signature by verifying spk with the server, or using a different channel other than the welcome message itself. Another possibility is out-of-band authentication, such as via safety numbers, a feature provided by some messaging services.

**Limitations.** A drawback of DGS is that enforcing different "levels of administration", for which IAS can be easily extended, is not straightforward. Nevertheless, one can still implement minor policies such as muting users at an application level (as done in practice). We also note that admins may not have a reliable view of the set of admins if CGKA$^*$ is susceptible to insider attacks that violate robustness, which is beyond the scope of our model that considers trusted admins. If these attacks are relevant, one can deploy heavier protocols such as P-Act-Rob in [7]. A third limitation is that admins

cannot give up their admin status immediately: they must send a self rem-adm proposal, erase their admin state, and wait for another admin to commit. In MLS, parties aware of any removals commit them before application messages are sent, ensuring speedy processing.

**Security mechanisms.** We note the conceptual simplicity of achieving PCS and FS in the group administration keys (in the adversarial model for $\mathsf{CGKA}^*$) given the existence of secure CGKA schemes in the literature, since both properties are ensured by $\mathsf{CGKA}^*$ itself. Update mechanisms are largely simplified due to a single admin key being used. Delegation and revocation of admin keys are also straightforward.

### 4.3 Integrating A-CGKA into MLS

The current MLS specification authenticates users via credentials, which are essentially public signature keys for each party that are certified by a PKI. These keys authenticate messages originating from that party. Therefore, one can extend the CGKA used in MLS to an A-CGKA in more efficiently than via a compiled A-CGKA construction. In practice, it is feasible to support secure administration in MLS via an *MLS extension*, a feature that enables additional proposal types and new actions in the protocol [13]. Constructing such an extension is almost straightforward; the three main challenges that we find are (1) update credentials to provide admins with FS and PCS, (2) add new proposal types and modify the commit and proc algorithms accordingly, and (3) maintain guarantees for incoming users by registering keys in a PKI.

**Protocol details.** The main modifications are to (1) the admins' credentials, which are regularly updated via upd-adm proposals and admin commits; and (2) in the introduction of the three additional proposal types from A-CGKA. For brevity, we omit several parts of the protocol, such as sanity checks (like **require** predicates), functionality that we do not need to modify and details on a higher level than CGKA (like the use of a MAC). Also for simplicity, we extend the CGKA state $\gamma$ to include the state variables used in IAS. Following [6], we split the processing algorithms in two – one for commit messages, and one for welcome messages. We provide a more detailed protocol description in Figure 7 in the appendices.

Overall, the overhead with respect to (bare-bones) MLS is minimal; we essentially only add the new types of proposals and refresh admin credentials for admin updates. Most of the protocol logic relates to updating signatures and the adminList. Note that proposals are always signed in MLS so signing within makeAdminProp can be foregone. We also support self-removal proposals committed by standard users.

**Correctness and security.** We leave it open to formally propose and prove correctness and security for an appropriate MLS extension; we sketch here how it could be done. The modelling of messaging and MLS in particular in [6] is more complex than ours. In particular, they consider CGKA as a sub-primitive that is used to build secure group messaging (SGM) alongside several other primitives. Thus, one could re-define SGM to account for new proposal types and administration as we have done for A-CGKA, including admin correctness guarantees and security upon injections from non-admins. Since admin proposals are tightly-coupled with protocol flow, proposing a model 'on top' of theirs to provide admin security seems infeasible, although modular guarantees would be ideal. Proving correctness and security then boils down to similar case analysis and reductions to ours.

## 5 Results

### 5.1 Security

**Theorem 1.** *Let* CGKA *be a correct and* $(t_{\mathsf{cgka}}, q, \varepsilon_{\mathsf{cgka}})$*-secure CGKA with respect to cleanness predicate* $\mathsf{C}_{\mathsf{cgka}}$*, according to Definition 3. Let* $\mathcal{S} = (\mathsf{SigGen}, \mathsf{Sig}, \mathsf{Ver})$ *be a* $(t_{\mathcal{S}}, q, \varepsilon_{\mathcal{S}})$ *SUF-CMA secure signature scheme. Let* $H_4$ *be a* $(t_F, 1, \varepsilon_F)$*-secure PRF. Then, the IAS protocol (Figures 2 and 6) is correct and* $(t, q, q \cdot \varepsilon_F + \varepsilon_{\mathsf{cgka}} + q^2 \cdot \varepsilon_{\mathcal{S}})$*-secure (Definition 3) with respect to predicates* $\mathsf{C}_{\mathsf{cgka}}, \mathsf{C}_{\mathsf{adm}}, \mathsf{C}_{\mathsf{forge}}$*, where* $t_{\mathsf{cgka}} \approx t_{\mathcal{S}} \approx t_F \approx t$*,* $\mathsf{C}_{\mathsf{adm}}$ *is defined in Figure 5 and* $\mathsf{C}_{\mathsf{forge}}$ *is defined in Appendix A.*

*Proof sketch.* We first bound an adversary $\mathcal{A}$'s advantage in distinguishing between the $\mathsf{KIND}^{\mathcal{A}}_{\mathsf{A\text{-}CGKA},\mathsf{C}_{\mathsf{cgka}},\mathsf{C}_{\mathsf{adm}}}$ game and a game $G_1$ which replaces calls to hash functions $H_i$ by uniformly sampling the output (modelling each $H_i$ in IAS as a PRF). Then, we divide $\mathcal{A}$'s behaviour in $G_1$ into two events based on whether they successfully query the $O^{\mathsf{Inject}}$ oracle (event $E_1$) or not (event $E_2$). Given $E_1$, we reduce security via a number of SUF-CMA adversaries. Otherwise, we reduce directly using a $\mathsf{KIND}^{\mathcal{A}}_{\mathsf{CGKA},\mathsf{C}_{\mathsf{cgka}}}$ adversary.

**Theorem 2.** *Let* CGKA *be a correct and* $(t_{\mathsf{cgka}}, q, \varepsilon_{\mathsf{cgka}})$*-secure CGKA with respect to* $\mathsf{C}_{\mathsf{cgka}}$*, according to Definition 3. Let* $\mathsf{CGKA}^*$ *be a correct and* $(t_{\mathsf{cgka}*}, q, \varepsilon_{\mathsf{cgka}*})$ *secure CGKA with respect to* $\mathsf{C}_{\mathsf{cgka}*}$*. Let* $\mathcal{S}$ *be a* $(t_{\mathcal{S}}, q, \varepsilon_{\mathcal{S}})$ *SUF-CMA secure signature scheme. Let* $H_4$ *be a* $(t_F, 1, \varepsilon_F)$*-secure PRF. Let* $H_{\mathsf{ro}}$ *be a random oracle queried at most* $q_{\mathsf{ro}}$ *times. Then, the DGS protocol (formalised in the full version of this paper [11]) is correct and* $(t, q, \varepsilon_{\mathsf{cgka}} + q \cdot (\varepsilon_F + \varepsilon_{\mathcal{S}} + q_{\mathsf{ro}} \cdot \varepsilon_{\mathsf{cgka}*} + 2^{-\lambda}))$*-secure (Definition 3) in the random oracle model with respect to predicates* $\mathsf{C}_{\mathsf{cgka}}, \mathsf{C}_{\mathsf{adm}}, \mathsf{C}_{\mathsf{forgery}}$*, where* $t_{\mathsf{cgka}} \approx t_{\mathsf{cgka}*} \approx t_{\mathcal{S}} \approx t_F \approx t$*,* $\mathsf{C}_{\mathsf{adm}}$ *is a function of* $\mathsf{C}_{\mathsf{cgka}*}$ *and* $\mathsf{C}_{\mathsf{forge}}{}^*$ *is defined in Appendix A.*

*Proof sketch.* $\mathsf{C}_{\mathsf{adm}}$ roughly ensures that the set of safe oracle queries for DGS adversary $\mathcal{A}$ given inject queries of the form $q_i = O^{\mathsf{Inject}}(\mathsf{ID}, m, t_a)$ are those that are safe for $\mathsf{CGKA}^*$ adversary $\mathcal{A}'$ under essentially the same queries, replacing $O^{\mathsf{Inject}}(\cdot, \cdot, t_a)$ queries with queries of the form $O^{\mathsf{Challenge}}(t_a)$. To prove security, we use a similar game-hopping argument

as in IAS. Given $E_2$, we reduce to CGKA security. Given $E_1$, we simulate depending on whether $\mathcal{A}$ makes a query to random oracle $H_{\mathsf{ro}}$ with a correct CGKA* key $k$ after injecting. Here, if $\mathcal{A}$ is successful, we reduce to CGKA* security using $k$; otherwise, we reduce to EUF-CMA security.

## 5.2 Benchmarking

We implemented the protocol in Section 4.3 to obtain a realistic estimate of the overhead of securely administrating a real-world messaging protocol. We modified an open-source implementation of MLS in Go[3] and compared the running times of MLS (which also performs e.g. parent hashing and non-admin proposal signing), with the running times of administrated MLS in different scenarios. In particular, we analyze the commit and proc algorithms in Figure 7, where the latter includes proc-CM and also processing proposals when relevant (done separately in the implementation). We ran our benchmarks on a laptop with a 4-core 11th Gen Intel i5-1135G7 processor and 16 GB of RAM using Go's `testing` package[4]. Core cryptographic operations were implemented as HPKE [14] with ciphersuite `DHKEM(P-256, HKDF-SHA256)`, `HKDF-SHA256`, `AES-128-GCM` from Go standard libraries. We measured the time taken for a *single group member* to perform the relevant operation. For each data point, we took the average over 100 iterations that randomized the group members and admins performing group operations, as performance can be affected by their position in MLS's TreeKEM.

Our results are displayed in Figure 3, where we show the running time of commit in different realistic scenarios. We run experiments where relevant, i.e., when there are no admin operations, using the original implementation as a baseline, as well as using our modified implementation, to demonstrate that the additional admin logic we introduce does not noticeably affect performance. In the appendix (Figure 8) we also show results for proc. In both cases, we vary the group size and the number of updates as described in the figures.

**Communication overhead.** In both the baseline and our implementations, proposals used 364 to 366 bytes, and admin proposals used 364 to 368 bytes (all proposals being signed). Commit message sizes in both implementations vary proportionally with the size of the group and the number of proposals. In the *baseline MLS implementation*, a typical commit for $|G| = 8$ and $|G| = 128$ with $t = 2$ and $t = 32$ update proposals uses 1.49 KB and 17.11 KB respectively. In *our implementation*, corresponding commits use 1.56 KB and 17.17 KB respectively. If $t/2$ admin updates are added (1 and 16, respectively), commits require 1.60 KB and 17.65 KB. In general, commits in our implementation, even with admin proposals, incur only a small amount of overhead (tens of bytes) over the baseline implementation when fixing the number of proposals.

[3] Our code is available at github.com/cryptographicadmins/impl. The baseline code is available at github.com/cisco/go-mls.

[4] https://pkg.go.dev/testing

# 6 Discussion

## 6.1 Efficiency

The results in Section 5.2 show that the additional cost (for users) of running a securely-administrated MLS is minimal. Figure 3 shows that the commit algorithm involves less than a 20% overhead when up to $|G|/8$ members carry out admin updates simultaneously. In Figure 8 we also show that the processing time of admin and standard updates is very similar.

Separately, we analyze the overhead of IAS and DGS for group members, assuming modular implementations of IAS and DGS not integrated with an existing CGKA as before. In IAS, admins generate a signature key pair and sign every time they commit or do a proposal, and verify a small amount $t$ of signatures (typically $t \leq |G^*|$) in admin proposals before a commit. The overhead of DGS depends heavily on the cost of CGKA* (an optimistic estimation can be $O(\log m)$ [5, 7, 31] but can be $O(m)$ in the worst case). CGKA operations only affect administrators. Note that a DGS admin-only commit is not sent to standard members (only the signed new admin key has to). Hence, DGS is very efficient for standard users.

The ratio of additional messages sent, which is application-specific, is hard to estimate. Admin-only commits and admin modifications are expected to be less frequent than standard operations. The number of update proposals (although very cheap) is expected to be at most linear in $|G|$. We conclude that IAS presents a generally affordable overhead for all users, while DGS introduces basically no cost for standard users and is more costly for administrators if $|G^*|$ is relatively large. Moreover, forward-secure signatures (for optimally-secure IAS) can be instantiated with essentially constant amortized overhead in space and time relative to a regular signature scheme, while supporting unbounded secret key updates [32]. When standard users are allowed to commit updates, the tree blanking issue with MLS TreeKEM is not worsened by administrators, hence efficiency should not decrease either.

## 6.2 On Modelling in Related Work

**CGKAs and MLS.** The CGKA abstraction has deviated from MLS and has become an object of study of its own [3, 8, 18], but in general still inherits important limitations from MLS. Among them, CGKAs rely on the availability of a well-behaved PKI, require total control message ordering, and fail to capture messaging solutions that deviate from group key agreement such as Signal or WhatsApp.

In MLS, there exists a strong architectural separation between the Delivery Service (DS, usually a central server) and a so-called Authentication Service (AS) whose design is left

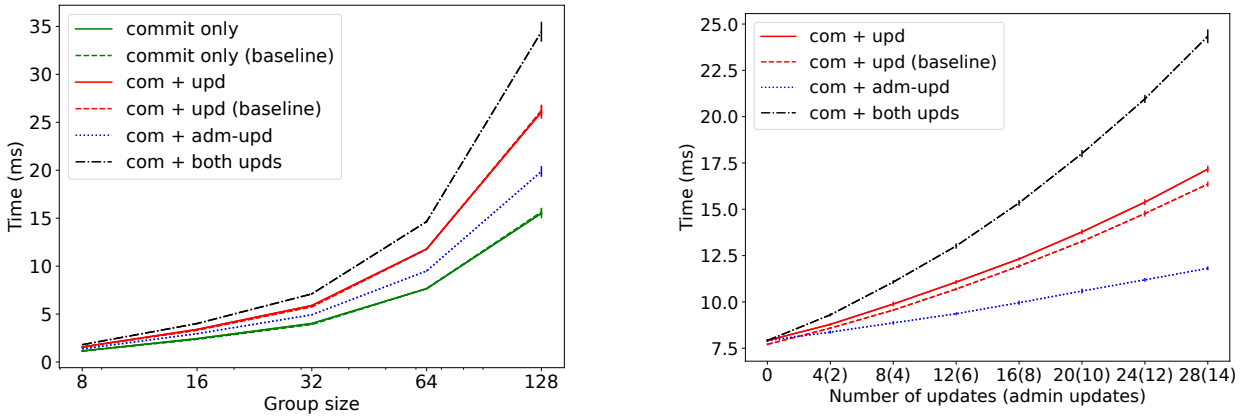


**Figure 3:** Benchmark of the commit algorithm in the following scenarios: (1) standard commits (com-type = std, omitted right), (2) standard commits with $t$ update proposals, (3) standard and admin commits (com-type = both) with $t/2$ admin-update proposals but no standard-update proposals, and (4) standard and admin commits with $t$ update and $t/2$ admin-update proposals. Original MLS is displayed as baseline. *Left:* running time with respect to group size $|G|$ on constant member/admin $|G|/|G^*| = 4$ ratio and constant number of updates $t = |G^*|$ ($t/2$ admin updates). *Right:* running time with respect to the number of updating users $t$ (and $t/2$ admin updates), for fixed $|G| = 64$.

to the infrastructure designers [13]. Following this separation, the Delivery Service is often modelled adversarially in CGKAs whereas the AS is abstracted as a PKI [5, 7, 31] as in this work. A compromised AS allows for the corruption of user credentials, resulting in user impersonation.

**PKI.** Both IAS and DGS rely on a PKI that we assume incorruptible. In IAS, parties use the PKI to verify the identity of administrators and self-removing users. In DGS, incoming users use it to retrieve the current group-wide admin signature public key. As the PKI is only used to establish an initial root of trust among parties, i.e., forward and post-compromise security are ensured without additional PKI calls, our modelling is consistent with the separation between delivery and authentication discussed above. Note that group administration aims to remove the trust in the DS (the server) but is still vulnerable to a corrupt AS. Previous CGKA work follows similar PKI abstractions [5, 31], or ignores the AS [26]. That is, in all group messaging works we are aware of, the PKI *always* behaves consistently and correctly for all users. One partial exception is [9], where malicious keys can be registered, although security degrades strongly for such users. By abstracting away the AS, our schemes are compatible with diverse authentication solutions such as out-of-band verification.

**Signal Private Groups.** In [22], a central server tracks membership of a group whilst hiding the set of group members from non-members (modulo metadata leaked to a network adversary). The main goal of this solution is to achieve user privacy. We believe that this approach could be extended to support secure administration; an advantage is that users no longer have to track group membership individually as in (A)-CGKA, which prevents consistency issues when users do not apply the same sequence of group updates locally. However, the system has not been analysed in composition with an underlying group messaging protocol (pairwise Signal) where concurrency issues can arise. Moreover, administration security has not been formally analysed and in their constructions the server applies administration updates.

## 6.3 Additional Admin Mechanisms

To conclude, we consider possible extensions of A-CGKA and corresponding construction ideas for future work.

**Admins beyond CGKA.** CGKA is not a suitable formalism for some group messaging protocols used in practice like pairwise Signal and Sender Keys (used in WhatsApp [40]). In these protocols, each user is associated with their own key or keying material rather than a common group secret. Nevertheless, an IAS-like protocol can be easily adapted to this setting. For Sender Keys, admins could replace their keying material at a low cost (a signature on their new signing key) for PCS authentication guarantees. We leave it as useful future work to formalise group administration beyond CGKA.

Telegram, although not end-to-end encrypted, offers fine-grained administration features like message filtering and delays. Some of these could be conceivably implemented cryptographically, e.g. leveraging admins and/or NIZKs.

**Private admins.** In some applications, it may be desirable to hide the set of admins from other users within a group. DGS could achieve some notion of administrative privacy if the underlying admin CGKA provides privacy guarantees. IAS could be modified to achieve anonymity guarantees using ring signatures, although involving an overhead.

In MLS' TreeKEM protocol, proposals are constant-sized, but commits are variable, which leaks information about the commit even if encrypted. Thus, padding is required at a minimum for privacy.

**External admins.** Our A-CGKA constructions assume that all admins are group members. Some applications may be better suited for *external* administration. For example, an online platform may wish to control the set of conversation participants to ensure they are subscribers but nevertheless ensure they are provided confidentiality. External admins who then attempt to add users that group members do not trust can be detected on the protocol level.

**Threshold admins.** To improve robustness, admins could use threshold cryptography such that e.g. several admins need to sign a commit message for it to be considered valid [36].

## Acknowledgments

We would like to thank all anonymous reviewers and our shepherd who helped improve our paper.

David Balbás is supported by the PICOCRYPT project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 101001283), partially supported by PRODIGY Project (TED2021-132464B-I00) funded by MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU / PRTR, and partially funded by Ministerio de Universidades (FPU21/00600). David Balbás carried out part of this work while at EPFL, Switzerland.

## References

[1] Martin R Albrecht, Sofía Celi, Benjamin Dowling, and Daniel Jones. Practically-exploitable cryptographic vulnerabilities in matrix. https://nebuchadnezzar-megolm.github.io/static/paper.pdf.

[2] Joël Alwen, Benedikt Auerbach, Mirza Ahad Baig, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. Grafting key trees: Efficient key management for overlapping groups. In Kobbi Nissim and Brent Waters, editors, *Theory of Cryptography*, pages 222–253, Cham, 2021. Springer International Publishing.

[3] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof 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.

[4] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 129–158, Cham, 2019. Springer International Publishing.

[5] 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.

[6] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular Design of Secure Group Messaging Protocols and the Security of MLS. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1463–1483, New York, NY, USA, 2021. Association for Computing Machinery.

[7] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In Rafael Pass and Krzysztof Pietrzak, editors, *Theory of Cryptography*, pages 261–290, Cham, 2020. Springer International Publishing.

[8] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. Server-aided continuous group key agreement. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 69–82, New York, NY, USA, 2022. Association for Computing Machinery.

[9] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the Insider Security of MLS. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 34–68, Cham, 2022. Springer Nature Switzerland.

[10] David Balbás. On Secure Administrators for Group Messaging Protocols, 2021. MSc Thesis, KTH Royal Institute of Technology.

[11] David Balbás, Daniel Collins, and Serge Vaudenay. Cryptographic administration for secure group messaging. Cryptology ePrint Archive, Paper 2022/1411, 2022. https://eprint.iacr.org/2022/1411.

[12] Fatih Balli, Paul Rösler, and Serge Vaudenay. Determining the Core Primitive for Optimally Secure Ratcheting. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020*, pages 621–650, Cham, 2020. Springer International Publishing.

[13] R Barnes, B Beurdouche, J Millican, E Omara, K Gohn-Gordon, and R Robert. The Messaging Layer Security (MLS) Protocol. https://messaginglayersecurity.rocks/mls-protocol/.

[14] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. Hybrid Public Key Encryption. Internet-Draft draft-irtf-cfrg-hpke-08, Internet Engineering Task Force. Work in Progress.

[15] Mihir Bellare and Sara K. Miner. A Forward-Secure Digital Signature Scheme. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 431–448, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[16] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted Encryption and Key Exchange: The Security of Messaging. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 619–650, Cham, 2017. Springer International Publishing.

[17] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous Decentralized Key Management for

Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS). 5 2018. https://hal.inria.fr/hal-02425247.

[18] Alexander Bienstock, Yevgeniy Dodis, Sanjam Garg, Garrison Grogan, Mohammad Hajiabadi, and Paul Rösler. On the worst-case inefficiency of cgka. In Eike Kiltz and Vinod Vaikuntanathan, editors, *Theory of Cryptography*, pages 213–243, Cham, 2022. Springer Nature Switzerland.

[19] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the Price of Concurrency in Group Ratcheting Protocols. In Rafael Pass and Krzysztof Pietrzak, editors, *Theory of Cryptography*, pages 198–228, Cham, 2020. Springer International Publishing.

[20] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use pgp. ACM Press, 2004.

[21] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Security analysis of the MLS key derivation. In *43rd IEEE Symposium on Security and Privacy (SP)*, 2022.

[22] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The signal private group system and anonymous credentials supporting efficient verifiable encryption. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1445–1459, 2020.

[23] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A Formal Security Analysis of the Signal Messaging Protocol. *Journal of Cryptology*, 33, 2020.

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

[25] 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*, pages 1802–1819, 2018.

[26] Cas Cremers, Britta Hale, and Konrad Kohbrok. The Complexities of Healing in Secure Group Messaging: Why {Cross-Group} Effects Matter. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1847–1864, 2021.

[27] Julien Devigne, Céline Duguey, and Pierre-Alain Fouque. MLS: how Zero-Knowledge can secure Updates. In *ESORICS 2021*, 2021.

[28] F. Betül Durak and Serge Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. In Nuttapong Attrapadung and Takeshi Yagi, editors, *Advances in Information and Computer Security*, pages 343–362, Cham, 2019. Springer International Publishing.

[29] The Matrix.org Foundation. Matrix specification v1.4, 2022. https://spec.matrix.org/latest/.

[30] Jonathan Katz and Ji Sun Shin. Modeling insider attacks on group key-exchange protocols. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 180–189, 2005.

[31] Karen Klein, Guillermo Pascual-Perez, Michael Walter, Chethan Kamath, Margarita Capretto, Miguel Cueto, Ilia Markov, Michelle Yeo, Joël Alwen, and Krzysztof Pietrzak. Keep the dirt: Tainted treekem, adaptively and actively secure continuous group key agreement. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 268–284. IEEE, 2021.

[32] Tal Malkin, Daniele Micciancio, and Sara Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 400–417. Springer, 2002.

[33] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm, 2016. https://signal.org/docs/specifications/doubleratchet/.

[34] Bertram Poettering, Paul Rösler, Jörg Schwenk, and Douglas Stebila. Sok: Game-based security models for group key exchange. In *Cryptographers' Track at the RSA Conference*, pages 148–176. Springer, 2021.

[35] Paul Rösler, Christian Mainka, and Jörg Schwenk. More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 415–429, 2018.

[36] Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000.

[37] Telegram. Group Chats on Telegram. https://telegram.org/tour/groups.

[38] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. CCS '21, page 2024–2045, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Matthew A Weidner. Group messaging for secure asynchronous collaboration, 2019. MSc Thesis, University of Cambridge.

[40] WhatsApp. WhatsApp Encryption Overview Technical white paper, v.3, October 2020. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf.

## A Cleanness Predicates

**CGKA predicate.** A more fine-grained characterization of this predicate is to write it as $\mathsf{C}_{\mathsf{cgka}} = \mathsf{C}_{\mathsf{cgka\text{-}opt}} \wedge \mathsf{C}_{\mathsf{cgka\text{-}add}}$, where $\mathsf{C}_{\mathsf{cgka\text{-}opt}}$ is an *optimal*, generic cleanness predicate that excludes only unavoidable trivial attacks, and $\mathsf{C}_{\mathsf{cgka\text{-}add}}$ is an additional cleanness predicate that depends on the scheme and excludes other attacks. We define $\mathsf{C}_{\mathsf{cgka\text{-}opt}}$ in a similar way to the safe predicate in [5]. Namely, we exclude the following cases: (i) the group secret in challenge epoch $t_s^*$ was already challenged or revealed, and (ii) a group member ID whose state was exposed in epoch $t_{\mathsf{exp}} \leq t_s^*$ did not update their keys (i.e., processed their own commit, or processed a commit in which they were involved in an add, remove, or update proposal) or was not removed before the challenge epoch $t_s^*$. The optimal cleanness predicate is given in Figure 4 for an adversary that makes queries $q_1, \ldots, q_n$ in the game.

$\mathsf{C}_{\mathsf{cgka\text{-}opt}}$ : $\forall\, (i, \mathsf{ID}, \mathsf{ctr} \in (0, \mathsf{exp\text{-}ctr}]) : q_i = O^{\mathsf{Challenge}}(t_i^*)$,
$(\mathsf{ID} \notin \mathsf{G}[t_i^*]) \vee$
$\big(\exists (t_i, c) : (\mathsf{tExp}(\mathsf{ID}, \mathsf{ctr}).t_s < t_i \leq t_i^*) \wedge$
$\mathsf{hasUpd}_{\mathsf{std}}(\mathsf{ID}, \mathsf{T}[(t_i, \cdot), \texttt{'com'}, c], \mathsf{T}[(t_i, \cdot), \texttt{'vec'}, c]) \wedge$
$(\mathsf{C}[(t_i, \cdot)] = c)\big) \vee \big(t_i^* < \mathsf{tExp}(\mathsf{ID}, \mathsf{ctr}).t_s\big)$.

**Figure 4:** Optimal CGKA predicate where the adversary makes oracle queries $q_1, \ldots, q_n$.

The predicate is the logical disjunction of three clauses: for every exposure, adversarial challenge, and party ID, we require that either 1) ID was not a group member at the challenge time, 2) the challenged epoch precedes the exposure (forward security), or 3) ID updated between the exposure and the (subsequent) challenge (post-compromise security). To avoid cluttering the predicate, our game already enforces that only one challenge or reveal can be performed per epoch (which is optimal for our game).

The function tExp is such that $\mathsf{tExp}(\mathsf{ID}, \mathsf{ctr}) = \mathsf{exp}[\mathsf{ID}, \mathsf{ctr}]$ if $\exists k : q_k = O^{\mathsf{Expose}}(\mathsf{ID})$, and $-1$ (for CGKA) or $(-1, -1)$ (for A-CGKA) otherwise. Given $\vec{P}$, the function $\mathsf{hasUpd}_{\mathsf{std}}(\mathsf{ID}, (T, \mathsf{com\text{-}type}), \vec{P})$ (sans com-type for CGKA) outputs true if either: (i) ID has processed a commit of his own, where $\mathsf{com\text{-}type} \in \{\mathsf{std}, \mathsf{both}\}$, or (ii) ID is a user affected by an add, update, or removal proposal in $\vec{P}$.

**Admin predicate.** $\mathsf{C}_{\mathsf{adm\text{-}opt}}$ is symmetric to $\mathsf{C}_{\mathsf{cgka\text{-}opt}}$ and excludes the following family of attacks: the adversary attempts a forgery on a member $\mathsf{ID}'$ at an administrative epoch $t_a^*$ while having exposed the state of $\mathsf{ID} \in G^*$ at epoch $t_{\mathsf{exp}} \leq t_a^*$, such that ID has not updated at some point between them. The predicate is optimal, as any attack that it excludes must occur while an administrator is directly under state exposure. In the game itself, we also require that $\mathsf{ID}'$ is in the challenge epoch specified by the adversary, i.e., $\mathsf{ep}[\mathsf{ID}'] = (\cdot, t_a^*)$. This predicate is unrelated to the common group secret and standard epochs $t_s$, and only relates to administration dynamics.

$\mathsf{C}_{\mathsf{adm\text{-}opt}}$ : $\forall\, (i, \mathsf{ID}, \mathsf{ID}', \mathsf{ctr} \in (0, \mathsf{exp\text{-}ctr}]) : q_i = O^{\mathsf{Inject}}(\mathsf{ID}', \cdot, t_i^*)$,
$(\mathsf{ID} \notin \mathsf{ADM}[t_i^*]) \vee$
$\big(\exists (t_i, c) : (\mathsf{tExp}(\mathsf{ID}, \mathsf{ctr}).t_a < t_i \leq t_i^*) \wedge$
$\mathsf{hasUpd}_{\mathsf{adm}}(\mathsf{ID}, \mathsf{T}[(\cdot, t_i), \texttt{'com'}, c], \mathsf{T}[(\cdot, t_i), \texttt{'vec'}, c]) \wedge$
$(\mathsf{C}[(\cdot, t_i)] = c)\big) \vee \big(t_i^* < \mathsf{tExp}(\mathsf{ID}, \mathsf{ctr}).t_a\big)$.

**Figure 5:** Optimal administrative predicate where the adversary makes oracle queries $q_1, \ldots, q_n$.

The optimal admin predicate $\mathsf{C}_{\mathsf{adm\text{-}opt}}$ is captured in Figure 5. In the expression, the function $\mathsf{hasUpd}_{\mathsf{adm}}$ is defined as in $\mathsf{hasUpd}_{\mathsf{std}}$, except it is defined with respect to $\mathsf{com\text{-}type} \in \{\mathsf{adm}, \mathsf{both}\}$ (rather than $\mathsf{com\text{-}type} \in \{\mathsf{std}, \mathsf{both}\}$).

**Forgery predicate.** $\mathsf{C}_{\mathsf{forge}}$ is defined as follows with respect to variables in $O^{\mathsf{Inject}}$ and the game. Suppose $m$ is input to $O^{\mathsf{Inject}}$. Consider $\mathsf{ST}[\mathsf{ID}].G, \mathsf{ST}[\mathsf{ID}].G^*$, and let $\vec{P} = \mathsf{props}(\mathsf{ST}[\mathsf{ID}], m)$. Consider $\vec{P}' = \{P \in \vec{P} : P' \in \mathsf{T}[\mathsf{ep}[\mathsf{ID}], \texttt{'prop'}, \cdot] \wedge \mathsf{prop\text{-}info}(\mathsf{ST}[\mathsf{ID}], P) = \mathsf{prop\text{-}info}(\mathsf{ST}[\mathsf{ID}], P')\}$.[5] Let $H = \{\mathsf{ID} : (\mathsf{gid}, \mathsf{rem}, \mathsf{ID}, \mathsf{ID}) = \mathsf{prop\text{-}info}(\mathsf{ST}[\mathsf{ID}], P) \wedge (P \in \vec{P}')\}$ and $H^* = \{\mathsf{ID} : (\mathsf{gid}, \mathsf{rem\text{-}adm}, \mathsf{ID}, \mathsf{ID}) = \mathsf{prop\text{-}info}(\mathsf{ST}[\mathsf{ID}], P) \wedge (P \in \vec{P}')\}$. Then $\mathsf{C}_{\mathsf{forge}}$ is true if and only if $(\mathsf{ST}[\mathsf{ID}].G \setminus H, \mathsf{ST}[\mathsf{ID}].G^* \setminus H^*) \neq (\gamma.G, \gamma.G^*)$. If there are no self-removes, i.e. $H = H^* = \emptyset$, this simplifies to $(\mathsf{ST}[\mathsf{ID}].G, \mathsf{ST}[\mathsf{ID}].G^*) \neq (\gamma.G, \gamma.G^*)$; let $\mathsf{C}_{\mathsf{forge}}{}^*$ be this simplified predicate.

## B Additional Figures

We include Figures 6, 7 and 8 which contain the auxiliary functions for IAS, the implementation of the MLS extension in Section 4.3, and the benchmark for the proc algorithm as done for commit in Section 5.2.

[5] The effect of the checks for prop-info is that a dishonest proposal $P'$ that has the same semantics as an honest proposal $P$ will not be considered a 'forgery' by $\mathsf{C}_{\mathsf{forge}}$.

VALID$_P$

/ Predicate checks validity of admin proposal
/ as defined in Section 4.1

makeAdminProp(gid, type, ID; $r_1, r_2$)

$P_0 \leftarrow \perp$
**if** type = add-adm
  $\mathsf{spk}_{\mathsf{pki}} \leftarrow \mathsf{getSpk}(\mathsf{ID}, \gamma.\mathsf{ME})$
  $P_0 \leftarrow (\mathsf{gid}, \mathsf{type}, \mathsf{ID}, \gamma.\mathsf{ME}, \mathsf{spk}_{\mathsf{pki}})$
**else if** type = rem-adm
  $P_0 \leftarrow (\mathsf{gid}, \mathsf{type}, \mathsf{ID}, \gamma.\mathsf{ME}, \perp)$
**else if** type = upd-adm
  **if** $(\gamma.\mathsf{ssk}', \gamma.\mathsf{spk}') \neq (\perp, \perp)$
    **return** $\perp$ / only one update per epoch
  $(\gamma.\mathsf{ssk}', \gamma.\mathsf{spk}') \leftarrow \mathsf{SigGen}(\gamma.1^\lambda; r_1)$
  $P_0 \leftarrow (\mathsf{gid}, \mathsf{type}, \gamma.\mathsf{ME}, \gamma.\mathsf{ME}, \gamma.\mathsf{spk}')$
**else return** $\perp$
**return** $(P_0, \mathsf{Sig}(\gamma.\mathsf{ssk}, P_0; r_2))$

c-Std(gid, $\vec{P}_0, \vec{P}_A; r_1$)

$(\gamma.\mathsf{s0}, C_0, W_0, k) \leftarrow$
    $\mathsf{CGKA.commit}(\gamma.\mathsf{s0}, \mathsf{gid}, \vec{P}_0; r_1)$
**if** $W_0 \neq \perp$ / list for new users only
  $\mathsf{adminList}' \leftarrow \mathsf{updAL}(\gamma.\mathsf{adminList}, \vec{P}_A)$
  **return** $(C_0, W_0, \mathsf{adminList}', k)$
**else return** $(C_0, \perp, \perp, k)$

verifyPropSigs($\vec{P}_0, \Sigma, \vec{P}_A$)

**for** $(P, \mathsf{ID}, \sigma) \in \Sigma$
  $\mathsf{spk} \leftarrow \mathsf{getSpk}(\mathsf{ID}, \gamma.\mathsf{ME})$
  **if** $\neg\mathsf{Ver}(\mathsf{spk}, P, \sigma) \vee$
  $P \notin \vec{P}_0 \vee \mathsf{adminList}[\mathsf{ID}] \neq \perp$
    **return** false
**for** $(P, \sigma_P) \in \vec{P}_A$ :
  $(\perp, \perp, \perp, \mathsf{ID}') \leftarrow \mathsf{prop\text{-}info}(P)$
  $\mathsf{spk}_P \leftarrow \mathsf{adminList}[\mathsf{ID}'].\mathsf{spk}$
  **if** $\neg(\mathsf{Ver}(\mathsf{spk}_P, P, \sigma_P) \wedge \mathsf{VALID_P})$
    **return** false
**return** true

propCleaner(gid, $\vec{P}$)

admReq $\leftarrow$ false; $\vec{P}_0, \vec{P}_A, \Sigma \leftarrow []$
**for** $P \in \vec{P}$ :
  $(\mathsf{gid}, P.\mathsf{type}, P.\mathsf{ID}, P.\mathsf{ID}') \leftarrow \mathsf{prop\text{-}info}(P)$
  **if** $(P.\mathsf{type} = *\text{-adm}) \wedge \mathsf{VALID}_P$
    $\vec{P}_A \leftarrow [\vec{P}_A, P]$
    admReq $\leftarrow$ true
  **else** / $\vec{P}_0$ is handled by CGKA
    **if** $P.\mathsf{type} \in \{\mathsf{add}, \mathsf{rem}\}$
      admReq $\leftarrow$ true
    **if** $P.\mathsf{type} = \mathsf{rem} \wedge$
    $(P.\mathsf{ID} = P.\mathsf{ID}') \wedge (P.\mathsf{ID} \notin \gamma[\mathsf{gid}].G^*)$
      admReq $\leftarrow$ false
      $(P', \sigma) \leftarrow P$;
      $\vec{P}_0 \leftarrow [\vec{P}_0, P']$; $\Sigma \leftarrow [\Sigma, (P, P.\mathsf{ID}, \sigma)]$
    **else**
      $\vec{P}_0 \leftarrow [\vec{P}_0, P]$
    / admin rem from $G \implies$ rem from $G^*$
    **if** $(P.\mathsf{type} = \mathsf{rem}) \wedge (P.\mathsf{ID} \in \gamma[\mathsf{gid}].G^*)$
      $P' \leftarrow \mathsf{makeAdminProp}(\mathsf{gid}, \mathsf{rem}, \mathsf{ID}; \perp)$
      $\vec{P}_A \leftarrow [\vec{P}_A, P']$
$(\vec{P}_0, \vec{P}_A) \leftarrow \mathsf{enforcePolicy}(\vec{P}_0, \vec{P}_A)$
**return** $(\vec{P}_0, \vec{P}_A, \Sigma, \mathsf{admReq})$

updAL(adminList, $\vec{P}_A$)

**for** $P \in \vec{P}_A$
  $(\mathsf{gid}, \mathsf{type}, \mathsf{ID}, \perp, \mathsf{spk}, \perp) \leftarrow P$
  **if** $\mathsf{type} \in \{\mathsf{add\text{-}adm}, \mathsf{upd\text{-}adm}\}$
    **if** $(\mathsf{type} = \mathsf{add\text{-}adm}) \wedge (\mathsf{ID} = \gamma.\mathsf{ME})$
      $\gamma.\mathsf{spk} \leftarrow \mathsf{spk}$
      $\gamma.\mathsf{ssk} \leftarrow \mathsf{getSsk}(\mathsf{spk}, \mathsf{ID})$
    $\mathsf{adminList}[\mathsf{ID}] \leftarrow (\mathsf{ID}, \mathsf{spk})$
  **if** $\mathsf{type} = \mathsf{rem\text{-}adm}$
    $\mathsf{adminList}[\mathsf{ID}] \leftarrow \perp$
    **if** $(\mathsf{ID} = \gamma.\mathsf{ME})$
      $(\gamma.\mathsf{ssk}, \gamma.\mathsf{spk}) \leftarrow (\perp, \perp)$
**return** adminList

p-Comm(gid, $T_C$)

$(\perp, \mathsf{ID}, C_0, C_A, \Sigma, \mathsf{spk}) \leftarrow T_C$
/ check signatures in proposals
**if** $C_A \neq \perp$
/ $C_A = \vec{P}_A$
  **if** $\neg\mathsf{verifyPropSigs}(C_A)$ **return** false
/ apply commit
**if** $C_0 \neq \perp$
  $(\gamma', \mathsf{acc}) \leftarrow \mathsf{CGKA.proc}(\gamma.\mathsf{s0}, C_0)$
  **if** acc = false **return** false
  **if** $\gamma.\mathsf{ME} \notin \gamma'.G$ / user removed
    $\gamma[\mathsf{gid}] \leftarrow \perp$ / reinitialize state (only gid)
  **else** $\gamma[\mathsf{gid}].\mathsf{s0} \leftarrow \gamma'$
/ set temporary updated keys
**if** $(\mathsf{ID} = \gamma.\mathsf{ME}) \vee (\exists P \in C_A : P.\mathsf{ID} = \gamma.\mathsf{ME})$
  $(\gamma.\mathsf{ssk}, \gamma.\mathsf{spk}) \leftarrow (\gamma.\mathsf{ssk}', \gamma.\mathsf{spk}')$
  $\gamma.\mathsf{ssk}', \gamma.\mathsf{spk}' \leftarrow \perp$
$\gamma.\mathsf{adminList} \leftarrow \mathsf{updAL}(\gamma.\mathsf{adminList}, C_A)$
/ committer's key
$\gamma.\mathsf{adminList}[\mathsf{ID}].\mathsf{spk} \leftarrow \mathsf{spk}$
**return** true

enforcePolicy($\vec{P}_0, \vec{P}_A$)

/ This method enforces policy consistency
**return** $(\vec{P}_0, \vec{P}_A)$

p-Wel(gid, $T_W, \sigma$)

$(\perp, \mathsf{ID}, W_0, \mathsf{adminList}) \leftarrow T_W$
$(\gamma[\mathsf{gid}].\mathsf{s0}, \mathsf{acc}) \leftarrow \mathsf{CGKA.proc}(\gamma.\mathsf{s0}, W_0)$
$\mathsf{acc} \leftarrow \mathsf{acc} \wedge$
  $\mathsf{Ver}(\mathsf{getSpk}(\gamma.\mathsf{ME}, \mathsf{ID}), (\mathsf{gid}, \perp, T_W), \sigma)$
**if** acc  $\gamma[\mathsf{gid}].\mathsf{adminList} \leftarrow \mathsf{adminList}$
**if** $\mathsf{acc} \wedge (\mathsf{adminList}[\mathsf{ME}] \neq \perp)$
  $\gamma.\mathsf{spk} \leftarrow \mathsf{adminList}[\mathsf{ME}].\mathsf{spk}$
  $\gamma.\mathsf{ssk} \leftarrow \mathsf{getSsk}(\mathsf{spk}, \mathsf{ME})$
**return** acc

**Figure 6:** Helper functions for the IAS construction in Figure 2.

prop(ID, type; $r_0$)

1 : **if** type $= *$-adm
2 : **require** $\gamma$.ME $\in \gamma$.adminList
3 : $(P, \perp) \leftarrow$ IAS.makeAdminProp(type, ID; $r_0$)
/ getSpk is replaced in makeAdminProp
4 : **if** type $\in$ {add, rem, upd}
5 : $(\gamma, P) \leftarrow$ CGKA.prop($\gamma$, ID, type; $r_0$)
/ Added users' keys retrieved from contact list/PKI
/ Proposals in MLS are each signed
6 : $\sigma \leftarrow$ Sig($\gamma$.ssk, $P$)
7 : **return** $(P, \sigma)$

getSpk(ID)

/ Get spk from ID's credential
1 : **return** Cred[ID].spk

updSpk($\gamma$, ID, spk$'$)

/ Register spk$'$ with the PKI
1 : $\gamma \leftarrow$ registerPKI($\gamma$, ID, spk$'$)
/ Update ID's credential
2 : $\gamma$.Cred[ID].spk $\leftarrow$ spk$'$

commit($(\vec{P}_0, \vec{P}_A)$, com-type; $r_0$)

1 : $(r_1, r_2, r_3) \leftarrow H_3(r_0, \gamma)$
2 : **if** com-type $\in$ {adm, both}
3 : **require** $\gamma$.ME $\in \gamma$.adminList
4 : **require** IAS.verifyPropSigs($\vec{P}_A$)
5 : $C_A \leftarrow \vec{P}_A$
6 : adminList$'$ $\leftarrow$ IAS.updAL(adminList, $\vec{P}_A$)
7 : $(\gamma$.ssk$', \gamma$.spk$') \leftarrow$ SigGen($\gamma.1^\lambda; r_1$)
8 : $\gamma \leftarrow$ updSpk($\gamma$, ID, spk$'$)
9 : **if** com-type $\in$ {std, both}
10 : $(\gamma, C_0, W_0, \perp) \leftarrow$ CGKA.commit($\vec{P}_0; r_2$)
11 : **if** $W_0 \neq \perp$ / share updated adminList
12 : Prepare wel. msgs as in [6] for $\vec{W}$
13 : **for** $W \in \vec{W}$ :
14 : $W \leftarrow W$||adminList$'$
15 : $\sigma \leftarrow\$$ Sig($\gamma$.ssk, $W$) / rand.
16 : $T \leftarrow$ ('com', $\gamma$.ME, $C_0, C_A, \gamma$.spk$'$)
17 : $\sigma \leftarrow$ Sig($\gamma$.ssk, $T; r_3$)
18 : **return** $((T, \sigma), \vec{W})$

proc-WM($W$)

/ ID is the committer of $W$
1 : **require** ID $\in W$.adminList
2 : Run Proc-WM($W$) in [6]
3 : $\gamma$.adminList $\leftarrow W$.adminList
4 : Check adminList[ID].spk with PKI

proc-CM($T, \sigma$)

1 : ('com', ID, $C_0, C_A$, spk$'$) $\leftarrow T$
2 : **if** ID $\notin$ adminList
3 : **require** Ver(getSpk(ID), $T, \sigma$)
4 : Run Proc-CM($T$) in [6]
5 : **require** no membership changes to $\gamma.G$ except self-removals
**if** ID $\in$ adminList
6 : **require** Ver(adminList[ID].spk, $T, \sigma$)
7 : **if** spk$' \neq \perp$
/ spk was registered
8 : **require** spk$'$ = getSpk(ID)
9 : Update keys and adminList as in IAS
10 : IAS.p-Comm($T$)
11 : Check new adminList keys with PKI

**Figure 7:** Main algorithms of an MLS extension that supports group administrators, effectively converting the CGKA in MLS into a A-CGKA. Highlighted lines correspond to our main modifications in the original SGM construction in [6]. Cred[·] denotes an array that stores the credentials of all ID's. We also use the abstract function registerPKI for standard PKI functionality for registering signature keys. Several technical details are omitted or simplified.

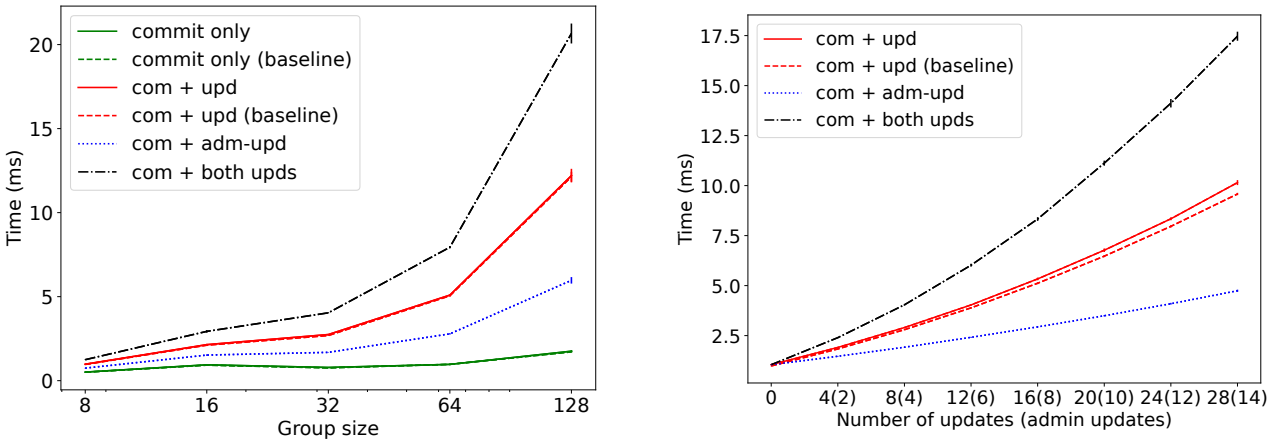


**Figure 8:** Benchmark of the proc algorithm when processing a commit message. The different scenarios are those of Figure 3.