Cryptographic Administration for Secure Group Messaging

David Balbás∗1,2, Daniel Collins3, and Serge Vaudenay3

1IMDEA Software Institute, Madrid, Spain
2Universidad Politécnica de Madrid, Spain
3EPFL, Switzerland

August 7, 2023
Full Version†

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. We embed A-CGKA with a novel correctness notion which provides guarantees for group evolution and consistency, and a security model that prevents even corrupted (non-admin) members from forging messages that add new users to a group.

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. Our constructions admit little overhead over running a CGKA and can be extended to support advanced admin functionalities.
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 [PM16] and OTR [BGB04] protocols, or for group conversations. In the group case ([BBR18, ACDT20, KPPW∗21, ACJM20, ACDT21a]), modern protocols are often designed to achieve forward security (FS) and post-compromise security (PCS) [CGCG16] to protect past and future communications, respectively, upon state compromise. The most common approach for designing a 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. [ACDT20] 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 [BBR18]) has been adopted by the Messaging Layer Security (MLS) [BBM∗] 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 diminishing the confidentiality of 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 [ACDT20, KPPW∗21], Sender Keys (WhatsApp, Signal Messenger) [Wha20] and Matrix [Fou22], 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 *burglary into a group attack* [RMS18], 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 [ACDJ22] 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 users in a group: group 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.

1.1.1 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 [Tel].

- 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.

1.1.2 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 [KS05, AJM20] on protocol execution. Insider 
adversaries, or compromised group members, will not be able to gain control of a group 
unless they are administrators\(^1\).

\(^1\)Note that denial-of-service attacks from malicious non-admin insiders as in [ACJM20] are not necessarily 
prevented; we also remark that this family of attacks does not affect confidentiality. This issue is discussed in 
later sections.
- Increase the resilience of implementations of messaging protocols, preventing pitfalls such as the *burgle into a group attack* [RMS18] or the recent attacks on Matrix [ACDJ22].
- Reduce concurrency issues, especially when the delivery service is not a central server [WKHB21], since only a reduced set of members are able to commit group changes.

### 1.1.3 Admin capabilities

Let $G = \{ID_1, \ldots, ID_n\}$ 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 $ID \in G^*$ 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 presented above. In the case of Telegram, their “fine-grained administration” is practical due to the lack of end-to-end encryption. By default, Telegram relies on a central server that 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; our main goal in this paper is to provide these. 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.

2. We introduce a novel game-based correctness notion for both CGKA and A-CGKA in Section 3.3 which emphasises the role of *group dynamics* which we argue is centrally linked to group administration.

3. Extending existing CGKA key indistinguishability security notions, we introduce a game-based security notion (Section 3.4) which further aims to prevent even fully corrupted non-admin users from modifying group membership.

4. 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 formalise both protocols in detail in Section 4, analyze their performance in Section 6.1, and prove correctness and security (Section 5.2).

5. We propose an extension to MLS in Section 4.3 that provides efficient secure administration that we also implement and benchmark locally (Section 5.3).

6. 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 [ACJM20, AJM20, ACDT21a]. 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. In A-CGKA, only admins can make admin proposals, and moreover only admins can authorise all types of commit messages. That is, users only process group changes that have been attested by an admin (except for users leaving the group by themselves, which can always be processed).

Correctness. Our notion of A-CGKA correctness enforces that users that process the same sequence of commit messages for a given group derive consistent views of both the evolution of group members and admins, and also of the shared key. Our game explicitly checks that only through processing commit messages that group membership can change. We also enforce that honest proposals have their intended effect when embedded in commit messages upon processing.

Early CGKA game-based correctness notions were limited to key consistency [ACDT20] or were not formally specified [KPPW+21]. It is only recently that group evolution guarantees have been considered, notably in the latest version of [ACJM20] which works in the UC and the monolithic security definition of [ACDT21b]. We emphasise this because, given the length of our IAS construction and corresponding correctness proof, it is possible that subtle bugs concerning group evolution are hidden in existing group messaging constructions and implementations.\footnote{For example, in Figure 8 (SGM) and Figure 17 (RTreeKEM) in [ACDT21b], commit processing is not well-defined for a user who is processing their own removal from the group.}

In particular, we spotted (resolved) inconsistencies while trying to prove our protocols correct.

Security. Our security notion captures two core guarantees. Firstly, like previous work [ACDT20, KPPW+21], 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 [DV19, ACDT20].

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. We model this by allowing the (semi-active) adversary to inject commit messages to particular parties which process the messages (albeit without updating their state). The adversary can adaptively corrupt participants and make challenge queries. Security is ensured insofar as the adversary does not trivially compromise an administrator, i.e., they are permitted to compromise many non-admins. As in the case of key indistinguishability, we also specify a separate admin cleanness predicate to capture trivial
attacks for this attack vector. Our security notion allows for FS and PCS guarantees with respect to the admin keying material.

**Constructions.** In this work, we provide two separate, modular constructions of A-CGKA from a CGKA that we describe in Section 4. We also introduce an extension of MLS that supports administration. The security of the authentication mechanisms in all our protocols matches the FS and PCS demands of group messaging.

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). This highlights the fact that administration can be done without authenticating single users. Instead of maintaining individual signatures, admins instead 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 over time and verify that commits are signed using it.

In Section 4.3, we embed the MLS protocol with A-CGKA functionality more organically by making use of MLS’s credential infrastructure. We describe the main modifications needed and propose an extension of MLS that admits secure administration. 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.** In the appendices, we formally prove that our protocols IAS and DGS are correct and secure with respect to our A-CGKA definitions. The main theorems are as follows:

**Theorem 1** (Simplified). Let CGKA be a correct and \((t_{cgka}, q, \epsilon_{cgka})\)-secure CGKA. Let \(S = (\text{SigGen}, \text{Sig}, \text{Ver})\) be a \((t_S, q, \epsilon_S)\) SUF-CMA secure signature scheme. Let \(H_4\) be a \((t_F, 1, \epsilon_F)\)-secure PRF. Then, the IAS protocol (Figures 6, 7 and 8) is correct and \((t, q, q \cdot \epsilon_F + \epsilon_{cgka} + q^2 \cdot \epsilon_S)\)-secure (Definition 4) where \(t_{cgka} \approx t_S \approx t_F \approx t\).

We prove IAS secure with respect to a sub-optimal admin cleanness predicate (somewhat weak forward secrecy). We argue that the protocol and proof can be very easily modified to satisfy optimal security using forward-secure signatures [BM99] with no asymptotic overhead; this is discussed in Section 4.1.4.

**Theorem 2** (Simplified). Let CGKA (resp. CGKA\(^\ast\)) be a correct and \((t_{cgka}, q, \epsilon_{cgka})\)-secure (resp. \((t_{cgka\ast}, q, \epsilon_{cgka\ast})\))-secure CGKA. Let \(S = (\text{SigGen}, \text{Sig}, \text{Ver})\) be a \((t_S, q, \epsilon_S)\) SUF-CMA secure signature scheme. Let \(H_4\) be a \((t_F, 1, \epsilon_F)\)-secure PRF and \(H_{\text{ro}}\) a random oracle queried at most \(q_{\text{ro}}\) times. Then, the DGS protocol (Figures 6 and 7) is correct and \((t, q, \epsilon_{cgka} + q \cdot (\epsilon_F + \epsilon_S + q_{\text{ro}} \epsilon_{cgka\ast} + 2^{-\lambda}))\)-secure (Definition 4) for security parameter \(\lambda\) in the random oracle model where \(t_{cgka} \approx t_S \approx t_F \approx t\).

**1.4 Additional Related Work**

**Messaging.** 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 [BGB04] and Signal [PM16] (the latter being formalized in [ACD19, CGCD\(^+\)20]).
The TreeKEM protocol in MLS, initially proposed in [BBR18], was inspired by Asynchronous Ratchet Trees [CGCG+18]. Later, variants of TreeKEM arose like Tainted TreeKEM [KPPW+21], Insider-Secure TreeKEM [AJM20], Re-randomized TreeKEM [ACDT20], and Causal TreeKEM [Wei19]. MLS as a whole is studied in [BCK22, ACDT21a].

CGKAs have been recently used to formally build full group messaging protocols [ACDT21a]. Besides TreeKEM, CGKA variants include [ACJM20, AHKM21, WKHB21, AAN+22]. Side works deal with multi-group security [CHK21], efficient key schedules for multiple groups [AAB+21], and concurrency [BDR20]. Separately, [PRSS21] surveys group key exchange protocols. Group admins were considered in [RMS18], although without a formal cryptographic approach, instead opting for security notions described more informally akin to a symbolic model of security using predicates.

An alternative approach towards securing group membership was taken in the Signal Private Group System [CPZ20] which we discuss in Section 6.2.

An earlier version containing some preliminary results appears in Balbas’ MSc thesis [Bal21].

2 Notation

A user, participant, or party is an entity that takes part in a protocol. Users are identified by a unique identity string ID, which is a public parameter. Groups are also uniquely identified by a public group identifier gid. Users keep an internal state γ with all information used for protocol execution. This includes keys, message records, dictionaries and parameters. If γ is leaked, we say ID suffers a state compromise or a corruption.

To assign the output of an algorithm Alg on input x to variable a, we write a ← Alg(x), and a ←$ Alg(x)$ for randomized algorithms (to make the randomness r explicit, we write a ← Alg(x;r)). Blank values are denoted by ⊥.

Our security games are played between a challenger and an adversary A, who can interact with the protocol via oracles. We let λ ∈ N be the security parameter. In our games, the predicate ‘require P’ enforces that a logical condition P is satisfied; otherwise the oracle/algorithm aborts and returns ⊥. 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: A[k] ← a adds the value a to the dictionary A under key k, overwriting if necessary. b ← A[k] retrieves A[k] and assigns it to variable b. A dictionary A is initialized as A[·] ← a; where all values are set to a. The use of the prefix operator Alg(++)x), is equivalent to writing first x ← x + 1 and then Alg(x).

3 (Administrated) Continuous Group Key Agreement

In this section, we introduce Continuous Group Key Agreement (CGKA) and then extend it to formalize our Administrated CGKA (A-CGKA) primitive. We also introduce our correctness and security definitions for both CGKA and A-CGKA.

3.1 Continuous Group Key Agreement

The aim of the Continuous Group Key Agreement (CGKA) primitive [ACDT20] 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 definition of CGKA is introduced below, in the so-called propose and commit paradigm [AJM20, ACJM20], in which different operation proposals in a given group (e.g., adding/removing members) are collated into a commit message by a group member which is processed by users. The evolution of a CGKA in time is captured by epochs; a group member advances to a new epoch every time they successfully process a commit message, at which point there may be a change in the shared secret and/or group structure from their perspective.

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 $\text{CGKA} = (\text{init}, \text{create}, \text{prop}, \text{commit}, \text{proc}, \text{prop-info}, \text{props})$ such that:

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

3.1.1 Protocol execution

For simplicity, we assume that all users and groups are associated with a unique identifier ID and gid, respectively. Once every user has initialized their state using init, a group is created when some party calls create on a list of IDs. The init algorithm can also serve to authenticate and register keys on a PKI when appropriate, as in [ACDT20, KPPW+21, ACJM20]. In Section 4.1, we expand on the use of PKI and authentication issues: for each protocol, we describe our...
assumptions on the PKI. The create algorithm outputs a control message $T$ that must be processed by prospective group members, including the group creator, to join the group $\text{gid}$.

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 type = upd to propose to update (i.e., refresh) their keying material, in which case the input ID is ignored. Following [ACDT21a], we define prop-info which outputs proposal attributes, rather than allowing for their direct access, to support possibly encrypted proposals (e.g., as in MLSCiphertext [BBM+]).

Proposed changes become effective once a user commits a (possibly empty) vector of proposals $\vec{P} = (P_1, \ldots, P_m)$ using commit. The commit algorithm 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 bit acc indicating success or failure. We note that proc and props do not require a group identifier as input; this models the standard behaviour of a messaging protocol where, upon reception of a message, the user needs to determine which group the message corresponds to. In the event that a group member needs to parse the proposals in a commit message $T$ without processing, it can do so via the props algorithm.

3.1.2 Example

Consider 5 parties \{ID$_1$, \ldots, ID$_5$\} executing a CGKA protocol. After they each initialize their states as $\gamma_i \leftarrow \text{init}(1^\lambda, \text{ID}_i)$, the following actions take place:

1. ID$_1$ calls create($\gamma_1, \text{gid}, \{\text{ID}_1, \text{ID}_2, \text{ID}_3, \text{ID}_4\}$), which updates $\gamma_1$ and outputs a control message $T_0$. At this stage, the group is still empty, $G = \emptyset$.

2. Each ID$_i$ (including ID$_1$) processes the group creation as proc($\gamma_i, T_0$), which updates each state $\gamma_i$ and outputs acc = true to each user. At this stage, $G = \{\text{ID}_1, \text{ID}_2, \text{ID}_3, \text{ID}_4\}$, and the group members share a common secret $k_1$.

3. Several users propose changes in the group:
   - ID$_2$ proposes to add ID$_5$ to the group by calling $(\gamma'_2, P_1) \leftarrow \text{prop}(\gamma_2, \text{gid}, \text{ID}_5, \text{add})$.
   - ID$_3$ wants to update its key material and thus calls $(\gamma'_3, P_2) \leftarrow \text{prop}(\gamma_3, \text{gid}, \text{ID}_3, \text{upd})$.
   - ID$_1$ proposes to remove ID$_4$ from the group and calls $(\gamma'_1, P_3) \leftarrow \text{prop}(\gamma_1, \text{gid}, \text{ID}_4, \text{rem})$.

This is shown in Figure 1. The group remains the same.

4. ID$_2$ collates all proposals in a commit message by calling $(\gamma'_2, T_1, k_2) \leftarrow \text{commit}(\gamma_2, \text{gid}, (P_1, P_2, P_3))$. The group remains the same, since parties have not yet processed $T_1$.

5. All parties process $T_1$ by calling proc($\gamma_i, T_1$) and updating their states. Now, $G = \{\text{ID}_1, \text{ID}_2, \text{ID}_3, \text{ID}_5\}$ and these members share a new common secret $k_2$, which is not known
to ID_4. In addition, ID_3 (due to the update) and ID_2 (due to the commit) have refreshed their entire keying material.

Figure 1: Diagram of a sample CGKA execution with 5 parties. Parties in green (ID_2, ID_3, ID_5) will update their key material after ID_2 commits, while ID_4 will leave the group.

3.1.3 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 extend the syntax of A-CGKA with a dedicated policy algorithm that defines this order as in [Bal21].

3.1.4 Alternative definitions

Particularly in older CGKA definitions [ACDT20, KPPW+21] and MLS drafts, group changes are made effective immediately by processing proposals. To this end, the commit and prop algorithms are replaced by specific ‘action’ algorithms such as add, remove, update. The propose and commit paradigm used in this work was introduced in version 8 of MLS to allow for multiple group changes and key updates to be applied at the same time to reduce latency [BBM+].

As mentioned in [KPPW+21], the older protocols can be written in the propose and commit paradigm, which is more flexible (and also better suited for group administrators as we will see).

A relevant difference between the definition in [ACDT20] (and other game-based formulations such as in [KPPW+21]) and ours is that we work in the multi-group setting, and so we consider group identifiers of the form gid. Multi-group CGKAs have not been formalized in the literature, although multi-group security has been studied [CHK21], as well as efficient key schedules for multiple groups [AAB+21]. According to our definition, a user can be in many groups, identified by different values gid and interleave operations from each group arbitrarily. There are formulations of CGKA in the universal composability (UC) model [ACJM20, AJM20] which use group identifiers. In fact, composability guarantees in the UC model rely on the existence of unique, a priori established ‘session identifiers’ [Can01, KT11]; these can be established in practice via a central server or a distributed protocol [BLR04]. Note [CHK21] considers cross-group security for messaging but does not treat CGKA as a primitive formally.

Finally, we note that there are other small differences in the literature. The semantics of Tainted TreeKEM [KPPW+21] enable users to speculatively execute operations; our syntax
could be modified to support this. In CGKA work such as [ACDT20], it is typical to make a distinction between standard commit and welcome messages. We implicitly incorporate this distinction in our constructions, but avoid it in the primitive syntax for simplicity. In [AJM20, ACJM20], additional algorithms such as getKey are provided; we treat $k$ as a state variable instead, which is output by commit. As mentioned, it is also possible to conceive a policy algorithm as in [Bal21]. Groups in [ACDT21a, AJM20] are initially of size 1. Different works formalise the role of the PKI to different degrees: [ACDT21a, AJM20] consider an explicitly modelled PKI where users can choose their own possibly malicious keying material. Nevertheless, all works on CGKI hitherto formally assume that the PKI acts consistently for all users.

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 $A$-CGKA = (init, create, prop, commit, proc, prop-info, props) such that:

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

Given ID’s state $\gamma$ and gid, $\gamma$[gid].$G$ and $\gamma$[gid].$k$ are defined as in Definition 1, and $\gamma$[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 which concern administrative changes. Namely, an admin can propose to add another admin to the group of administrators, revoke the admin capabilities from a party, or update their administrative key material, 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 will ensure this holds throughout execution. Thus, the group administrators are always a subset of the group members. We take this approach following previous CGKAs [ACDT20, KPPW+21, ACJM20] and group messaging protocols [BBM+, ACDT21a] where only group members can perform commits or make changes in the group. In these works, it is impossible for an external user to administrate a group, since external commits are not permitted. We elaborate on this in Section 6.
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 6.3.

3.3 Correctness

Due to their similarity, we define the correctness of CGKA and A-CGKA together. Correctness of an (A)-CGKA scheme \((A)-\text{CGKA}\) under the notion \(\text{CORR}_{(A)}\) is defined by game \(\text{CORR}^A_{(A)}\text{-CGKA}_{\text{corr}}\) played by adversary \(A\) in Figure 2. The main properties captured by the game are the following:

- **View consistency:** All users who transition to the same epoch (i.e. which process the same sequence of commit messages) have the same group view (i.e., \(G, G^*\) and key \(k\)).

- **Message processing:** The group structure \((G/G^*)\) and \(k\) can only be modified due to calls to proc.

- **Forking states:** If the group is partitioned into subgroups that process different sequences of commit messages (thus leading to different group views), the game ensures that members in each partition have consistent views.

- **Multiple groups:** The adversary may create groups via \(O\text{-Create}\) on behalf of different users, and interact with different IDs in multiple groups.

Separately, we ensure that a user’s state is not modified whenever a particular algorithm call fails. As observed for two-party messaging [BSJ+17], we require incorrect inputs to not affect the functionality of the protocol, and in particular to not cause a denial of service.

Overview. The game starts by setting up several public dictionaries. The main two are \(ST[\cdot]\), which is a dictionary indexed by ID which keeps the states of each of the users ID throughout the game; and \(T[\cdot]\), which keeps all control messages and proposals generated by the A-CGKA algorithms. A message \(T\) stored in \(T\) is indexed by the corresponding gid, epoch, message type (‘prop’, ‘com’ or ‘vec’, standing for proposal, commit, or proposal vector respectively), and a message counter prop-ctr or com-ctr. After initialization, we let the adversary \(A\) interact with the oracles with respect to multiple groups. The variable \(\text{win}\) is set to 1 if one of the reward clauses is true, which leads to \(A\) winning given the (optional) predicate \(C_{\text{corr}}\) is also true when \(A\) finishes executing.

Epochs. Control messages output by successful create and commit calls are labelled uniquely by the challenger. For correctness, an epoch is a pair \((t, c)\), where \(t\) is an integer relative to a particular group and party which increments upon each successful proc call while in the group, and \(c\) is the value of the global variable com-ctr when the corresponding control message was output. For a given group, each party’s epoch value is stored in \(ep[\cdot]\) and initialised to \((-1, -1)\), and is set to \(\bot\) when they leave a group. To this end, we model correctness in the presence of
Figure 2: Correctness game for (A)-CGKA with respect to predicate $C_{com}$. Highlighted code is executed only when considering an A-CGKA. Note that when reward $P$ is true for predicate $P$, the variable $\text{win}$ is set to 1.
an adversary who maintains arbitrary network partitioning, so long as they provide a consistent view of messages to parties in each ‘partition’.

**Group consistency.** We enforce that, for each group member, that each group is only (possibly) updated upon a successful call to \texttt{proc} (via \texttt{CheckSameGroupState}). For A-CGKA, we ensure that, for a group gid, \(\emptyset \neq G^* \subseteq G\) must hold at all times. In \texttt{proc}, we enforce that all users who transition to the same epoch have the same view of the group and set of admins when relevant (via \texttt{UpdateView}). The dictionary \texttt{ep-view} stores the state of the first party who transitions to a given epoch \((t, c)\) for a gid. We also require that, even if there are multiple calls to \texttt{create}, only one of them is processed by group members (i.e. states do not fork from the initial epoch). To achieve this, the variable \texttt{first-crt} tracks the commit number of the first successfully processed create message for a given gid. This check is made in \textit{UniqueCreatePerGID}.

**Key partnering.** Note that new epoch keys are derived upon successful \texttt{proc} calls, and that a new key \(k\) (for group members) is always derived in this case. Correctness ensures that all users who transition to the same epoch derive the same key \(k\). The consistency between the key output by \texttt{commit} and the actual epoch keys is also verified. Moreover, whenever a user derives a key \(k \neq \bot\), they must be a group member (line 11 of \texttt{O Deliver}).

**Liveness.** We enforce that some algorithms, such as \texttt{prop}, always succeed on ‘valid’ input, since without such a check, a (A)-CGKA with algorithms that always fail is considered correct. One example of a liveness check is an equality check between proposals output by \texttt{props} and the input proposals in \texttt{O Commit}. Since the precise semantics of (A)-CGKA vary between applications, additional checks are delegated to a correctness predicate \(C_{corr}\) which, in part, parameterises the correctness game. In the previous example, \(C_{corr}\) may depend on the protocol policy for the application of proposals. Without extra checks, the predicate should be set to \(C_{corr} = \text{true}\).

**Definition 3** (\textit{CORR} \((A)\)-CGKA). A CGKA (resp. A-CGKA A-CGKA) is correct w.r.t. a predicate \(C_{corr}\) if, for all 1\(^\lambda\) and all computationally unbounded adversaries \(\mathcal{A}\), it holds that:

\[
\Pr[\text{CORR} \((A)\)-CGKA, \(C_{corr} \)(1\(^\lambda\)) = 1] = 0
\]

where the probability is taken over the choice of the random coins of the challenger and adversary.

Note that this notion can easily be relaxed to consider correctness that holds with high probability w.r.t a computationally-bounded adversary (e.g. to capture decryption failures from lattice-based cryptography). We compare our correctness notion to related work in the introduction.

### 3.4 Security

A-CGKA is a primitive that extends the functionality of a CGKA to provide secure administration, but whose end purpose is that the members of a given group derive a common group key. Therefore, any A-CGKA construction must satisfy at least ‘standard’ CGKA security (key indistinguishability).

The main additional goal of A-CGKA over standard CGKA is to prevent unauthorized (standard) users from deciding on changes to a group, capturing the security of the group
evolution. Note that an A-CGKA in which the adversary fully controls a standard group member is not secure with respect to key indistinguishability, but it should still be secure with respect to group evolution. We define (A)-CGKA security in Definition 4.

**Definition 4** (Security of (A)-CGKA). A CGKA CGKA (resp. A-CGKA A-CGKA) is \((t, q, \epsilon)\)-secure w.r.t. the predicates \(C_{cgka}\), \((C_{adm}, C_{forgery})\) if, for any adversary \(A\) limited to \(q\) oracle queries and running time \(t\), the advantage of \(A\) in the \(\text{KIND}_{(A)}-\text{CGKA}\) game (Figure 3) given by

\[
\Pr[\text{KIND}_{(A)}\text{-CGKA}, C_{cgka}, \text{(Cadm, Cforgery)}](1^\lambda) = 1 - \frac{1}{2}
\]

is bounded by \(\epsilon\), where the probability is taken over the choice of the challenger and adversary's random coins.

### 3.4.1 Overview

At its core, the security game in Figure 3 is a key indistinguishability game that captures the security of the common group secret for a single group (we implicitly assume a fixed gid), extending the game in [ACDT20]. The game considers a partially active adversary 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 \(\text{proc}\) is called. The main consequence of this is that injected proposals cannot be included in commits; nevertheless, the adversary can make commits on arbitrary vectors of proposals created via \(O_{\text{Prop}}\).

Informally, the adversary can win the game if it plays a clean game where it either 1) correctly guesses the challenge bit by distinguishing between correct and uniformly sampled keys or (for A-CGKA) 2) manages to forge a message which, after being processed by a user, changes its view of \((G, G^*)\). We assume that the dependency on the PKI is implicit in the game; we describe the PKI functionality we assume for each protocol as they are introduced. A detailed description follows.

### 3.4.2 Epochs

Messages output by successful create, commit, and prop calls are uniquely labelled by the challenger via counters prop-ctr, com-ctr. Whenever such a call is made, the corresponding messages are stored in variable \(T\) with (incremented) last argument ++com-ctr. The evolution of the group after parties process such control messages is modelled using epochs (differing from the epochs considered for correctness), which are each represented as an integer \(t_s\) (for CGKA) or a pair of integers \((t_s, t_a)\) (for A-CGKA). The standard epoch \(t_s\) represents the time between two successive key evolutions, where 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 (i.e. between two sequences of simultaneous admin updates, adds and/or removals).

For CGKA, epochs \((t_s)\) advance every time a commit is processed. For A-CGKA, \(t_s\) advances if the commit type \(\text{com-type} \in \{\text{std, both}\}\) and \(t_a\) advances if \(\text{com-type} \in \{\text{adm, both}\}\). Group members can be in different epochs, captured by the variable \(\text{ep}[\text{ID}]\) which stores the current epoch pair for a given group member \(\text{ID}\). If a participant \(\text{ID}\) is not in the group, then \(\text{ep}[\text{ID}] = -1\) (CGKA) or \(\text{ep}[\text{ID}] = (-1, -1)\) (A-CGKA) holds.
Figure 3: Key indistinguishability (KIND) security game for (single-group) (A)-CGKA, parametrized by the $C_{cgka}$ and $C_{adm}$ predicates. Highlighted code is executed only when considering an A-CGKA.
Challenges. At any point in the game, the adversary can challenge with respect to a standard epoch $t_s$ by calling $O_{\text{Challenge}}$. In a challenge, the adversary is given the group key $K[t_s]$ 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 $C_{\text{cgka}}$ is true or, for A-CGKA, the adversary makes a forgery and the admin cleanness predicate is true. Cleanness ensures that no trivial attacks on the game are possible; we elaborate on this below.

Exposure mechanisms. In order to capture group key ratcheting (for FS and PCS), the adversary has two mechanisms to obtain secret group material: it can expose a user ID and reveal the group secret $k$. An exposure leaks the entire current state of ID stored in $ST[ID]$. We keep track of the specific epochs in which each ID was exposed using the $\exp[\cdot]$ variable. On the other hand, a reveal leaks the group key to the adversary on a specified epoch $t_s$ – in this case, $\text{chall} [t_s]$ is set to true to prevent the adversary from challenging on $t_s$ (conversely, the reveal fails when $\text{chall} [t_s] = \text{true}$).

Injections. For A-CGKA, the adversary can also win the game by successfully injecting a forged commit. An injection can be attempted by calling $O_{\text{Inject}}(\text{ID}, m, t_a)$, 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 $C_{\text{adm}}$ is not violated. The adversary wins the game if the forgery is accepted by any group member $\text{ID} \in G$ and if the forgery predicate $C_{\text{forgery}}$ that we define below is true. As discussed below, $C_{\text{adm}}$ captures administration security by excluding trivial attacks. An trivial scenario excluded by any predicate is when the adversary has exposed an administrator immediately before a forgery attempt.

3.4.3 CGKA forgery predicate

For A-CGKA, we define security under active attacks performed using the $O_{\text{Inject}}$ oracle with respect to a predicate $C_{\text{forgery}}$. The predicate we describe captures the fact that if admins have not been corrupted, then nonadmins can only make group changes for self-removes, i.e. when non-admins want to remove themselves. Moreover, we require that self-removes cannot be forged themselves (i.e. the only acceptable self-remove operations are those that were generated honestly). If there are no self-removes operations, then the predicate reduces to the fact that non-admins cannot affect changes in the group.

The predicate $C_{\text{forgery}}$ is defined as follows with respect to variables in $O_{\text{Inject}}$ and in the game in general. Suppose $m$ is input to $O_{\text{Inject}}$. Let $\tilde{P} = \text{props}(ST[\text{ID}], m)$. Consider $\tilde{P}' = \{P \in \tilde{P} : P' \in T[\text{ep}[\text{ID}], 'prop', \cdot] \land \text{prop-info}(ST[\text{ID}], P) = \text{prop-info}(ST[\text{ID}], P')\}$.\(^3\) Let $H = \{\text{ID} : (\text{gid}, \text{rem}, \text{ID}, \text{ID}) = \text{prop-info}(ST[\text{ID}], P) \land (P \in \tilde{P}')\}$ and $H* = \{\text{ID} : (\text{gid}, \text{remadm}, \text{ID}, \text{ID}) = \text{prop-info}(ST[\text{ID}], P) \land (P \in \tilde{P}')\}$. Then $C_{\text{forgery}}$ is true if and only if $(ST[\text{ID}], G \setminus H, ST[\text{ID}], G* \setminus H*) \neq (\gamma, G, \gamma, G*)$. If there are no self-removes, i.e. $H = H* = \emptyset$, this simplifies to the predicate $(ST[\text{ID}], G, ST[\text{ID}], G*) \neq (\gamma, G, \gamma, G*)$; let $C_{\text{forgery}}^*$ be this simplified predicate.

\(^3\)The effect of the equality check with respect to $\text{prop-info}$ is that a dishonest proposal $P'$ that has the same semantics as an honest proposal $P$ will not be considered a ‘forgery’ by $C_{\text{forgery}}$. 

18
∀ (i, ID, ctr ∈ (0, exp-ctr)) : q_i = C^{Challenge}(t^*_i),
(ID /∈ G[t^*_i]) ∨
( ∃ (t_i, c) : (tExp(ID, ctr), t_s < t_i ≤ t^*_i) ∧
hasUpd_{std}(ID, T[[t_i, .], t^{com}, c], T[[t_i, .], t^{vec}, c]) ∧
(C[[t_i, .]] = c) ) ∨ (t^*_i < tExp(ID, ctr), t_s).

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

3.4.4 CGKA cleanness predicate

The security game in Figure 3 is parametrized by two cleanness predicates, C_{cgka} and C_{adm}. The first predicate C_{cgka} follows approaches like [BRV20, ACDT20] to parametrize the security of the common (A)-CGKA key. Namely, this predicate excludes trivial attacks on the protocol, i.e., those that break security unavoidably such as exposing a user and issuing a challenge before its key material has been updated. Further, it captures the exact security of the protocol (with respect to key indistinguishability), which in our case comprises forward security and post-compromise security after updates. If an independent CGKA is used to construct an A-CGKA, the predicate C_{cgka} may mostly depend on the security of the CGKA. An example, as we show in later sections, is our second construction DGS.

A more fine-grained characterization of this predicate is to write it as C_{cgka} = C_{cgka-opt} ∧ C_{cgka-add}, where C_{cgka-opt} is an optimal, generic cleanness predicate that excludes only unavoidable trivial attacks, and C_{cgka-add} is an additional cleanness predicate that depends on the scheme and excludes other attacks. We define C_{cgka-opt} in a similar way to the safe predicate in [ACDT20]. 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_{exp} ≤ 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 oracle queries q_1, ..., q_n in the game.

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).

We have used the following auxiliary functions. The function tExp is such that tExp(ID, ctr) = exp[ID, ctr] if ∃ k : q_k = C^{Expose}(ID), and -1 (for CGKA) or (-1, -1) (for A-CGKA) otherwise. Given \overline{P}, the function hasUpd_{std}(ID, (T, com-type), \overline{P}) (sans com-type for CGKA) outputs true if either: (i) ID has processed a commit of his own, where com-type ∈ {std, both}, or (ii) ID is a user affected by an add, update, or removal proposal in \overline{P}.

3.4.5 Admin cleanness predicate

The second predicate C_{adm} models administration security. This predicate should be more permissive in some aspects than C_{cgka}, since a forgery attempt should be permitted even if the adversary knows the state of a (standard) group member. Following the approach above, we
can decompose $C_{adm} = C_{adm-opt} \land C_{adm-add}$.

$C_{adm-opt}$ is symmetric to $C_{cgka-opt}$ and excludes the following family of attacks: the adversary attempts a forgery on a member $ID'$ at an administrative epoch $t_a^*$ while having exposed the state of an administrator $ID \in G^*$ at an administrative epoch $t_{exp} \leq t_a^*$, 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 $ID'$ is in the challenge epoch specified by the adversary, i.e., $ep[ID'] = (\cdot, t_a^*)$. Notice that this predicate is unrelated to the common group secret and standard epochs $t_s$, and only relates to administration dynamics.

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

### 3.4.6 Limitations

Our security definition does not allow arbitrary message injections to participants. Thus, attacks on robustness are not captured by our security model. In particular, 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 [ACJM20, DDF21]. In any case, we note that confidentiality is not compromised under this family of attacks, as their main consequence is to “disconnect” users from the protocol (in particular, 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 some non-strongly robust variants of TreeKEM, such as the one used in MLS [BBM+] (which is now an RFC at the time of writing). Standard users can still attain FS and PCS guarantees, and in particular PCS when their update proposals are committed.

Among the broader family of group key agreement protocols, where long-lived sessions and PCS are not always considered, modelling fully active adversaries is common [PRSS21]. We also do not model authentication (we implicitly assume an incorruptible PKI) and randomness manipulation, and we do not explicitly model (via a no-deletion oracle or similar [ACDT21a]) parties who do not delete their state as instructed by the algorithm and are then exposed. We leave these for future work.

Multi-group security can be captured rather easily. The main difference (besides increased notation complexity introduced by the $gids$ as in Figure 2) appears in the state exposure oracle: exposing the state of a party implies a security loss in all groups that the party is a member of simultaneously. The feature is not included as our security proofs are in the single-group
setting.

Our $O^{\text{inject}}$ oracle does not allow the adversary to inject welcome messages. Of course, the adversary can always make a new group with whatever users it chooses. It is nonetheless possible to extend our security notion to allow for injections, such that the adversary can only create a group for ‘valid’ users, i.e. those who have registered their keys with the PKI (which would be checked when a user processes a welcome message).

4 Constructions

A first attempt of A-CGKA is to simply require group members to keep a list of administrators over time. Whenever an admin wants to make a commit, it can simply check whether the admin-changing proposals have been made by administrators, then commit them, and the other users will verify the admin condition upon processing. This approach is functional, but not secure in our model due to a lack of admin authentication. An adversary can easily forge a commit message and impersonate an admin unless this message is authenticated (for example signed). Many notions of CGKA security [ACDT20, KPPW+21] 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 ID $\in G^*$ maintains their own signature key pair $(ssk, spk)$. 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 each control message. Proposed changes to the group and to the administration are signed using an admin’s keying material.

4.1.1 Protocol

The IAS construction is presented in Figures 6, 7 and 8. The first figure describes the A-CGKA algorithms, and the second and third describe helper functions and auxiliary methods. We note that the algorithms defined in Figure 6 are incomplete without the helper functions; therefore, the construction spans the three figures.

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

21
All implemented A-CGKA algorithms, including \texttt{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.

In our functions in Figures 6 and 7, 8, we often omit the group identifier of the state to simplify presentation. We assume that $\gamma$ refers to $\gamma[gid]$ whenever $gid$ is a subject of the algorithm, such as when it is a parameter of the function, and sometimes omit $gid$ when it is clear from context. We note that our scheme nevertheless supports multiple groups.

**Randomness.** In our construction, 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 [BRV20]: without prior knowledge of $\gamma$ (and assuming it has a sufficient entropy), an adversary that reads or manipulates $r_0$ will not be able to derive a corresponding $r_i$ value. This is an additional feature that aims to maintain certain security properties in stronger adversarial models than considered in this work presently, and does not interfere with the rest of the protocol.

**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 $ID'$ 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 \texttt{getSpk} algorithm, which on input $(ID, ID', gid)$ for subject $ID$ and caller $ID'$ outputs $spk$ relevant to the context the call is made in. We also assume a method of the form \texttt{getSsk}(spk, ID, gid) that returns the $ssk$ associated to $spk$ when called by ID given they uploaded it. During protocol execution, parties upload signature key pairs $(ssk, spk)$ to the PKI via an abstract \texttt{registerKeys}(ID) method both in initialisation and during the two aforementioned scenarios.\footnote{This abstraction is made to reduce notational complexity.} Formally, the adversary is only exposed to \texttt{getSpk}; we assume the other functions are called as needed in the security game though.

### 4.1.2 Description

**Initialization.** Before the creation of a group, a participant starts by calling the \texttt{init} method, which initializes the state $\gamma$. In turn, \texttt{init} calls \texttt{CGKA.init} from the underlying CGKA to initialize its state $\gamma.s0$. $(ssk, spk)$ and $(ssk', spk')$ are two signature key pairs for group administration. The first pair is the valid admin signature key pair using during protocol execution, while the second pair stores updated keys after a commit or a key update operation is done by the participant but before it is processed (i.e. acts as a temporary variable). After successfully processing a commit message, the second key pair replaces the first.
init(1^3, ID; r_0)  
1: γ.s_0 ← CGKA.init(1^3, ID; r_0)  
2: γ.ME ← ID; γ.1^3 ← 1^3  
3: if γ[.]adminList[.] ← ⊥  // stores (ID, spk) pairs  
4: γ[.]ssk, γ[.]spk ← ⊥  // active admin key pair  
5: γ[.]ssk', γ[.]spk' ← ⊥  // temporary key pair  
6: registerKeys(ID)  // Upload keys to PKI

prop(gid, ID, type; r_0)  
1: P ← ⊥; (r_1, r_2, r_3, r_4) ← H_4(r_0, γ)  
2: if type = upd-adm  
   // Note if type = upd-adm, keys are updated  
3: require γ.ME ∈ γ.adminList  
4: P ← makeAdminProp(gid, type, ID; r_1, r_2)  
5: else (γ.s_0, P) ← CGKA.prop(γ.s_0, gid, ID, type; r_1)  
6: if (type = rem) ∧ (ID = ME) ∧ (ID /∈ γ.s_0.G')  
7: (ssk, spk) ← SigGen(γ.1^3; r_3)  
8: P ← (P, Sig(ssk', P; r_4))  
9: return P

commit(gid, P, com-type; r_0)  
1: require γ.ME ∈ γ.s_0.G  
2: require com-type ∈ {adm, std, both}  
3: (r_1, ..., r_8) ← H_4(r_0, γ)  
4: (P_0, P_1, Σ, admReq) ← propCleaner(gid, P)  
5: require verifyPropSig(P_0, Σ, P_1)  
6: if admReq ∨ (com-type ∈ {adm, both})  
7: require γ.ME ∈ γ.adminList  
8: if com-type ∈ {adm, both}  
9: C_A ← P_0  
10: if com-type ∈ {std, both}  
11: (C_0, W_0, adminList, k) ← c-Std(gid, P_0, P_1; r_1)  
12: require C_0 /∈ ⊥  
13: // generate new key pair and sign new spk  
14: (γ.ssk', γ.spk') ← SigGen(γ.1^3; r_2)  
15: T_C ← (comm', γ.ME, C_0, C_A, ⊥, γ.spk')  
16: if W_0 /∈ ⊥  // share updated admin list  
17: T_W ← (well', γ.ME, W_0, adminList)  
18: else T_W ← ⊥  
19: σ_T ← Sig(γ.ssk, (gid, T_C, T_W); r_3)  
20: else  // only self-removes - no admin sig  
21: (C_0, ⊥, ⊥, k) ← c-Std(gid, P_0, P_1; r_1)  
22: T_C ← (comm', γ.ME, C_0, ⊥, ⊥, Σ)  
23: T_W ← ⊥; σ_T ← ⊥  
24: return ((gid, T_C, T_W, σ_T), k)

create(gid, G, G'; r_0)  
1: require (γ.ME ∈ G') ∧ (G' ⊆ G)  
2: (r_1, r_2) ← H_2(r_0, γ)  
3: (γ.s_0, W_0) ← CGKA.create(γ.s_0, gid, G; r_1)  
4: if W_0 = ⊥ return ⊥  
5: adminList[.] ← ⊥  // this is not γ.adminList  
6: for ID ∈ G'  
7: adminList[ID] ← (ID, getSpk(ID, γ.ME))  
8: γ.spk' ← adminList[ME]  
9: γ.ssk' ← getGsk(γ.spk', ME)  
10: T_W ← ('well', γ.ME, W_0, adminList)  
11: return (gid, ⊥, T_W, Sig(ssk', (gid, ⊥, T_W); r_2))

proc(T)  
1: (gid, T_C, T_W, σ_T) ← T; acc ← false  
2: if (γ.ME /∈ γ[.]adminList) ∧ (∀x ∈ γ[.]adminList)  
3: (msg-type, · · · ) ← T_W  
4: require msg-type = 'well'  
5: acc ← p-Well(gid, T_W, σ_T)  // Welcome helper  
6: else if γ[.]ME ∈ γ[.]adminList  
7: (msg-type, ⊥, C_A, · · · ) ← T_C  
8: require msg-type = 'comm'  
9: for σ : (P, ID, σ) ∈ Σ  
10: if ¬Ver(getSpk(ID, ME), σ, P)  
11: ∨ (ID ∈ γ[.]adminList) return false  
12: if σ_C = ⊥  // no sign - check only self-removes  
13: (γ', acc) ← CGKA.proc(γ[.]s_0, C_0)  
14: SR ← {ID : (·, ID) ∈ Σ}  
15: if ¬acc ∨ γ'[.]s_0.G ∪ SR /∈ γ[.]s_0.G  
16: return false  
17: γ[.]s_0 /∈ γ'; return true  
18: if ¬(ID ∈ γ[.]adminList) ∧ 
19: (Ver(γ[.]adminList[.]ID, spk, (gid, T_C, T_W), σ_T)))  
20: return false  // verification failed  
21: acc ← p-Comm(gid, T_C)  // Admin commit helper  
22: return acc

prop-info(P)  
1: if P is of the form (P, σ)  // self-removes  
2: (P, σ) ← P  
3: if P is a CGKA proposal  
4: (P.gid, P.type, P.ID, P.ID') ← 
   CGKA.prop-info(γ.s_0, P)  
5: else if P is an admin proposal  
6: (P.gid, P.type, P.ID, P.ID', ⊥, ⊥) ← P  
7: return (P.gid, P.type, P.ID, P.ID')

Figure 6: Individual admin signatures (IAS) construction of an A-CGKA, built from a CGKA, a signature scheme 𝑆 = (SigGen, Sig, Ver), and 𝑛-PRFs 𝐻_𝑛 : 𝑅 × 𝑆𝑇 → 𝑅^𝑛 for 𝑛 ≤ 4, randomness space 𝑅 and state space 𝑆𝑇. The state values representing the group, the admins, and the group key are assigned as: 𝜋[.]gid.G = 𝜋[.]gid.s_0.G, 𝜋[.]gid.G' = ⊃ { gid[.]adminList[.]ID }, and 𝜋[.]gid.k = 𝜋[.]gid.s_0.k.
VALID_P

// Predicate checks validity of admin proposal
1: (P gid, P type, P ID, P ID') \rightarrow \propinfo(P)
2: S1 := (P gid = gid) // correct group
3: S2 := (P ID \in \gamma gid, G) // ID member
4: S3 := (P ID' \in \gamma gid, G') // ID' admin
5: C1 := (P type = adm-adm)
6: S4 := (P ID \in \gamma gid, G) \cap ID admin
7: C2 := (P type = adm-adm)
8: return S1 \land S2 \land S3 \land C1 \land S4

makeAdminProp(gid, type, ID; r1, r2)
1: P0 := ⊥
2: if type = adm-adm
3: spk = getSsk(ID, γ, ME)
4: P0 := (gid, type, ID, γ, ME, spk)
5: else if type = rem-adm
6: P0 := (gid, type, ID, γ, ME, ⊥)
7: else if type = upd-adm
8: if (γ, spk, γ) ≠ (∞, ⊥)
9: return (γ, spk', γ, spk) // only update per epoch
10: (γ, spk', γ, spk') := SigGen(γ, ID; r1)
11: P0 := (gid, type, γ, ME, γ, ME, γ, spk')
12: else return ⊥
13: return (P0, Sig(γ, spk, P0; r2))

c-Std(gid, P0, P0, ⊥)
1: (γ, s0, C0, W0, k) := CGKA.commit(γ, s0, gid, P0; r1)
2: if W0 ≠ ⊥ // list for new users only
3: adminList' := updAL(γ, adminList, P0)
4: return (C0, W0, adminList', k)
5: else return (C0, ⊥, ⊥, k)

verifyPropSig(gid, P0, P0; P0)
1: for (P ID, σ) ∈ Σ
2: spk := getSsk(ID, γ, ME)
3: if ~Ver(spk, P, σ) ∨ P \notin P0 ∨ adminList[ID] ≠ ⊥
4: return false
5: for (P, σ) ∈ P0:
6: (⊥, ⊥, ⊥, ID) := \propinfo(P)
7: spk := adminList[ID].spk
8: if ~Ver(spk, P, σ) \land \propinfo(P)
9: return false
10: return true

propCleaner(gid, P0)
1: admReq := false; P0, P0, Σ := []
2: for P ∈ P0:
3: (γ, spk, g0, acc) := CGKA.proc(γ, spk, σ, W0)
4: acc := acc \land Ver(getSpk(γ, ME, ID), (γ, ⊥, T0, σ))
5: if acc = γ gid, adminList := adminList
6: γ, acc := acc \land (adminList[ID] = ⊥)
7: γ, spk := adminList[ME].spk
8: return acc

updAL(adminList, P0)
1: for P ∈ P0:
2: (γ, spk, g0, acc) := CGKA.proc(γ, spk, σ, W0)
3: acc := acc \land Ver(getSpk(γ, ME, ID), (γ, ⊥, T0, σ))
4: if acc = γ gid, adminList := adminList
5: γ, spk := spk
6: γ, acc := acc \land (adminList[ID] = ⊥)
7: adminList[ID] := (ID, spk)
8: if type = rem-adm
9: adminList[ID] := ⊥
10: if (ID = γ, ME)
11: γ, acc := acc \land (⊥, ⊥)
12: return adminList

Figure 7: Helper functions for the IAS construction in Figure 6 (part I).
p-Comm\((\text{gid}, T_C)\)

\[
\begin{align*}
1 & : (\ldots, \text{ID}, C_0, C_A, \Sigma, \text{spk}) \leftarrow T_C \\
& \quad \quad \text{// check signatures in proposals} \\
2 & : \text{if } C_A \neq \bot \\
3 & : \quad \text{if } \neg \text{verifyPropSigs}(C_A) \text{ return false} \quad \text{// apply commit} \\
4 & : \quad \text{if } C_0 \neq \bot \\
5 & : \quad (\gamma', \text{acc}) \leftarrow \text{CGKA.proc}(\gamma, s_0, C_0) \\
6 & : \quad \text{if } \text{acc} = \text{false} \text{ return false} \\
7 & : \quad \text{if } \gamma.\text{ME} \not\in \gamma.G \quad \text{// user removed} \\
8 & : \quad \gamma[\text{gid}] \leftarrow \bot \quad \text{// reinitialize state (only for gid)} \\
9 & : \quad \text{else } \gamma[\text{gid}].s_0 \leftarrow \gamma' \\
10 & : \quad \text{// set temporary updated keys} \\
11 & : \quad \text{if } (\text{ID} = \gamma.\text{ME}) \lor (\exists P \in C_A : P.\text{ID} = \gamma.\text{ME}) \\
12 & : \quad (\gamma.\text{ssk}, \gamma.\text{spk}) \leftarrow (\gamma.\text{ssk}', \gamma.\text{spk}') \\
13 & : \quad \gamma.\text{adminList} \leftarrow \text{updAL}(\gamma.\text{adminList}, C_A) \\
14 & : \quad \gamma.\text{adminList[ID]}.\text{spk} \leftrightarrow \text{spk} \quad \text{// committer’s key} \\
15 & : \quad \text{return true}
\end{align*}
\]

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

\[
\text{// This method can be extended to other policies}
\]

\[
\begin{align*}
1 & : \text{numAdmins} \leftarrow |G^*| \\
2 & : \text{for } P \in [\vec{P}_0, \vec{P}_A] \\
3 & : \quad (\text{gid}, P.\text{type}, P.\text{ID}, P.\text{ID}') \leftarrow \text{prop-info}(P) \\
4 & : \quad \text{if } P.\text{type} = \text{rem} \quad \text{// If duplicates, removal prevails} \\
5 & : \quad \text{delete any other } P' \text{ s.t. } P.\text{ID} = P'.\text{ID} \\
6 & : \quad \text{except for rem-adm proposals} \\
7 & : \quad \text{else if } P.\text{type} = \text{rem-adm} \\
8 & : \quad \text{delete any other admin } P' \text{ s.t. } P.\text{ID} = P'.\text{ID} \\
9 & : \quad \text{else if } P.\text{type} = \text{rem-adm}, \text{ numAdmins}-- \\
10 & : \quad \text{else if } P.\text{type} = \text{add-adm}, \text{ numAdmins}++ \\
11 & : \quad \text{require numAdmins} \geq 1 \quad \text{// Ensures } \emptyset \neq G^* \subseteq G \\
12 & : \quad \text{return } (\vec{P}_0, \vec{P}_A)
\end{align*}
\]

\text{props}(T)

\[
\text{// Supports non-welcome control messages}
\]

\[
\begin{align*}
1 & : (T_C, T_W, \sigma_T) \leftarrow T \\
2 & : \vec{P}_0 \leftarrow \text{props}(T_C, C_0); \vec{P}_A \leftarrow T_C, C_A \\
3 & : \text{return } \vec{P}_0||\vec{P}_A
\end{align*}
\]

\textbf{Figure 8:} Helper functions for the IAS construction in Figure 6 (part II).

\begin{itemize}
  \item **Group creation.** The create algorithm creates the group \text{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 \((\text{ID}, \text{spk}_\text{ID})\) for parties \(\text{ID} \in G^*\). 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 \((\gamma.\text{ssk}', \gamma.\text{spk}')\).
  
  \item **Proposals.** Any group member can use \text{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 \text{type} = \text{upd-adm}. In the case of an add-adm proposal to promote \text{ID} to admin status, the proposer \(\gamma.\text{ME}\) retrieves a public signature key \text{spk} of \text{ID} from the PKI using getSpk. In the case of an \text{rem} proposal where \text{ID} = \text{ME} (i.e., a self-remove), the caller samples a new signature key pair, registers the public key \text{spk} with the PKI and signs their proposal.
  
  The \text{prop-info} method simply retrieves the main information of a proposal. As mentioned in the previous section, it could be adapted to support CGKAs and A-CGKAs where proposals are encrypted\(^5\) (under some key derived from the group key, for instance).
\end{itemize}

\(^5\)Special precaution must be taken with respect to security when proposals are encrypted under the CGKA key, as the adversary gains access to multiple additional ciphertexts which can result in a security loss.
**Commits.** The commit algorithm can only be called by group administrators (except for the special case in which only key updates and self-removes are proposed, when standard users can commit), and performs the following actions:

1. Clean the input vector of proposals $\vec{P}$, ensuring that they are well-formed. This is done via the propCleaner algorithm, which in turn calls the enforcePolicy method. For security reasons, we adopt the main features of the MLS policy (removing duplicates and prioritizing removals) in our construction [BBM+], but extensions to this policy can be implemented. In addition, we verify the legitimacy of the admin proposals and the fact that self-remove proposals are correctly signed via verifyPropSigs. Then, the predicate $\text{VALID}_P$ verifies that the gid matches, that added users (respectively admins) do not belong to $G$ (resp. $G^*$), that removed users do belong to $G$ (resp. $G^*$), and that the proposer is an admin. Finally, we ensure that all 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. We split the CGKA commit in two components $C_0$ and $W_0$ as is usual in the literature [ACDT20, AJM20, KPPW+21, ACJM20]. If the CGKA does not allow for this, it is easy to modify the protocol without compromising security.

3. Generate a new (temporary) administrative signature key pair ($\gamma.\text{ssk}', \gamma.\text{spk}'$).

4. Produce the final control message $T$ which includes the new 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 $\gamma.\text{ssk}$.

The props method, given a commit, retrieves the list of proposals that it implements; this simply calls the underlying CGKA.props algorithm and combines the output with the list of admin updates directly contained in a well-formed commit message.

**Processing control messages.** The proc method takes a control message $T$ as input and updates the state accordingly. The algorithm returns an acceptance bit acc which is 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, attempts to process the message via the underlying CGKA, and updates the state given this succeeds. 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).

---

6This division is made for clarity, but it may be used to improve efficiency too. Namely, the welcome part $W_0$ of a commit message does not need to be processed by existing group members, so in principle $C_0$ can be sent only to these. In A-CGKAs such as IAS, this can also be applied if signatures are handled carefully.
4.1.3 Features

We first note that the IAS protocol can be built over any CGKA. Since signatures are often already present in CGKAs such as [AJM20], the extension from CGKA to A-CGKA can be more direct (and thus incur less overhead) than presented here. This also holds true for any group messaging scheme, such as the administrated MLS extension we describe in Section 4.3.

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), although, as previously discussed, adversarial group members may deny service if the underlying CGKA is not robust. We also enforce that standard users cannot propose administrative changes (even if these could be later ignored by admins), and similarly can be allowed as required by an application.

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 by the message delivery service. Separately, additional protections (i.e. checking members are registered on the PKI) are needed to ensure that parties are not invited to fake groups where the list of group administrators is forged.

4.1.4 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 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 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 [ACDT20].

Optimal security can be easily achieved by replacing regular signatures with forward-secure signatures [BM99]. 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. The reason is that they authenticate admin messages using an admin signature key that is shared among all admins. Notably, group members do not need to keep track of an administrator list; admins implicitly track this via their CGKA.

### 4.2.1 Protocol

The DGS protocol is introduced in Figures 9 and 10. 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 either for performance reasons or if, for instance, stronger FS and PCS guarantees are required for the administrative CGKA. For simplicity of exposition, DGS as written does not support self-signed removal operations that non-admins can commit directly, but we note that the technique to implement them is identical to IAS.

#### States.

Each party stores γ.s₀, corresponding to the primary CGKA, as well as γ.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 gid₁ and gid∗₂ are distinct for all gid₁, gid₂. Besides these fields, the state includes the administrative public key γ[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 γ[gid].G = γ.s₀[gid].G, γ[gid].G* = γ.sA[gid∗].G, and γ[gid].k = γ.s₀[gid].k.

#### Authentication.

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 getSpk(gid) (which is only required by DGS for incoming group members). Authentication could be implemented while ensuring k-anonymity such that a member authenticates his group membership but not his identity; such a feature cannot be provided by IAS without modification.

### 4.2.2 Description

#### Initialization.

The init procedure calls the CGKA.init and CGKA∗.init algorithms to initialize γ.s₀ and γ.sA, respectively, and sets γ[.].spk, γ[.].ssk ← ⊥.

#### 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 W₀ and W₁, 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.
Figure 9: Dynamic group signature (DGS) construction of an $A$-CGKA, built from two (possibly different) CGKAs, a signature scheme $\mathcal{S} = (\text{SigGen}, \text{Sig}, \text{Ver})$, and $n$-PRFs $H_n : R \times ST \to R^m$ for $n \leq 4$, randomness space $R$ and state space $ST$. The state values representing the group, the admins, and the group key are assigned as: $\gamma[gid].G = \gamma[gid].s_0.G$, $\gamma[gid].G^* = \gamma[gid].s_A.G$, and $\gamma[gid].k = \gamma[gid].s_0.k$. 

$$init(1^\lambda, ID)$$

1. $\gamma.s_0 \leftarrow $$CGKA.init(1^\lambda, ID)$
2. $\gamma.s_A \leftarrow $$CGKA*.init(1^\lambda, ID)$
3. $\gamma.ME \leftarrow ID; \ 1^\lambda \leftarrow 1^\lambda$
4. $\gamma[\cdot].spk, \gamma[\cdot].ssk \leftarrow \bot$

$$create(id, G, G^*; r_0)$$

1. require $(\gamma.ME \in G^*) \land (G^* \subseteq G)$
2. $(r_1, r_2, r_3, r_4) \leftarrow H_4(r_0, \gamma)$
3. $(W_0, \gamma.s_0) \leftarrow CGKA.create(\gamma.s_0, gid, G; r_1)$
4. $(W_A, \gamma.s_A) \leftarrow CGKA*.create(\gamma.s_A, gid^*, G^*; r_2)$
5. $T_{CR} \leftarrow \text{create'}, W_0, W_A$
6. $(\gamma[\cdot].spk, \gamma[\cdot].ssk) \leftarrow \text{SigGen}(\gamma.1^\lambda; r_3)$
7. $\sigma_T \leftarrow \text{Sig}(\gamma.ssk, (\gamma, T_{CR}); r_4)$
8. return $(\gamma, T_{CR}, \bot, \bot, \sigma_T)$

$$prop(gid, ID, type; r_0)$$

1. $(r_1, r_2) \leftarrow H_2(r_0, \gamma)$
2. if type = adm
3. require $\gamma.ME \in \gamma.s_A.G$
4. $(\gamma.s_A, P_0) \leftarrow CGKA*.prop(\gamma.s_A, gid^*, ID, type; r_1)$
5. $P \leftarrow (P_0, \text{Sig}(\gamma.ssk, P_0; r_2))$
6. else
7. $(\gamma.s_0, P) \leftarrow CGKA.prop(\gamma.s_0, gid, ID, type; r_1)$
8. return $P$

$$prop-info(P)$$

1. if $P$ is a CGKA proposal
2. $(P.gid, P.type, P.ID, P.ID') \leftarrow \text{CGKA.prop-info}(\gamma.s_0, P)$
3. else if $P$ is an admin proposal
4. $(P.gid, P.type, P.ID, P.ID') \leftarrow \text{CGKA*.prop-info}(\gamma.s_A, P)$
5. $P.type \leftarrow P.type||\text{adm}$
6. return $(P.gid, P.type, P.ID, P.ID')$

$$commit(gid, \vec{P}, \text{com-type}; r_0)$$

1. require $\gamma.ME \in \gamma.G$
2. require com-type $\in \{\text{adm, std, both}\}$
3. $(r_1, r_2, r_3) \leftarrow H_3(r_0, \gamma)$
4. $C_0, C_A, W_0, W_A, k \leftarrow \bot$
5. $(\vec{P}, \vec{P}_A, \text{admReq}) \leftarrow \text{propCleaner}(\gamma, \vec{P})$
6. if admReq $\lor$ (com-type $\in \{\text{adm, both}\}$)
7. require $\gamma.ME \in \gamma.G^*$
8. $(\text{ssk}, \text{spk}) \leftarrow \text{getSigKey}(\gamma.sA.k) \ \# \text{old keys}$
9. if com-type $\in \{\text{adm, both}\}$ $\# \text{update spk}$
10. $(\text{spk}, C_A, W_A) \leftarrow c\text{-Adm}(\gamma, \vec{P}_A; r_1)$
11. if com-type $\in \{\text{std, both}\}$
12. $(C_0, W_0) \leftarrow c\text{-Std}(\gamma, \vec{P}_0, k; r_2)$
13. $T_C \leftarrow \text{create'}(\gamma, C_0, C_A, W_A, \text{spk})$
14. $T_W \leftarrow \text{create'}(\text{wel}, W_0, \text{spk})$
15. $\sigma_T \leftarrow \text{Sig}(\gamma.ssk, (\gamma, T_{CR}, T_W); r_3)$
16. else $\# \text{can be done by non-admins}$
17. $(C_0, \bot, k) \leftarrow c\text{-Std}(\vec{P}_0; r_1)$
18. $T_C \leftarrow (\text{comm'}, C_0, C_A, W_A, \text{spk})$
19. $T_W, \sigma_T \leftarrow \bot$
20. if $k = \bot \ k \leftarrow \gamma.s_0.k$
21. return $((\bot, T_C, T_W, \sigma_T), k)$

$$proc(T)$$

1. $(\gamma, T_{CR}, T_W, T_C, \sigma_T) \leftarrow T; \ \text{acc} \leftarrow \text{false}$
2. if $T_{CR} \neq \bot$
3. if $\gamma.ME \in \gamma[gid].s_0.G$ return false
4. acc $\leftarrow p\text{-Create}(\gamma, T_{CR}, \sigma_T)$
5. else if $(\gamma.ME \notin \gamma[gid].G) \land (T_W \neq \bot)$
6. acc $\leftarrow p\text{-Well}(\gamma, T_C, T_W, \sigma_T)$
7. else if $(\gamma.ME \notin \gamma[gid].G) \land (T_C \neq \bot)$
8. acc $\leftarrow p\text{-Comm}(\gamma, T_C, T_W, \sigma_T)$
9. return acc
c-Adm\((\text{gid}, \vec{P}_A; r_1)\)
1: for \(P \in \vec{P}_A\)
2: if \(P\) type = add-adm
3: require \(P; ID \in \gamma.G\)
4: \((C_A, W_A, k, \gamma.sA) \leftarrow \text{CGKA}^*\text{commit}(\gamma.sA, \text{gid}^*, \vec{P}_A; r_1)\)
5: if \(C_A = \perp\) return \(\perp\)
6: \((\text{ssk}, \text{spk}) \leftarrow \text{getSigKey}(k)\)
7: return \((\text{spk}, C_A, W_A)\)

propCleaner\((\text{gid}, P)\)
1: \(\text{admReq} \leftarrow \text{false}; \vec{P}_0, \vec{P}_A \leftarrow []\)
2: for \(P \in \vec{P}\)
3: \((\text{gid}', P\text{.type}, P; ID) \leftarrow \text{prop-info}(P)\)
4: if \(\text{gid}' = \text{gid}^* \land P\text{.type} = \text{adm} \land \text{IAS.VAlID}\)
5: \(\vec{P}_A \leftarrow [\vec{P}_A, P]\)
6: \text{admReq} \leftarrow \text{true}
7: else if \(\text{gid}' = \text{gid}\)
8: \(\vec{P}_0 \leftarrow [\vec{P}_0, P]\)
9: if \(P\text{.type} \in \{\text{adm, rem}\}\)
10: \(\text{admReq} \leftarrow \text{true}\)
11: // \text{admin rem from } G \implies \text{rem also from } G^*
12: \(\text{if } (P\text{.type} = \text{rem}) \land (P; ID \in \gamma.G^*)\)
13: \(P' \leftarrow \text{CGKA}^*\text{.prop}(\gamma.sA, \text{gid}^*, P; ID, \text{rem})\)
14: \(\vec{P}_A \leftarrow [\vec{P}_A, P']\)
15: \((\vec{P}_0, \vec{P}_A) \leftarrow \text{enforcePolicy}(\vec{P}_0, \vec{P}_A)\)
16: return \((\vec{P}_0, \vec{P}_A, \text{admReq})\)

getSigKey\((r)\)
1: \((\text{ssk}, \text{spk}) \leftarrow \text{SigGen}(1^\lambda; H_\text{oa}(r))\)
// Deterministic key generation from \(r\)
// Random oracle \(H_\text{oa}\)
2: return \((\text{ssk}, \text{spk})\)

enforcePolicy\((\vec{P}_0, \vec{P}_A)\)
1: // As in IAS
2: return \((\vec{P}_0, \vec{P}_A) \leftarrow \text{IAS.enforcePolicy}(\vec{P}_0, \vec{P}_A)\)

p-Well\((\text{gid}, T_C, T_W, \sigma_T)\)
1: \((\text{msg-type}, W_0, \text{spk}) \leftarrow T_W\)
2: require \text{msg-type} = \textquote{well} \land \text{getSpk}(\text{gid}) = \text{spk}
3: if \neg \text{Ver}(\text{spk}, (\text{gid}, T_C, T_W), \sigma_T) \text{return false}\n4: \((\gamma', \text{acc}) \leftarrow \text{CGKA.proc}(\gamma.s0, W_0)\)
5: if \neg \text{acc} \text{return false}\n6: \(\gamma\text{[gid]} s0 \leftarrow \gamma'\)
7: \(\gamma\text{[gid]} \text{spk} \leftarrow \text{spk}\)
8: return \text{true}

p-Comm\((\text{gid}, T_C, T_W, \sigma_T)\)
1: \((\text{msg-type}, C_0, C_A, W_A, \text{spk}) \leftarrow T_C\)
2: require \text{msg-type} = \textquote{comm}
3: \(\gamma' \leftarrow \gamma.sA\)
4: if \(\sigma_T = \perp \land \text{no sig} \implies \text{check no changes to } G\)
5: \((\gamma', \text{acc}) \leftarrow \text{CGKA.proc}(\gamma.s0, C_0)\)
6: if \neg \text{acc} \lor (\gamma'[\text{gid}], G \neq \gamma[\text{gid}], G)\)
7: return false\n8: \(\gamma\text{[gid]} s0 \leftarrow \gamma'\)
9: return \text{true}\n10: else if \neg \text{Ver}(\gamma[\text{gid}], \text{spk}, (\text{gid}, T_C, T_W), \sigma_T)\)
11: return false\n12: if \gamma.ME \in \gamma.G^*
13: \((\gamma', \text{acc}) \leftarrow \text{CGKA}^*.\text{proc}(\gamma.sA, C_A)\)
14: if \neg \text{acc} \text{return false}\n15: else if \(W_A \neq \perp\)
16: \((\gamma', \perp) \leftarrow \text{CGKA}^*.\text{proc}(\gamma.sA, W_A)\)
17: if \(C_0 \neq \perp\)
18: \((\gamma^1, \text{acc}^1) \leftarrow \text{CGKA.proc}(\gamma.s0, C_0)\)
19: if \neg \text{acc}^1 \text{return false}\n20: \(\gamma\text{[gid]} s0 \leftarrow \gamma^1\)
21: if \gamma.ME \notin \gamma^1.G \land \text{removed user}\n22: \(\gamma\text{[gid]} \leftarrow \perp\)
23: return \text{true}\n24: \(\gamma\text{[gid]} \text{spk} \leftarrow \gamma'; \gamma\text{[gid]} \text{spk} \leftarrow \text{spk}\)
25: return \text{true}

\textbf{Figure 10:} Helper functions for the DGS construction in Figure 9 with respect to random oracle \(H_\text{oa}\).
Proposals. The prop algorithm generates a proposal message $P$ by using CGKA.prop the input type is standard and CGKA*.prop when it is administrative (i.e. of the form $\ast$-adm). As in IAS, a validity check on the caller ID' and the target ID of the proposal is made using the VALID$_P$ predicate. Administrative proposals are signed with $\gamma$.spk; this is done to protect against insider adversaries that may re-send previously crafted administrative proposals (i.e., those that are legitimate but correspond to a previous epoch), or even create new ones if these are sent in plaintext. The prop-info algorithm is fully based on the respective CGKA's prop-info algorithms (which may not necessarily output ID' if anonymous proposals are allowed); we assume props is likewise inherited from the underlying CGKAs.

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 $k_{adm}$, that is used to update the admin key pair ($ssk'$, $spk'$).

The new admin key $spk'$ 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 $spk'$), the committer signs the whole commit message including $C_A$, $C_0$ and $spk'$ with the old admin key $\gamma$.ssk. In addition, the committer must verify all proposal signatures in advance.

As before, a commit can be split into a welcome message $T_W$ for newly added users, and a commit message $T_C$ for group members. These are signed jointly in our construction to simplify the security proof, but may also be signed separately. In $T_C$, we also include the welcome messages to CGKA*, since they must always be addressed to current group members (i.e. of $G$).

Commits in both CGKAs are independent: CGKA can be updated while CGKA* is not, and vice-versa.

Processing control messages. The proc method takes a control message $T$, determines the type of message (create, welcome, or commit) and the gid, and updates the state only if processing succeeds (acc = true). Newly added users verify the admin signature, process the welcome message using CGKA.proc and store the new public admin key ($spk'$, 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 $\gamma$.spk $\leftarrow$ spk'. In case $T$ contains a create message $T_{CR}$, both CGKAs process the respective welcome messages contained in $T_{CR}$ separately.

4.2.3 Features

DGS allows the use of two distinct and independent CGKA protocols that authenticate admins as a group, providing some notable features that differ from IAS. One can also imagine a ‘hybrid’ approach where users run IAS except that some IAS keys are maintained and updated via their own CGKA.

Minimal information reveal. As opposed to IAS, 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). We discuss this further in Section 6.3.2.
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\(^7\). If these attacks are relevant, one can deploy heavier protocols such as the P-Act-Rob in [ACJM20]. 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. This occurs generally in CGKA when a member leaves a group; in MLS the policy enforces removals to be committed before any application message is sent. This problem can nevertheless be solved using the same approach as for self-removes in IAS.

Security mechanisms. We note the conceptual simplicity of achieving PCS and FS in the group administration keys (in the adversarial model for CGKA*) given the existence of secure CGKA schemes in the literature, since both properties are ensured by 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

Some group messaging protocols already authenticate group members via signatures and public-key infrastructure. The MLS specification [BBR+23] relies on credentials, which are essentially public signature keys for each protocol user that are certified by a PKI; these keys authenticate messages originating from that user\(^8\). Therefore, it is possible to extend the CGKA used in MLS to an A-CGKA in a more efficient way than using a compiled A-CGKA construction resembling IAS. We note that, in practice, it is feasible to support secure administration in MLS via an MLS extension, a feature that enables additional proposal types and actions in the protocol [BBM+]. Constructing such an extension is almost straightforward and we identify three main necessary changes:

- Credentials are not necessarily refreshed in MLS, meaning that admins (and users in general) whose state is compromised at some point lack forward security and post-compromise security on their authentication keys (unless they proactively update them). Our solution is to introduce an IAS-like credential update mechanism for admin signature keys (providing post-compromise security) which may be invoked without updating the core CGKA secret.

\(^7\)This scenario is out of the scope of our security model where admins are fully trusted.

\(^8\)Signatures play an important role in MLS: “...group members can verify a message originated from a particular member of the group. This is guaranteed by a digital signature on each message from the sender’s signature key. The signature keys held by group members are critical to the security of MLS against active attacks...” [BBM+].
- Group members need to keep track of the administrators (for an IAS-like extension). To this end, we propose to introduce new admin proposal types and enforce that admin proposals are signed, alongside corresponding update policies and modifications to commit and proc.

- As in IAS, admins register their keys with the PKI as they are updated over time.

4.3.1 Modifications

We propose an extension of the main algorithms of the MLS protocol (in particular, of the CGKA-related prop, commit and proc) in Figure 11, that we also benchmark in Section 5.3. Our goal is to show how IAS can be easily integrated with relatively low overhead. We follow [ACDT21a] (in particular, Figure 8 in the full version [ACDT21b]), as this is the most comprehensive formalization of MLS in the literature at the time of writing; therefore, we also work in the single-group setting and omit gids. We omit the send and rcv algorithms as these are used to send application messages only. We note also that their create method supports only one initial participant; hence its integration with IAS is trivial.

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 [ACDT21a], we split the processing algorithms in two – one for commit messages, and one for welcome messages – that a committer produces for each incoming user separately.

Overall, the overhead with respect to (bare-bones) MLS is minimal; we essentially only need to support the new types of proposals and to 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 that can be 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 [ACDT21a] 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.
Figure 11: Construction of an MLS extension that supports group administrators, effectively turning the CGKA in MLS into an A-CGKA. Highlighted lines correspond to our main modifications in the original SGM construction in [ACDT21a]. Cred[] denotes an array that stores the credentials of all ID’s. We also use the abstract function registerPKI for standard PKI functionality of registering signature keys. Some technical details are omitted.
5 Results

5.1 Correctness

Proposition 1. Let CGKA be a correct CGKA (Definition 3) and $S = (\text{SigGen}, \text{Sig}, \text{Ver})$ be a correct signature scheme. Then, the IAS protocol (Figures 6 and 7) is correct with respect to Definition 3.

Proposition 2. Let CGKA and CGKA* be correct CGKAs, and $S = (\text{SigGen}, \text{Sig}, \text{Ver})$ be a correct signature scheme. Then, the DGS protocol (Figures 9 and 10) is correct with respect to Definition 3.

The correctness proofs for both protocols can be found in Appendix C.

5.2 Security

Observe that IAS, which uses a (regular) digital signature scheme, does not provide optimal forward security. Therefore, we prove security with respect to a sub-optimal admin cleanness predicate $C_{adm}$ where $C_{adm} = C_{adm-opt} \land C_{adm-add}$ and $C_{adm-opt}$ is defined in Section 3.4. We define $C_{adm-add}$ in Appendix B, together with the full security proof.

Theorem 1. Let CGKA be a correct and $(t_{cgka}, q, \epsilon_{cgka})$-secure CGKA with respect to cleanness predicate $C_{cgka}$, according to Definitions 3 and 4. Let $S = (\text{SigGen}, \text{Sig}, \text{Ver})$ be a deterministic $(t_S, q, \epsilon_S)$ SUF-CMA secure signature scheme, as in Definition 7. Let $H_1$ be a $(t_F, 1, \epsilon_F)$-secure 4-PRF, as in Definition 5. Then, the IAS protocol (Figures 6 and 7) is $(t, q, q \cdot \epsilon_F + \epsilon_{cgka} + q^2 \cdot \epsilon_S)$-secure (Definition 4) with respect to predicates $C_{cgka}, C_{adm}, C_{forgery}$, where $t_{cgka} \approx t_S \approx t_F \approx t$, $C_{adm}$ is defined in Figure 14 and $C_{forgery}$ is defined in Section 3.4.3.

Proof idea. We first bound an adversary $A$’s advantage in distinguishing between the KIND$_{A-CGKA}$ 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 $A$’s behaviour in $G_1$ into two events based on whether they successfully query the $O^{inject}$ 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 with KIND$_{CGKA}$ adversary, at which point the claim follows.

Theorem 2. Let CGKA be a correct and $(t_{cgka}, q, \epsilon_{cgka})$-secure CGKA with respect to $C_{cgka}$, according to Definitions 3 and 4. Let CGKA* be a correct and $(t_{cgka*}, q, \epsilon_{cgka*})$ secure CGKA with respect to $C_{cgka*}$. Let $S$ be a $(t_S, q, \epsilon_S)$ SUF-CMA secure signature scheme, as in Definition 7. Let $H_1$ be a $(t_F, 1, \epsilon_F)$-secure 4-PRF, as in Definition 5. Let $H_{ro}$ be a random oracle queried at most $q_{ro}$ times. Then, the DGS protocol (Figures 9 and 10) is $(t, q, \epsilon_{cgka} + q \cdot (\epsilon_F + \epsilon_S + q_{ro} \cdot \epsilon_{cgka} + 2^{-\lambda}))$-secure (Definition 4) in the random oracle model with respect to predicates $C_{cgka}, C_{adm}, C_{forgery}$, where $t_{cgka} \approx t_{cgka*} \approx t_S \approx t_F \approx t$, $C_{adm}$ is a function of $C_{cgka}$ and $C_{forgery}$ is defined in Section 3.4.3.

Proof idea. We first describe $C_{adm}$. Intuitively, $C_{adm}$ ensures that the set of safe oracle queries for DGS adversary $A$ given inject queries of the form $q_i = O^{inject}(ID, m, t_a)$ are those that are safe for CGKA* adversary $A'$ under essentially the same queries, replacing $O^{inject}(\cdot, \cdot, t_a)$ queries with queries of the form $O^{Challenge}(t_a)$. To prove security, we use a similar game-hopping argument as in IAS. We first replace $H_i$ calls using the PRF assumption. We then consider $E_1$ (a successful injection is made) and $E_2$ (otherwise) as in IAS. Given $E_2$, we can simulate directly via the CGKA adversary. Given $E_1$, we simulate differently depending on whether $A$
makes a query to random oracle $H_{ro}$ with a correct CGKA$^*$ key before the successful injection or not. Here, if $A$ is successful, we reduce security to the CGKA$^*$ adversary by intercepting the relevant random oracle query and guessing the correct bit using information from $O_{\text{Challenge}}$. Otherwise, we can simulate via an EUF-CMA adversary as in IAS since the signature key is now uniform from the adversary’s perspective.

5.3 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$^9$ and compare 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 11, 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$^{10}$. Core cryptographic operations were implemented as HPKE [BBLW22] 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 Figures 12 and 13, where we show the running time of the commit (Figure 12) and proc (Figure 13) algorithms 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.

On the one hand, we present the running time of both algorithms for varying group size $|G|$ with a fixed member/admin ratio $|G|/|G^*|=4$. In the event of updating users, there are $t = |G^*|$ users updating and/or $t/2$ admins doing an admin-update. On the other hand, we benchmark our algorithms for fixed group size $|G|=64$, $|G^*|=16$, while varying the number of updates $t$ (and/or $t/2$ updating admins) in a commit.

For both cases, we compare the committing and processing times of (1) standard commits ($\text{com-type} = \text{std}$, omitted in fixed group size benchmarks), (2) standard commits with $t$ update proposals, (3) standard and admin commits ($\text{com-type} = \text{both}$) 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.

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.

$^9$The original source code is available at https://github.com/cisco/go-mls.

$^{10}$https://pkg.go.dev/testing
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.

![Figure 12: 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.](image)

![Figure 13: Benchmark of the proc algorithm when processing a commit message. The different scenarios are those of Figure 12.](image)

6 Discussion

6.1 Efficiency

6.1.1 Protocols

The results in Section 5.3 show that the additional cost (for users) of running a securely-administrated MLS is minimal. Figure 12 shows that the commit algorithm involves less than a
20% overhead when up to $|G|/8$ members carry out admin updates simultaneously (note that admin updates also involve standard updates). Figure 13 shows that the processing time of admin and standard updates is very similar, and increases linearly in the number of updates.

Separately, we analyze the overhead of IAS and DGS for group members, both for number of operations and for message size. We note that this assumes that IAS and DGS are implemented modularly and not integrated with an existing CGKA as before. In IAS, admins generate a signature key pair and sign every time they carry out a commit or a proposal, and verify a small amount $t$ of signatures (typically $t \leq |G^*|$) in admin proposals before a commit. If we denote the cost (time/length) of a message signature or verification by $s$, and the cost (time/length) of a signature key pair generation by $k$, we obtain the values\(^{11}\) in Table 1. Note that $s = \mathcal{O}(\lambda)$ and $k = \mathcal{O}(\lambda)$ (i.e. are constant) for security parameter $\lambda$.

The overhead of DGS depends heavily on the cost of CGKA* (an optimistic estimation can be $\mathcal{O}(\log m)$ [ACDT20, KPPW+21, ACJM20] but can be $\mathcal{O}(m)$ 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.

<table>
<thead>
<tr>
<th></th>
<th>Length (Adm)</th>
<th>Length (All)</th>
<th>Time (Adm)</th>
<th>Time (All)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAS</td>
<td>$ts + tk$</td>
<td>$ts + tk$</td>
<td>$\mathcal{O}(ts + k)$</td>
<td>$\mathcal{O}(ts)$</td>
</tr>
<tr>
<td>DGS</td>
<td>$C + s + k$</td>
<td>$s + k$</td>
<td>$\mathcal{O}(C + s + k)$</td>
<td>$\mathcal{O}(s + k)$</td>
</tr>
</tbody>
</table>

\(^{11}\)We neglect the size of user identifiers $\text{ID}s$ as we assume it small and constant with respect to the security parameter. Besides, identifiers may already be present in protocols that construct admins at the application level.

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 $|G|$.

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 [MMM02].

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.

### 6.1.2 Admins in TreeKEM variants

The Tainted TreeKEM protocol provides efficiency advantages if only a subset of users carry out tree-changing operations (adds and removals) [KPPW+21]. Tainted TreeKEM, however, is not formalised in the propose-and-commit-paradigm, which complicates the efficiency comparison; such an analysis thus remains open. 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

6.2.1 CGKAs and MLS

The CGKA abstraction has deviated from MLS and has become an object of study of its own [AHKM21, BDG+21, AAN+22], 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 to the infrastructure designers [BBM+]. Following this separation, the Delivery Service is often modelled adversarially in CGKAs whereas the AS is abstracted as a PKI [ACDT20, KPPW+21, ACJM20] as in this work. A compromised AS allows for the corruption of user credentials, resulting in user impersonation.

6.2.2 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 [ACDT20, KPPW+21], or ignores the AS [CHK21]. 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 [AJM20], 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.

6.2.3 Signal Private Groups

In [CPZ20] and as deployed in Signal, a central server manages the membership of a group whilst hiding the set of group members from non-members (modulo metadata leaked to a network adversary). The main goals of this solution are to achieve user privacy and act as a single source of truth for the membership of a group. 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. We note that Signal Private Groups however does not fully protect from server and network attacks as our A-CGKA constructions do: for example, it is possible for the server to re-add removed users. In addition, the system has not been analysed in composition with an underlying group messaging protocol (pairwise Signal) where concurrency issues can arise.
6.3 Additional admin mechanisms

We consider possible extensions of A-CGKA as a primitive and corresponding construction ideas. We note that these extensions may provide stronger security guarantees, or additional functionality, at reduced cost if the number of admins is small.

6.3.1 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 [Wha20]). 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. by entrusting admins to process messages or through NIZKs.

6.3.2 Private administrators

In some applications, it may be desirable to hide the set of admins from (non-admin) users within a group (or between themselves). 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 [RST01]. However, there is overhead with ring signatures over regular signatures, at a minimum to parse the anonymity set, and privacy is compromised e.g. if a single admin performs an admin update then signs after, so admins would need to batch updates for privacy.

In MLS’ TreeKEM protocol, proposals are constant-sized, but commits are variable, which leaks information about the contents of the commit even if it is encrypted. Thus, padding is required at a minimum for privacy. In the MLS standard, ciphertexts, i.e. MLSCiphertexts, leak the group ID, epoch and message content type (proposal or commit) in plaintext, which would need to be hidden for additional privacy. In practice, additional attack vectors like timing and traffic analysis preclude privacy also, which are considered by some messaging systems [TGL+17, CSM+20] that do not, however, provide FS and PCS; it remains open to e.g. adapt CGKA to defend against these attack vectors. As discussed in the introduction, the Signal Private Group System [CPZ20] hides the group membership from non-members (although not all metadata); the mechanism could be extended for admins but adapting the technique to (A)-CGKA is also open. A step towards anonymity in messaging has been recently made in [DHRR22], including the study of anonymity under state exposure.

6.3.3 External admins

Our A-CGKA constructions assume the admins comprise a subset of all group members, i.e. $G^* \subseteq G$. 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, rather than the less well-defined application level as previously done.

Given that the underlying CGKA allows for external commits, it is straightforward to administrate IAS and DGS-based groups oup externally. Namely, the admin who is approving the change can inspect proposals and make commit messages for the corresponding parties. However, TreeKEM and its variants are not ideal for this since the committer is the party who derives new group secrets and can thus violate confidentiality. One way around this issue is to essentially write an wrapper around each CGKA algorithm which declares that some CGKA group members are not actually in the group. Here, the wrapper would also force admins to delete group secrets as soon as they derive them, and would not consider admins as part of the group; the solution is however clearly vulnerable to corrupted admins.

One conceptually simple solution is to allow commit messages from regular users which play the role of proposals which have to be “committed” (e.g. signed) by admins. Additional machinery like NIZKs is however required in the malicious setting to enable admins to verify that such commits are well-formed.

6.3.4 Hierarchical admins

In messaging apps like Telegram and WhatsApp, the group creator has stronger capabilities than other admins. For instance, the group creator can never be removed by another admin. Extending this concept, one can conceive a hierarchy of administrators of several levels, e.g., of the form $G^{**} \subseteq G^{*} \subseteq G$, where $G^{**}$ are super-administrators. Extending IAS, one can imagine using signatures that attest to other signatures in a chain-of-trust fashion. DGS can be extended by considering many CGKAs where CGKA $i$ must sign commit messages for CGKA $i+1$ for each $i \geq 1$. Attribute-based admins would enable greater flexibility.

6.3.5 Muting admins

It is possible to provide some cryptographic guarantees to the process of muting conversation participants. Although we do not explicitly consider group messaging, we sketch how such a solution would look. One solution entails a DGS-like construction in which members must sign messages using a common signature key $spk$ derived from a secondary CGKA; honest group members would then process application messages if and only if they are signed using the common signature key (i.e. only from the set of unmuted users). Then, muted members will be able to filter messages from other muted users (since they could still be informed of the state of $spk$ over time), but they will not able to sign their own messages. Mechanisms that enable a central server to filter messages while maintaining privacy like [HS20] can be also be integrated into the encryption layer over A-CGKA. Nevertheless, we note that in messaging services where the identity of the group members is known, muted members can generally bypass a ban by sending individual messages to all group members using two-party messages. At an application level, muting group members is a functionality supported by both Signal and Telegram.

6.3.6 Threshold admins

One issue with our A-CGKA constructions is that security breaks down if a single admin is compromised. To improve the robustness of the protocol, a protocol can enforce that some $k > 1$ admins must attest to a particular commit before it may be processed, which can be achieved simply using threshold cryptography [Sho00].
6.3.7 Decentralized admins

To allow for network decentralization, it is straightforward in theory for a given messaging group $G$ to simply execute a state machine replication protocol [CL$^+99$] to order commit messages and require that users reliably broadcast [BT85] all proposal messages. Given that group members who are expected to execute the protocol on e.g. cellphones may not be available often, thus leading to liveness (and possibly unintended safety) violations in protocol execution, a natural solution is to entrust administrators to provide messages to users. These admins could indeed execute consensus.

Acknowledgements

This work has received funding in part from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program under project PICOCRYPT (grant agreement No. 101001283), by the Spanish Government under projects SCUM (ref. RTI2018-102043-B-I00), and RED2018-102321-T, by the Madrid Regional Government under project BLOQUES (ref. S2018/TCS-4339), by a research grant from Nomadic Labs and the Tezos Foundation, and by Ministerio de Universidades (FPU21/00600). We are grateful to the anonymous reviewers of this work for their valuable suggestions and comments.

References


[ACDT21a] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of mls. In Proceedings


A Primitives

Definition 5 (PRF). We say that a function $f_n : K \times X \rightarrow Y^n$ is a $(t, q, \epsilon)$-secure $n$-pseudorandom function ($n$-PRF) if, for any polynomial-time adversary $A$ limited to $q$ oracle queries, the advantage of $A$ in the $\text{PRF}_{f_n, 1}^A$ game below given by

$$\Pr[\text{PRF}_{f_n, 1}^A(1^\lambda) = 1] - \Pr[\text{PRF}_{f_n, 0}^A(1^\lambda) = 1]$$

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

\begin{align*}
\text{PRF}_{f_n, b}^A(1^\lambda) & \quad \text{O}^{\text{Eval}}(m) \\
1: & \quad k \leftarrow \mathcal{K} \\
2: & \quad \text{sample function } F : X \rightarrow Y^n \\
3: & \quad b' \leftarrow A^{\text{O}}(\lambda) \\
4: & \quad \text{return } b' \\
1: & \quad \text{if } b = 0 \\
2: & \quad \text{return } F(m) \\
3: & \quad \text{return } f_n(k, m)
\end{align*}

Definition 6 (Digital signature). A digital signature scheme is a triple of algorithms $(\text{SigGen}, \text{Sig}, \text{Ver})$ such that:

- $(sk, pk) \leftarrow \text{SigGen}(1^\lambda)$ creates a public-private key pair.
- $\sigma \leftarrow \text{Sig}(sk, m)$ generates a signature $\sigma$ from a message $m$ and the secret key $sk$.  

46
We say that the signature scheme is correct if for any \( \lambda \) any query game below given by

\[
\Pr[\text{secure against strong existential forgery under chosen message attacks (SUF-CMA)} \mid \lambda]
\]

After proving security, we discuss how one can (easily) extend the proofs and ensure optimal

Let \( \text{Cadm} \) be a predicate that we prove IAS secure with respect to. Towards proving security, we prove the following lemma.

\[\text{Cadm} \rightarrow \forall (i, ID, ID', ctr \in (0, \text{exp-ctrl})) : q_i = O^{\text{inject}}(ID', i, t_i^*) \]

\[
(\exists (t_i, c) : (t\text{Exp}(ID, ctr), t_a < t_i \leq t_i^*) \land
\text{hasUpdadm}(ID, T[\cdot, t_i], \cdot, \text{com}', c, t[\cdot, t_i], \cdot, \text{vec}', c) \land
(C[\cdot, t_i] = c)).
\]

Fig. 14: Sub-optimal administrative predicate where the adversary makes oracle queries \( q_1, \ldots, q_n \).

- \( b \leftarrow \text{Ver}(pk, \sigma, m) \) outputs \( b \in \{0, 1\} \), indicating acceptance or rejection, given a signature \( \sigma \), a message \( m \) and a public key \( pk \).

We say that the signature scheme is correct if for any \( \lambda, m \in \mathcal{P} \), and all choices of randomness, if \((sk, pk) \leftarrow \text{SigGen}(1^\lambda)\) and \( \sigma \leftarrow \text{Sig}(sk, m) \), then \( \text{Ver}(pk, \sigma, m) = 1 \).

**Definition 7** (Digital signature security: SUF-CMA). A digital signature scheme \( \Pi \) is \((t, q, \epsilon)\)-secure against strong existential forgery under chosen message attacks (SUF-CMA) if, for any polynomial-time adversary \( \mathcal{A} \) limited to \( q \) oracle queries, the advantage of \( \mathcal{A} \) in the SUF-CMA\( \Pi \) game below given by \( \Pr[\text{SUF-CMA}\Pi(1^\lambda) = 1] \) is bounded by \( \epsilon \), where the probability is taken over the choice of the challenger and adversary’s random coins.

\[
\begin{align*}
\text{SUF-CMA}\Pi(1^\lambda) & \quad \text{\( O^{\text{Sign}}(m) \)} \\
1: & (sk, pk) \leftarrow \Pi.\text{SigGen}(1^\lambda) & 1: & \sigma \leftarrow \Pi.\text{Sig}(sk, m) \\
2: & Q \leftarrow \emptyset & 2: & Q \leftarrow Q \cup \{(m, \sigma)\} \\
3: & (m, \sigma) \leftarrow A^{O}(pk) & 3: & \text{return } \sigma \\
4: & \text{require } (m, \sigma) \notin Q &
5: & \text{return } 1_{\Pi.\text{Ver}(pk, m, \sigma)}
\end{align*}
\]

## B Security Proofs

### B.1 Proof of Theorem 1 (IAS security, Section 5.2)

Note that IAS uses a (regular) signature scheme, which results in sub-optimal security since an admin may not update their signature key at the beginning of every (admin) epoch. We start with the definition of the predicate \( \text{Cadm} \) predicate that we prove IAS secure with respect to. After proving security, we discuss how one can (easily) extend the proofs and ensure optimal security using forward-secure signatures. \( \text{Cadm} \) is presented in Figure 14. Note that it differs from the optimal predicate (Figure 5) only by the lack of \((t_i^* < t\text{Exp}(ID, ctr), t_a)\) condition, so it holds that \( \text{Cadm} \land \text{Cadm-opt} = \text{Cadm} \). That is, the forward security guarantees are weaker, since e.g. if a party updates their admin key in admin epoch 3 then if they are exposed in epoch 5 then the adversary can make a trivial forgery in the construction (and thus it is considered a trivial attack by \( \text{Cadm} \)). Towards proving security, we prove the following lemma.

**Lemma 1.** Let \( \mathcal{A} \) be a KIND\( \Pi \)-CGKA,\( \text{cgka} \)-\( \text{Cadm} \) adversary playing with respect to IAS. Consider any query \( \mathcal{A} \) makes of the form \( O^{\text{inject}}(ID, m, t_a) \) which results in a response \( v \neq \bot \). Then \( \mathcal{A} \) can parse \( m = (\text{gid}, T_C, T_W, \sigma_T) \) and efficiently derive \( pk \) such that \( \text{Ver}(pk, \sigma_T, (\text{gid}, T_C, T_W)) = 1 \).
Proof. Consider $\mathcal{A}$’s query $O^{\text{Inject}}(ID, m, t_a)$ that outputs $v \neq \perp$. Given $v \neq \perp$ and by definition of $O^{\text{Inject}}$, a call $(\gamma, \cdot) \leftarrow \text{proc(ST[ID], m)}$ was previously made by the challenger such that $C_{\text{forgery}} = \text{true}$. Note $\mathcal{A}$ cannot register signature keys with the PKI, and keys are assumed to be bound to the context that they are used in, e.g. for self-removals. In addition, any non-admin commit orissuing of group changes that only consists of self-removes will not result in the adversary winning by construction of IAS, and if a self-remove is created then even if another ‘dishonest’ self-remove can be created for that party, $C_{\text{forgery}}$ will consider it equivalent to the ‘original’ self-remove. Thus, self-removes do not affect security. Now, $O^{\text{Inject}}$ disallows $t_a \neq -1$, which is the case if and only if $\text{ID} \notin G$, and that unsigned control messages cannot change the group structure. Thus, we only need to consider control messages that are input to p-Comm in proc and result in output $\text{acc} = \text{true}$ when $\sigma_T \neq \perp$. To reach p-Comm, the proc call must be such that $\text{Ver}(\gamma[\text{gid}.\text{adminList[ID].spk}], (\text{gid, T}_C, T_W), \sigma_T) = 1$. Since $\gamma[\text{gid}.\text{adminList[ID].spk}$ must have been previously sent in a message by construction of IAS, it follows that $\text{pk} = \gamma[\text{gid}.\text{adminList[ID].spk}$ is efficiently computable.

Now we are ready to prove the theorem which we restate below:

**Theorem 1** (IAS security). Let CGKA be a correct and $(t_{\text{cgka}}, q, \epsilon_{\text{cgka}})$-secure CGKA with respect to cleanness predicate $C_{\text{cgka}}$, according to Definitions 3 and 4. Let $S = (\text{SigGen}, \text{Sig}, \text{Ver})$ be a deterministic $(t_S, q, \epsilon_S)$ SUF-CMA secure signature scheme, as in Definition 7. Let $H_4$ be a $(t_F, 1, \epsilon_F)$-secure 4-PRF, as in Definition 5. Then, the IAS protocol (Figures 6, 7 and 8) is 4; we assume for simplicity that each $\text{cgka}$, $\epsilon$-secure 4-PRF, as in Definition 5. Then, the IAS protocol (Figures 6, 7 and 8) is $A$-CGKA with $C_{\text{cgka}}, C_{\text{adm}}, C_{\text{forgery}}$, where $t_{\text{cgka}} \approx t_S \approx t_F \approx t$, $C_{\text{adm}}$ is defined in Figure 14 and $C_{\text{forgery}}$ is defined in Section 3.4.3.

Proof. Let $G_0$ be the KIND$_{\text{IASCgka}, C_{\text{adm}}}$ game. Let $G_1$ be as in $G_0$, except that all calls of the form $(r_1, \ldots, r_i) \leftarrow H_i(r_0, \gamma)$ are replaced with calls of the form $(r_1, \ldots, r_i) \leftarrow R^i$ where $R$ is the space of random coins used by each (A)-CGKA algorithm. Note that in IAS we always have $i \leq 4$; we assume for simplicity that each $H_i$ for $i \leq 4$ is implemented by calling $H_4$ and then truncating the output as needed.

Let $G_{0,0} = G_0$. Let $G_{0,j}$ be $G_0$ except that the first $j$ calls of the form $H_i(r_0, \gamma)$ that adversary $\mathcal{A}$ makes results in at most one call to a function of the form $H_i(\cdot, \cdot)$, $G_{0,k} = G_1$ for some $k \leq q$.

Consider $G_{0,j-1}$ and $G_{0,j}$ where $j \geq 1$; suppose these games are played by adversary $\mathcal{A}$. Let $\mathcal{A}'$ be a 4-PRF adversary as in Definition 5. $\mathcal{A}'$ simulates directly except when $\mathcal{A}$ makes their oracle query which leads to the $j$-th call to a function of the form $H_i$. Upon this call, $\mathcal{A}$ simulates this call by calling $O^{\text{Eval}}(\gamma)$ for the input $\gamma$, and truncates the response $(r_1, \ldots, r_4)$ to $(r_1, \ldots, r_4)$ when necessary before continuing execution (i.e. its simulation). Clearly $\mathcal{A}'$ perfectly simulates $G_{0,j-1}$ when $\mathcal{A}$’s challenger’s bit is 1. When $\mathcal{A}$’s challenger’s bit is 0, note that the output of $\mathcal{A}$’s $O^{\text{Eval}}$ call, namely $(r_1, \ldots, r_4)$, is of the form $F(\gamma)$ for a uniformly random function $F : \text{ST} \rightarrow R^4$, where $\text{ST}$ is the A-CGKA state space. Since $F$ is randomly chosen, this output is distributed identically to $(r_1, \ldots, r_4)$ where each $r_i$ is uniformly sampled from $R$. It follows that $\mathcal{A}'$ perfectly simulates $G_{0,j}$ when its challenge bit is 0. By combining the sequence of game hops, it follows that

$$\left| \Pr[G_0^A(1^\lambda) = 1] - \frac{1}{2} \right| \leq \left| \Pr[G_1^A(1^\lambda) = 1] - \frac{1}{2} \right| + q \cdot \epsilon_F,$$

where $\epsilon_F$ is the advantage of PRF adversary $\mathcal{A}'$. 48
In the following, we will simulate for a $G_1$ adversary via either a SUF-CMA or CGKA adversary. Without hopping from $G_0$ to $G_1$, the simulation would not have been identical when e.g. using the SUF-CMA $O^{\text{Sign}}$ oracle. Since we have transitioned to $G_1$, which uses randomness normally, there are no issues regarding simulation and randomness.

Let $A$ be a $G_1$ adversary. Let $E_1$ be the event that $A$ makes a query to $O^{\text{Inject}}$ such that $O^{\text{Inject}}$ outputs value $v \neq \bot$ (i.e. the challenge bit). Let $E_2$ be the event that this does not occur; clearly $\Pr[E_1] + \Pr[E_2] = 1$. We consider each event separately. Without loss of generality, we restrict our simulations given $E_1$ to the case where $C_{\text{adm}}$ is true as otherwise the adversary cannot win, and similarly given $E_2$ to the case where $C_{\text{gka}}$ is true. It suffices to observe that these two predicates are efficiently computable such that the adversary can abort during unclean executions.

$E_1$: By construction of IAS and Lemma 1, note, given that $O^{\text{Inject}}$ outputs $v \neq \bot$, that a signature forgery has occurred where the signature keying material is sampled due to an oracle call. Given $E_1$, we need to determine which input values ID and $t_a$ are used by $A$ on the first $O^{\text{Inject}}$ call which outputs $v \neq \bot$. By construction of IAS, this can happen as a result of a query to $O^{\text{Create}}$, $O^{\text{Prop}}$ or $O^{\text{Commit}}$. However, $A$ may make at most $q$ injection attempts with respect to this key pair, each of which may be a winning one. Thus, the reduction has to guess both 1) the $O^{\text{Inject}}$ query which first outputs $v \neq \bot$ and 2) the query $q_i$ which generates the signature key pair corresponding to this injection.

Let $E_{1,i,j}$ be the event that $A$ makes query $q_i$ which leads to the generation of signature key pair $(\text{ssk}, \text{spk})$ such that query $q_j$ is the first query to $O^{\text{Inject}}$ resulting in output $v \neq \bot$. Note that IAS is such that each oracle query $q_i$ results in at most one new signature key pair being sampled by the challenger. By the union bound, we have:

$$
\Pr[G_1^{A}(1^\lambda) = 1 \land E_1] \leq \sum_{i,j \in \{1,\ldots,q\}} \Pr[G_1^{A}(1^\lambda) = 1 \land E_{1,i,j}].
$$

Suppose $E_{1,i,j}$ holds. Let $A'$ be an SUF-CMA adversary that simulates for $G_1$ adversary $A$. We aim to show that $\Pr[G_1^{A'}(1^\lambda) = 1 \land E_{1,i,j}] \leq \epsilon_S$.

$A'$ simulates as follows. $A'$ simulates variable initialisation as in $G_1$ except that it lazily samples init calls as needed (ensuring it remains polynomially-bounded). $A'$ simulates the first $i - 1$ of $A$'s oracle queries locally, i.e. simulates all relevant behaviour resulting in corresponding state and game variables being set and updated. This includes queries of the form getSsk and getSpk which $A'$ simulates via signature scheme $S$.

Consider $A'$'s $i$-th oracle query $q_i$. Let $(\text{ssk}^*, \text{spk}^*)$ denote the signature key pair sampled by the SUF-CMA challenger; recall that SUF-CMA adversary $A'$ has access to oracle $O^{\text{Sign}}$. $A'$ simulates as follows:

- If $q_i$ is to $O^{\text{Create}}$, then $(\text{ssk}^*, \text{spk}^*)$ plays the role of $(\gamma, \text{ssk}'_1, \gamma, \text{spk}')$. Namely, $A'$ sets the output of getSpk(ID, $\gamma$, ME) to $\text{spk}^*$ and that of getSsk($\gamma, \text{spk}'_1$, ME) to $\text{ssk}^*$ after embedding $\text{spk}^*$ in adminList[ME] at line 7 of create. $A'$ also calls $O^{\text{Sign}}((\text{gid}, \bot, T_W))$ which outputs $\sigma$. $A'$ otherwise simulates and returns the result of $A$ (which includes $\sigma$ in part).

- If $q_i$ is to $O^{\text{Prop}}$ with input type = upd-admin, then $(\text{ssk}^*, \text{spk}^*)$ plays the role of $(\gamma, \text{ssk}'_1, \gamma, \text{spk}')$ (line 10 of makeAdminProp). Similarly to above, $A'$ embeds $\text{spk}^*$ in $P_0$ (line 11 of makeAdminProp), and then simulates the rest of the oracle call.
• Otherwise, \( q_i \) is to \( O^{\text{Commit}} \). Note we assume \( E_{1,i,j} \) holds. Thus, the branch at line 6 in \text{commit} must be executed. As before, \((\text{ssk}^*, \text{spk}^*)\) plays the role of \((\gamma, \text{ssk}^*, \gamma, \text{spk}^*)\); \( A' \) simulates the remainder of the call.

All other oracle queries, except to \( O^{\text{Inject}} \), are simulated locally by \( A' \) except when signatures with respect to \( \text{spk}^* \) are required, in which case SUF-CMA oracle \( O^{\text{Sign}} \) is used, or when \( \text{spk}^* \) is to be embedded in a message. Note that at most \( q \) \( O^{\text{Sign}} \) queries are made by \( A' \) since each oracle query \( A \) makes produces at most one signature (which may or may not require \( O^{\text{Sign}} \) to simulate).

Note by construction of IAS that each signature key pair is sampled by a single party and is never revealed/embedded in another message, and that each key pair is uniformly and independently sampled. Thus, the simulation is valid, ignoring \( O^{\text{Expose}} \) queries, since the challenge key pair is independent of all other keying material and challenge secret key \( \text{ssk}^* \) is never revealed. Similarly, since \( C_{\text{adm}} \) is true (since we assume \( E_1 \)), \( A \) will never query \( O^{\text{Expose}} \) with respect to the challenge key pair, a query which would otherwise not be able to be simulated.

For \( A \)'s query \( q_j' = O^{\text{Inject}}(\text{id}) \) such that \( j' \neq j \), \( A' \) simulates by returning \( \perp \) to \( A \). When \( A \) makes query \( q_j = O^{\text{Inject}}(\text{id}, m, t_0) \), \( A' \) inspects \( m = (\text{gid}, T_C, T_W, \sigma_T) \) and returns the message/signature pair \(((\text{gid}, T_C, T_W), \sigma_T)\) to \( A \) which, by Lemma 1, exists. These two steps are valid by definition of \( E_{1,i,j} \) and that \( j' < j \) in the first step since we only simulate up to query \( j \).

It thus follows that the simulation is perfect. Noting that \( A \) wins at most as often as \( A' \) (since e.g., \( A \) may come up with a forgery \((m, \sigma)\) nevertheless rejected by \( \text{proc} \)), we have:

\[
\Pr[G^A_1(1^\lambda) = 1 \land E_{1,i,j}] \leq \epsilon_{\text{sig}}.
\]

\( E_2 \): Note first that given \( E_2 \), we can deduce that every query to \( O^{\text{Inject}} \) will have output \( \perp \). Let \( A' \) be a \( \text{KIND}^{\text{cgka}}_{\mathcal{A}, \mathcal{C}_{\text{cgka}}} \) adversary. \( A' \) simulates for \( \text{KIND}^{\text{cgka}}_{\mathcal{A}, \mathcal{C}_{\text{cgka}}} \) adversary \( A \) as follows. When \( A \) makes an oracle query, \( A' \) executes all code in IAS except for that which makes use of CGKA algorithms, which are processed via \( A' \)'s oracles; \( A' \) then returns each response to \( A \). One case we must deal with is when \( \text{proc} \) is called at line 13; the call may succeed but the caller may ignore the state update, undoing the changes made; this happens given the group state changes as a result of the call. Note that \( A' \) can determine whether this case occurs or not using the fact that commit messages are honestly delivered by construction of the KIND game and by correctness which ensures that honestly-generated and delivered commit messages are accepted. In particular, \( A' \) can use the policy to determine whether or not the call would update the group state, and only call its \( O^{\text{Deliver}} \) when the group state is not changed. Finally, \( A' \) outputs the same bit as \( A \).

To see that the simulation is perfect, first note that CGKA algorithms are used as a black box in IAS. Moreover, except in the case dealt with above, CGKA state variables \( s_0 \) for each ID are not used except as input to CGKA algorithms, after which they are immediately overwritten, exactly as done by the CGKA \( \mathbb{KIND} \) challenger given correctness and in particular the fact that failing algorithm calls do not update the state. Thus, it suffices to simulate CGKA code using \( A \)'s oracles. Thus:

\[
\Pr[G^A_1(1^\lambda) = 1 \land E_2] = \Pr[\text{KIND}^{\mathcal{A}'}_{\mathcal{A}, \mathcal{C}_{\text{cgka}}}(1^\lambda) = 1] = \epsilon_{\text{cgka}}
\]
We then have:
\[
\begin{align*}
&\left| \Pr[G_A^1(1^\lambda) = 1] - \frac{1}{2} \right| \\
= &\left| \Pr[G_A^1(1^\lambda) = 1 \land (\lor_{i,j} E_{1,i,j} \lor E_2)] - \frac{1}{2} \right| \\
\leq &\sum_{i,j} \Pr[G_A^1(1^\lambda) = 1 \land E_{1,i,j}] + \Pr[G_A^1(1^\lambda) \land E_2] - \frac{1}{2} \\
\leq &\sum_{i,j} \Pr[G_A^1(1^\lambda) = 1 \land E_{1,i,j}] \\
&+ \left| \Pr[G_A^1(1^\lambda) = 1 \land E_2] - \frac{1}{2} \right| \\
\leq & q^2 \cdot \epsilon_S + \epsilon_{cgka}
\end{align*}
\]
where the second and third lines follow from the union bound and triangle inequality, respectively. The result follows by combining this with the game hop earlier. 

**Optimal forward security.** We can achieve optimal forward security, and thus optimal admin security (i.e. security with respect to $C_{adm}$-opt) by replacing signatures with forward-secure signatures [BM99, MMM02]. Forward-secure signatures divide signature key pairs into epochs based on how many secret key update calls are made by the secret key holder. Relative to regular signatures, the security game for forward-secure signatures provides two additional oracles to 1) expose a secret key at a given epoch and 2) transition the secret key to a new epoch. Recall that the construction change we suggested to IAS comprised of parties calling the key update function whenever they transition to a new epoch such that do not ‘update’ their keys in the CGKA sense. The logic of the security reduction is very similar to that presented with regular signatures above. The main difference is that forward-secure signature calls are replaced by oracle calls, including possibly key exposure calls, which, conditioned on the optimal admin predicate being true, will not result in a trivial attack in the forward-secure signature game.

**B.2 Proof of Theorem 2 (DGS security, Section 5.2)**

**Predicate $C_{adm}$**. $C_{adm}$ is tailored to DGS and is a function of the underlying CGKA$^*$ predicate $C_{cgka}^*$. Intuitively, $C_{adm}$ ensures that the set of safe oracle queries for DGS adversary $A$ given inject queries of the form $q_i = O^{\text{Inject}}(ID, m, t_a)$ are those that are safe for CGKA$^*$ adversary $A'$ (i.e., the predicate $C_{cgka}^*$) under roughly the same queries, replacing at most one $O^{\text{Inject}}(\cdot, \cdot, t_a)$ queries with a $O^{\text{Challenge}}(t_a)$ query. For example, noting the symmetry between predicates $C_{adm}$-opt and $C_{cgka}$-opt, if CGKA$^*$ is secure with respect to CGKA predicate $C_{cgka}$-opt, then DGS is secure with respect to admin predicate $C_{adm}$-opt.

We define $C_{adm}$ more formally. Let $Q = (q_i)_I$ be the ordered sequence of oracle queries made by the DGS adversary $A$. To define $C_{adm}$, we construct an ordered sequence of queries $Q^*$ that are made by the CGKA$^*$ adversary $A'$ in the security proof below by replacing, inserting and/or deleting queries in-order. Let $\ell \in [1, q_{inj}] \cup \{\perp\}$ where $q_{inj}$ is the number of $O^{\text{Inject}}$ queries made by $A$. To this end, consider each $q_i \in Q$ and, for each $\ell$, define $q_i^*$ to be either a single query or a sequence of queries in $Q^*$ as follows:
• $q_i = O_{\text{Create}}(\text{ID}, G, G^*)$: Set $q_i^* = O_{\text{Create}}(\text{ID}, G^*)$.

• $q_i = O_{\text{Prop}}(\text{ID}, \text{ID}', \text{type})$: Set $q_i^* = \perp$ if type $\neq *$-adm and $q_i^* = O_{\text{Prop}}(\text{ID}, \text{ID}', \text{type}^*)$ otherwise where type $= \text{type}^*$-adm.

• $q_i = O_{\text{Commit}}(\text{ID}, (i_1, \ldots, i_k), \text{com-type})$: If the condition $(\text{admReq} \lor \ldots)$ at line 6 of commit is false, com-type $= \text{std}$ or ID is not currently an admin, set $q_i^* = \perp$. Otherwise, let \{ID$_1$, \ldots, ID$_j$\} be the (possibly empty) set of parties for which CGKA$^*$ rem proposals are introduced by propCleaner (line 12). Let $\tilde{P}_A$ be the value input to CGKA$^*$ commit at line 4 of c-Adm (or $\tilde{P}_A = \perp$ if the line is not reached), and (i$_1$, \ldots, i$_k$) the corresponding proposal indices in the CGKA$^*$ KIND game. Let $q'$ be the key reveal query that reveals the key $k$ output by commit in the $O_{\text{Commit}}$ call if the corresponding signature key pair is not used for the first successful $O_{\text{Inject}}$ query, and $\perp$ otherwise. Then, set $q_i^*$ to the sequence $(O_{\text{Prop}}(\text{ID}, \text{ID}_1, \text{rem}), \ldots, O_{\text{Prop}}(\text{ID}, \text{ID}_j, \text{rem}), O_{\text{Commit}}(\text{ID}, (i_1, \ldots, i_k)), q')$.

• $q_i = O_{\text{Challenge}}(t_s)$: Set $q_i^* = \perp$.

• $q_i = O_{\text{Deliver}}(\text{ID}, (t_s, t_a), c)$: Let $(T, \text{com-type}) = T[(t_s, t_a), \text{com}', c]$ for DGS KIND game variable T. If proc is called by the game, ID $\in G$ holds, $T_c \neq \perp$ holds and either ID $\in G^*$ holds or $W_A$ (contained in $T$) is $\neq \perp$, set $q_i^* = O_{\text{Deliver}}(\text{ID}, t_a, c^*)$, where $c^* \leq c$ is the number of times CGKA$^*$ commit was called after $c$ queries to $O_{\text{Commit}}$. Otherwise, set $q_i^* = \perp$.

• $q_i = O_{\text{Reveal}}(t_s)$: Set $q_i^* = \perp$.

• $q_i = O_{\text{Expose}}(\text{ID})$: Set $q_i^* = q_i$.

• $q_i = O_{\text{Inject}}(\text{ID}, m, t_a)$: Set $q_i^* = \perp$ if $q_i^*$ is the $j$-th query to $O_{\text{Inject}}$ where $j \neq \ell$ (possibly $\perp$) and $O_{\text{Challenge}}(t_a)$ otherwise ($\ell \neq \perp$).

Then, $C_{\text{adm}}$ is defined to be true if $C_{\text{cgka}}$ is true for runs where the first successful $O_{\text{Inject}}$ query is the $\ell$-th query to $O_{\text{Inject}}$ (where the $\perp$-th query denotes no successful injection). We restate the theorem to be proved:

**Theorem 2** (DGS security). Let CGKA be a correct and $(t_{\text{cgka}}, q, \epsilon_{\text{cgka}})$-secure CGKA with respect to $C_{\text{cgka}}$, according to Definitions 3 and 4. Let CGKA$^*$ be a correct and $(t_{\text{cgka}}^*, q, \epsilon_{\text{cgka}}^*)$ secure CGKA with respect to $C_{\text{cgka}}^*$. Let $S$ be a $(t_S, q, \epsilon_S)$ SUF-CMA secure signature scheme, as in Definition 7. Let $H_1$ be a $(t_F, 1, \epsilon_F)$-secure 4-PRF, as in Definition 5. Let $H_0$ be a random oracle queried at most $q_{\text{ro}}$ times. Then, the DGS protocol (Figures 9 and 10) is $(t, q, \epsilon_{\text{cgka}} + q \cdot (\epsilon_F + \epsilon_S + q_{\text{ro}} \cdot \epsilon_{\text{cgka}} + 2^{-\lambda}))$-secure (Definition 4) in the random oracle model with respect to predicates $C_{\text{cgka}}, C_{\text{adm}}, C_{\text{forgery}}$, where $t_{\text{cgka}} \approx t_{\text{cgka}}^* \approx t_S \approx t_F \approx t$, $C_{\text{adm}}$ is a function of $C_{\text{cgka}}^*$ and $C_{\text{forgery}}^*$ is defined in Section 3.4.3.

**Proof.** Following the proof of Theorem 1 we let $G_0$ be the KIND$^*_A, CGKA_{\text{cgka}}, C_{\text{adm}}$ game, and $G_1$ be as in $G_0$ except that all calls of the form $(r_1, \ldots, r_i) \leftarrow H_i(r_0, \gamma)$ are replaced with calls of the form $(r_1, \ldots, r_i) \leftarrow s R^i$. We transition between $G_0$ and $G_1$ exactly as in Theorem 1. That is, we define hybrids $G_{0,j}$ where $G_{0,0} = G_0$, $G_{0,j} = G_1$ when $j \geq q$ and $G_{0,j}$ differs from $G_0$ for appropriate $0 < j < q$ by replacing the first $j$ calls to functions of the form $H_i$ with uniformly
To prove security, we will reduce to the security of the primary CGKA, i.e. \( \epsilon_{CGKA} \) is called by the challenger at some point. Let \( F \) before Lemma 1) \( O \) corresponds to the signature key pair used in the successful key pair is derived from a given \( CGKA \) \( q \) Note that there are at most \( \epsilon_{CGKA} \) adversary \( A \) plays \( \epsilon_{CGKA} \) calls can be easily simulated); let \( E \) be the event that this does not occur; clearly \( \Pr[E_1] + \Pr[E_2] = 1 \). To prove security, we will reduce to the security of the primary CGKA, i.e. CGKA, given \( E_2 \), and to \( CGKA^* \) and signature security given \( E_1 \).

We first consider the simpler \( E_2 \) case where no successful injection is made (and thus \( O^{Inject} \) calls can be easily simulated); let \( A' \) be a KIND adversary w.r.t. CGKA simulating for \( G_1 \) adversary \( A \) given \( E_2 \). \( A' \) simulates as follows. For each oracle query, \( A' \) simulates relevant \( CGKA^* \) calls locally unless stated otherwise. In particular, \( A' \) calls \( init(1^\lambda, ID) \) for \( ID \) only as needed (i.e. lazily). Then:

- \( O^{Create}(ID, G, G^*) \): \( A' \) calls \( O^{Create}(ID, G) \) if needed and otherwise locally simulates.

- \( O^{Prop}(ID, ID', \text{type}) \): \( A' \) simulates locally if \( \text{type} \) is of the form \( *-adm \) and simulates via \( O^{Prop}(ID, ID', \text{type}) \) otherwise.

- \( O^{Commit}(ID, (i_1, \ldots, i_k), \text{com-type}) \): Since CGKA.commit is called after \( CGKA^*.commit \), \( A' \) can simulate \( CGKA^* \) calls locally and call \( O^{Commit}(ID, J = (j_1, \ldots, j_k)) \), where \( J \) corresponds to the set of relevant CGKA proposal indices derived from \( (i_1, \ldots, i_k) \), to simulate CGKA.commit calls.

- \( O^{Deliver}(ID, (t_s, t_a), c) \): \( A' \) simulates the relevant CGKA.proc call (there is at most one such call made by construction of DGS proc) via \( O^{Deliver}(ID, t_s, c') \) where \( c' \) is the index of the relevant CGKA control message. \( A' \) simulates locally otherwise.

- \( O^{Reveal}(t_s) \) and \( O^{Challenge}(t_s) \): \( A' \) simulates directly via their respective oracles.

- \( O^{Expose}(ID) \): \( A' \) calls \( O^{Expose}(ID) \) and simulates the rest of the call locally.

- \( O^{Inject} \): By definition of \( E_2 \), \( O^{Inject} \) always returns \( \perp \), and since \( O^{Inject} \) does not modify the state, \( A' \) simply outputs \( \perp \) upon each \( O^{Inject} \) call.

By construction, DGS inherits the (normal) CGKA cleanness predicate \( C_{cgka} \) from CGKA, and so the simulation is perfect and it follows thus that:

\[
\Pr[G_1^A(1^\lambda) = 1 \wedge E_2] \leq \epsilon_{CGKA}.
\]

Now, consider adversary \( A \) playing \( G_1 \) given \( E_1 \) occurs, i.e. \( A \) makes a successful \( O^{Inject} \) call. Note that there are at most \( q \) different \( CGKA^* \) epochs during a given execution and consequently at most \( q \) different \( CGKA^* \) signature key pairs computed by correct parties (since each signature key pair is derived from a given \( CGKA^* \) epoch secret).

Given \( E_1 \), let \( F_1 \) be the event that \( G_1 \) adversary \( A \) calls random oracle \( H_{ro} \) with input \( r \) that corresponds to the signature key pair used in the successful \( O^{Inject} \) query (guaranteed to exist by Lemma 1) before the injection is made. That is, \( r \) is such that \( (ssk, spk) \leftarrow \text{SigGen}(1^\lambda, H_{ro}(r)) \) is called by the challenger at some point. Let \( F_2 \) be the complementary event (i.e., either such
a $H_{ro}$ query is made after the successful injection or not at all). We consider the case with $F_1$ (by simulating the CGKA* KIND game) and $F_2$ (via the SUF-CMA game) separately.

Consider $F_1$. Let $F_{i,j}$ be the event that the aforementioned $H_{ro}(r)$ query is the $i$-th query to $H_{ro}$ and is with respect to the $j$-th CGKA* key pair sampled by oracle queries during the game’s execution; clearly at most $q \cdot q_{ro}$ such events occur. Let $A’$ be a CGKA* adversary who simulates for $G_1$ adversary $A$ as follows given $F_{i,j} \land E_1$. We assume KIND is such that $A’$ can reveal secrets $k$ output by commit from oracle $O_{Commit}$. Note that our predicate $C_{adm}$ is designed for this simulation.

$A’$ simulates by calling relevant CGKA* oracles and simulating locally otherwise. When $A_i$ calls $\text{getSigKey}$ while simulating $\text{commit}$ calls, $H_{ro}$ is queried with input $r$; $A_i$ lazily samples in this case. When $A$ queries $O_{Inject}$, $A’$ simply returns $\bot$. Note that $O_{Commit}$ invokes commit but does not output the key $k$ that commit outputs. When $A’$ first derives key $k$ from $O_{Commit}$ for the $x$-th CGKA* key pair, for $x \neq j$, $A$ reveals secret $k$ output by $\text{commit}$; $A$ can thus simulate DGS algorithm $\text{commit}$ perfectly. When $A’$ makes their $i$-th query to $H_{ro}$ with input $r$, $A’$ makes an $O_{Challenge}$ query for the corresponding epoch, which outputs $k’$; $A_i$ finishes simulating and returns bit 0 if and only if $k’ = r$. The simulation is perfect, and when $b = 0$ $A_i$ wins iff $A$ outputs 0 only if the correct key or signature public key in the simulation, and when $b = 0$ the challenge oracle outputs the correct key. The $b = 1$ case is similar except in the case that $b = 1$ and the $r$ sampled by the game is the same as the real key (which happens with probability $\frac{1}{2^i}$); it follows that

$$\Pr[G^{A}_1(1^\lambda) = 1 \land E_1 \land F_1] \leq q \cdot q_{ro} \cdot \epsilon_{cgka*} + \frac{q}{2^i}.$$  

We consider $F_2$. Let $F’_i$ be the event, for $1 \leq i \leq q$, that a successful injection is made which uses the $i$-th CGKA* key pair sampled by oracle queries during the game’s execution. Note that such a key pair must exist and that a signature forgery, by algorithm construction, is a necessary but not sufficient condition to make an $O_{Inject}$ query with a non-bottom response. Consider $A_i$ who simulates as follows. $A_i$ simulates all queries locally except that $A_i$ embeds his challenge key in the $i$-th such CGKA* key pair in relevant control messages and uses the $O_{Sign}$ oracle to produce signatures as necessary. Note that the safety predicate is such that, conditioned on $F’_i$, that an $O_{Exposure}$ query that leaks the corresponding signature key is disallowed. In addition, $A$ does not make any $H_{ro}$ query that would lead to a trivial exposure by definition of $F_2$, and thus the distribution of the challenge key pair is uniform (i.e. correct). It follows that the simulation is perfect and that:

$$\Pr[G^{A}_1(1^\lambda) = 1 \land E_1 \land F_2] \leq q \cdot \epsilon_{S}.$$  

The proof is completed by combining the sequence of game hops considered hitherto. 

\[ \square \]

C Correctness Proofs

C.1 Proof of Proposition 1 (IAS correctness)

We want to prove that no adversary $A$ can win $\text{CORR}_{A-CGKA_{corr}}$ (where we set $C_{corr} = \text{true}$) played with respect to IAS (Figures 6 and 7) given that CGKA is correct. We analyze the different game oracles separately and sketch parts derived by direct inspection or based on CGKA correctness.

For $O^{\text{Prop}}$, $A$ can win the game in $O^{\text{Prop}}$ if either $\text{prop}$-info incorrectly interprets the proposal or if the $\text{prop}$ call changes the view of the group. In the first case, correctness follows from the
correctness of CGKA.prop-info if the proposal is standard, and by inspection of IAS otherwise. In the second case, the group view is never changed by prop (as makeAdminProp only updates γ.ssk', γ.spk' in case of an admin proposal, and CGKA.prop is correct in the case of a standard proposal).

OCreate and OCommit can be proven correct by inspection in a similar way.

ODeliver: We examine each reward clause. Note that in line 2 of ODeliver, T is either a commit message created by create or by commit.

We start with the clause \((\emptyset \neq \gamma[\text{gid}].G^* \subseteq \gamma[\text{gid}].G)\). If T is a create message, then line 1 of create and the fact that variable adminList is populated ensures that this condition is fulfilled for T. Upon processing, and after a correct PKI retrieval, p-Wel overwrites \(\gamma\text{.adminList}\) and \(\gamma\text{.s0}\) (via CGKA.proc), so the condition holds. If T is a (standard) commit made by a non-admin user – that is, one without a signature – then there are no changes to the group as checked explicitly by the proc algorithm from line 12. Otherwise, if T an admin commit, then it is processed by p-Comm. We can distinguish some further cases depending on the proposals contained in T:

- If T contains proposals of types upd, upd-adm, add only, then the condition is trivially met.
- If T additionally contains rem proposals, for every removed ID a corresponding rem-adm proposal is generated by an honest admin in propCleaner, hence \(G^* \subseteq G\).
- If T additionally contains add-adm proposals, the VALIDP predicate checks that the added admins are already group members (via \(S_2\)), hence \(G^* \subseteq G\).
- If T additionally contains rem-adm proposals, the final check in enforcePolicy ensures that \(G \neq \emptyset\).
- Any other combination of several contradicting proposals affecting the same ID is handled by enforcePolicy, which prioritizes removals (while preserving admin removals for the same user) which performs a final check on the size of \(G^*\).

We conclude that, for any possible combination of proposals, the condition is always met provided that \(\text{acc} = \text{true}\) with respect to T.

The next cases are the reward props\((ST[\text{ID}], T) \neq T[\text{gid}, (t, c), \langle \text{vec'}, c'\rangle]\) and the reward \(\gamma[\text{gid}], k \neq \bot\) condition given proc outputs \(\gamma\) such that \(\text{ID} \notin \gamma[\text{gid}].G\). It is straightforward to see that both conditions are met by correctness of CGKA.props.

The check by UpdateView rewards the adversary if two users processing the same commit message (on epoch \((t, c)\)) differ in their group view. We show correctness by induction. Suppose \(\text{ID}_1\) and \(\text{ID}_2\) process the same commit message \(T\). If they are in epoch \((-1, -1)\) and process a create message, correctness is easily seen. For the inductive step, we assume that their group views were equal in \((t, c)\), and we want to show that they remain equal after moving to epoch \((t + 1, c')\). The commit is handled by p-Comm, and the only parts that can change for different users are the if condition in line and updAL. The behaviour of both sections of code varies only on the modification of \(\gamma\)'s signature keys, but not on the group structure. Hence, and assuming CGKA correctness, we conclude that \(\text{ID}_1\) and \(\text{ID}_2\) end up having consistent views.

The edge case in which a user is just added to the group is handled by p-Wel, and follows from CGKA correctness and the fact that \(\gamma\text{.adminList} \leftarrow \text{adminList}\) is executed where \text{adminList} is directly provided in T.
Finally, the check \( \text{reward} \gamma[gid].k \neq T[gid, (t, c), 'key', c'] \) follows from the correctness of the \( \text{CGKA.commit} \) algorithm which outputs the new group key \( k \).

C.2 Proof of Proposition 2 (DGS correctness)

We prove that no adversary \( A \) can win \( \text{CORR}_{A, \text{CGKA}, \text{corr}} \) (where we set \( \text{corr} = \text{true} \)) played with respect to DGS (Figures 9 and 10) given that \( \text{CGKA}, \text{CGKA}^* \) are correct. We omit some details which are analogous to IAS’ correctness proof (note that IAS and DGS are designed such that they share sections of their code).

For \( \text{O}^{\text{Prop}} \), the correctness of \( \text{prop-info} \) follows from the correctness of \( \text{CGKA}.\text{prop-info}, \text{CGKA}^*.\text{prop-info} \) by assumption. Also, the adversary cannot win after the group membership check as \( \text{prop} \) only modifies the state by calling \( \text{CGKA}.\text{prop} \) and \( \text{CGKA}^*.\text{prop} \); which are correct by assumption.

The group membership check must always pass in \( \text{O}^{\text{Create}} \) and \( \text{O}^{\text{Commit}} \) for identical reasons.

\( \text{O}^{\text{Deliver}} \): We examine each \( \text{reward} \) clause as done previously with IAS. As before, note that in line 2 of \( \text{O}^{\text{Deliver}} \), \( T \) is a commit message created either by \( \text{create} \) or by \( \text{commit} \). Also, it is easy to see that the \( \text{CGKA}.\text{props} \) check in \( \text{O}^{\text{Prop}} \) holds.

The clause \((\emptyset \neq \gamma[gid].G^* \subseteq \gamma[gid].G)\) is met upon generation of any create message \( T \) (produced by \( \text{O}^{\text{Create}} \)) by construction (line 1 of \text{create}). When any message is processed by \( \text{p-Create} \), the condition is enforced again.

If \( T \) is a (standard) commit made by a non-admin user – that is, one without a signature – then there are no changes to the group in the commit as checked explicitly by the auxiliary \( \text{c-Std} \) upon commit. The message must only contain a \( T_C \) which is processed by \( \text{p-Comm} \), which again enforces this condition.

If \( T \) an admin commit, then it is processed by \( \text{p-Comm} \) or by \( \text{p-Wel} \). In both cases, commit messages are processed by the underlying \( \text{CGKA}, \text{CGKA}^* \) methods. Therefore, correctness depends on the \text{commit} algorithm. The case distinction follows the exact same logic as in IAS, since the algorithms \text{propCleaner} and \text{enforcePolicy}, and the predicate \text{VALID$_P$} enforce the same conditions.

The next case is the \( \text{reward} \gamma[gid].k \neq \perp \) for a removed (or non-member) ID, which is enforced in DGS by \( \text{p-Comm} \) and by the correctness of the \( \text{CGKAs} \).

For the last check by \text{UpdateView}, one can proceed by induction as in IAS. The main difference is that the admin update is not done manually as in IAS (i.e. modifying \text{adminList}), but rather by the underlying \text{proc} algorithms of \( \text{CGKA}^* \), which yields the result easily. The last check for the consistency of the derived group key also follows as in IAS. We omit the details.