How to Hide MetaData in MLS-Like Secure Group Messaging: Simple, Modular, and Post-Quantum

Keitaro Hashimoto*
Tokyo Institute of Technology & AIST
Tokyo, Japan
hashimoto.k.au@m.titech.ac.jp

Shuichi Katsumata†
AIST & PQShield Ltd.
Tokyo, Japan & Oxford, U.K.
shuichi.katsumata@aist.go.jp

Thomas Prest
PQShield SAS
Paris, France
thomas.prest@pqshield.com

ABSTRACT

Secure group messaging (SGM) protocols allow large groups of users to communicate in a secure and asynchronous manner. In recent years, continuous group key agreements (CGKAs) have provided a powerful abstraction to reason on the security properties we expect from SGM protocols. While robust techniques have been developed to protect the *contents* of conversations in this context, it is in general more challenging to protect *metadata* (e.g. the identity and social relationships of group members), since their knowledge is often needed by the server in order to ensure the proper function of the SGM protocol.

In this work, we provide a simple and generic wrapper protocol that upgrades non-metadata-hiding CGKAs into metadata-hiding CGKAs. Our key insight is to leverage the existence of a unique continuously evolving group secret key shared among the group members. We use this key to perform a group membership authentication protocol that convinces the server in an *anonymous* manner that a user is a legitimate group member. Our technique only uses a standard signature scheme, and thus, the wrapper protocol can be instantiated from a wide range of assumptions, including post-quantum ones. It is also very efficient, as it increases the bandwidth cost of the underlying CGKA operations by at most a factor of two.

To formally prove the security of our protocol, we use the universal composability (UC) framework and model a new ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{mh}}$ capturing the correctness and security guarantee of metadata-hiding CGKA. To capture the above intuition of a "wrapper" protocol, we also define a restricted ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$, which roughly captures a non-metadata-hiding CGKA. We then show that our wrapper protocol UC-realizes $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{mh}}$ in the $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$-hybrid model, which in particular formalizes the intuition that any non-metadata-hiding CGKA can be modularly bootstrapped into metadata-hiding CGKA.

KEYWORDS

secure group messaging; metadata-hiding; messaging layer security; continuous group key agreement; generic construction; post-quantum security

1 INTRODUCTION

A *secure group messaging* (SGM) protocol allows a group of users to asynchronously communicate in an *end-to-end encrypted* (E2EE) fashion. With the advent of Signal, SGM protocols have seen exponential growth in practical use, and currently, IETF is almost finished standardizing the Messaging Layer Security (MLS) SGM protocol [14] known to offer good scalability properties.

The baseline security notion guaranteed by all SGM protocols is that no adversary, including the server, should be able to read the messages sent among the group members. However, this does not prevent the adversary from collecting *metadata*, such as the identities of the sender and of other group members, which can both be leaked from the exchanged encrypted contents. It has been shown in numerous real-world scenarios [23, 34, 49, 51, 54] that knowledge of metadata alone can cause damaging repercussions, sometimes enough to defeat the purpose of using SGM protocols. These also have negative impacts on the activity and safety of some users, e.g., journalists and activists [47, 48]. Recent media articles [18, 42] report that, in the United States, metadata collection by law enforcement agencies on *secure messaging applications* is a widespread practice, supported by a legal [42] and technical framework [18] that gives it a wide reach.

Metadata in SGM. In SGM protocols, we can informally divide sensitive information into the following three layers:

$$\left\{\begin{array}{ll} \text{1st layer:} & \text{group secret keys \& messages} \\ \text{2nd layer:} & \text{static, explicit metadata} \\ \text{3rd layer:} & \text{dynamic, implicit metadata} \end{array}\right\} =: \text{“metadata”}$$

Securing the 1st layer is the *default* goal of any SGM protocol; exchanging messages in an E2EE fashion is only possible if secure group secret keys[1] are shared among the group. Since the server is not considered an endpoint of the conversation, state-of-the-art SGM protocols aim at protecting the 1st layer from the server.

The 2nd and 3rd layers together constitute the metadata. Since they help the server to ensure the functionality of the SGM protocol, they are often only encrypted using a transport layer encryption protocol (like TLS or Noise [52]) between the server and the participants. In this case, the server has access to this information and may expose it if legally compelled, as discussed earlier.

The 2nd layer captures any *static* metadata that is *explicitly* leaked from the content transmitted over the channel. For instance, the exchanged content may explicitly include the identity of the sender in the clear or the identity of a member being added, as in e.g. vanilla Signal [45] and MLSPlaintext [14]. Static metadata are also defined as a collection of *sender information* and *handshake messages* in the MLS standard draft [14, Sec. 10.1].

The 3rd layer captures any *dynamic* metadata that is *implicitly* leaked from the access pattern between group members and the

*The author was partially supported by JSPS KAKENHI Grant Number 22J13963, Japan.
†The author was partially supported by JSPS KAKENHI Grant Number 22K17892, Japan and JST AIP Acceleration Research JPMJCR22U5, Japan.

[1]There could either be a unique group secret key shared among the entire group as in MLS or multiple group secret keys, where different segments of the keys are shared among different members of the group as in Signal to perform pairwise communications.

Table 1: SGM protocols and the corresponding layers they protect. "✔✔" (resp. "✘") indicates that there is a (resp. are no) security proof. "✔" indicates that there is a proof capturing the security of the 2nd & 3rd layers but not of the 1st layer.

Layer	1st	1st & 2nd	1st, 2nd & 3rd
Signal	Vanilla Signal	⊥	Private Groups
Security proof	✘	⊥	✔ [25]
MLS	MLSPlaintext	MLSCiphertext	⊥
Security proof	✔✔ E.g., [9, 12]	✘	⊥

server via the communication channel. For example, in MLSCiphertext that hides up to the 2nd layer, users connect to the server using a non-anonymous protocol such as TLS. Then, since the user implicitly identifies itself to the server via the channel, the server learns the group member's identities (and how many times each member accessed) by observing the users accessing the same group identity. In particular, this happens regardless of protecting the 2nd layer metadata. To hide the user identity on the channel, users can use anonymous protocols (e.g., Tor [1, 31]) instead. However, even if an anonymous protocol is used when a user fetches information about a group they belong to, the exact subset of accessed information may be correlated to this user's identity. This may for example be the case for SGM protocols that arrange group members in complex data structures such as trees. Specifically, hiding the metadata only at the 2nd layer while using anonymous channels is insufficient since similar information may be inferred from the 3rd layer, which incorporates all implicit leakages of dynamic metadata.

In this work, only when all three layers are secured do we say that an SGM protocol is *metadata-hiding*.[2]

Existing metadata-hiding SGM. Existing SGM protocols and the level of layers they protect are depicted in Tab. 1. Signal recently proposed a metadata-hiding SGM protocol that we call *Private Groups* [3]. This is an extension of *Sealed Sender* [2] — a metadata-hiding *two-user* secure messaging protocol. The main building block of Private Groups is an efficient *MAC-based keyed-verification anonymous credential* (KVAC) [24] that leverages the specific properties of tools from classical group-based cryptography, such as the ElGamal PKE and Schnorr PoK. While there is no formal security proof for Signal's vanilla SGM, recently Chase, Perrin, and Zaverucha [25] proposed a new security model to capture exactly the metadata layers (2nd & 3rd layers) and provided a partial security proof of Private Groups.

MLS [14] comes in two variants: MLSPlaintext and MLSCiphertext, each corresponding to protocols protecting the 1st and 1st & 2nd layers, respectively. The security of MLSPlaintext has been scrutinized over the past few years [8, 9, 12, 17, 19] and we now have a good understanding of it. However, no formal security proof for MLSCiphertext is known. Moreover, unlike Signal's Private Groups, constructing any variant of MLS that further hides the *dynamic* metadata is unanswered. Considering that dynamic metadata leaks part of, if not all, static metadata, MLSCiphertext may be leaking more metadata in practice than ideally expected.

[2]Note that there are other types of metadata we can consider such as access timing [46] and geolocation of users.

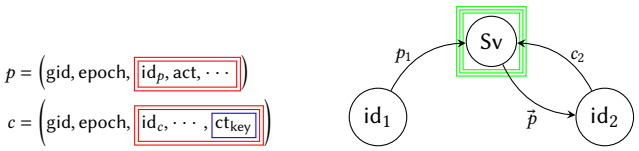


Figure 1: (Left) p and c are proposals and commits. (Right) id_1 uploads a proposal p_1 to the server Sv; id_2 downloads all the stored proposals $\vec{p}$ and uploads a commit c_2 for the next epoch. The single (resp. double, triple) bordered box indicates the 1st (resp. 2nd, 3rd) layer information.

1.1 Goal of This Work

In this work, we focus on *continuous group key agreement* (CGKA) — an abstraction that captures the core protocol underlying the MLS protocol, i.e., TreeKEM [16], and many other MLS-inspired SGM protocols [5, 8–12, 30, 36, 41].[3] In brief, CGKA allows an evolving group of users to agree on a continuous sequence of group secret keys. Other than the simple function of sharing a group secret key, CGKA further models the strong notions of *forward secrecy* (FS) and *post-compromise security* (PCS) [7, 28, 29], which allow to greatly limit the scope of a compromise.

In a nutshell, a CGKA works as follows (see also Fig. 1):[4] A group member may either (a) add a new member, (b) remove a member, and/or (c) update its keys by sending a *proposal* p. In an arbitrary interval, a group member may download the list of proposals $\vec{p} = \{ p_i \}_i$ from the server and take them into effect by transmitting a *commit* c — this creates a new epoch where the group state is updated according to $\vec{p}$. Importantly, a commit also updates the group secret key to achieve PCS.

A proposal p consists of five elements: (i) a string gid identifying the group; (ii) a counter epoch that specifies the current group state; (iii) the identity id_p of the member creating p; (iv) a string act specifying whether p corresponds to (a), (b), or (c); and (v) other information typically required for authentication. A commit c has a similar structure, where id_c denotes the committer and $\mathsf{ct}_{\mathsf{key}}$ is a ciphertext encrypting a key used to update the group secret.

As depicted in Fig. 1, key is the 1st layer information, and any other static information included in p and c other than (gid, epoch) belong to the 2nd layer. Here, (gid, epoch) needs to be clear so that the receiver can download the appropriate p and c from the server. In the past few years, we have seen several increasingly stronger or different types (e.g., game-based, simulation-based) of security models for CGKA [5, 8–12, 19, 37, 41, 57], however these models only capture security at the 1st layer. Although it is straightforward to construct a CGKA that intuitively secures the 2nd layer once a group secret key is established, it is not clear whether this intuition is correct. Indeed, MLSPlaintext has undergone 13 iterations, and formal security analyses of the 1st layer [8, 12] uncovered some subtle bugs. Thus, our first goal is the following:

(G1) *Propose a security model capturing the security of the 1st & 2nd layers and prove the security of existing CGKAs.*

As discussed above, securing the 2nd layer alone is insufficient. At first glance, it is tempting to replace the use of TLS for client-server communication with a client-anonymized authenticated channel

[3]To be accurate, MLS was inspired by *asynchronous ratchet trees (ART)* [30].
[4]We base the explanation on the most recent iterations of TreeKEM (i.e., after version 8 on MLS) following a "*propose-and-commit*" flow.

(e.g., VPN or an anonymized proxy such as Tor [1, 31]) in order to hide the 3rd layer. Unfortunately, this introduces another issue since, without any authentication on the client side, any adversary who knows (gid, epoch) can upload arbitrary garbage proposals and commits to the server, causing a denial of service (DoS) against the group. It could be possible to rely on the efficient MAC-based KVAC used by Signal's Private Groups [3], however, their construction is highly limited to a classical, pre-quantum setting, and the security proof is in the generic group model [55]. Considering the modularity of the vanilla Signal and the MLS protocol, having a generic construction that can be efficiently instantiated from versatile assumptions, including but not limited to *post-quantum* assumptions, is highly desirable. Of independent interest, we note that in the face of a compromise or removal of a group member, Private Groups must restart a new group [25]. It remains an interesting problem to construct a protocol that offers any (non-trivial) PCS. This brings us to our second goal:

(G2) *Propose an efficient and generic metadata-hiding CGKA achieving the same level of FS and PCS offered by existing non-metadata-hiding CGKAs.*

Finally, Chase, Perrin, and Zaverucha [25] proposed a security model capturing the 2nd & 3rd layers of Signal. However, it does not capture the 1st layer of security, i.e., confidentiality and integrity of exchanged messages, nor the notion of PCS. Moreover, this model is tailored to the specific construction of Signal's Private Groups [3] and seems unfit for CGKA. Thus, we arrive at our final goal:

(G3) *Propose a security model for metadata-hiding CGKA.*

1.2 Our Contributions

UC model for the 2nd layer. We propose the first security model of CGKA capturing the security of the 2nd layer, i.e., static metadata. Our security model extends the state-of-the-art universal composability (UC) security model used by Alwen et al. [12] and Hashimoto et al. [36] to analyze TreeKEM version 10 in MLS and Chained CmPKE, respectively — we denote the ideal functionality as $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$.[5] The main new ingredients we introduce are *leakage functions* that allow us to formally model the leaked static metadata (e.g., sender's identity) from the proposals and commits. Similar to the state-of-the-art ideal functionalities, ours captures a strong model where *active adversaries* can tamper with or inject messages, and *malicious insiders* can invite malicious members to the group and arbitrarily fork the group state. With this formalization effort, we answer the first half of (**G1**), see Sec. 3 for details.

Chained CmPKE$^{\text{ctxt}}$ UC-realizes $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. We prove that a ciphertext variant of Chained CmPKE by Hashimoto et al. [36], coined as Chained CmPKE$^{\text{ctxt}}$, UC-realizes the ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$.[6] Considering the similarity between Chained CmPKE and TreeKEM, we believe the ciphertext variant of TreeKEM can also be proven secure following a similar proof. The reason why we focused on the former is that it is in some sense a generalization of the latter — it allows members to *selectively download* updates from the server, also known as *filtered* CGKA [11]. This generalization allows obtaining a concretely efficient CGKA even in the *post-quantum* regime, which otherwise could be quite inefficient [36]. This security proof addresses the second half of (**G1**), see Sec. 3 for details.

UC model for the 3rd layer. We propose the first UC security model of CGKA capturing the security of the 3rd layer, i.e., group access pattern — we denote the ideal functionality as $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$. Any CGKA that UC-realizes $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ is a *metadata-hiding* CGKA. The model captures the fact that a group member performing an upload or download remains anonymous and unlinkable from the server, while also restricting non-group members from accessing the group contents. To formalize the latter property, our ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ captures an *honest-but-curious* server for the first time. All prior models only considered *malicious* servers so it was not possible to define a "correct" behavior of the server, i.e., shutting out non-group members. $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ allows the adversary to corrupt the server, in which case it becomes identical to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ defined above. This answers (**G3**), see Sec. 5 for details.

A generic and efficient protocol UC-realizing $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$. We provide a simple and generic wrapper protocol W^{mh} that UC-realizes $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ in the $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$-hybrid model. Specifically, given an *arbitrary* CGKA Π_{ctxt} that UC-realizes $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$, W^{mh} in composition with Π_{ctxt} UC-realizes $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$. Unlike Signal's Private Groups, we do not rely on complex tools such as a MAC-based KVAC, whose known efficient instantiations require classical group-based assumptions. Our key insight is to leverage the unique group secret key shared among the members (which is non-existing in Signal) to perform a proof of membership to the server. The concrete construction of our wrapper protocol only requires a standard signature scheme, which can be efficiently instantiated using either classical or post-quantum assumptions. Our metadata-hiding CGKA inherits all the FS and PCS properties satisfied by the underlying CGKA satisfying $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. For instance, using MLSCiphertext as the underlying CGKA, the upload cost of a key update of our metadata-hiding CGKA can be $O(\log N)$, as opposed to $O(N)$ for Private Groups. This provides a theoretic answer to (**G2**), see Sec. 4 for details.

Instantiation and efficiency analysis. We provide concrete instantiations of our protocols under either classical or post-quantum assumptions. We then study the bandwidth impact of W^{mh} when applied to Chained CmPKE$^{\text{ctxt}}$. The impact of W^{mh} is moderate, as it never increases the bandwidth cost of the principal operations ("update", "add" and "remove" proposals, as well as commit or application messages) by more than a factor of two. In practice, the concrete overhead may be even lower. This illustrates that our notion of metadata-hiding CGKA can be realized at a moderate cost. This covers the efficiency aspects of (**G2**), see Sec. 6.2 for details.

Statistical leakage from metadata-hiding CGKA. Our security model allows us to prove that a CGKA UC-realizes an ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ *with respect to a specific leakage function*, which defines any inherent metadata leakage that cannot be hidden. What the adversary can learn from this leakage function is another question.

We initiate a discussion on the nature and extent of the information that can be inferred from this leakage. We study several CGKAs [5, 11, 12, 36, 41] and find that all of them leak information through the size of protocol messages (welcome, proposal and/or commit), sometimes in surprising and indirect ways. At this point, we emphasize that the authors of these protocols never claimed

[5] The subscript "ctxt" is inspired by the protocol name MLSCiphertext
[6] This variation mirrors what MLSCiphertext does to MLSPlaintext, in the sense that it encrypts the static metadata by applying a layer of encryption.

them to be metadata-hiding, so this leakage does not reflect the shortcomings of the designs. We believe that a systematic study of this leakage, as well as proposing countermeasures to provably mitigate it, constitutes a valuable and exciting research direction. For now, this discussion explores the limitations of **(G3)**, see Sec. 7 for details.

We would like to clarify the limitations of this work. First, we only consider the CGKA aspect of SGMs. While it is believed that CGKA captures the essence of SGMs — which is supported by the vast amount of research focusing solely on CGKA [5, 8–12, 19, 36, 41, 57] — to argue a provably secure metadata-hiding SGM, we would need to extend our security model to cover the message exchanging layer as well. It was only recently that a security model that captures the entire SGM at the 1st layer was proposed [9]. Second, our security model and protocol do not prevent an adversary from anonymously registering numerous fake groups on the server. We only prevent an outsider from accessing an existing group. Technically, this seems efficiently solvable using standard anonymous credentials [26] and we leave it as future work to incorporate these into our security model. Finally, metadata outside the scope of our models, such as access timing [46] and device fingerprinting, may circumvent the privacy guarantees provided by a metadata-hiding CGKA.

2 BACKGROUND

We now provide some background on the existing definition of CGKA that models the default security of the 1st layer, i.e., *group secret key*.[7] In this work, we focus on CGKA defined in the UC framework. With the nice composability property, we will be able to construct a metadata-hiding CGKA in a modular manner.

Due to page limitation, we refer the readers to App. B for some background on the UC framework and a formal definition of the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$ capturing the UC security of the 1st layer.

2.1 Syntax of CGKA

All the group members internally store the group identifier gid, the current epoch of the group, the current group secret key, and information to identify the current members. Moreover, each member (roughly) stores a public key whose associated secret key is known only to themselves. This is called the *key material*.

Members can upload *proposals* p to the server, which can take on three types: addition or removal of some members, and update of key materials. Members can download any subset of these proposals $\vec{p}$ at an arbitrary interval to create a *commit* c. Any members can then *process* this commit to update their group state to the next epoch according to the content of the proposals that were committed. The group secret key is always updated after a process — if every group member's key material was uncompromised at that epoch, then this effectively *heals* the group and offers PCS.

In this work, we follow the syntax of Hashimoto et al. [36] (which extends the syntax of Alwen et al. [12]) that models *selective downloading*. This allows the members to only download part of the commit required to move to the next epoch. Specifically, a commit c is divided into $(c_0, \vec{c})$, where c_0 is the member-*independent* commit and $\vec{c} = (\widehat{c}_{\mathsf{id}})_{\mathsf{id}}$ is the list of member-*dependent* commit. Member id only needs to download $(c_0, \widehat{c}_{\mathsf{id}})$ from the server to advance its epoch. Here, we assume there is a canonical ordering of the group members, and member id requests its index in the list $\vec{c}$ to the server to download $\vec{c}[\mathsf{index}] = \widehat{c}_{\mathsf{id}}$. Such selective downloading can bring great efficiency gain — especially in the post-quantum regime where asymmetric ciphertexts tend to be much larger than classical ones (see [36] for further motivations). This formalization was later coined as *filtered* CGKA [11].

Informally, CGKA is defined by the following algorithms, where we assume id is the executing party and omit it from the input.

Group Creation ($\mathtt{Create}$)**:** It initializes a new group state with party id as the only member.[8]

Proposals ($\mathtt{Propose}$, act) $\to p$**:** It outputs a proposal p for the *action* act that can take on the value 'add'-id_t, 'rem'-id_t, or 'upd'. The first two actions dictate the adding or removal of id_t. The last updates id's key material.

Commit ($\mathtt{Commit}, \vec{p}$) $\to (c_0, \vec{c}, \vec{w})$**:** It commits a vector of proposals $\vec{p}$ and outputs a commit $(c_0, \vec{c})$. c_0 is a member-independent commit while $\vec{c} = (\widehat{c}_{\mathsf{id}'})_{\mathsf{id}'}$ is a list of member-dependent commits, where $|\vec{c}|$ is equal to the current group size. If $\vec{p}$ contains an add proposal, then it outputs a welcome message $\vec{w} = (\widehat{w}_{\mathsf{id}_t})_{\mathsf{id}_t}$, where id_t denotes the added members.[9]

Process ($\mathtt{Process}, c_0, \widehat{c}_{\mathsf{id}}, \vec{p}$)**:** It processes a commit $(c_0, \widehat{c}_{\mathsf{id}})$ with the associated proposals $\vec{p}$, and advances id's internal group state to the next epoch.

Join ($\mathtt{Join}, \widehat{w}_{\mathsf{id}}$)**:** It allows id to join the group using the welcome message $\widehat{w}_{\mathsf{id}}$. id's group state is synced with any member who processes the commit made at the same epoch.

Key ($\mathtt{Key}$) $\to$ **k:** It outputs the current group secret key k.

2.2 Default UC Security Model of CGKA

All prior works on CGKA capture the rough security notion that the group secret key should remain hidden from the adversary. We follow the state-of-the-art UC security model of [12, 36]. In this model, the malicious server is modeled as an *active adversary* that can tamper with or inject messages. Moreover, a rogue group member is modeled as a *malicious insider* that can invite other malicious members (e.g., server) to the group and arbitrarily fork the group state. The UC security model captures the fact that even in face of such strong adversaries, the group secret key remains indistinguishable from random under certain conditions. Below, we explain the high-level description of the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$.

History graphs and the safe predicate. The core concept underlying the definition is the so-called *history graph* [9, 12] maintained by $\mathcal{F}_{\mathrm{CGKA}}$. A history graph is a symbolic representation of the group's evolution, where each node on a graph roughly corresponds to a group state at a particular epoch.[10] It tracks all the generated commits and proposals, and group members' positions

[7] As mentioned in Sec. 1.2, CGKA only captures the security of the group secret key k. The message, also included in the 1st layer, is handled by a different protocol. Roughly, the messages are sent through E2EE using the established k.

[8] Following prior definitions [10, 12, 36], we assume Group Creation is run only once.

[9] Although we can also structure $\vec{w}$ to have a party independent w_0 and dependent part, we chose not to do so since it leads to a less secure metadata-hiding protocol. Roughly, by looking at w_0, the server can infer who will be added to the same group.

[10] The formal definition is made with more care since in case an adversary forks the group state, two different nodes with the same epoch can be created.

on the history graph. To define the security of the group secret key, $\mathcal{F}_{\text{CGKA}}$ is parameterized by a predicate **safe**, which takes the history graph and a node as input, and assigns a random group secret key $k \leftarrow \mathcal{K}$ to a node where **safe** is true. This formalization allows modeling FS and PCS naturally. For example, an adversary $\mathcal{A}$ can corrupt a member at some epoch, thus making **safe** false at epoch. If the member is *healed* at a later epoch$'$ > epoch, then **safe** becomes true again at epoch$'$. Importantly, **safe** is a scheme-specific predicate that can be defined arbitrarily to capture different levels of FS and PCS.

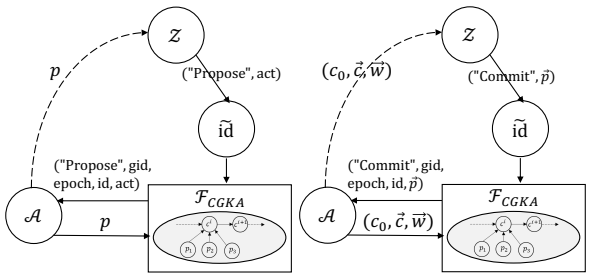


Figure 2: Group operations in the UC security model. $\mathcal{Z}$ invokes the dummy party $\widetilde{\text{id}}$, and $\mathcal{F}_{\text{CGKA}}$ informs the adversary $\mathcal{A}$ of this invocation. $\mathcal{A}$ simulates the corresponding proposal or commit and sends it to $\mathcal{F}_{\text{CGKA}}$. W.l.o.g., we assume $\mathcal{A}$ sends the same information to $\mathcal{Z}$, denoted by a dashed line. The shaded region denotes the history graph maintained by $\mathcal{F}_{\text{CGKA}}$.

Example: Ideal propose and commit. We depict the ideal `Propose` and `Commit` functions in Fig. 2, where $\mathcal{Z}$ denotes the environment. For example, $\mathcal{Z}$ can invoke the (dummy) party $\widetilde{\text{id}}$ to execute a commit on an arbitrary list of proposals $\vec{p}$. $\mathcal{F}_{\text{CGKA}}$ then provides to $\mathcal{A}$ the proposals $\vec{p}$ along with all the static metadata (gid, epoch, id) included in a commit, where id denotes the identity of the committer. $\mathcal{A}$ interprets the proposals $\vec{p}$ and simulates the commit $(c_0, \vec{c})$, where it further simulates the welcome message $\vec{w}$ if $\vec{p}$ includes an add proposal. The commit and welcome messages are sent to $\mathcal{F}_{\text{CGKA}}$ and are registered in the history graph as a new node indicating that a new epoch has been created. Here, if $\vec{p}$ was generated by a group member, then $\mathcal{F}_{\text{CGKA}}$ mandates correctness by checking if the commit output by $\mathcal{A}$ was consistent with the actions included in $\vec{p}$. Otherwise, in case some $p \in \vec{p}$ was not generated by a group member, then this implies that the server or some rogue insider injected a *malicious* proposal.

3 HIDING STATIC METADATA IN CGKA

As a first step, we propose a UC security model capturing the security of the 1st & 2nd layers (i.e., group secret keys and *static* metadata) by defining a new ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. Similarly to the **safe** predicate used in $\mathcal{F}_{\text{CGKA}}$ to control the levels of FS and PCS, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ comes with five *leakage functions* allowing to control the amount of static metadata leaked from create group, proposal, commit, process, and join. By defining the leakage functions to leak all the static metadata, then $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ (essentially) recovers the prior ideal functionality $\mathcal{F}_{\text{CGKA}}$.

We then prove that a ciphertext variant of Hashimoto et al.'s Chained CmPKE [36] satisfies $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$, where the leakage functions are defined to only leak the minimal static metadata matching our intuition.

3.1 UC Security Model for Static Metadata

$\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ has the same user interface (or syntax) as $\mathcal{F}_{\text{CGKA}}$. The only difference is how the internals of the ideal functionalities are defined. The full details on $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ is provided in App. B. Below, we explain the two main points at which $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ differs from $\mathcal{F}_{\text{CGKA}}$.

Modeling static metadata leakage. Recall how $\mathcal{F}_{\text{CGKA}}$ defined the ideal proposal function (see Fig. 2). When a party id is invoked on (`Propose`, act) from the environment $\mathcal{Z}$, $\mathcal{F}_{\text{CGKA}}$ informs the adversary $\mathcal{A}$ with (`Propose`, gid, epoch, id, act). $\mathcal{A}$ then simulates a proposal p. This models the fact that p in the real world is allowed to leak information on (gid, epoch), the party id creating p, and the type of proposal act $\in$ { 'add'-id_t, 'rem'-id_t, 'upd' } included in p.

We control the amount of such leakage from p by a leakage function ***leak-prop**. Informally, ***leak-prop** takes as input the identity id of the member creating the proposal and the current epoch. In case **safe** is false at epoch, ***leak-prop** outputs all the static metadata since the group secret key is compromised at that epoch. Otherwise, it only outputs the static metadata the CGKA is allowed to leak. For example, to model MLSCiphertext, we define ***leak-prop**(id, epoch) to only output (gid, epoch, |id|, |act|) when **safe** is true, where |act| leaks the size of the action included in p. In case every member identity id_t is encoded in the same bit-length, then this implies for instance that p does not leak who created the p and who was added and removed.

Similarly to ***leak-prop**, we define four more leakage functions ***leak-create**, ***leak-proc**, ***leak-com**, and ***leak-wel**. For instance, the last two controls the amount of static metadata leaked from commits and welcome messages, respectively. Unlike proposals, the static metadata that is inherently leaked from commits and welcome messages differs between CGKAs. Thus, care is required when formally defining them. For instance, in MLSCiphertext, a welcome message for member id_t includes a hash of id_t's key package kp_{id_t} (i.e., key materials used by id_t). To model this fact, ***leak-wel** must also output kp_{id_t} to the adversary (see App. C.2 for more discussion). Another subtlety is that the size of a commit in MLSCiphertext is dictated by how many blank nodes exist in the tree. In Sec. 7, we discuss the real-world consequences of these inherent leakages of static metadata.

Using semantics for nodes in history graphs. In this work, we update the prior definition of history graphs to use the *semantics* of a transcript to identify the nodes in a history graph. That is, we identify a node by a counter that informally counts the number of group operations leading to the node. In previous works [12, 36] (which did not capture the security of the static metadata), each node was identified by the *value* of the *non-encrypted* member-independent commit c_0. Since c_0 uniquely defined an epoch and group state, it made intuitive sense to identify each node by c_0.

Unfortunately, this intuition is lost when we try to further secure the static metadata. This is because c_0 is now an *encryption* of the actual commit content. Namely, there can be two distinct c_0 and c_0' that decrypt to the same commit content. While we would like to

assign different members processing c_0 and c_0' to the same node in the history graph, it is not clear which c_0 and c_0' to use to identify the node. Such an issue disappears by using the semantics since the group operation defined inside c_0 and c_0' are identical.

Independently, Alwen et al. [11] also uses the semantics to define nodes in their security model. This was crucial to capture a more advanced form of (non-metadata-hiding) CGKA coined as *server-aided* CGKA.

Restricted adversary due to commitment problem. In the UC framework, it is typical to restrict the adversary $\mathcal{A}$ from performing corruptions that would cause the so-called *commitment problem*. Informally, this is a type of attack where $\mathcal{A}$ can adaptively choose to corrupt some states *after* being provided with some challenges with respect to the state. While this attack is prohibited by default in any natural game-based definition (e.g., the adversary cannot obtain a secret key after being provided the challenge ciphertext), we need to make this restriction explicit in the UC-based definition.

Compared to prior works in the UC framework [10, 12, 36], the description of the restriction we require is more strict. Previously, when a new node was created due to a commit, the predicate **safe** was undefined for that node. Roughly, this is because the group secret key inside that node was never explicitly used during the real protocol and $\mathcal{A}$ was given the freedom to adaptively decide whether to corrupt that node at an arbitrary moment of the security game. However, in the static metadata-hiding setting, this freedom of the adversary needs to be restricted. This is because when a new node is created, the group secret key inside this node is explicitly used to encrypt the static metadata of the commit content. When **safe** is true at this node, the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ (roughly) wants to assign a random ciphertext to model the fact that the commit does not leak any information. Since $\mathcal{A}$ is explicitly given the ciphertext, it cannot later decide to corrupt the node (i.e., **safe** remains true), as otherwise, it could trivially distinguish a valid ciphertext from a random one.

3.2 Proof of Static Metadata-Hiding CGKA

Hashimoto et al. [36] proposed Chained CmPKE and showed that it UC-realizes $\mathcal{F}_{\mathrm{CGKA}}$. We show that the ciphertext variant of Chained CmPKE, which we call Chained CmPKE$^{\mathrm{ctxt}}$, UC-realizes $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$. The construction itself is a variation mirroring what MLSCiphertext does to MLSPlaintext. Thus, the technical novelty of this section is the first formal security proof that models static metadata-hiding. Considering that the overall structure of Chained CmPKE is similar to TreeKEM, except that the former further supports selective downloading, we expect our proof techniques to naturally translate to the ciphertext variant of TreeKEM. We leave it as an important future work to formally validate this. Below, we provide the core insights of our security proof. The full detail is provided in App. C.

Recycling previous proofs. As explained earlier, the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ *almost* degenerates to $\mathcal{F}_{\mathrm{CGKA}}$ when the leakage functions are defined to leak all static metadata. Our proof takes advantage of this fact. Recall that to prove security in the UC framework, we informally need to construct a simulator $\mathcal{S}$ that simulates the real world adversary $\mathcal{A}$. We thus try to construct a simulator $\mathcal{S}^{\mathrm{ctxt}}$ for $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ that internally executes $\mathcal{S}$ for $\mathcal{F}_{\mathrm{CGKA}}$. Using the description of $\mathcal{S}$ provided in Hashimoto et al. [36], the hope is to recycle all the proofs provided by them (which spanned 30 pages!). At a high level, during the hybrids where the leakage functions do not hide any static metadata, $\mathcal{S}^{\mathrm{ctxt}}$ can use knowledge of the group encryption key to encrypt whatever non-encrypted proposals and commits $\mathcal{S}$ outputs, and decrypt whatever $\mathcal{S}$ inputs to perform a process. This allows us to recycle all the proofs that ignore the security of the static metadata. We then gradually modify the definition of the leakage functions to arrive at our ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$. In the final hybrid, $\mathcal{S}^{\mathrm{ctxt}}$ no longer knows the group encryption key for those epochs where **safe** is true, and $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ (roughly) takes care of generating random encryption of a proposal and commit.

While the high-level idea of recycling the proof of Hashimoto et al. [36] sounds straightforward, it turns out to be easier said than done. The non-triviality comes from the fact that the history graph created by $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ while using the null-leakage function is *not* identical to those created by $\mathcal{F}_{\mathrm{CGKA}}$ – it is only *almost* identical. As explained earlier, our new history graph uses the semantics to identify the nodes and captures settings unique to $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$. Since all the security arguments boil down to how the predicate **safe** is defined over the history graph constructed during the security proof, it is not clear how to relate the proof of Hashimoto et al. to ours. To this end, we define the notion of *isomorphisms* of history graphs so that a security proof translates within the same class of history graphs. At a high level, two history graphs are isomorphism if the symbolic representation of the group evolution is identical. We prove that the two history graphs created in $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ and $\mathcal{F}_{\mathrm{CGKA}}$ are isomorphic.

Polynomial security loss. We like to point out that our security proof only admits a *polynomial* security loss. This is in contrast to Hashimoto et al. [36] that admitted an *exponential* security loss.[11]

The main reason for this disparity is because Chained CmPKE$^{\mathrm{ctxt}}$ captures a larger part of a CGKA compared to Chained CmPKE. As explained earlier, Chained CmPKE does not use the group secret key to encrypt proposals and commits, while Chained CmPKE$^{\mathrm{ctxt}}$ does. Effectively, in the former, an adversary was able to *adaptively* corrupt the group secret key while not trivially winning the security game. However, once the group secret key is used in a higher-level protocol as in Chained CmPKE$^{\mathrm{ctxt}}$, this is no longer the case. An adversary capable of corrupting the group secret key *after* seeing the ciphertext can trivially win the security game.[12] Since the exponential security loss in Hashimoto et al. [36] was due to the adaptivity of the adversary, we can remove this loss by naturally restricting such adversaries in our UC-security model.

Remark 1 (Adversary-controlled randomness). *In case of corruption, [12, 36] allow the adversary $\mathcal{A}$ in $\mathcal{F}_{CGKA}$ to set the randomness to be used by the corrupted member* id *by altering the value* RandCorr[id]. *While our ideal functionality $\mathcal{F}_{CGKA}^{\mathrm{ctxt}}$ models such strong adversaries, we restrict the adversary in our security proof to never alter the randomness for simplicity and better readability.*

[11] We note that they were able to achieve a polynomial security loss relying on a stronger form of multi-recipient PKE secure against adaptive corruptions.

[12] We note that this problem may theoretically be solved using *non-committing* encryptions [22], but we did not consider them as a viable option as it will add a noticeable overhead in efficiency.

Recent definitions [9, 11] also intentionally disregard this type of attack.[13] *We leave it as future work to incorporate adversary-controlled randomness into the proof.*

4 CONSTRUCTING METADATA-HIDING CGKA

We now construct a simple and modular CGKA construction that enables us to secure the 3rd layer (i.e., *dynamic* metadata). This results in the first *metadata-hiding* CGKA. Technically, we construct a wrapper protocol W^{mh} specifically taking care of the 3rd layer security in the $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$-hybrid model. Specifically, W^{mh} can be wrapped around *any* CGKA Π_{ctxt} that UC-realizes $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ and bootstraps Π_{ctxt} into a metadata-hiding CGKA.

The formal security proof of our metadata-hiding CGKA is deferred to Sec. 5, where we propose a new UC security model capturing the 3rd layer. We refer the readers to App. E for the full detail on the construction of W^{mh}. Below, we provide an overview of our protocol.

4.1 Goal of the Wrapper Protocol W^{mh}

To claim that CGKA secures the 3rd layer, our goal is to informally construct a wrapper protocol W^{mh} with the following properties:

(1) **Anonymous upload.** A group member id can anonymously upload a proposal p or a commit $(c_0, \vec{c})$ to the server.
(2) **Anonymous download.** A group member id can anonymously download a commit $(c_0, \vec{c})$ from the server.
(3) **Unlinkability.** A member performing multiple uploads and downloads remains unlinkable from the server.
(4) **Group authentication.** Only members of the group — excluding the *honest* server — can upload to and download from the server.

Here, unlinkability (Item (3)) is a strictly stronger notion compared to anonymity (Item (1)). Even if the member remains anonymous, some protocol may allow the server to link whether two uploads came from the same member. We also note that Item (4) is only relevant when the server is *honest* — a malicious server can always allow a non-member to perform an upload or a download on behalf of the group.

A wrapper protocol W^{mh} satisfying the above conditions is sufficient to bootstrap any CGKA that UC-realizes $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ into a metadata-hiding CGKA. However, if the underlying CGKA supports *selective downloading*, such as Chained CmPKE, Item (2) fails to provide the same efficiency offered by the underlying selective downloading CGKA. We thus strengthen Item (2) as follows.

(2^+) **Anonymous selective download.** A group member id can anonymously and *selectively* download a partial commit $(c_0, \widehat{c}_{\mathsf{id}})$ from the server.

Specifically, the wrapper protocol W^{mh} allows the download cost to remain independent of the group member size in case a selective downloading CGKA is internally used.

Finally, even if the group secret key becomes compromised, we want all the above properties to hold again once the group state is healed. Namely, we seek a protocol with the following property.

(5) **Compromise Resilience.** W^{mh} inherits the FS+PCS guarantees offered by the underlying CGKA UC-realizing $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$.

Insufficiency of client-anonymized authenticated channel. One natural idea is to use a client-anonymized authenticated channel such as a VPN or an anonymized proxy like Tor [1, 31]. While such a channel solves Item (1) (and Item (2)), this alone cannot solve Items (2^+) to (5). For instance, when selective downloading is performed, member id needs to specify its index in the group to retrieve the partial commit $(c_0, \widehat{c}_{\mathsf{id}})$. Even if index may not directly leak id, the second time id performs a download on the same index, it will break linkability (Item (3)). Moreover, since the client-side is unauthenticated, it does not prevent external adversaries to perform a denial of service (DoS) attack on the group by uploading garbage contents, thus contradicting Item (4). Recall here that the server can no longer explicitly check if the uploaded contents come from genuine group members since the static metadata including the identity of the uploading member is hidden.

In summary, a client-anonymized authenticated channel alone is not enough to hide dynamic metadata.

4.2 High Level Description of W^{mh}

We provide an overview of our wrapper protocol W^{mh}. The main idea is to use the unique group secret key k exchanged among the group to perform an efficient proof of membership to the server. To make the presentation simple, we deliberately provide an informal description of our protocol.

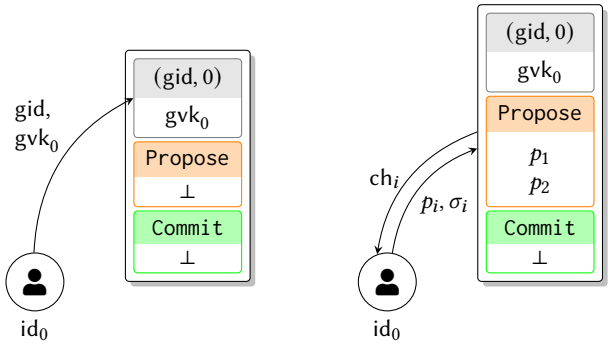
Below, we assume all parties have access to the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$. Moreover, we assume the party communicates with the server via a client-anonymized authenticated channel, except when a new member is retrieving a welcome message from the server.

Fig. 3a: Group registration. Assume party id_0 wishes to create a group of three members $(\mathsf{id}_0, \mathsf{id}_1, \mathsf{id}_2)$. id_0 first registers a new *empty* group to the server. Informally, id_0 invokes $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ on input (`Create`) and initializes a new group identifier gid and a group secret key k_0 for epoch = 0. It then *deterministically* creates a group specific signature key $(\mathsf{gvk}_0, \mathsf{gsk}_0) \leftarrow \mathsf{KeyGen}(1^\kappa; \mathsf{PRF}(\mathsf{k}_0, \text{'auth'}))$ from the group secret key k_0. We call this verification key gvk_0 as the *group statement* for epoch = 0. Party id_0 then uploads the pair $(\mathsf{gid}, \mathsf{gvk}_0)$ to the server.[14] Finally, the server creates a new database for the group gid.

Fig. 3b: Initial proposal to add members. With the database for gid set up on the server, id_0 next adds id_1 and id_2 to the group. Specifically, id_0 invokes $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ on input (`Propose`, 'add'-id_i) and generates an add proposal p_i for $i \in [2]$. To upload p_i on the server, id_0 proves that it is a member of the group gid by essentially executing an identification protocol with the server. The server sends a random challenge $\mathsf{ch}_i \leftarrow \{0,1\}^\kappa$ and id_0 creates a signature $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{gsk}_0, \mathsf{ch}_i)$. The server verifies that σ_i is a valid signature with respect to the group statement gvk_0 at epoch = 0. If so, it adds p_i to the database. Due to the unforgeability of the signature scheme, no party without the group signing key gsk_0 can impersonate a group member.

[13] Alwen et al.[9] does not allow adversaries to manipulate randomness used for symmetric key encryption.

[14] Our protocol W^{mh} does not prevent a malicious id_0 from registering multiple groups. As explained in Sec. 1.2, one possible way to thwart such a DoS attack would be to use anonymous credentials [26].



(a) Group registration **(b) Initial proposals from id_0**

Figure 3: Creation of a group by a party id_0. The three-part box represents the group state at a given epoch, as stored on the server. The top box stores in the clear the group identifier gid, epoch and group statement gvk_0. The middle box stores the (encrypted) proposals created during the epoch. The bottom box stores the (encrypted) commit message which concluded this epoch, if it exists.
To prove membership, the server sends a challenge ch_i and id_0 responds with a signature σ_i (Fig. 3b). The contents are exchanged over a client-anonymized authenticated channel.

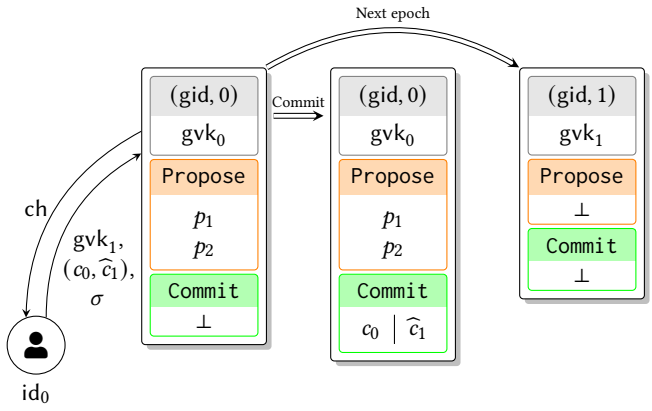


Figure 4: An initial commit made by the group creator id_0. The server freezes the state of the current epoch = 0, and initializes a new epoch in the database.

Fig. 4: Initial commit to execute initial proposals. To create a new group with members id_1 and id_2, id_0 must commit the proposals $\vec{p} = (p_i)_{i\in[2]}$. It first invokes $\mathcal{F}^{\mathsf{ctxt}}_{\mathsf{CGKA}}$ on input $(\mathtt{Commit}, \vec{p})$ and generates $(c_0, \widehat{c}_0, \vec{w} = (\widehat{w}_i)_{i\in[2]})$[15] and (roughly) updates the group secret key k_1 for the next epoch = 1. Similar to the group registration phase, id_0 creates a group statement gvk_1 for epoch = 1. To upload the commit $(c_0, \widehat{c}_0)$, id_0 performs the same identification protocol as in Fig. 3b to prove that he is indeed a member of the group gid. If the identification protocol succeeds, the server stores the commit on the database and further creates a new column for the next epoch = 1.

Finally, id_0 uploads the welcome messages $\vec{w}$ to the server. The welcome messages are stored on a party-dependent database and work identically to $\mathcal{F}^{\mathsf{ctxt}}_{\mathsf{CGKA}}$. In particular, id_1 and id_2 can retrieve $\widehat{w}_1$ and $\widehat{w}_2$, respectively, from the server and execute $\mathcal{F}^{\mathsf{ctxt}}_{\mathsf{CGKA}}$ on input $(\mathtt{Join}, \widehat{w}_i)$ to have the same group state as id_0. Effectively, they become a member of the group gid at epoch = 1.

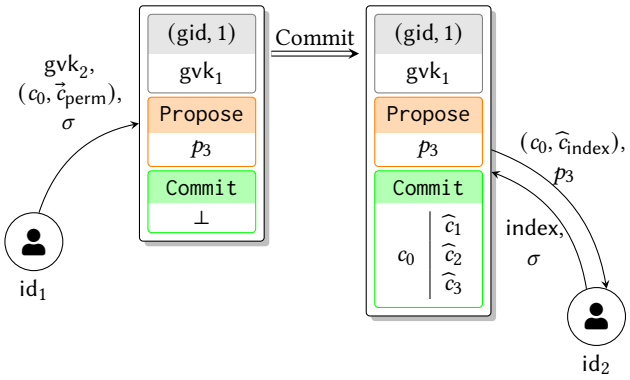


Figure 5: Left: a commit sent by id_1. Right: a subsequent process made by id_2. Member-dependent commits $\vec{c}_{\mathsf{perm}} := (\widehat{c}_1, \widehat{c}_2, \widehat{c}_3)$ are randomly permuted and id_2 specifies an index to fetch the commit from the server. Challenges sent by the server are omitted for readability.

Commits without selective downloading. Now that id_1 and id_2 joined the group gid at epoch = 1 (i.e., share the same group secret key k_1), they can upload proposals and commits to the database defined with respect to the group statement gvk_1.

We now explain the structure of a commit message when the group has more than one member. Assume some member made an update proposal p_3, and id_1 wishes to commit this proposal (left half of Fig. 5). Then, following the same procedure as the initial commit, id_1 invokes $\mathcal{F}^{\mathsf{ctxt}}_{\mathsf{CGKA}}$ on input $(\mathtt{Commit}, p_3)$ and generates $(c_0, \vec{c} = (\widehat{c}_i)_{i\in[3]})$, along with an updated group statement gvk_2 for the next epoch = 2. If selective downloading is not performed, then id_1 can simply upload $(c_0, \vec{c})$ to the server. The server then initiates a new column for epoch = 2 and the other members can anonymously download the entire commit from the server (by performing the aforementioned identification protocol).

Issues with naive selective downloading. Unfortunately, if selective downloading is naively applied, the above method leaks the access pattern of group members.

Recall that when selective downloading is performed in $\mathcal{F}^{\mathsf{ctxt}}_{\mathsf{CGKA}}$, a member sends an index and receives the corresponding member-dependent commit $\widehat{c}_{\mathsf{index}}$ from the server. When a member selectively downloads commit messages relative to two distinct epochs, they send the same index for both epochs. The server can infer that both requests were made by the same party, contradicting Item (3).

Even worse, suppose that the group secret key, and thus, the group member list is compromised. Although the group secret key may heal after PCS of $\mathcal{F}^{\mathsf{ctxt}}_{\mathsf{CGKA}}$ kicks in, the access pattern will never

[15] Note that id_0 must process the commit $(c_0, \widehat{c}_1)$ to move to the next epoch = 1.

heal; the server who learned the member-index correspondence can permanently break *anonymity* when selective downloading is performed by simply looking at the same index used at every epoch, contradicting Items (2^+) and (5).

Commits with *oblivious* selective downloading. While the problem exposed above is known to be solvable using relatively complex tools such as private information retrieval (PIR) [27], we provide a much simpler solution using a pseudorandom permutation PRP by taking advantage of the fact that selective downloading is performed by each group member *once per epoch*. Continuing with our above example, when id_1 generates a commit $(c_0, \vec{c} = (\widehat{c}_i)_{i\in[3]})$, it further *deterministically* generates a PRP key $\mathsf{permKey} \leftarrow \mathsf{PRF}(\mathsf{k}_1, \text{'perm'})$ which defines a permutation over the group size, which is $[3] = \{1, 2, 3\}$ for our example. It then creates a permuted member dependent commit $\vec{c}_{\mathsf{perm}}$ so that $\widehat{c}_i$ is placed at entry $\mathsf{PRP}(\mathsf{permKey}, i) \in [3]$. Finally, id_1 uploads $(\mathsf{gvk}_2, (c_0, \vec{c}_{\mathsf{perm}}))$ to the server by performing the identification protocol using gsk_1.

When id_2 performs a selective download to retrieve the proposal and (partial) commit from the server, it computes its permuted $\mathsf{index} = \mathsf{PRP}(\mathsf{permKey}, 2)$, where id_2 generates an identical permKey as id_1. It performs an identification protocol using gsk_1, sends index, and retrieves $(c_0, \widehat{c}_{\mathsf{index}})$ and the proposal p_3 from the server. This is illustrated in the right half of Fig. 5. id_1 can then invoke $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ on input $(\mathtt{Process}, (c_0, \widehat{c}_{\mathsf{index}}), p_3)$ and move to the next epoch = 2.

Observe that a member never performs a selective download more than once per epoch. This is the main reason why a PRP suffices — the access would have been linkable if selective downloading was performed more than twice per epoch using the same PRP key. Moreover, since the group secret key is updated at each epoch, the PRP key is also updated, thus satisfying FS and PCS (Item (5)).

Remark 2 (Non-Interactive Membership Identification). *We provided a challenge-response type* interactive *identification protocol to prove group membership. By allowing the server to perform additional checks on the database and further reasonably weakening the security guarantee (i.e., Item (4) is guaranteed only for uploads), we are able to make the protocol completely* non-interactive. *At a high level, the party simply needs to sign the proposal or commit (rather than a challenge message) to upload, and perform no membership identification to download. The full detail is provided in App. E.2.*

5 FORMAL MODEL FOR METADATA-HIDING CGKA

We define a UC security model capturing the security of the entire 1st, 2nd & 3rd layers (i.e., group secret keys, static and *dynamic* metadata) by defining a new ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$. Any CGKA UC-realizing $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ is provably a *metadata-hiding* CGKA.

Reusing most of the description of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ handling the 1st and 2nd layers, the description of $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ can focus mainly on the 3rd layer. Our model succinctly captures all the properties explained in Sec. 4.1, Items (1) to (5). We then show that the wrapper protocol W^{mh} presented in the previous section UC-realizes $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ in the $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$-hybrid model. The full details of this section is provided in App. D. Below, we provide an overview of our idea.

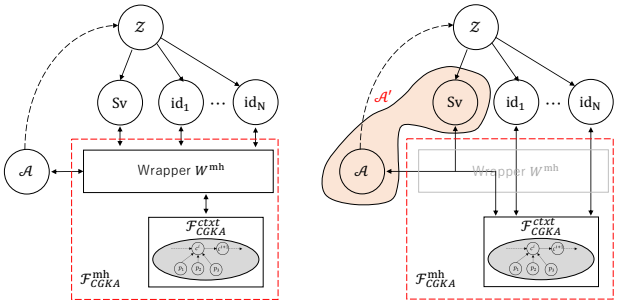


Figure 6: (Left) metadata-hiding CGKA $\Pi^{\mathsf{mh}}_{\mathbf{CGKA}}$ when the server Sv is honest and (Right) when Sv is malicious. The red dotted box denotes the entire protocol $\Pi^{\mathsf{mh}}_{\mathbf{CGKA}}$, where it is further decomposed as a combination of the wrapper protocol W^{mh} and ideal functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathbf{CGKA}}$. The red shaded region denotes that Sv is corrupted and that $(\mathcal{A}, \mathrm{Sv})$ are viewed as a single adversary $\mathcal{A}'$. In this case, W^{mh} is ignored and $\Pi^{\mathsf{mh}}_{\mathbf{CGKA}}$ degenerates to a UC-realization of $\mathcal{F}^{\mathsf{ctxt}}_{\mathbf{CGKA}}$.

5.1 Modeling an Honest but Curious Sever

In previous constructions of CGKA, the server was assumed to be *always* malicious. This is for instance captured in the ideal functionalities $\mathcal{F}_{\mathrm{CGKA}}$ and $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ by observing that the `Commit` and `Process` take as input arbitrary proposals and commits — not just those created by the honest group members.

While assuming the server to be always malicious allows to capture a strong level of security against group secret keys and static metadata, this is far too inflexible for our use case. Recall Sec. 4.1, Item (4). To properly model that any non-member cannot upload and download from the server on behalf of the group, we must model an *honest but curious* server — a server that honestly follows the protocol but tries to learn as much metadata as possible.

To this end, we explicitly incorporate a server into our model as depicted in Fig. 6. We allow the server to be in two states: honest[16] or corrupt. When the server is honest, we are able to properly model Sec. 4.1, Item (4). Otherwise, since a malicious server can arbitrarily choose to accept or reject the identification protocol executed by the wrapper protocol W^{mh}, W^{mh} does not provide any meaningful functionality. In particular, our metadata-hiding CGKA $\Pi^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ degenerates to offer the same functionality as $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

5.2 UC Security Model for Dynamic Metadata

As already mentioned, the new ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ inherits all functionalities offered by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ comes with seven additional functionalities:

- `RegisterGroup`,
- `PublishProposal`, `FetchProposals`,
- `PublishCommit`, `FetchCommit`,
- `PublishWelcome`, `FetchWelcome`.

As the name indicates, `Publish-`$*$ (resp. `Fetch-`$*$) is invoked to upload (resp. download) a proposal, commit, or welcome message from

[16]We drop "but curious" for simplicity.

the server. These functions are mainly invoked during an execution of `Create`, `Propose`, `Commit`, `Process`, and `Join`. For instance, when `Commit` is invoked, the member first runs `FetchProposals` to retrieve the proposals $\vec{p}$ from the server and invokes $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ on input $(\texttt{Commit}, \vec{p})$. It then uploads the commit and welcome message output by $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ using `PublishCommit`.

These functions are defined differently depending on (i) whether the calling party is an honest group member or an adversary,[17] and (ii) whether the server is honest or malicious. As explained above, if the server is malicious, then we let the adversary $\mathcal{A}$ (i.e., server) decide whether `Publish-`$*$ or `Fetch-`$*$ succeeds. In this case, $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{mh}}$ becomes functionally identical to $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$, modulo some syntactical difference due to the inclusion of the server into the model.

Otherwise, if the server is honest, then $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{mh}}$ captures the correctness and security guarantees. For correctness, if the calling party is a group member and the group statement for the next epoch was honestly generated, then the functionality demands the server to accept the upload or download. On the other hand, for security, if the calling party is the adversary, then we require the server to reject the upload or download as long as the predicate **safe** at that epoch is true. That is, if the group secret key is not compromised, then the adversary should not be able to upload and download on behalf of the group.

Additionally, the database stored by the server (see Figs. 3 to 5) is modeled in $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{mh}}$ by two lists PropDB and ComDB, each maintaining the proposals and commit for group gid at epoch. $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{mh}}$ also models the permutation-based selective downloading explained in Sec. 4.2 by an ideal (helper) function `*permute-commit`. Finally, due to the already complex nature of the metadata-hiding CGKA, we did not model adversarially controlled randomness in $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{mh}}$. We leave this as an important future work.

5.3 Proof of Dynamic Metadata-Hiding CGKA

We prove that the wrapper protocol W^{mh} UC-realizes the metadata-hiding ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{mh}}$ in the $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$-hybrid model. The full proof is provided in App. E.3.

The proof is relatively simple and modular since we defined W^{mh} in the $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$-hybrid model. We can in essence ignore all the description of $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{mh}}$ that relates to the 1st and 2nd layers' correctness and security since the same checks can be handled by the simulator $\mathcal{S}$ internally simulating $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$. Specifically, our proof only needs to focus on the 3rd layer of correctness and security. The proof is standard and consists of invoking the security of the pseudorandom permutation and signature scheme.

6 INSTANTIATION AND EFFICIENCY

We now discuss how to instantiate W^{mh} and Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$. We target the so-called "NIST Level I"[18], which informally states that breaking the protocol is no easier than key-recovery on a block cipher with a 128-bit key (e.g. AES128). This provides a meaningful baseline to discuss post-quantum security and ignoring quantum attacks corresponds to a classical security level of 128 bits. Given a cryptographic object x, we note |x| its size in bytes.

[17] In our security definition (in App. D), we use `Publish-`$*$`-Adv` and `Fetch-`$*$`-Adv` to indicate that the calling party is the adversary.

[18] https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/evaluation-criteria/security-(evaluation-criteria)

Table 2: Comparison of selected mPKEs. An mPKE ciphertext has the form $\left(\mathsf{ct}_0, (\widehat{\mathsf{ct}}_{\mathsf{id}})_{\mathsf{id}\in[N]}\right)$ when uploaded, and $\left(\mathsf{ct}_0, \widehat{\mathsf{ct}}_{\mathsf{id}}\right)$ when downloaded by id. All sizes are in bytes.

mPKE	Reference	$\lvert\mathsf{ek}\rvert$	$\lvert\mathsf{ct}_0\rvert$	$\lvert\widehat{\mathsf{ct}}_{\mathsf{id}}\rvert$
ElGamal-based	[43] + [40]	32	32	32
SIKEp434-based	[36]	330	330	16
Ilum512	[36]	768	704	48
Bilbo640	[36]	10240	10240	24

6.1 Instantiation

The main cryptographic primitives used in our protocols are: (a) two signature schemes SIG and SIG′, (b) a multi-recipient PKE mPKE, and finally (c) symmetric primitives: a pseudo-random function PRF, a symmetric encryption scheme SKE and a pseudo-random permutation PRP.

Multi-recipient PKE. For pre-quantum security, one may combine Kurosawa's multi-recipient variant [43] of ElGamal with the transform of [40], whose decomposability property makes it amenable to the selective downloading performed by Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$. For post-quantum security, one may readily use one of the mPKEs proposed in [36]: Ilum512, Bilbo640 or a SIKEp434-based mPKE. For completion, we recall their performance profiles in Tab. 2.

Signature schemes. We choose signature schemes that complement our chosen mPKEs nicely, either by having similar performances, being based on similar assumptions, or both:

- The ECDSA standard is based on similar assumptions as of the ElGamal-based mPKE, and it has a similar performance profile as well.
- The NIST PQC finalist Falcon [53] complements our SIKEp434-based mPKE, as they both have small communication costs.
- The NIST PQC finalist Dilithium [44] is based on Module-LWE (plus Module-SIS), just like Ilum512.
- SPHINCS$^+$ [38] and Bilbo640 are both based on assumptions perceived as conservative (hash-based assumptions and unstructured LWE), and they both have comparatively larger communication costs than other schemes.

For simplicity, we consider that SIG and SIG′ use the same scheme, but using distinct schemes may lead to interesting trade-offs.

Pseudo-random permutation. We require a PRP in order to permute the set of $[N]$ group members. There are at least two viable approaches:

- *Shuffling.* One may use a shuffling algorithm whose randomness is provided by passing permKey into a PRF. If cache attacks on group members' devices are not a concern, the Fisher-Yates shuffle is a good choice since it is simple, performs $N-1$ swaps, and its entropy consumption is optimal. If cache attacks are a concern, one may need to use so-called *oblivious* algorithms. For example, with the Thorp

shuffle [50], each member may compute their permuted index in time $O(\log N)$.
- *Sorting.* This approach assigns to each member id a pseudorandom value $r_{\mathsf{id}} = \mathsf{H}(\mathsf{permKey}, \mathsf{id})$, then sorts the id's according to the r_{id}'s. The sorting step can be done obliviously in time $O(N \log^2 N)$ using sorting networks, for example, Batcher networks. Assuming H is collision-resistant, this provides a PRP over $[N]$.

While the solutions proposed above are not optimal when used in `*permuted-commit-index` (each group member needs to shuffle/sort *all* indices before finding their position), we expect them to be significantly less costly than public-key cryptographic operations, even for concretely large groups.

6.2 Efficiency

We now study the impact of W^{mh} on bandwidth efficiency, by applying it to Chained CmPKE$^{\mathsf{ctxt}}$. The results are summarized in Tab. 3, with the overhead of W^{mh} represented in bold red font **(+X)**.

Let us make two observations. First, W^{mh} adds the same overhead to any CGKA protocol realizing the ideal functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathsf{CGKA}}$ (e.g., our Chained mPKE$^{\mathsf{ctxt}}$ and MLSCiphertext[19]). Second, as is obvious in Tab. 3, the bandwidth overhead is added only in the *upload* direction, since the additional signatures serve to convince the server that a user is a legitimate group member.

For a more concrete perspective on the numbers provided in Tab. 3, we study the impact of applying W^{mh} over Chained CmPKE$^{\mathsf{ctxt}}$, when these protocols are instantiated with the four mPKE-signature pairs selected in Sec. 6.1. The results are given in Tab. 4. For our examples, W^{mh} increases the bandwidth cost of `Propose`-'upd', `Propose`-'add', `Propose`-'rem', `Commit`, `Process` and application messages by at most 44%, 63%, 100%, 57%, 39% and 100%, respectively.[20] We believe this to be a very reasonable overhead if protecting metadata is considered important.

7 LIMITATION OF EFFICIENT METADATA-HIDING CGKA

We conclude this work by discussing some inherent limitations of metadata-hiding CGKAs. Our model and solution hide metadata from messages but allow leaking message size. This may reveal information about the structure and activity of the group. While remaining informal, our discussion highlights that the nature and extent of this information depend on the inner workings of the CGKA, as well as the precise topology of the group at a given time (especially for tree-based CGKAs).

7.1 Chained CmPKE and TreeKEM

We first discuss Chained CmPKE [36], as well as variants of TreeKEM *without server assistance* [8, 9, 12, 14, 41], meaning that the server forwards messages to group members without editing them.

Leakage from proposal messages. When the size of proposal messages depends on their type, these messages leak the proposal type (i.e., 'upd', 'add', 'rem'). For Chained CmPKE, this is evident from Tab. 4. Even though this can be fixed by padding, we note that at least for Chained CmPKE, commit messages subsequent to a successful 'add' (resp. 'rem') will increase (resp. decrease) by $|\widehat{\mathsf{ct}}_{\mathsf{id}}|$ bytes. For TreeKEM, a less obvious yet similar correlation exists. On the bright side, the size of proposals is independent of the group size and the sender's identity.

Leakage from commit messages.

Chained CmPKE [36]. The size of an uploaded commit message is affine in the group size N. Thus, the server can infer N from commit messages. Chained CmPKE allows selective downloading but the wrapper W^{mh} hides the index of each party-dependent message by randomizing indices via per-epoch random permutation. Thus, from the server's point of view, indices look random.

TreeKEM [8, 9, 12, 14]. The number of PKE ciphertexts in a commit message depends on the topology of the ratchet tree and the sender's position in this tree, therefore the size of a commit message can leak information about both elements. We provide two examples.

First, suppose that the tree is full (i.e., no blank node) but the group size is not a power of two. Since TreeKEM uses left-balanced binary trees, parties assigned to leaves "at the left" of the tree will send longer commit messages than the ones "on the right". Hence the server can easily partition the set of parties into two groups depending on the length of commit messages.

Second, we consider the tree in Fig. 7, which has some blank nodes. In this tree, the number of cryptographic materials sent by each party is as follows: parties 1 and 2 (resp. 3 and 4, resp. 5 and 6, resp. 7) send 3 encryption keys + 5 ciphertexts (resp. 3 + 4, resp. 3 + 3, resp. 2 + 2 encryption keys and ciphertexts).

Tainted TreeKEM [41]. In this variant of TreeKEM, commit messages contain cryptographic materials related to tainted nodes, nodes managed by a party other than on the direct path. Message size becomes larger if the sender manages more tainted nodes.

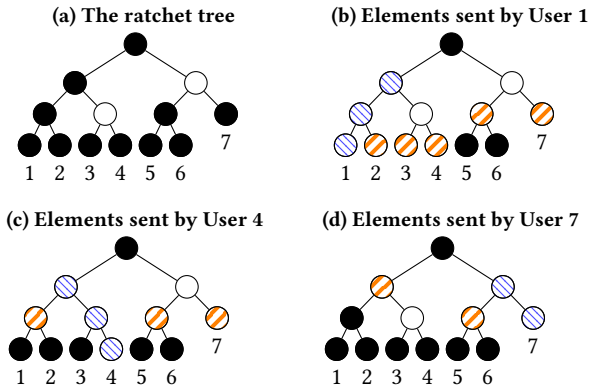


Figure 7: Example of the TreeKEM ratchet tree with blank nodes. The numbers indicate party identities. Figs. 7b to 7d: sending encryption keys ▨ and ciphertexts ▨ when party 1, 4 or 7 commits, respectively.

[19] Although no formal proof exists, it is believed that MLSCiphertext realizes $\mathcal{F}^{\mathsf{ctxt}}_{\mathsf{CGKA}}$.
[20] For simplicity, numbers for `Commit` and `Process` in Tab. 4 consider an idealized setting where no proposal was made during the last epoch. In practice, this is not the case and the bandwidth overhead of W^{mh} for `Commit` and `Process` will be even lower in percentage.

Table 3: Bandwidth overhead of our wrapper W^{mh} applied to Chained CmPKE$^{\mathsf{ctxt}}$, for a group of N members in terms of public key cryptography elements. The nominal bandwidth cost of Chained CmPKE$^{\mathsf{ctxt}}$ is written in normal font (X). The overhead of W^{mh} is written in bold red (+X). U (resp. A, R) stands for the number of 'upd' (resp. 'add', 'rem') proposals published during the last epoch.

Procedure	Upload					Download				
	|ek|	$\|\mathsf{ct}_0\|$	$\|\widehat{\mathsf{ct}}_{\mathsf{id}}\|$	|sig|	|svk|	|ek|	$\|\mathsf{ct}_0\|$	$\|\widehat{\mathsf{ct}}_{\mathsf{id}}\|$	|sig|	|svk|
Propose-'upd'	1			2 **(+1)**							
Propose-'add'	1			1 **(+1)**	1						
Propose-'rem'				1 **(+1)**							
Commit	1	1	N	2 **(+2)**	**(+1)**	U+A			2U+A+R	A	
Process				**(+1)**		U+A+1	1	1	2U+A+R+2	A	
Applications messages				1 **(+1)**							

Table 4: Concrete overhead of W^{mh} in terms of public key cryptography elements, when instantiated with: (a) ElGamal-mPKE + ECDSA (E+E), (b) SIKEp434-mPKE + Falcon (S+F), (c) Ilum512 + Dilithium (I+D), (d) Bilbo640 + SPHINCS$^+$ (B+S). We assume a group of $N = 256$ members, and numbers for Commit and Process assume no proposal was made during the last epoch. The nominal bandwidth cost of Chained CmPKE$^{\mathsf{ctxt}}$ is written in normal font (X), and the overhead of W^{mh} is written in bold red (+X). All sizes are in bytes.

Procedure	E+E		S+F		I+D		B+S	
Propose-'upd'	160	**+64**	1 662	**+666**	5 608	**+2 420**	44 416	**+17 088**
Propose-'add'	128	**+64**	1 893	**+666**	4 500	**+2 420**	27 360	**+17 088**
Propose-'rem'	64	**+64**	666	**+666**	2 420	**+2 420**	17 088	**+17 088**
Commit	8 384	**+160**	6 088	**+2 229**	18 600	**+6 152**	60 800	**+34 208**
Process	224	**+64**	2 008	**+666**	6 360	**+2 420**	54 680	**+17 088**
Applications messages	64	**+64**	666	**+666**	2 420	**+2 420**	17 088	**+17 088**

Leakage from welcome messages. In both protocols, welcome messages leak the receiver's identity, the key package's hash, and the group size. Note that the receiver's identity must be in the clear for the server to deliver welcome messages. Key package hashes are used by the receivers to identify which decryption key is necessary to process the welcome message.

First of all, the server always gets to know when a welcome message was fetched. In addition, if the welcome message did not hide the dynamic metadata (i.e., uploader's identity), then the server can infer that the party creating the welcome message and the party fetching it belong to the same group. Since groups are created by adding new members through a welcome message, this minor leakage of metadata can be used to trace the entire group member.

However, even if the welcome message hides the dynamic metadata, the server may still be able to link welcome messages to a specific group in some scenarios. For example, assume a party at an insecure $\mathsf{epoch}_{\mathsf{cur}}$ commits an add proposal and moves to a secure $\mathsf{epoch}_{\mathsf{next}}$. Using a client-anonymized authenticated channel to upload the welcome message, the party uploading the welcome message remains anonymous, thus protecting against the above attack. However, since the server gets to see the key package included in the add proposal issued at the insecure $\mathsf{epoch}_{\mathsf{cur}}$, the server can link this key package to the key package hash included in the welcome message to infer that the new member joins the group related to $\mathsf{epoch}_{\mathsf{cur}}$. In particular, while the key package hash included in the welcome message is good for efficiency, it may have non-trivial side effects.

A simple way to prevent such information leakage is to remove key package hashes from welcome messages. However, this would require recipients to try decrypting with all their registered decryption keys since they no longer can determine which decryption key can be used to decrypt the received welcome message. Finally, we note that welcome messages will always leak the size of the group to which the party was newly added. This is because a welcome message in essence sends all the current group states to the newly added party.

7.2 Server-Aided Variants of TreeKEM

We discuss recent efficient variations of TreeKEM [5, 11]. These allow the server to perform special tasks, e.g., editing signatures or ratchet trees in addition to delivering messages. They improve efficiency by revealing metadata to the server. Devising metadata-hiding variants of these schemes would likely require devising ways to perform these editing operations in an oblivious way, similarly to the PRP-based solution, we proposed for Chained CmPKE$^{\mathsf{ctxt}}$.

SAIK^{21} [11]. This TreeKEM variant uses reducible signatures, a variant of redactable signatures, to allow parties to selectively download parts of commit messages while still guaranteeing their validity with the same signature that was initially uploaded by the sender. This improves the overall communication cost. However, the server

needs to know the identities of both the sender and receiver in order to reduce the commit message before forwarding it to the receiving party. In SAIK, this reduction is more involved than the selective downloading of Chained CmPKE. Even assuming that the reduction itself can be performed obliviously, the size of the reduced message depends on the positions of both the sender and receiver, so it would still leak information about both parties.

CoCoA[22] [5]. This variant allows the server to merge multiple commit messages for the purpose of reducing communication costs. The server keeps the public part of the group state and creates the next group state by merging concurrently issued commit messages. Then, it forwards to each group member the part of the new state they need. This results in $O(\log N)$ upload and download costs, and a $O(\log^2 N)$ total cost. However, the server needs to know the public part of group states, which in particular includes the group member list.

One can say that these schemes improve efficiency by giving more information to the server. In other words, there is a trade-off between efficiency and privacy. We view it as an interesting research direction to construct more efficient (and practical) CGKA protocols while still protecting metadata.

REFERENCES

[1] [n.d.]. Orbot: Proxy with Tor. https://guardianproject.info/apps/org.torproject.android/ https://guardianproject.info/apps/org.torproject.android/.
[2] 2018. Technology preview: Sealed sender for Signal. https://signal.org/blog/sealed-sender/ https://signal.org/blog/sealed-sender/.
[3] 2019. Technology Preview: Signal Private Group System. https://signal.org/blog/signal-private-group-system/ https://signal.org/blog/signal-private-group-system/.
[4] Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. 2022. How to Abuse and Fix Authenticated Encryption Without Key Commitment. To Appear at USENIX 2022.
[5] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. 2022. CoCoA: Concurrent Continuous Group Key Agreement. In *EUROCRYPT 2022, Part II (LNCS)*. Springer, Heidelberg, 815–844. https://doi.org/10.1007/978-3-031-07085-3_28
[6] Joël Alwen, Margarita Capretto, Miguel Cueto, Chethan Kamath, Karen Klein, Ilia Markov, Guillermo Pascual-Perez, Krzysztof Pietrzak, Michael Walter, and Michelle Yeo. 2021. Keep the Dirt: Tainted TreeKEM, Adaptively and Actively Secure Continuous Group Key Agreement. In *2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Los Alamitos, CA, USA, 596–612. https://doi.org/10.1109/SP40001.2021.00035
[7] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. 2019. The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol. In *EUROCRYPT 2019, Part I (LNCS, Vol. 11476)*, Yuval Ishai and Vincent Rijmen (Eds.). Springer, Heidelberg, 129–158. https://doi.org/10.1007/978-3-030-17653-2_5
[8] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. 2020. Security Analysis and Improvements for the IETF MLS Standard for Group Messaging. In *CRYPTO 2020, Part I (LNCS, Vol. 12170)*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer, Heidelberg, 248–277. https://doi.org/10.1007/978-3-030-56784-2_9
[9] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. 2021. Modular Design of Secure Group Messaging Protocols and the Security of MLS. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 1463–1483. https://doi.org/10.1145/3460120.3484820
[10] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. 2020. Continuous Group Key Agreement with Active Security. In *TCC 2020, Part II (LNCS, Vol. 12551)*, Rafael Pass and Krzysztof Pietrzak (Eds.). Springer, Heidelberg, 261–290. https://doi.org/10.1007/978-3-030-64378-2_10
[11] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. 2021. Server-Aided Continuous Group Key Agreement. Cryptology ePrint Archive, Report 2021/1456. https://eprint.iacr.org/2021/1456.
[12] Joël Alwen, Daniel Jost, and Marta Mularczyk. 2022. On the Insider Security of MLS. In *CRYPTO 2022, Part II (LNCS)*. Springer, Heidelberg, 34–68. https://doi.org/10.1007/978-3-031-15979-4_2
[13] Manuel Barbosa and Pooya Farshim. 2007. Randomness reuse: Extensions and improvements. In *IMA International Conference on Cryptography and Coding*. Springer, 257–276.
[14] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. 2022. *The Messaging Layer Security (MLS) Protocol*. Internet-Draft draft-ietf-mls-protocol-13. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-13 Work in Progress.
[15] Mihir Bellare and Chanathip Namprempre. 2000. Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT 2000 (LNCS, Vol. 1976)*, Tatsuaki Okamoto (Ed.). Springer, Heidelberg, 531–545. https://doi.org/10.1007/3-540-44448-3_41
[16] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. 2018. *TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS)*. Research Report. Inria Paris. https://hal.inria.fr/hal-02425247
[17] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. 2019. *Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS*. Research Report. Inria Paris. https://hal.inria.fr/hal-02425229
[18] Thomas Brewster. 2022. Meet The Secretive Surveillance Wizards Helping The FBI And ICE Wiretap Facebook And Google Users. Forbes. https://www.forbes.com/sites/thomasbrewster/2022/02/23/meet-the-secretive-surveillance-wizards-helping-the-fbi-and-ice-wiretap-facebook-and-google-users/.
[19] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. 2021. Cryptographic Security of the MLS RFC, Draft 11. Cryptology ePrint Archive, Report 2021/137. https://eprint.iacr.org/2021/137.
[20] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd FOCS*. IEEE Computer Society Press, 136–145. https://doi.org/10.1109/SFCS.2001.959888
[21] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. 2007. Universally Composable Security with Global Setup. In *TCC 2007 (LNCS, Vol. 4392)*, Salil P. Vadhan (Ed.). Springer, Heidelberg, 61–85. https://doi.org/10.1007/978-3-540-70936-7_4
[22] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. 1996. Adaptively Secure Multi-Party Computation. In *28th ACM STOC*. ACM Press, 639–648. https://doi.org/10.1145/237814.238015
[23] Bjorn Carey. 2015. Stanford computer scientists show telephone metadata can reveal surprisingly sensitive personal information. https://news.stanford.edu/2016/05/16/stanford-computer-scientists-show-telephone-metadata-can-reveal-surprisingly-sensitive-personal-information/.
[24] Melissa Chase, Sarah Meiklejohn, and Greg Zaverucha. 2014. Algebraic MACs and Keyed-Verification Anonymous Credentials. In *ACM CCS 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM Press, 1205–1216. https://doi.org/10.1145/2660267.2660328
[25] Melissa Chase, Trevor Perrin, and Greg Zaverucha. 2020. The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption. In *ACM CCS 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, 1445–1459. https://doi.org/10.1145/3372297.3417887
[26] David Chaum. 1982. Blind Signatures for Untraceable Payments. In *CRYPTO'82*, David Chaum, Ronald L. Rivest, and Alan T. Sherman (Eds.). Plenum Press, New York, USA, 199–203.
[27] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private Information Retrieval. In *36th FOCS*. IEEE Computer Society Press, 41–50. https://doi.org/10.1109/SFCS.1995.492461
[28] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. 2017. A Formal Security Analysis of the Signal Messaging Protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. 451–466. https://doi.org/10.1109/EuroSP.2017.27
[29] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. 2016. On Post-compromise Security. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. 164–178. https://doi.org/10.1109/CSF.2016.19
[30] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. 2018. On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees. In *ACM CCS 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, 1802–1819. https://doi.org/10.1145/3243734.3243747
[31] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. 2004. Tor: The Second-Generation Onion Router. In *USENIX Security 2004*, Matt Blaze (Ed.). USENIX Association, 303–320.
[32] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. 2018. Fast Message Franking: From Invisible Salamanders to Encryptment. In *CRYPTO 2018, Part I (LNCS, Vol. 10991)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, Heidelberg, 155–186. https://doi.org/10.1007/978-3-319-96884-1_6
[33] Pooya Farshim, Claudio Orlandi, and Razvan Rosie. 2017. Security of symmetric primitives under incorrect usage of keys. *IACR Transactions on Symmetric*

[21] SAIK stands for Server-Aided Insider-Secure TreeKEM.
[22] CoCoA stands for COncurrent COntinuous group key Agreement.

Cryptology (2017), 449–473.
[34] Ola Flisbäck. 2015. Stalking anyone on Telegram. https://oflisback.github.io/telegram-stalking/.
[35] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. 2017. Message Franking via Committing Authenticated Encryption. In *CRYPTO 2017, Part III (LNCS, Vol. 10403)*, Jonathan Katz and Hovav Shacham (Eds.). Springer, Heidelberg, 66–97. https://doi.org/10.1007/978-3-319-63697-9_3
[36] Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. 2021. A Concrete Treatment of Efficient Continuous Group Key Agreement via Multi-Recipient PKEs. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 1441–1462. https://doi.org/10.1145/3460120.3484817
[37] Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. 2021. A Concrete Treatment of Efficient Continuous Group Key Agreement via Multi-Recipient PKEs. Cryptology ePrint Archive, Report 2021/1407. https://eprint.iacr.org/2021/1407.
[38] Andreas Hulsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. 2020. *SPHINCS+*. Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.
[39] Daniel Jost, Ueli Maurer, and Marta Mularczyk. 2019. A Unified and Composable Take on Ratcheting. In *TCC 2019, Part II (LNCS, Vol. 11892)*, Dennis Hofheinz and Alon Rosen (Eds.). Springer, Heidelberg, 180–210. https://doi.org/10.1007/978-3-030-36033-7_7
[40] Shuichi Katsumata, Kris Kwiatkowski, Federico Pintore, and Thomas Prest. 2020. Scalable Ciphertext Compression Techniques for Post-quantum KEMs and Their Applications. In *ASIACRYPT 2020, Part I (LNCS, Vol. 12491)*, Shiho Moriai and Huaxiong Wang (Eds.). Springer, Heidelberg, 289–320. https://doi.org/10.1007/978-3-030-64837-4_10
[41] Karen Klein, Guillermo Pascual-Perez, Michael Walter, Chethan Kamath, Margarita Capretto, Miguel Cueto, Ilia Markov, Michelle Yeo, Joël Alwen, and Krzysztof Pietrzak. 2021. Keep the Dirt: Tainted TreeKEM, Adaptively and Actively Secure Continuous Group Key Agreement. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 268–284. https://doi.org/10.1109/SP40001.2021.00035
[42] Andy Kroll. 2021. FBI Document Says the Feds Can Get Your WhatsApp Data — in Real Time. Rolling Stone. https://www.rollingstone.com/politics/politics-features/whatsapp-imessage-facebook-apple-fbi-privacy-1261816/.
[43] Kaoru Kurosawa. 2002. Multi-recipient Public-Key Encryption with Shortened Ciphertext. In *PKC 2002 (LNCS, Vol. 2274)*, David Naccache and Pascal Paillier (Eds.). Springer, Heidelberg, 48–63. https://doi.org/10.1007/3-540-45664-3_4
[44] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. 2020. *CRYSTALS-DILITHIUM*. Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.
[45] Moxie Marlinspike and Trevor Perrin. 2016. The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/ https://signal.org/docs/specifications/doubleratchet/.
[46] Ian Martiny, Gabriel Kaptchuk, Adam Aviv, Dan Roche, and Eric Wustrow. 2021. Improving Signal's sealed sender. (2021). To appear at NDSS 2021.
[47] Susan E. McGregor, Polina Charters, Tobin Holliday, and Franziska Roesner. 2015. Investigating the Computer Security Practices and Needs of Journalists. In *USENIX Security 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 399–414.
[48] Susan E. McGregor, Franziska Roesner, and Kelly Caine. 2016. Individual versus Organizational Computer Security and Privacy Concerns in Journalism. *PoPETs* 2016, 4 (Oct. 2016), 418–435. https://doi.org/10.1515/popets-2016-0048
[49] Vaishnavi Krishna Mohan. 2021. WhatsApp's New Privacy Policy: Collecting Metadata and Its Implications. https://www.globalviews360.com/articles/whatsapps-new-privacy-policy-collecting-metadata-and-its-implications.
[50] Ben Morris, Phillip Rogaway, and Till Stegers. 2018. Deterministic Encryption with the Thorp Shuffle. *Journal of Cryptology* 31, 2 (April 2018), 521–536. https://doi.org/10.1007/s00145-017-9262-z
[51] Kurt Opsahl. 2013. Why Metadata Matters. https://www.eff.org/deeplinks/2013/06/why-metadata-matters.
[52] Trevor Perrin. [n.d.]. The Noise Protocol Framework. The Noise Protocol Framework. http://www.noiseprotocol.org/noise.pdf.
[53] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. 2020. *FALCON*. Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.
[54] Charlie Savage. 2013. Court Rejects Appeal Bid by Writer in Leak Case. The New York Times. http://www.nytimes.com/2013/10/16/us/court-rejects-appealbid-by-writer-in-leak-case.html.
[55] Victor Shoup. 1997. Lower Bounds for Discrete Logarithms and Related Problems. In *EUROCRYPT'97 (LNCS, Vol. 1233)*, Walter Fumy (Ed.). Springer, Heidelberg, 256–266. https://doi.org/10.1007/3-540-69053-0_18
[56] Nigel P. Smart. 2005. Efficient Key Encapsulation to Multiple Parties. In *SCN 04 (LNCS, Vol. 3352)*, Carlo Blundo and Stelvio Cimato (Eds.). Springer, Heidelberg, 208–219. https://doi.org/10.1007/978-3-540-30598-9_15
[57] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. 2021. Key Agreement for Decentralized Secure Group Messaging with Strong Security Guarantees. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 2024–2045. https://doi.org/10.1145/3460120.3484542

A DEFINITION OF GENERAL CRYPTOGRAPHIC PRIMITIVES

A.1 Notation

We prepare some notations and keywords used to define the security and construct CGKAs. They are taken almost verbatim from [36, Sec. A.1].

We denote the set of natural numbers (non-negative integers) by $\mathbb{N}$ and the security parameter by $\kappa \in \mathbb{N}$. For an algorithm A, we write $A(\cdot; r)$ to denote that A is run with the explicit randomness r. For $k, n \in \mathbb{N}$ such that $k \leq m$, we write $[k : n]$ to denote the set $\{k, \ldots, n\}$. We use the shorthand $[n]$ when $k = 1$. We use $v \leftarrow x$ and $v := x$ to denote assigning the value x to the variable v, and use $v \leftarrow\!\!\$\, S$ to denote sampling an element v uniformly and randomly from a set S. We denote by $[cond]$ the bit that is 1 if the boolean statement $cond$ is true, and 0 otherwise.

Data structure. If V is a set, we write $V \xleftarrow{+} x$ and $V \xleftarrow{-} x$ as shorthands for $V \leftarrow V \cup \{x\}$ and $V \leftarrow V \setminus \{x\}$, respectively. For another set W, we write $V \xleftarrow{+} W$ and $V \xleftarrow{-} W$ as shorthands for $V \leftarrow V \cup W$ and $V \leftarrow V \setminus W$, respectively. For lists (vectors) $x := (x_1, \ldots, x_n)$ and $y := (y_1, \ldots, y_m)$, we denote the concatenation by $x \| y = (x_1, \ldots, x_n, y_1, \ldots, y_m)$ and use $x \xleftarrow{\shortparallel} v$ as a shorthand for $x \leftarrow x \| (v)$. We further use associative arrays and use $A[i] \leftarrow x$ and $y \leftarrow A[i]$ to assignment and retrieval of element i, respectively. We denote by $A[*] \leftarrow v$ the Initialization of the array to the default value v. For simplicity, we use the wildcard notation when dealing with sets of tuples and multi-argument associative arrays. For example, for an array with domain $\mathcal{I} \times \mathcal{J}$, we write $A[*, j] := \{A[i, j] \mid i \in \mathcal{I}\}$ and for a set $S \subseteq \mathcal{I} \times \mathcal{J}$, we write $(i, *) \in S$ as a shorthand for the condition $\exists j \in \mathcal{J} : (i, j) \in S$.

Keywords. We use the following keywords:

- **req** *cond* denotes that if the condition *cond* is false, then the current function unwinds all state changes and immediately returns $\perp$.
- **parse** $(m_1, \ldots, m_n) \leftarrow m$ denotes an attempt to parse a message m as a tuple. If m is not of the correct format, the current function unwinds all state changes and immediately returns $\perp$.
- **try** $y \leftarrow \mathsf{func}(x)$ is a shorthand notation for calling a helper function or subroutine func and executing **req** $y \neq \perp$.
- **assert** *cond* is only used to describe functionalities. It denotes that if *cond* is false, then the functionality permanently halts, making the real and ideal worlds trivially distinguishable (this is used to validate inputs of the simulator).

A.2 Decomposable Multi-Recipient Public Key Encryption

A multi-recipient PKE (mPKE) [13, 43, 56] is a type of PKE that allows a party to send the *same* message to a set of recipients of size N, more efficiently than executing N parallel runs of a standard PKE. In this work, we use a *decomposable* mPKE introduced in [40] that mandates a mPKE ciphertext to be decomposable into a recipient-dependent and independent part. [40] showed that many assumptions known to imply PKE (e.g., DDH, LWE, SIDH) can naturally be used to construct an IND-CCA decomposable mPKE in the random oracle model.

Definition A.1 (Decomposable Multi-Recipient Public-Key Encryption). A (single-message) decomposable multi-recipient public-key encryption (mPKE) over a message space $\mathcal{M}$ consists of the following algorithms:

- $\mathsf{mSetup}(1^\kappa) \to \mathsf{pp}$: On input the security parameter 1^κ, it outputs a public parameter pp.
- $\mathsf{mGen}(\mathsf{pp}) \to (\mathsf{ek}, \mathsf{dk})$: On input a public parameter pp, it outputs a pair of encryption key and decryption key $(\mathsf{ek}, \mathsf{dk})$.
- $\mathsf{mEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \mathsf{M}; \mathsf{r}_0, (\mathsf{r}_i)_{i\in[N]}) \to (\mathsf{ct}_0, \vec{\mathsf{ct}} = (\widehat{\mathsf{ct}}_i)_{i\in[N]})$: The (decomposable) encryption algorithm splits into a pair of algorithms $(\mathsf{mEnc}^{\mathsf{i}}, \mathsf{mEnc}^{\mathsf{d}})$:
  - $\mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{r}_0) \to \mathsf{ct}_0$: On input a public parameter pp, it outputs an (encryption key independent) ciphertext ct_0.
  - $\mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \mathsf{M}; \mathsf{r}_0, \mathsf{r}_i) \to \widehat{\mathsf{ct}}_i$: On input a public parameter pp, an encryption key ek_i, a message $\mathsf{M} \in \mathcal{M}$, it outputs an (encryption key dependent) ciphertext $\widehat{\mathsf{ct}}_i$.
- $\mathsf{mDec}(\mathsf{dk}, \mathsf{ct}_0, \widehat{\mathsf{ct}}_i) \to \mathsf{M}$ or $\perp$: On input a decryption key dk and a ciphertext $(\mathsf{ct}_0, \widehat{\mathsf{ct}}_i)$, it outputs either $\mathsf{M} \in \mathcal{M}$ or $\perp \notin \mathcal{M}$.

Definition A.2 (Correctness). A mPKE is correct if we have

$$\mathbb{E}\left[\max_{\mathsf{M}\in\mathcal{M}} \Pr\left[\mathsf{M} = \mathsf{mDec}(\mathsf{dk}, (\mathsf{ct}_0, \widehat{\mathsf{ct}})) : \begin{array}{c} \mathsf{ct}_0 \leftarrow \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}), \\ \widehat{\mathsf{ct}} \leftarrow \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}, \mathsf{M}) \end{array}\right]\right] \geq 1 - \mathsf{negl}(\kappa),$$

where the expectation is taken over $\mathsf{pp} \leftarrow \mathsf{mSetup}(1^\kappa)$ and $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{mGen}(\mathsf{pp})$.

Ciphertext-spreadness defines how much entropy a properly generated ciphertext has. For the application of CGKA, we require the party-independent ciphertext to maintain enough min-entropy. Note that [36] defined ciphertext-spreadness with the entire ciphertext but the proof relies on the high min-entropy of the party independent part.

Definition A.3 (Ciphertext-Spreadness of Party Independent Ciphertext). A mPKE has ciphertext-spreadness (with respect to the party independent ciphertext of), if for all $\mathsf{pp} \in \mathsf{mSetup}(1^\kappa)$, we have

$$\mathbb{E}[\max_{\mathsf{ct}_0} \Pr[\mathsf{ct}_0 = \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp})] \leq \mathsf{negl}(\kappa),$$

where the expectation is taken over $\mathsf{pp} \leftarrow \mathsf{mSetup}(1^\kappa)$ and the encryption randomness.

*Definition A.4 (*IND-CCA*).* The security notion is defined by a game illustrated in Fig. 8, where we say the adversary $\mathcal{A}$ *wins* if the game outputs 1. A mPKE is IND-CCA *secure* if for all PPT adversaries $\mathcal{A}$, we have $|\Pr[\mathcal{A}\ \textit{wins}] - 1/2| \leq \mathsf{negl}(\kappa)$.

GAME IND-CCA

$\mathsf{pp} \leftarrow \mathsf{mSetup}(1^\kappa)$
foreach $i \in [N]$ **do**
  $(\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{mGen}(\mathsf{pp})$
$(\mathsf{M}_0, \mathsf{M}_1) \leftarrow \mathcal{A}^{\mathcal{D}(\cdot)}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]})$
$b \leftarrow\$ \{0, 1\}$
$(\mathsf{ct}_0^*, \vec{\mathsf{ct}}^* := (\widehat{\mathsf{ct}}_i^*)_{i\in[N]}) \leftarrow \mathsf{mEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \mathsf{M}_b)$
$b' \leftarrow \mathcal{A}^{\mathcal{D}(\cdot)}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \mathsf{ct}_0^*, \vec{\mathsf{ct}}^*)$
return $[b = b']$

Decapsulation Oracle $\mathcal{D}(i, \mathsf{ct}_0, \mathsf{ct})$

req $(\mathsf{ct}_0, \widehat{\mathsf{ct}}) \neq (\mathsf{ct}_0^*, \widehat{\mathsf{ct}}_i^*)$
$\mathsf{M} \leftarrow \mathsf{mDec}(\mathsf{dk}_i, \mathsf{ct}_0, \widehat{\mathsf{ct}})$
return M

Figure 8: IND-CCA **security of** mPKE**.**

A.3 Secret Key Encryption

We define secret key encryption SKE.

Definition A.5 (Secret-Key Encryption). Secret-key encryption (SKE) over a key space $\mathcal{K}$ (implicitly parameterized by the security parameter) and message space $\mathcal{M}$ consists of the following two algorithms:

- $\mathsf{Enc}_\mathsf{s}(\mathsf{k}, \mathsf{M}) \to \mathsf{ct}$: On input a secret key $\mathsf{k} \in \mathcal{K}$ and a message $\mathsf{M} \in \mathcal{M}$, it outputs a ciphertext ct.
- $\mathsf{Dec}_\mathsf{s}(\mathsf{k}, \mathsf{ct}) \to \mathsf{M}$ or $\perp$: On input a secret key k and a ciphertext ct, it deterministically outputs either $\mathsf{M} \in \mathcal{M}$ or $\perp \notin \mathcal{M}$.

Definition A.6 (Correctness). A SKE is correct if $\Pr[\mathsf{Dec}_\mathsf{s}(\mathsf{k}, \mathsf{Enc}_\mathsf{s}(\mathsf{k}, \mathsf{M})) = \mathsf{M}] \geq 1 - \mathsf{negl}(\kappa)$ holds for all $\kappa \in \mathbb{N}$, $\mathsf{M} \in \mathcal{M}$ and $\mathsf{k} \in \mathcal{K}$.

We define IND-CPA and IND-CCA security for SKE by the left-or-right version of game-based indistinguishability [15].

*Definition A.7 (*IND-CPA *and* IND-CCA *for* SKE*).* The security notion is defined by a game illustrated in Fig. 9, where we say the adversary $\mathcal{A}$ *wins* if the game outputs 1. A SKE is IND-CPA *secure* (resp. IND-CCA *secure*) if for all PPT adversaries $\mathcal{A}$, we have $|\Pr[\mathcal{A}\ \textit{wins}$ in IND-CPA game (resp. IND-CCA secure)$] - 1/2| \leq \mathsf{negl}(\kappa)$.

We finally define the *key-committing* property [33] which roughly states that it is difficult to find two secret keys that correctly decrypt the same ciphertext (to possibly different messages). As in prior works [4, 32, 33, 35, 36], we define this notion by providing the (non-uniform) adversary oracle access to Enc_s and Dec_s, where we implicitly assume these two algorithms are implemented using an internal hash function modeled as a random oracle.

Definition A.8 (Key-Committing). A SKE has *key-committing* property if for all PPT adversary $\mathcal{A}$, we have

$$\Pr\left[\begin{array}{l} \mathsf{Dec}_\mathsf{s}(\mathsf{k}_0, \mathsf{ct}) \neq \perp \\ \wedge \mathsf{Dec}_\mathsf{s}(\mathsf{k}_1, \mathsf{ct}) \neq \perp \end{array} : (\mathsf{k}_0, \mathsf{k}_1, \mathsf{ct}) \leftarrow \mathcal{A}^{\mathsf{Enc}_\mathsf{s}, \mathsf{Dec}_\mathsf{s}}(1^\kappa) \right] \leq \mathsf{negl}(\kappa).$$

GAME IND-CPA

$\mathsf{k} \leftarrow\!\$\ \mathcal{K}$
$b \leftarrow\!\$\ \{0,1\}$
$b' \leftarrow \mathcal{A}^{\mathcal{LR}(\cdot,\cdot)}(1^\kappa)$
return $[b = b']$

Left-or-Right Oracle $\mathcal{LR}(\mathsf{M}_0, \mathsf{M}_1)$

$\mathsf{ct} \leftarrow \mathsf{Enc}_s(\mathsf{k}, \mathsf{M}_b)$
$S \xleftarrow{+} \mathsf{ct}$
return ct

GAME IND-CCA

$\mathsf{k} \leftarrow\!\$\ \mathcal{K}$
$b \leftarrow\!\$\ \{0,1\}$
$S \leftarrow \emptyset$
$b' \leftarrow \mathcal{A}^{\mathcal{LR}(\cdot,\cdot),\mathcal{D}(\cdot)}(1^\kappa)$
return $[b = b']$

Decryption Oracle $\mathcal{D}(\mathsf{ct})$

req $\mathsf{ct} \notin S$
$\mathsf{M} \leftarrow \mathsf{Dec}_s(\mathsf{k}, \mathsf{ct})$
return M

Figure 9: IND-CPA **and** IND-CCA **security of** SKE**. If the condition following req does not hold, the game terminates by returning a random bit.**

A.4 Digital Signatures

We provide the standard notion of digital signatures.

Definition A.9 (Signature Scheme). A signature scheme SIG over a message space $\mathcal{M}$ consists of the following algorithms:

- $\mathsf{Setup}(1^\kappa) \to \mathsf{pp}$: On input the security parameter 1^κ, it outputs a public parameter pp.
- $\mathsf{KeyGen}(\mathsf{pp}) \to (\mathsf{svk}, \mathsf{ssk})$: On input a public parameter pp it outputs a pair of verification key and signing key (svk, ssk).
- $\mathsf{Sign}(\mathsf{pp}, \mathsf{ssk}, \mathsf{m}) \to \mathsf{sig}$: On input a public parameter pp, a signing key ssk and a message m, it outputs a signature sig.
- $\mathsf{Verify}(\mathsf{pp}, \mathsf{svk}, \mathsf{m}, \mathsf{sig}) \to \top/\bot$: On input a public parameter pp, a verification key ssk, a message m and a signature sig, it outputs $\top$ or $\bot$.

Definition A.10 (Correctness). A signature scheme SIG is correct if for all $\kappa \in \mathbb{N}$, all messages $\mathsf{m} \in \mathcal{M}$ and all $\mathsf{pp} \in \mathsf{Setup}(1^\kappa)$,

$$\Pr\left[\mathsf{Verify}(\mathsf{pp}, \mathsf{svk}, \mathsf{m}, \mathsf{sig}) = \top : \begin{array}{l}(\mathsf{svk}, \mathsf{ssk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp});\\ \mathsf{sig} \leftarrow \mathsf{Sign}(\mathsf{pp}, \mathsf{ssk}, \mathsf{m})\end{array}\right] \geq 1 - \mathsf{negl}(\kappa).$$

*Definition A.11 (*sEUF-CMA*).* A signature scheme is strongly EUF-CMA (sEUF-CMA) secure if for all PPT adversary $\mathcal{A}$, we have

$$\Pr\left[\begin{array}{l}\mathsf{Verify}(\mathsf{pp}, \mathsf{svk}, \mathsf{m}^*, \mathsf{sig}^*) = \top\\ \wedge (\mathsf{m}^*, \mathsf{sig}^*) \notin L\end{array} : \begin{array}{r}\mathsf{pp} \leftarrow \mathsf{Setup}(1^\kappa);\\ (\mathsf{svk}, \mathsf{ssk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp});\\ (\mathsf{m}^*, \mathsf{sig}^*) \leftarrow \mathcal{A}^{\mathcal{S}(\cdot)}(\mathsf{pp}, \mathsf{svk})\end{array}\right] \leq \mathsf{negl}(\kappa),$$

where $\mathcal{S}(\cdot)$ is defined as $\mathsf{Sign}(\mathsf{ssk}, \cdot)$, and L is the set of pairs of messages and signatures generated by the signing oracle. A signature scheme is EUF-CMA secure if we only restrict $(\mathsf{m}^*, *) \notin L$ above.

A.5 Message Authentication Codes

We define message authentication codes (MAC).

*Definition A.12 (*MAC*).* A (deterministic) message authentication code MAC over a key space $\mathcal{K}$ and a message space $\mathcal{M}$ consists of the following algorithms:

- $\mathsf{TagGen}(\mathsf{k}, \mathsf{m}) \to \mathsf{tag}$: On input a key $\mathsf{k} \in \mathcal{K}$ and a message $\mathsf{m} \in \mathcal{M}$, it (deterministically) outputs a tag tag.
- $\mathsf{TagVerify}(\mathsf{k}, \mathsf{m}, \mathsf{tag}) \to \bot/\top$: On input a key k, a message m and a tag tag, it (deterministically) outputs $\top$ or $\bot$.

Definition A.13 (Correctness). A MAC is correct if for all keys $\mathsf{k} \in \mathcal{K}$ and all messages $\mathsf{m} \in \mathcal{M}$,

$$\Pr\left[\mathsf{TagVerify}(\mathsf{k}, \mathsf{m}, \mathsf{TagGen}(\mathsf{k}, \mathsf{m})) = \top\right] = 1.$$

Following previous works [12, 36], we use a MAC based on a hash function modeled as a random oracle. For instance, we can use HMAC, also used by MLS. Since the security of the MAC is implicitly invoked during in the *generalized selective decryption* (GSD) security game [6, 10, 12, 36], we do not explicitly define the notion of unforgeability.

A.6 HKDF

HKDF is a key derivation function (KDF) based on HMAC. It consists of the two algorithms HKDF.Extract and HKDF.Expand. The extraction algorithm $\mathsf{k} \leftarrow \mathsf{HKDF.Extract}(s_0, s_1)$ outputs a uniformly random key k if either s_0 or s_1 has high min-entropy. The expansion algorithm $\mathsf{k}_{\mathsf{lbl}} \leftarrow \mathsf{HKDF.Expand}(\mathsf{k}, \mathsf{lbl})$, on input a key k and a public label lbl, outputs a uniformly random key $\mathsf{k}_{\mathsf{lbl}}$ for lbl. Both HKDF.Extract and HKDF.Expand were modled as a random oracle to prove security of Chained CmPKE [36].

A.7 Pseudorandom Function

Let $\mathsf{F} : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ be a function family with key space $\mathcal{K}$, domain $\mathcal{D}$ and finite range $\mathcal{R}$. We define a pseudorandom function as follows.

Definition A.14 (Pseudorandom Function). We say F is a pseudorandom function (PRF) if for all PPT adversary $\mathcal{A}$, we have

$$\left|\Pr\left[\begin{array}{c} b = b' \end{array} : \begin{array}{c} b \leftarrow\!\$\ \{0,1\}; \mathsf{K} \leftarrow\!\$\ \mathcal{K}; \mathsf{RF} \leftarrow\!\$\ \mathcal{RF};\\ b' \leftarrow \mathcal{A}^{\mathcal{F}(\cdot)}(1^\kappa)\end{array}\right] - \frac{1}{2}\right| \leq \mathsf{negl}(\kappa),$$

where $\mathcal{RF}$ is a set of all functions with domain $\mathcal{D}$ and range $\mathcal{R}$, and $\mathcal{F}(\cdot)$ is defined as $\mathsf{F}(\mathsf{K}, \cdot)$ if $b = 0$, and $\mathsf{RF}(\cdot)$ otherwise.

A.8 Pseudorandom Permutation

Let $\phi : \mathcal{K} \times \mathcal{R} \to \mathcal{R}$ be a function family of one-to-one functions from $\mathcal{R}$ to $\mathcal{R}$ with key space $\mathcal{K}$. We define a pseudorandom permutation as follows.

Definition A.15 (Pseudorandom Permutation). We say ϕ is a pseudorandom permutation (PRP) if for all PPT adversary $\mathcal{A}$, we have

$$\left|\Pr\left[\begin{array}{c} b = b' \end{array} : \begin{array}{c} b \leftarrow\!\$\ \{0,1\}; \mathsf{K} \leftarrow\!\$\ \mathcal{K}; \mathsf{RP} \leftarrow\!\$\ \mathcal{RP};\\ b' \leftarrow \mathcal{A}^{\mathcal{P}(\cdot)}(1^\kappa)\end{array}\right] - \frac{1}{2}\right| \leq \mathsf{negl}(\kappa),$$

where $\mathcal{RP}$ is the set of all permutations over $\mathcal{R}$, and $\mathcal{P}(\cdot)$ is defined as $\phi(\mathsf{K}, \cdot)$ if $b = 0$, and $\mathsf{RP}(\cdot)$ otherwise.

B STATIC METADATA-HIDING CGKA: DEFINITION

In this section, we propose a UC security model capturing the security of the 1st & 2nd layers (i.e., group secret keys and *static* metadata) by defining a new ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. This is an extension of the ideal functionality $\mathcal{F}_{\text{CGKA}}$ [10, 12, 36] that captures the security of the 1st layer.

B.1 Background

B.1.1 Universal Composable Security. We briefly recall the UC framework. We refer to [20, 21] for the full descriptions. The UC security is formalized by the indistinguishability of real and ideal protocols. In the real protocol, parties execute a protocol Π, where an *adversary* $\mathcal{A}$ may corrupt some of the parties. In the ideal protocol, the parties are replaced by *dummy* parties that interact with an *ideal functionality* $\mathcal{F}$, where a *simulator* $\mathcal{S}$ may corrupt some of the dummy parties. The dummy parties are defined to be the identity function that simply outputs whatever is fed as input. In addition, there is another entity called the *environment* $\mathcal{Z}$ that tries to distinguish the two protocols. In the real (resp. ideal) protocol, $\mathcal{Z}$ can interact arbitrary with $\mathcal{A}$ (resp. $\mathcal{S}$), and it can also invoke any non-corrupted parties (resp. dummy parties) to honestly run the protocol Π (resp. the ideal functionality $\mathcal{F}$), where the output is always reported back to $\mathcal{Z}$. The goal of UC-security is then, given any adversary $\mathcal{A}$, to construct a simulator $\mathcal{S}$ such that any environment $\mathcal{Z}$ cannot distinguish between the real and ideal protocols. We say the real protocol Π *UC-realizes* the ideal functionality $\mathcal{F}$ if such $\mathcal{S}$ can be constructed. Put differently, whatever $\mathcal{A}$ can learn from the real protocol Π can be simulated using the information provided by the ideal functionality $\mathcal{F}$, which is secure by definition.

We often construct a protocol in a setting where a copy of an ideal functionality $\mathcal{G}$ is available. We call such a model $\mathcal{G}$-hybrid model. If a real protocol Π UC-realizes $\mathcal{F}$ while providing access to an ideal functionality $\mathcal{G}$, then we say Π UC-realizes $\mathcal{F}$ in the *$\mathcal{G}$-hybrid model*. For a real protocol Π in the $\mathcal{G}$-hybrid model, and a real protocol Π' that realizes $\mathcal{G}$ (in the standard model), we can naturally define a composed protocol $\Pi^{\Pi'}$ in the standard model, in which calls for $\mathcal{G}$ from Π are answered by Π' instead of $\mathcal{G}$. Canetti [20] proved a *universal composition theorem* stating that, if Π UC-realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model and Π' UC-realizes $\mathcal{G}$, then $\Pi^{\Pi'}$ UC realizes $\mathcal{F}$.

The Corruption Model. We define what we mean by "corrupting" a party. In this work, we use the corruption model of continuous state leakage (transient passive corruptions) and adversarially chosen randomness of [10]. This is a standard in CGKA literature, but this is non-standard for typically UC-security. This corruption model allows the adversary to repeatedly corrupt parties by sending them two types of corruption messages: (1) a message `Expose` causes the party to send its current state to the adversary (once), (2) a message $(\texttt{CorrRand}, b)$ sets the party's rand-corrupted flag to b. If b is set, the party's randomness-sampling algorithm is replaced by the adversary providing the coins instead. Ideal functionalities are activated upon corruptions and can adjust their behavior accordingly.

Restricted Environments and Adversaries. To avoid the so-called commitment problem, caused by adaptive corruptions in simulation-based frameworks, we restrict the environment (and thus the adversary) not to corrupt parties at certain times. This roughly corresponds to ruling out "trivial attacks" in game-based definitions, e.g., the adversary cannot compromise the secret key after being provided with the challenge ciphertext. In simulation-based frameworks, such attacks are no longer trivial, but security against them requires relatively strong and inefficient cryptographic tools, e.g., non-committing encryption, and is not achieved by most protocols. We follow prior works [10, 12, 36, 39] and consider a weakened variant of UC-security that only quantifies over a restricted set of so-called admissible environments that do not exhibit the commitment problem. Whether an environment is admissible or not is defined by the ideal functionality $\mathcal{F}$ with statements of the form **restrict** *cond* and an environment is called admissible (for $\mathcal{F}$), if it has negligible probability of violating any such *cond* when interacting with $\mathcal{F}$.

B.1.2 PKI functionality. As in [12, 36], we define our CGKA in a hybrid model where parties can access an ideal functionality that models an (untrusted) PKI. In the real protocol, the parties can interact with the Authentication Service (AS) and Key Service (KS) PKI functionalities. For instance, the environment can instruct the AS (via the party's protocol) to register a new key for a party. As a result, the AS generates a new key pair for the party and hands the public key to the environment, making the secret key available to the party's protocol upon request. We note that the adversary can register arbitrary signature keys for any party to capture an insider adversary.

Authentication Service (AS). The authentication service (AS) certifies the ownership of a signature key. The AS is formalized by the functionality $\mathcal{F}_{\text{AS}}$ defined in Fig. 10. The definition is identical to that used in [37]. $\mathcal{F}_{\text{AS}}$ allows parties to register fresh signature key pairs via `register-svk` query and to check whether a verification key svk is registered by a party id′ via the `verify-cert` query. On registration, the new key pair for a party id is generated by $\mathcal{F}_{\text{AS}}$ using a genSSK algorithm (whose concrete specification depends on the CGKA). If id's current randomness source is corrupted (i.e., RandCorr[id] = 'bad'), $\mathcal{F}_{\text{AS}}$ asks the adversary to provide the randomness. After registration, id receives the new verification key svk. Also, parties can retrieve their signing keys via `get-ssk` query and delete registered signing keys via `del-ssk` query. The adversary can register arbitrary verification keys in the name of any party. When a party is corrupted, all signing keys except for the deleted ones are leaked to the adversary. Security is modeled by the ideal-world variant of $\mathcal{F}_{\text{AS}}$, called $\mathcal{F}_{\text{AS}}^{\text{IW}}$. It marks leaked signing keys by storing them in the ExposedSvk array (see boxes in Fig. 10). $\mathcal{F}_{\text{AS}}$ allows the Key Service functionality $\mathcal{F}_{\text{KS}}$ (see below) to signal that a certain ssk is leaked. $\mathcal{F}_{\text{KS}}$ sends this signal when the signature key is leaked due to a compromise of a key package. Finally, $\mathcal{F}_{\text{AS}}^{\text{IW}}$ always leaks all registered signing keys to the simulator.

Key Service (KS). The Key Service (KS) allows parties to upload one-time key packages used to add them to groups while they are offline. The KS is formalized by the functionality $\mathcal{F}_{\text{KS}}$ defined in Fig. 11. The functionality is identical to that used in [37] except that KS checks the validity of maliciously registered key packages

and parties can check whether the key package is registered to KS. In our definition, when the adversary registers a key package to $\mathcal{F}_{\mathrm{KS}}$, $\mathcal{F}_{\mathrm{KS}}$ checks whether the registering key packages are valid by the `*validate-kp` function. Thus, $\mathcal{F}_{\mathrm{KS}}$ ensures the registered key packages are valid in the sense that the `*validate-kp` function returns `true`. In addition, parties also check the key package is registered to $\mathcal{F}_{\mathrm{KS}}$ via a `has-kp` query. This allows parties to check the validity of a key package via the `has-kp` query since $\mathcal{F}_{\mathrm{KS}}$ ensures the validity of registered key packages. We introduce these functions to make the syntax of an add and update proposal to look more similar. When a party id adds a party id_t, it first fetches an id_t's key package from KS and invokes a CGKA protocol (or $\mathcal{F}_{\mathrm{CGKA}}$) on input (`Propose`, 'add'-$\mathsf{id}_t$-$\mathsf{kp}_t$). The party (and $\mathcal{F}_{\mathrm{CGKA}}$) can check the validity of kp_t through the `has-kp` query to $\mathcal{F}_{\mathrm{KS}}$. This is syntactically similar to an update proposal where, the updated signing key svk is validated via the `verify-cert` query.

Other functionalities are identical to that used in [37]. Similar to $\mathcal{F}_{\mathrm{AS}}$, parties can register key packages via the `register-kp` query. Upon receiving the `register-kp` query, $\mathcal{F}_{\mathrm{AS}}$ generates a new key package using a $\mathsf{genKP}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk})$ algorithm (whose concrete specification depends on the CGKA), which takes a party's identity id and a signature key pair $(\mathsf{svk}, \mathsf{ssk})$ and outputs a key package and the corresponding decryption key. Parties can request another party's key package via `get-kp` query. The returned key package is specified by the adversary. This reflects that the adversary can maliciously inject key packages that were not registered by honest parties. Finally, the ideal-world KS functionality $\mathcal{F}_{\mathrm{KS}}^{\mathrm{IW}}$ always leaks all decryption keys to the simulator.

B.1.3 History Graph. We use a so-called *history graph* [9, 10, 12, 36] to define the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$.

Overview. A history graph is a labeled directed graph that acts as a symbolic representation of a group's evolution. It has two types of nodes: commit and proposal nodes, representing all executed commit and propose operations, respectively. Each party is uniquely assigned to a commit node indicating that a party is in a group of members that processed the commit assigned to that specific commit node. The nodes' labels, furthermore, keep track of all the additional information relevant for defining security. For instance, proposal nodes have a label that stores the proposed action, and commit nodes to have labels that store the epoch's application secret and the set of parties corrupted in the given epoch. Security of the application secrets is then formalized by the functionality of choosing a random and independent key for each commit node whenever security is guaranteed; otherwise, the simulator gets to choose the key. Whether security is guaranteed in a given node, is determined via an explicit safe predicate on the node and the history graph. In addition to the secrecy of the keys, the functionality also formalizes authenticity by appropriately disallowing injections.

Formal Defintion. As explained in Sec. 3.2, we deviate from the definition of prior history graphs used to define CGKA in the UC framework. Each node in the history graph is identified by node pointers: $\mathsf{prop\text{-}id} \in \mathbb{N}$ for proposal nodes and $\mathsf{node\text{-}id} \in \{0\} \cup \mathbb{N}$ for commit nodes. In contrast, prior works [9, 11] used concrete (non-encrypted) proposals and commits to identify each node. This formalization was well-defined in prior works since each proposal and commit identified a unique group operation in the real protocol. However, when considering *static metadata-hiding*, two distinct (encrypted) proposals or commits may encrypt the same group operation, in which case, we would like to assign these distinct proposals and commits to the same node. Otherwise, two parties can be in the same group in the real protocol, while they are included in a different commit node in the history graph. Roughly, pointers prop-id and node-id define the *semantics* of a group operation and allow us to assign semantically equivalent proposals and commits to the same node. The pointer values of a proposal or a commit will be arbitrarily assigned by the simulator, and the ideal functionality checks whether the created history graph maintains consistency, authenticity, and confidentiality.

All nodes in the history graph store the following values:

- orig: the identity of the party who created the node, i.e., the message sender.
- par: the parent commit node, representing the sender's current epoch.
- $\mathsf{stat} \in \{$ 'good', 'bad', 'adv' $\}$: the flag indicating whether the secrets corresponding to the node are known to the adversary. 'good' means this node is secure, 'bad' means this node is created with adversarial randomness (hence it is well-formed but the adversary knows the secret), and 'adv' means this node is created by the injected message from the adversary.

Proposal nodes further store the following values:

- $\mathsf{act} \in \{$ 'upd'-kp, 'add'-id_t-kp_t, 'rem'-id_t $\}$: the proposal action. 'upd'-kp means the corresponding party updates its key package to kp. 'add'-id_t-kp_t means id_t is added with the key package kp_t[23].

Commit nodes further store the following values. In this work, history graphs keep gid, epoch, and new variable conthide in addition to the values used in the previous work [12, 37]:

- gid: the group identifier.
- epoch: the current epoch number.
- prop: the ordered list of committed proposals.
- mem: the list of a pair of group member's identity and its key package, which is sorted by dictionary order in identities.
- vcom: the list of party-dependent commitments associated with this node.
- key: the group (application) secret.
- exp: the set keeping track of corrupted parties in this node.
- chall: the flag indicating whether the group secret is challenged. That is, chall = `true` if a random group key was generated for this node, and `false` if the key was set by the adversary (or not generated).
- conthide: the flag indicating whether the static metadata protection is assured at this epoch. That is, conthide = `true` means the messages issued with this epoch's group secret hide metadata, and `false` means the metadata is leaked. This value is initialized when one of proposal/commit/welcome messages is first created at this epoch.

For convenience, we define the following helper function.

- $\mathsf{indexOf}(\mathsf{id})$: returns the index of id in the list mem.

[23] The previous models only kept signature keys in kp. To capture metadata-hiding property, we need to manage which key packages are being added/updated.

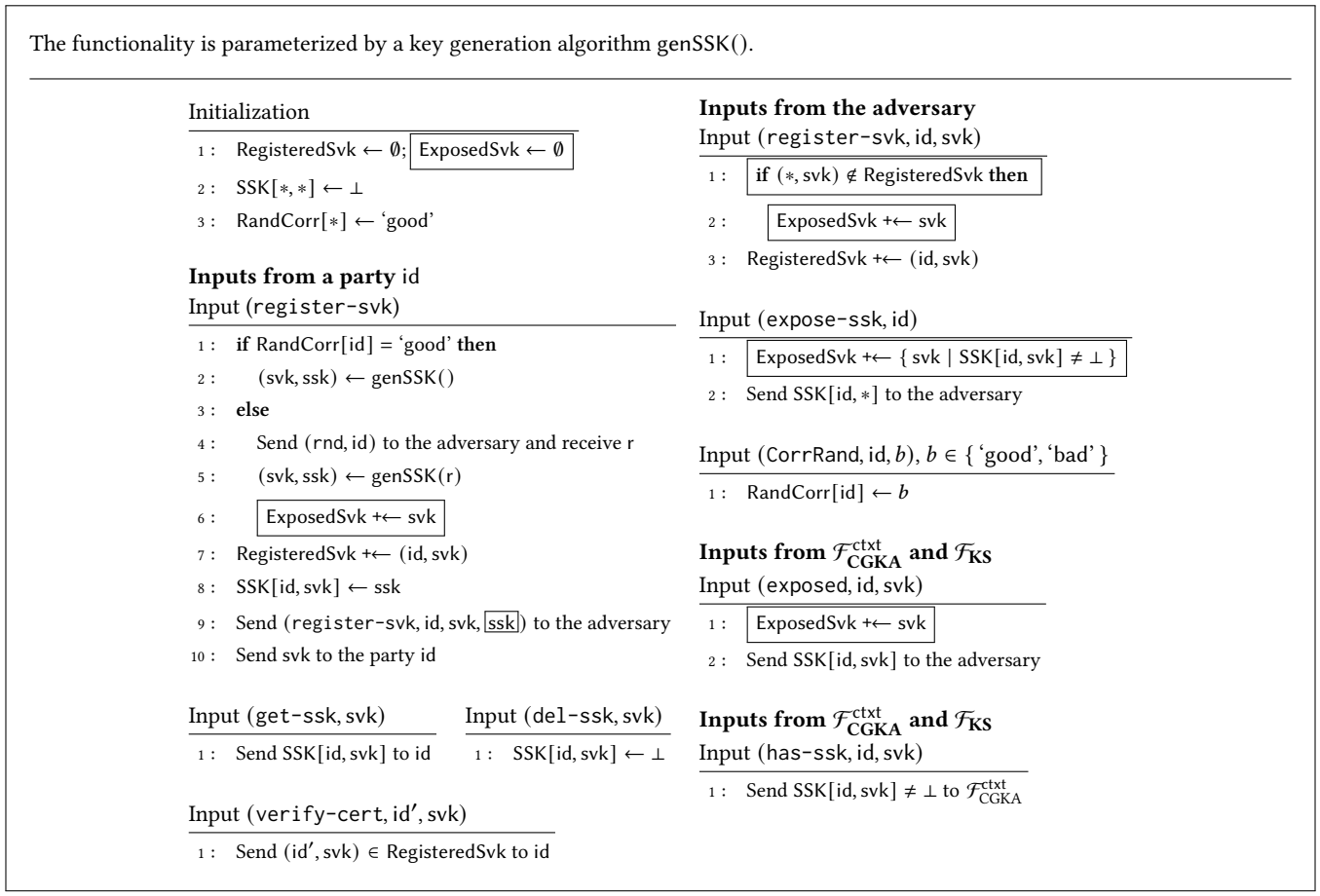


Figure 10: The ideal authentication service functionality $\mathcal{F}_{\mathbf{AS}}$ and its variant $\boxed{\mathcal{F}_{\mathbf{AS}}^{\mathbf{IW}}}$ used during the security proof.

B.2 UC Security Model and $\mathcal{F}_{\mathbf{CGKA}}^{\mathsf{ctxt}}$

We propose a new ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ capturing the static metadata-hiding property of CGKAs. $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ is based on the prior ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$ [36, 37] that only captured the security of group secret keys. The ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ is formally defined in Figs. 12 to 14, along with several helper functions in Figs. 16 to 20 to aid the readability. By setting the flag $\mathtt{flag}_{\mathsf{contHide}}$ to false and $\mathtt{flag}_{\mathsf{selDL}}$ to false (resp. true) in the "Initialization," $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ becomes identical to $\mathcal{F}_{\mathrm{CGKA}}$ used in [37] (resp. [36] capturing *selective downloading*).

To specify the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$, we also need to define the following:

***Safety Predicates*: safe**, **sig-inj-allowed**, and **mac-inj-allowed** specify which epoch secrets are secure and when authenticity is guaranteed,

***Leakage Functions*: *leak-create**, ***leak-prop**, ***leak-com**, ***leak-wel**, and ***leak-proc** specify information leaked from protocol messages.

The safety predicates and leakage functions are protocol specific. For instance, some CGKA may leak the type of proposal, while others may not. Put differently, a specific CGKA UC-realizes the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ *with respect to* a particular choice of safety predicates and leakage functions. By tuning the choice, $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ allows modeling a wide variety of CGKAs. A concrete choice of such safety predicates and leakage functions is provided in App. C.2, where we prove UC-security of our CGKA Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$.

Below, we provide an overview of the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

States. Different from the previous functionality in [36], $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ maintains parties' key packages kp instead of signature keys only. Also, it keeps track of party-specific secrets (e.g., CmPKE decryption key) by managing the PendDK array (which stores pending secrets of the key package kp), and the CurrDK array (which stores the current secrets of id). In other words, this modification means that $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ manages key packages as if the key service does. Moreover, $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ switches its functionality according to the flag $\mathtt{flag}_{\mathsf{selDL}}$ and $\mathtt{flag}_{\mathsf{contHide}}$: $\mathtt{flag}_{\mathsf{selDL}}$ is set to `true` if it performs selective downloading, and $\mathtt{flag}_{\mathsf{contHide}}$ is set to `true` if it offers static metadata-hiding. The other states are identical to $\mathcal{F}_{\mathrm{CGKA}}$

The functionality is parameterized by a key-package generation algorithm genKP(id, svk, ssk) and a key package validation function *validate-kp(kp, id).

Initialization

RegisteredKp ← ∅
DK[∗, ∗] ← ⊥; SVK[∗, ∗] ← ⊥
RandCorr[∗] ← 'good'

Inputs from a party id

Input (register-kp, svk, ssk)

if RandCorr[id] = 'good' **then**
  (kp, dk) ← genKP(id, svk, ssk)
  if kp = ⊥ **then return**
else
  Send (rnd, id) to the adversary and receive r
  (kp, dk) ← genKP(id, svk, ssk; r)
  if kp = ⊥ **then return**
  Send (exposed, id, svk) to $\mathcal{F}_{\text{AS}}$
RegisteredKp +← (id, kp)
DK[id, kp] ← dk; SVK[id, kp] ← svk
Send (register-kp, id, svk, kp, $\boxed{\text{dk}}$) to the adversary
Send kp to the party id

Input (get-dks)

Send { (kp, DK[id, kp]) | DK[id, kp] ≠ ⊥ } to id

Input (get-kp, id′)

Send (get-kp, id, id′) to the adversary and receive kp′
try *validate-kp(kp, id)
RegisteredKp +← (id′, kp′)
Send kp′ to id

Input (del-kp, kp)

DK[id, kp], SVK[id, kp] ← ⊥

Inputs from id **and** $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$

Input (has-kp, id, kp)

Send (id, kp) ∈ RegisteredKp

Inputs from the adversary

Input (CorrRand, id, b), $b \in$ { 'good', 'bad' }

RandCorr[id] ← b

Inputs from the adversary and $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$

Input (exposed, id)

Send DK[id, ∗] to the adversary
foreach svk ∈ SVK[id, ∗] **s.t.** svk ≠ ⊥ **do**
  Send (exposed, id, svk) to $\mathcal{F}_{\text{AS}}$

Figure 11: The ideal key service functionality $\mathcal{F}_{\text{KS}}$ and and its variant $\boxed{\mathcal{F}_{\text{KS}}^{\text{IW}}}$ used during the security proof.

in [36] with some syntactical change. $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ maintains the history graph. As explained in App. B.1.3, it identifies proposal nodes by a pointer prop-id $\in \{0\} \cup \mathbb{N}$ and commit nodes by a pointer node-id $\in \{0\} \cup \mathbb{N}$. We assume one group is created by an honest party (see Create in Fig. 12). This creates a root (commit) node called the *main root* identified by the pointer node-id = 0. We call the group starting from the main root *main group*. Moreover, other roots may be created without a commit message (e.g., when a party processes an injected welcome message that is not directly related to the main group). Such roots are called *detached root*. $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ also stores a pointer Ptr[id] for each party id. Ptr[id] identifies id's current commit node (i.e., current epoch). If id is not in the group, Ptr[id] = ⊥.

Interface. $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ offers interfaces to create a group, create a proposal, commit to a list of proposals, process a commit, join a group, and retrieve the group secret key. All interfaces except create and join are for group members only (i.e., parties for which Ptr[id] ≠ ⊥). We explain each interface in more detail below.

Group creation (See Fig. 12) $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ allows one main group to be created by a designated party $\text{id}_{\text{creator}}$. Initially, the main group has a single party $\text{id}_{\text{creator}}$, and it can invite additional members by issuing add proposals and committing to them. $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ checks the validity of $\text{id}_{\text{creator}}$'s signature key by *valid-svk. Then, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ informs the adversary $\mathcal{S}$[24] of the creation of a new group by sending the message (Create) to $\mathcal{S}$. (This models the fact that a server knows when a group is created.) If $\text{flag}_{\text{contHide}}$ = false, the adversary also receives the identity and signature key of the group creator. Otherwise, the adversary receives ***leak-create**(id, svk). The adversary returns the new group's identity gid. Then, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ generates the initial key package by the *update-kp function. The *update-kp function generates a new key package by itself if both $\text{flag}_{\text{contHide}}$ and **safe** are true; otherwise asks the key package to the adversary $\mathcal{S}$. This models, in the ideal metadata-hiding CGKA protocol, an honest party generates a new key package, but it is hidden from the adversary.[25] Then, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ initializes the root node. Note that the epoch of an initial group is set to 0.

Creating proposals (See Fig. 12) A party id can be invoked by the environment $\mathcal{Z}$ to create a proposal with a specific action act. $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ informs the adversary $\mathcal{S}$ that a proposal message is being created.

[24] In the UC framework, it is conventional to call $\mathcal{S}$ appearing in the ideal functionality as the "adversary." We use the term "simulator" during the security proof.
[25] If the flag $\text{flag}_{\text{contHide}}$ is false, the key package is asked to the adversary $\mathcal{S}$; this means the key package is known to the adversary because the group creator is known.

$\mathcal{S}$ receives *__leak-prop__(id, act) and returns a flag *ack*, an ideal proposal p and a node pointer prop-id. If $\texttt{flag}_{\texttt{contHide}} = \texttt{false}$, then $\mathcal{S}$ obtains all the information (Ptr[id], id, act) = *__leak-prop__(id, act) that can be inferred from a non-encrypted proposal. $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ allows $\mathcal{S}$ to send *ack* = `false` to report that the protocol fails. If act = 'upd', $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ updates act with the new updated key package returned from `*update-kp` function. The `*update-kp` function generates a new key package by itself if both $\texttt{flag}_{\texttt{contHide}}$ and **safe** are true; otherwise asks the key package to the adversary $\mathcal{S}$. This models, in the ideal metadata-hiding CGKA protocol, an honest party generates a new key package, but the proposal message hides the key package from the adversary.[26] If the protocol succeeds, and if no node associated with p exists, $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ creates a new proposal node Prop[propCtr] and assigns propCtr to p (setting PropID[p] $\leftarrow$ propCtr). In certain situations, $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ may not create a new proposal node. For example, id proposes to remove the same party twice in the same epoch. Another example is when the adversary $\mathcal{S}$ controls the party's randomness (via setting RandCorr = 'bad') and the party proposes to update using the same randomness twice. In these cases, $\mathcal{S}$ can specify to attach the created proposal p to an existing proposal node prop-id. $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ then enforces that the states on the existing proposal node are consistent with the expected one using `*consistent-prop`. $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ marks whether the current epoch is secure or not using `*mark-content-hiden-epoch`. This information is used to determine epochs the adversary is allowed to corrupt (see the **restrict** check run within `Expose` in Fig. 14). Finally, if all check passes, $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ returns the proposal message p to the calling (dummy) party id, which simply relays it to the environment $\mathcal{Z}$.

Committing to proposals (See Fig. 12) A party id can be invoked by the environment $\mathcal{Z}$ to create a commit with a list of proposals $\vec{p}$, along with a (possibly fresh) signature verification key svk. $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ informs the adversary $\mathcal{S}$ that a commit message is sent and provides *__leak-com__(id, $\vec{p}$, svk). If $\texttt{flag}_{\texttt{contHide}} = \texttt{false}$, then $\mathcal{S}$ obtains all the information (Ptr[id], id, $\vec{p}$, svk, mem) = *__leak-com__(id, $\vec{p}$, svk) that can be inferred from a non-encrypted commit. $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ receives a flag *ack*, a commit node node-id, and a commit message $(c_0, \vec{c})$. Here, $\vec{c}$ is a list of party dependent messages $(\widehat{c}_{\text{id}})_{\text{id}}$; if selective downloading is performed (i.e., $\texttt{flag}_{\texttt{selDL}}$ is true), then party id only needs to retrieve $(c_0, \widehat{c}_{\text{id}})$ from the server. The adversary $\mathcal{S}$ sets *ack* := `false` to report that the protocol fails. $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ then obtains the new updated key package via the `*update-kp` function. This models, in the ideal metadata-hiding CGKA protocol, an honest committer generates a new key package, and it is hidden from the adversary. If the commit protocol succeeds, $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ first asks $\mathcal{S}$ to interpret the injected proposals, i.e., proposals where no node has been created, by calling `*fill-prop`. It then computes the new member set resulting from applying $\vec{p}$ to the current member set by calling `*next-members` (which returns $\perp$ if $\vec{p}$ contains invalid proposals).

$\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ then checks the format of $\vec{c}$ specified by $\mathcal{S}$. If $\texttt{flag}_{\texttt{selDL}}$ is true, $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ requires that $\vec{c}$ contains the same number of party-dependent messages as the number of the current members. Else, $\vec{c}$ must be $\perp$.

$\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ then either creates a new commit node or verifies that the existing node is consistent by `*consistent-com`. The adversary $\mathcal{S}$ can specify an existing node-id. This case may happen for example when the adversary makes a party process an injected commit message c_0 and then makes another party commit the same c_0 by controlling its randomness. If the specified node Node[node-id] is a detached root, $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ attaches it to id's current node by calling `*attach`. Once the detached root is attached to the main group, the root's tree achieves the same security guarantee as the main tree. Since attaching a detached root changes the topology of the history graph, $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ enforces two invariants: `cons-invariant` enforces the consistency of the graph, and `auth-invariant` enforces the authenticity guarantee.

When add proposals are committed (i.e., addedMem $\neq \perp$), $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ informs the adversary $\mathcal{S}$ that a welcome message is sent and provides *__leak-wel__ on input the id's current epoch Ptr[id], the new epoch Node[c_0] and the receiver's identity id_t. If $\texttt{flag}_{\texttt{contHide}} = \texttt{false}$, then $\mathcal{S}$ obtains all the information that can be inferred from a non-encrypted welcome message. $\mathcal{S}$ returns a simulated welcome message $\widehat{w}$ to $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$. We note that in prior definitions [36, 37], the commit and welcome messages were simultaneously simulated by $\mathcal{S}$. We consciously divide this process into two. This allows us to model the fact that a welcome message does not necessarily leak information about the group. That is, the server can observe that a party id is invited to some group but will not know which group.[27]. $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ assigns the welcome message to the commit node created above. Finally, $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ marks whether the current epoch is secure or not using `*mark-content-hiden-epoch` and returns $(c_0, \vec{c}, \vec{w})$ to the calling (dummy) party id, which simply relays it to the environment $\mathcal{Z}$

Processing commits (See Fig. 13) A party id can be invoked by the environment $\mathcal{Z}$ to process a commit message with an associating list of proposals $(c_0, \widehat{c}, \vec{p})$. We explain the case where selective downloading is performed (i.e., $\texttt{flag}_{\texttt{selDL}}$ is true). $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ first checks that $\widehat{c}$ is the correct id-dependent message associated with c_0, and outputs $\perp$ if it is incorrect. (In case $\texttt{flag}_{\texttt{selDL}}$ is false, $\widehat{c}$ must be $\perp$.) $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ then calls $\mathcal{S}$ on input *__leak-prop__(id) and $(c_0, \widehat{c}, \vec{p})$. $\mathcal{S}$ sets *ack* := `false` to report that the protocol failed. If the process succeeds, $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ first asks the adversary to interpret the injected proposals by calling `*fill-prop`. $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ then either creates a new commit node or verifies that the existing node is consistent. The adversary can specify the existing node-id. If the node corresponding to c_0 does not exist and the adversary does not specify any existing node, $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ checks the validity of $\vec{p}$ and creates a new commit node with the committer identity orig′ and its signature key svk′ which are $\mathcal{S}$ interprets from $(c_0, \vec{c}, \vec{p})$. Note that the new node holds the same group identity as id's current node and the epoch is incremented. If the commit message was assigned a node (i.e., Node[c_0] $\neq \perp$) or the adversary $\mathcal{S}$ specifies an existing node, $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ checks the validity of the group identity and epoch and enforces that it is a valid successor of id's current node by calling `*valid-successor`. If c_0 is assigned to a detached root, $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ attaches the root to id's current node. If c_0 is not assigned a node, $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ assigns the adversary-specified node-id to c_0.

[26] If the flag $\texttt{flag}_{\texttt{contHide}}$ is false, the key package is asked to the adversary $\mathcal{S}$; this means the key package is known to the adversary.

[27] Note that we can capture the situation where a welcome message leaks the group by defining *__leak-wel__ to output the node pointer of the commit message.

Finally, if c_0 removes id, $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ sets $\mathsf{Ptr}[\mathsf{id}] = \bot$. Otherwise, $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ updates id's secrets if necessary and moves $\mathsf{Ptr}[\mathsf{id}]$ to the new commit node. The calling party receives the committer's identity, the semantics of the applied proposals, and the list of $(\mathsf{id}, \mathsf{svk})$-pair.

Joining a group (See Fig. 13) A party id can be invoked by the environment $\mathcal{Z}$ to join a group using the welcome message $\widehat{w}$. $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ forwards $(\mathsf{id}, \widehat{w})$ to the adversary $\mathcal{S}$ and receives the interpreted result. As usual, the adversary sets $ack := \mathtt{false}$ to report that the protocol failed. If the process succeeds, $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ identifies the commit node $\mathsf{node\text{-}id} = \mathsf{Wel}[\mathsf{id}, \widehat{w}]$ corresponding to $\widehat{w}$. If this is the first time $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ sees $\widehat{w}$, i.e., $\mathsf{Wel}[\mathsf{id}, \widehat{w}] = \bot$, $\mathcal{S}$ can specify $\mathsf{node\text{-}id}'$. If the commit node for $\mathsf{node\text{-}id}'$ does not exist (i.e., $\mathsf{Node}[\mathsf{node\text{-}id}'] = \bot$), $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ creates a new detached root where all the stored values are chosen by $\mathcal{S}$. Finally, $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ updates id's secrets (registered in the key service $\mathcal{F}_{\mathrm{KS}}$) and returns the state of the joining group (the committer's identity, the group identity, the epoch, and the list of $(\mathsf{id}, \mathsf{kp})$-pair) to the calling (dummy) party id.

Group keys (See Fig. 13) Parties can fetch the current group secret via the Key query. The returned group secret k is random if the protocol guarantees its confidentiality (identified by the **safe** predicate). Otherwise, k is set by the adversary. Unlike prior definitions [36, 37], we also prepare two extra group secret keys: the metadata key k_{mh} that can be obtained using $\mathsf{Key}_{\mathsf{mh}}$, and the next metadata key $\mathsf{k}_{\mathsf{mh}}{}'$ that can be obtained using $\mathsf{NextKey}_{\mathsf{mh}}$. While $\mathsf{Key}_{\mathsf{mh}}$ is defined identically to Key, $\mathsf{NextKey}_{\mathsf{mh}}$ is a function that allows obtaining the metadata key at the next epoch. These keys are explicitly used to hide the *dynamic* metadata. That is, if we only care about hiding static metadata, these two group secret keys can be safely omitted from the definition.

Corruptions (See Fig. 14) The adversary $\mathcal{S}$ can corrupt a party id using the Expose query. When the adversary $\mathcal{S}$ inputs $(\mathsf{Expose}, \mathsf{id})$ to $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$, the ideal functionality records the following leaked information:

- The current group secret keys and id's key materials (e.g., encryption key and signing key). This is recorded by adding id to the exposed set of id's current node (cf. Line 2 in $(\mathsf{Expose}, \mathsf{id})$ query).
- The key materials created by id during an update or a commit at the current epoch. This is recorded by setting the status of all the child commit nodes created by id (i.e., nodes with $\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$) to 'bad' (cf. `*update-stat-after-exp` function).
- The current signature signing key ssk. This is recorded by signaling to $\mathcal{F}_{\mathrm{AS}}$ that svk is exposed and sends ssk to the adversary (cf. Line 5 in $(\mathsf{Expose}, \mathsf{id})$ query).

Then, the ideal functionality gives id's current epoch $\mathsf{Ptr}[\mathsf{id}]$, the associated information $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]]$, and id's current secret keys stored in the CurrDK array. (Note that $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ manages users' key packages and secret information, see above.) Also, the adversary is allowed to corrupt a non-group member id as well. In such a case, the key packages that id registered to the key service $\mathcal{F}_{\mathrm{KS}}$ are leaked. $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ signals to $\mathcal{F}_{\mathrm{KS}}$ that key packages (including the signing key) are exposed and send the corresponding decryption keys and signing keys to the adversary (cf. Line 7 in $(\mathsf{Expose}, \mathsf{id})$ query).

If an adversary is allowed to compromise key materials that can be used to compute a group secret key, which $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ has already assigned random values, then $\mathcal{Z}$ trivially distinguishes a real protocol from an ideal protocol. For instance, if $\mathcal{Z}$ queries Key for a commit node where the predicate **safe** is true, then $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ assigns a random value to the group secret key k. Then, if the adversary at some later point compromises a party id via an $(\mathsf{Expose}, \mathsf{id})$ and can compute the *real* group secret key k' from the compromised key materials, $\mathcal{Z}$ can distinguish the two protocols by checking if $\mathsf{k} = \mathsf{k}'$.

To avoid such trivial attacks, we restrict the environment to not be able to corrupt key materials for those commit nodes with $\mathsf{chall} = \mathtt{true}$ or $\mathsf{conthide} = \mathtt{true}$. The former is identical to those used in prior works [36, 37]; if a random group secret key was set, then $\mathcal{Z}$ cannot corrupt a party that allows recovering of the real group secret key. The latter is new to this work. For static metadata-hiding, recall that we must encrypt the proposal and commits. This is modeled in $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ by requiring the adversary $\mathcal{S}$ to create the (encrypted) proposal and commits *without* knowing the message when the predicate **safe** is true. If $\mathcal{Z}$ were to compromise a party that allows recovering the real group secret key, it can try to decrypt the encrypted proposal or commit to trivially distinguish between a real and ideal protocol. $\mathsf{conthide} = \mathtt{true}$ indicates that a commit node created random encryption and restricts $\mathcal{Z}$ from later corrupting it. Note that this implies that the predicate **safe** for honestly generated commit nodes cannot be switched from true to false once created. This is in sharp contrast to previous definitions since the group secret key was never implicitly used as part of the real protocol. We note that this is not a weakness of our security model but rather a natural consequence of considering CGKAs in a larger system.

Initialization

Ptr[∗], Prop[∗], Node[∗], Wel[∗] ← ⊥
propCtr, nodeCtr ← 1
/ Flag is set to true if selective downloading is performed.
$\texttt{flag}_{\text{selDL}}$ = true
/ Flag is set to true if propose and commit contents are hidden
$\texttt{flag}_{\text{contHide}}$ = true
PropID[∗], NodeID[∗] ← ⊥
PendDK[∗], CurrDK[∗] ← ⊥
RandCorr[∗] ← 'good'

Inputs from a party $\text{id}_{\text{creator}}$

Input (Create, svk)

req Ptr[$\text{id}_{\text{creator}}$] = ⊥
Send (Create, ***leak-create**($\text{id}_{\text{creator}}$, svk)) to $\mathcal{S}$ and receive gid
req ∗valid-svk($\text{id}_{\text{creator}}$, svk)
(kp, dk) ← ∗update-kp(id, svk)
mem ← { ($\text{id}_{\text{creator}}$, kp) }; CurrDK[id] ← dk
Node[0] ← ∗create-root(gid, 0, $\text{id}_{\text{creator}}$, mem, RandCorr[id])
Ptr[$\text{id}_{\text{creator}}$] ← 0

Inputs from a party id

Input (Propose, act), act ∈ { 'upd'-svk, 'add'-id_t-kp_t, 'rem'-id_t }

req Ptr[id] ≠ ⊥
Send (Propose, ***leak-prop**(id, act)) to $\mathcal{S}$ and receive (ack, prop-id, p)
req ack
if act = 'upd'-svk **then**
  req ∗valid-svk(id, svk)
  (kp, dk) ← ∗update-kp(id, svk)
  act ← 'upd'-kp
  PendDK[kp] ← dk
if act = 'add'-id_t-kp_t **then req** ∗valid-kp(id_t, kp_t)
if PropID[p] = ⊥ ∧ prop-id = ⊥ **then**
  Prop[propCtr] ← ∗create-prop(Ptr[id], id, act, RandCorr[id])
  PropID[p] ← propCtr; propCtr++
else
  if PropID[p] = ⊥ **then** (prop-id′, PropID[p]) ← (prop-id, prop-id)
  else prop-id′ ← PropID[p]
  ∗consistent-prop(prop-id′, id, act)
if act = 'upd'-svk ∧ RandCorr[id] = 'bad' **then**
  Send (exposed, id, svk) to $\mathcal{F}_{\text{AS}}$
/ Mark whether generated messages hide contents (static metadata)
∗mark-content-hiden-epoch(Ptr[id])
return p

Input (Commit, $\vec{p}$, svk)

req Ptr[id] ≠ ⊥
Send (Commit, ***leak-com**(id, $\vec{p}$, svk)) to $\mathcal{S}$ and receive (ack, node-id, c_0, $\vec{c}$)
req ∗valid-svk(id, svk)
(kp_{new}, dk_{new}) ← ∗update-kp(id, svk)
PendDK[kp_{new}] ← dk_{new}
req ∗succeed-com(id, $\vec{p}$, kp_{new}) ∨ ack
∗fill-prop($\vec{p}$)
(mem′, ∗) ← ∗next-members(Ptr[id], id, $\vec{p}$, kp_{new})
assert mem′ ≠ ⊥ ∧ (id, kp_{new}) ∈ mem′
/ If selective downloading is performed, then member specific $\vec{c}$ has the same size
/ as the current member. Otherwise, $\vec{c}$ is ⊥
if $\texttt{flag}_{\text{selDL}}$ **then**
  assert $|\vec{c}|$ = |Node[Ptr[id]].mem|
else
  assert $\vec{c}$ = ⊥
if NodeID[c_0] = ⊥ ∧ node-id = ⊥ **then**
  Node[nodeCtr] ← ∗create-child(Ptr[id], id, $\vec{p}$, $\vec{c}$, mem′, RandCorr[id])
  NodeID[c_0] ← nodeCtr; nodeCtr++
else
  if NodeID[c_0] = ⊥ **then** (node-id′, NodeID[c_0]) ← (node-id, node-id)
  else node-id′ ← NodeID[c_0]
  ∗consistent-com(node-id′, id, $\vec{p}$, mem)
  if Node[node-id′].par = ⊥ **then**
    ∗attach(node-id′, id, $\vec{p}$)
/ Create welcome message for added members
addedMem ← Node[NodeID[c_0]].mem \ Node[Ptr[id]].mem
$\vec{w}$ ← ∅
foreach (id_t, ∗) ∈ addedMem **do**
  Send (Welcome, ***leak-wel**(Ptr[id], NodeID[c_0], id_t)) to $\mathcal{S}$ and
    receive (ack, $\widehat{w}$)
  req ack
  parse (id_t, ∗) ← $\widehat{w}$
  assert Wel[id_t, $\widehat{w}$] ∈ { ⊥, NodeID[c_0] }
  Wel[id_t, $\widehat{w}$] ← NodeID[c_0]
  $\vec{w}$ +← $\widehat{w}$
assert cons-invariant ∧ auth-invariant
if RandCorr[id] = 'bad' **then**
  Send (exposed, id, svk) to $\mathcal{F}_{\text{AS}}$
/ Mark whether generated messages hide contents (static metadata)
∗mark-content-hiden-epoch(Ptr[id])
if $\vec{w}$ ≠ ∅ **then** ∗mark-content-hiden-epoch(NodeID[c_0])
return (c_0, $\vec{c}$, $\vec{w}$)

Figure 12: The ideal static metadata-hiding CGKA functionality $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$: Create, Propose, Commit. The modifications in order for the functionality to keep track of key packages are highlighted in in orange.

Input (Process, $c_0, \widehat{c}, \vec{p}$)

1 : **req** Ptr[id] $\neq \perp$
2 : / If c_0 and $\vec{p}$ were generated honestly, then use the existing node-id $\neq \perp$.
/ Otherwise, the group secret must be exposed and $\mathcal{S}$ decides where to attach Ptr[id].
3 : Send (Process, ***leak-proc**(id), $c_0, \widehat{c}, \vec{p}$) to $\mathcal{S}$ and
4 : receive (ack, node-id, orig′, kp′)
5 : **req** *succeed-proc(id, $c_0, \widehat{c}, \vec{p}$) $\vee$ ack
6 : *fill-prop($\vec{p}$)
7 : / If selective downloading is performed,
/ then only the member-specific commitment $\widehat{c}$ is accepted when c_0 is honestly generated.
if $\texttt{flag}_{\texttt{selDL}}$ **then**
8 : node-id ← NodeID[c_0]
9 : **if** node-id $\neq \perp \wedge$ Node[node-id].stat = 'good' **then**
10 : index_{id} ← Node[Ptr[id]].indexOf(id)
11 : **req** $\widehat{c}$ = Node[node-id].vcom[index_{id}]
12 : **else**
13 : **assert** $\widehat{c} = \perp$ / Otherwise, $\widehat{c}$ is $\perp$.
14 : / If node-id = $\perp$, create a new node; Otherwise, check consistency with existing node.
if NodeID[c_0] $= \perp \wedge$ node-id $= \perp$ **then**
15 : **try** (mem′, $*$) ← *next-members(Ptr[id], orig′, $\vec{p}$, kp′)
16 : **assert** mem′ $\neq \perp$
17 : Node[nodeCtr] ← *create-child(Ptr[id], orig′, $\vec{p}$, mem′, 'adv')
18 : (NodeID[c_0], node-id) ← (nodeCtr, nodeCtr)
19 : nodeCtr++
20 : **else**
21 : **if** NodeID[c_0] $= \perp$ **then** (node-id′, NodeID[c_0]) ← (node-id, node-id)
22 : **else** node-id′ ← NodeID[c_0]
23 : / After processing, require gid to remain the same and epoch to be incremented by 1.
24 : **assert** Node[Ptr[id]].gid = Node[node-id′].gid
25 : **assert** Node[Ptr[id]].epoch = Node[node-id′].epoch + 1
26 : id_c ← Node[node-id′].orig; kp_c ← Node[node-id′].mem[id_c]
27 : (mem′, $*$) ← *next-members(Ptr[id], $\text{id}_c, \vec{p}, \text{kp}_c$)
28 : **assert** mem′ $\neq \perp$
29 : *valid-successor(node-id′, $\text{id}_c, \vec{p}$, mem′)
30 : **if** Node[node-id′].par $= \perp$ **then** *attach(node-id′, id, $\vec{p}$)
31 : / Mark whether generated messages hide contents (static metadata)
32 : *mark-content-hiden-epoch(Ptr[id])
33 : **if** $\exists p \in \vec{p}$: Prop[p].act = 'rem'-id **then** Ptr[id] ← $\perp$
34 : **else**
35 : **assert** (id, $*$) $\in$ Node[NodeID[c_0]].mem
36 : / Fetch new dk if key package is updated
37 : **if** Node[Ptr[id]].mem[id] $\neq$ Node[NodeID[c_0]].mem[id] **then**
38 : kp ← Node[NodeID[c_0]].mem[id]
39 : CurrDK[id] ← PendDK[kp]
40 : Ptr[id] ← NodeID[c_0]
41 : **assert** cons-invariant $\wedge$ auth-invariant
42 : **return** *output-proc(node-id′)

Input (Join, id, $\widehat{w}$)

1 : **req** Ptr[id] $= \perp$
2 : / If $\widehat{w}$ was generated honestly, then use the existing node-id $\neq \perp$.
3 : Send (Join, id, $\widehat{w}$) to $\mathcal{S}$ and
receive (ack, node-id′, gid′, epoch′, orig′, mem′)
4 : **req** *succeed-wel(id, $\widehat{w}$) $\vee$ ack
5 : node-id ← Wel[id, $\widehat{w}$]
6 : **if** node-id $= \perp$ **then**
7 : **if** Node[node-id′] $\neq \perp$ **then** node-id ← node-id′
8 : **else**
9 : Node[nodeCtr] ← *create-root(gid′, epoch′, orig′, mem′, 'adv')
10 : node-id ← nodeCtr
11 : nodeCtr++
12 : Wel[id, $\widehat{w}$] ← node-id
13 : kp ← Node[node-id].mem[id]
14 : CurrDK[id] ← DK[id, kp] / Fetch the registered dk used to join
15 : Ptr[id] ← node-id
16 : **assert** (id, $*$) $\in$ Node[node-id].mem
17 : **assert** cons-invariant $\wedge$ auth-invariant
18 : **return** *output-join(node-id)

Input (Key)

1 : **req** Ptr[id] $\neq \perp$
2 : **if** Node[Ptr[id]].key $= \perp$ **then** *set-key(Ptr[id])
3 : **return** Node[Ptr[id]].key

Input (Key_{mh})

1 : **req** Ptr[id] $\neq \perp$
2 : **if** Node[Ptr[id]].$\text{k}_{\text{mh}} = \perp$ **then** *set-key(Ptr[id])
3 : **return** Node[Ptr[id]].k_{mh}

Input ($\text{NextKey}_{\text{mh}}, c_0$)

1 : **req** Ptr[id] $\neq \perp \wedge$ NodeID[c_0] $\neq \perp$
2 : **req** Node[NodeID[c_0]].par = Ptr[id] $\wedge$ Node[NodeID[c_0]].orig = id
3 : **if** Node[NodeID[c_0]].$\text{k}_{\text{mh}} = \perp$ **then** *set-key(NodeID[c_0])
4 : **return** Node[NodeID[c_0]].k_{mh}

Figure 13: The ideal static metadata-hiding CGKA functionality $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$: Process, Join, Key, Key_{mh}, and $\text{NextKey}_{\text{mh}}$. The last two Key_{mh} and $\text{NextKey}_{\text{mh}}$ are to be used in a higher layer protocol. The modifications in order for the functionality to keep track of key packages are highlighted in in orange

Input (Expose, id)

1 : **if** Ptr[id] ≠ ⊥ **then**
2 : Node[Ptr[id]].exp +← id
3 : *update-stat-after-exp(id) / Pending secrets are marked as exposed.
4 : svk ← Node[Ptr[id]].mem[id].svk / Take svk from id's key package
5 : Send (exposed, id, svk) to $\mathcal{F}_{\text{AS}}$
6 : Send (Ptr[id], Node[Ptr[id]]) to $\mathcal{S}$ / All information stored in Node[Ptr[id]] is sent to $\mathcal{S}$.
7 : Send CurrDK[id] to $\mathcal{S}$ / id's secret key is sent to $\mathcal{S}$.
8 : Send (exposed, id) to $\mathcal{F}_{\text{KS}}$
9 : **restrict** ∀node-id :
10 : **if** Node[node-id].chall = true **then safe**(node-id) = true
11 : **if** Node[node-id].conthide = true **then safe**(node-id) = true

Input (CorrRand, id, b), $b \in \{$ 'good', 'bad' $\}$

1 : RandCorr[id] ← b

Figure 14: The static metadata-hiding CGKA functionality $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$: Corruptions from the adversary $\mathcal{S}$. The difference between those of $\mathcal{F}_{\text{CGKA}}$ [37] is highlighted in yellow. The modifications in order for the functionality to keep track of key packages are highlighted in orange

*create-root(gid, epoch, id, mem, stat)

1 : **return** new node with par ← ⊥, orig ← id, gid ← gid
epoch ← epoch, prop ← ⊥, mem ← mem, stat ← stat.

*create-child(node-id, id, $\vec{p}$, $\vec{c}$, mem, stat)

1 : (gid, epoch) ← (Node[Ptr[id]].gid, Node[Ptr[id]].epoch)
2 : / Create a new node with an incremented epoch.
3 : **return** new node with par ← node-id, orig ← id,
epoch ← epoch + 1, prop ← $\vec{p}$, vcom ← $\vec{c}$,
mem ← mem, stat ← stat.

*create-prop(node-id, id, act, stat)

1 : **return** new node with par ← node-id, orig ← id,
act ← act, stat ← stat.

*fill-prop($\vec{p}$)

1 : **foreach** $p \in \vec{p}$ **s.t.** PropID[p] = ⊥ **do**
2 : Send (Propose, Ptr[id], p) to $\mathcal{S}$ and receive (prop-id, orig, act)
3 : **if** prop-id = ⊥ **then**
4 : Prop[propCtr] ← *create-prop(Ptr[id], orig, act, 'adv')
5 : PropID[p] ← propCtr
6 : propCtr++
7 : / If $\text{flag}_{\text{contHide}}$ = true and p hides the same content, then $\mathcal{S}$ outputs prop-id ≠ ⊥.
/ In this case, check consistency with the exiting node.
8 : **else**
9 : *consistent-prop(prop-id, orig, act)
10 : PropID[p] ← prop-id

*set-key(node-id)

1 : **if safe**(node-id) **then**
2 : Node[node-id].key ←$ $\mathcal{K}$
3 : Node[node-id].chall ← true
4 : **else**
5 : Send (Key, id) to $\mathcal{S}$ and receive k
6 : Node[node-id].key ← k
7 : Node[node-id].chall ← false

*mark-content-hiden-epoch(node-id)

1 : **if safe**(node-id) **then**
2 : Node[node-id].conthide ← true
3 : **else**
4 : Node[node-id].conthide ← false

*update-stat-after-exp(id)

1 : **foreach** prop-id **s.t.** Prop[prop-id] ≠ ⊥
2 : ∧ Prop[prop-id].par = Ptr[id]
3 : ∧ Prop[prop-id].orig = id
4 : ∧ Prop[prop-id].act = 'upd'-∗ **do**
5 : Prop[prop-id].stat ← 'bad'
6 : **foreach** node-id **s.t.** Node[node-id] ≠ ⊥
7 : ∧ Node[node-id].par = Ptr[id]
8 : ∧ Node[node-id].orig = id **do**
9 : Node[node-id].stat ← 'bad'

Figure 15: The helper functions for creating and maintaining the history graph.

```
*valid-kp(id, kp)
1 : ack ← query (has-kp, id, kp) to F_KS
2 : return ack
```

```
*valid-svk(id, svk′)
1 : if Ptr[id] ≠ ⊥ then
2 :     svk ← Node[Ptr[id]].mem[id]
3 :     if svk′ ≠ ⊥ ∧ svk = svk′ then
4 :         return true
5 : ack ← query (has-ssk, id, svk′) to F_AS
6 : return ack
```

```
*update-kp (id, svk)
1 : if flag_contHide ∧ safe(Ptr[id]) then
2 :     ssk ← query (get-ssk, svk) to F_AS on behalf of id
3 :     if RandCorr[id] = 'good' then
4 :         (kp, dk) ← genKP(id, svk, ssk)
5 :     else
6 :         Send (rnd, id) to the adversary and receive r
7 :         (kp, dk) ← genKP(id, svk, ssk; r)
8 : else
9 :     Receive (kp, dk) from S
10 : return (kp, dk)
```

Figure 16: The helper functions related to keys. The `*update-kp` function highlighted in in orange is newly introduced to manage key packages in the functionality.

```
*output-proc(node-id)
1 : id_c ← Node[node-id].orig
2 : kp_c ← Node[node-id].mem[id_c]
3 : (*, propSem) ← *next-members(node-id, id_c, Node[node-id].prop, kp_c)
4 : return (Node[node-id].orig, propSem, Node[node-id].mem)
```

```
*output-join(node-id)
1 : gid ← Node[node-id].gid
2 : epoch ← Node[node-id].epoch
3 : mem ← Node[node-id].mem
4 : id_c ← Node[node-id].orig
5 : return (id_c, gid, epoch, mem)
```

Figure 17: The helper functions define output of process and join protocols.

*next-members(node-id, $\mathrm{id}_c, \vec{p}, \mathrm{kp}_c$)

1 : **if** Node[node-id] $\neq \bot \wedge (\mathrm{id}_c, *) \in$ Node[node-id].mem
  $\wedge \forall p \in \vec{p} : \mathrm{Prop}[p] \neq \bot \wedge \mathrm{Prop}[p].\mathrm{par} = \text{node-id}$ **then**
2 : $\vec{p}_{\text{'upd'}} \| \vec{p}_{\text{'rem'}} \| \vec{p}_{\text{'add'}} \leftarrow$ *sort-proposals($\vec{p}$)
3 : mem $\leftarrow$ Node[node-id].mem
4 : mem $-\!\leftarrow (\mathrm{id}_c, *)$; mem $+\!\leftarrow (\mathrm{id}_c, \mathrm{kp}_c)$
5 : $L \leftarrow \{ \mathrm{id}_c \}$ / set of updated parties
6 : **foreach** $p \in \vec{p}_{\text{'upd'}}$ **do**
7 : $(\mathrm{id}_s, \text{'upd'-kp}) \leftarrow (\mathrm{Prop}[p].\mathrm{orig}, \mathrm{Prop}[p].\mathrm{act})$
8 : **if** $\neg((\mathrm{id}_s, *) \in \text{mem} \wedge \mathrm{id}_s \notin L)$ **then return** $(\bot, \bot)$
9 : mem $-\!\leftarrow (\mathrm{id}_s, *)$; mem $+\!\leftarrow (\mathrm{id}_s, \mathrm{kp})$
10 : $L +\!\leftarrow \mathrm{id}_s$
11 : **foreach** $p \in \vec{p}_{\text{'rem'}}$ **do**
12 : $(\mathrm{id}_s, \text{'rem'-id}_t) \leftarrow (\mathrm{Prop}[p].\mathrm{orig}, \mathrm{Prop}[p].\mathrm{act})$
13 : **if** $\neg((\mathrm{id}_s, *) \in \text{mem} \wedge (\mathrm{id}_t \in \text{mem} \wedge \mathrm{id}_t \notin L))$ **then return** $(\bot, \bot)$
14 : mem $-\!\leftarrow (\mathrm{id}_t, *)$
15 : **foreach** $p \in \vec{p}_{\text{'add'}}$ **do**
16 : $(\mathrm{id}_s, \text{'add'-id}_t\text{-kp}_t) \leftarrow (\mathrm{Prop}[p].\mathrm{orig}, \mathrm{Prop}[p].\mathrm{act})$
17 : **if** $\neg((\mathrm{id}_s, *) \in \text{mem} \wedge (\mathrm{id}_t, *) \notin \text{mem})$ **then return** $(\bot, \bot)$
18 : mem $+\!\leftarrow (\mathrm{id}_t, \mathrm{kp}_t)$
19 : $P \leftarrow ((\mathrm{Prop}[\mathrm{PropID}[p]].\mathrm{orig}, \mathrm{Prop}[\mathrm{PropID}[p]].\mathrm{act}) : p \in \vec{p}_{\text{'upd'}} \| \vec{p}_{\text{'rem'}} \| \vec{p}_{\text{'add'}})$
20 : **return** (mem, P)
21 : **else**
22 : **return** $(\bot, \bot)$

*sort-proposals($\vec{p}$)

1 : $\vec{p}_{\text{'upd'}}, \vec{p}_{\text{'rem'}}, \vec{p}_{\text{'add'}} \leftarrow ()$
2 : **foreach** $p \in \vec{p}$ **do**
3 : $\mathrm{act}_p \leftarrow \mathrm{Prop}[\mathrm{PropID}[p]].\mathrm{act}$
4 : **if** $\mathrm{act}_p =$ 'upd'-$*$ **then**
5 : $\vec{p}_{\text{'upd'}} \,{+\!\!+}\!\leftarrow p$
6 : **elseif** $\mathrm{act}_p =$ 'rem'-$*$ **then**
7 : $\vec{p}_{\text{'rem'}} \,{+\!\!+}\!\leftarrow p$
8 : **if** $\mathrm{act}_p =$ 'add'-$*$ **then**
9 : $\vec{p}_{\text{'add'}} \,{+\!\!+}\!\leftarrow p$
10 : **return** $\vec{p}_{\text{'upd'}} \| \vec{p}_{\text{'rem'}} \| \vec{p}_{\text{'add'}}$

Figure 18: The helper functions to determine the group state after applying a commit. *sort-proposals($\vec{p}$) orders applying proposals.

*consistent-prop(prop-id, id, act)

1 : **assert** Prop[prop-id].par = Ptr[id] $\wedge$ Prop[prop-id].orig = id
2 : $\wedge$ Prop[prop-id].act = act

*consistent-com(node-id, id, $\vec{p}$, mem)

1 : *valid-successor(node-id, id, $\vec{p}$, mem)
2 : **assert** RandCorr[id] = 'bad' $\wedge$ Node[node-id].orig = id

*valid-successor(node-id, id, $\vec{p}$, mem)

1 : **assert** Node[node-id] $\neq \bot \wedge$ Node[node-id].mem = mem
  $\wedge$ Node[node-id].prop $\in \{ \bot, \vec{p} \}$
  $\wedge$ Node[node-id].par $\in \{ \bot, \mathrm{Ptr}[\mathrm{id}] \}$

*attach(node-id, id, $\vec{p}$)

1 : / Cannot attach to the original honest root node-id = 0
2 : **assert** node-id $\neq 0$
3 : Node[node-id].par $\leftarrow$ Ptr[id]
4 : Node[node-id].prop $\leftarrow \vec{p}$

*succeed-com(id, $\vec{p}$, kp)

1 : **return** $\forall p \in \vec{p}$: (prop-id := PropID[p] $\neq \bot \wedge$ Prop[prop-id].stat $\neq$ 'adv')
2 : $\wedge$ *next-members(Ptr[id], id, $\vec{p}$, kp) $\neq (\bot, \bot)$

*succeed-proc(id, $c_0, \widehat{c}, \vec{p}$)

1 : node-id $\leftarrow$ NodeID[c_0]
2 : $\mathrm{index}_{\mathrm{id}} \leftarrow$ Node[node-id].indexOf(id)
3 : **return** node-id $\neq \bot \wedge$ Node[node-id] $\neq \bot \wedge$ Node[node-id].par = Ptr[id]
4 : $\wedge$ Node[node-id].prop = $\vec{p} \wedge$ Node[node-id].stat $\neq$ 'adv'
5 : $\wedge \forall p \in \vec{p}$: Prop[PropID[p]].stat $\neq$ 'adv'
6 : $\wedge$ Node[node-id].vcom[$\mathrm{index}_{\mathrm{id}}$] = $\widehat{c}$

*succeed-wel(id, $\widehat{w}$)

1 : node-id $\leftarrow$ Wel[id, $\widehat{w}$]
2 : **return** node-id $\neq \bot$
3 : $\wedge$ Node[node-id] $\neq \bot \wedge$ Node[node-id].stat $\neq$ 'adv'
4 : $\wedge$ (id, $*$) $\in$ (Node[node-id].mem \ Node[Node[node-id].par].mem)

Figure 19: The helper functions for consistency and correctness.

`auth-invariant`

return `true` **iff**

(a) $\forall$node-id with node-id$_p$:= Node[node-id].par, node-id$_p \neq \perp$
and id := Node[node-id].orig :
if Node[node-id].stat = 'adv' **then**
sig-inj-allowed(node-id$_p$, id) $\wedge$ **mac-inj-allowed**(node-id$_p$) and

(b) $\forall p$ with node-id$_p$:= Prop[p].par and id := Prop[p].orig :
if Prop[p].stat = 'adv' **then**
sig-inj-allowed(node-id$_p$, id) $\wedge$ **mac-inj-allowed**(node-id$_p$) and

(c) $\forall$node-id with Node[node-id].par = $\perp$ and id := Node[node-id].orig :
sig-inj-allowed(node-id, id)

`cons-invariant`

return `true` **iff**

(a) $\forall$node-id **s.t.** Node[node-id].par $\neq \perp$:
Node[node-id].prop $\neq \perp \wedge$
$\forall p \in$ Node[node-id].prop :
Prop[PropID[p]].par = Node[node-id].par and

(b) $\forall$id **s.t.** Ptr[id] $\neq \perp$: (id, $*$) $\in$ Node[Ptr[id]].mem and

(c) the history graph contains no cycle

Figure 20: The history graph invariants.

C STATIC METADATA-HIDING CGKA: CONSTRUCTION AND SECURITY PROOF

In this section, we first modify Chained CmPKE of Hashimoto et al. [36] into a protocol, which we call Chained CmPKE$^{\text{ctxt}}$, that further secures the static metadata. We then prove that Chained CmPKE$^{\text{ctxt}}$ UC-realizes the ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. This results in the first CGKA that provably secures the 2nd layer.

C.1 Constructing Chained CmPKE$^{\text{ctxt}}$

We provide the description of Chained CmPKE$^{\text{ctxt}}$. The protocol state and the related helper method are shown in Tabs. 5 to 8. The main protocol is depicted in Figs. 22 and 23, and the associated helper functions are depicted in Figs. 25 to 33.

Chained CmPKE$^{\text{ctxt}}$ is almost identical to Chained CmPKE [37]. The main differences are explained below and highlighted in yellow in the figures. For a detailed description of the common parts of the protocols, we refer the readers to [37].

- Chained CmPKE$^{\text{ctxt}}$ additionally generates an *encryption secret* encSecret and a *welcome secret* welcomeSecret from the joiner secret.[28] encSecret is used to encrypt the contents in the proposal and commit messages, excluding the group identifier gid, epoch, and the message type, which are necessary for message delivery (cf. `*enc-prop` and `*enc-commit` functions in Fig. 32). welcomeSecret is used to encrypt the group information and the signature in the welcome message (cf. `*enc-welcome` function in Fig. 33).
- Chained CmPKE$^{\text{ctxt}}$ creates a separate welcome message for each new member. In contrast, Chained CmPKE included a member-independent welcome message w_0 that is transmitted to every new group member (see Footnote 9). However, this will allow the server to infer that the recipients with the same w_0 will join a (possibly unknown) group.
- In Chained CmPKE$^{\text{ctxt}}$, parties explicitly sort received proposals. In contrast, Chained CmPKE and MLS's TreeKEM implicitly assume that when parties fetch the proposals, the server sorts the proposals according to predetermined rules[29]. However, since proposals are encrypted in Chained CmPKE$^{\text{ctxt}}$, the server can no longer sort them. Therefore, we make parties sort fetched proposals via `*dec-and-sort-proposals` shown in Fig. 32 before they commit or process the proposals.
- (Optional for hiding the dynamic metadata) Chained CmPKE$^{\text{ctxt}}$ additionally generates an *metadata secret* metaKey from the joiner secret. This is used when hiding the *dynamic* metadata (see App. E for more detail).

[28] An encryption secret and welcome secret are also generated in MLS [14, Table 3] used to secure the static metadata.

[29] MLS stipulates that proposals are to be applied in the order Update, Remove, Add [14, Sec. 13.2.2].

Table 5: The protocol state. The additional component from [37] are highlighted in yellow.

G.gid	The identifier of the group.
G.epoch	The current epoch number.
G.confTransHash	The confirmed transcript hash.
G.confTransHash-w.o-'id_c'	The confirmed transcript hash without the committer identity.
G.interimTransHash	The interim transcript hash for the next epoch.
G.member[$*$]	A mapping associating party id with its state.
G.memberHash	A hash of the public part of G.member[$*$].
G.certSvks[$*$]	A mapping associating the set of validated signature verification keys to each party.
G.pendUpd[$*$]	A mapping associating the secret keys for each pending update proposal issued by id.
G.pendCom[$*$]	A mapping associating the new group state for each pending commit issued by id.
G.id	The identity of the party.
G.ssk	The current signing key.
G.appSecret	The current epoch's shared key.
G.membKey	The key used to MAC proposal packages.
G.encSecret	The key used to encrypt proposal and commit messages.
G.metaKey	The key used to hide the dynamic metadata in a higher level protocol.
G.initSecret	The next epoch's init secret.

Table 6: The party id's state stored in G.member[id] and helper method.

id	The identity of the party.
ek	The encryption key of a mPKE scheme.
dk	The corresponding decryption key.
svk	The signature verification key of a signature scheme.
sig	The signature for (id, ek, svk) under the signature singing key corresponding to svk.
kp()	Returns (id, ek, svk, sig) (if G.member[id] $\neq \perp$).

Table 7: The helper methods on the protocol state. The additional method from [37] are highlighted in yellow.

G.clone()	Returns (independent) copy of G.
G.memberIDs()	Returns the list of party ids sorted by dictionary order.
G.memberIDsvks()	Returns the list of party ids and its associating svk sorted by dictionary order in the ids.
G.memberPublicInfo()	Returns the public part of G.member[$*$].
G.groupCont()	Returns (G.gid, G.epoch, G.memberHash, G.confTransHash).
G.indexOf(id)	Returns the index of id in the sorted member list returned by G.memberIDs().

Table 8: The protocol state maintained only during the proof. The additional component from [37] are highlighted in yellow.

G.joinerSecret	The current epoch's joiner secret.
G.comSecret	The current epoch's commit secret.
G.confKey	The key used to MAC for commit and welcome messages.
G.welcomeSecret	The key used to encrypt group information and signature included in welcome messages.
G.confTag	The MAC tag included either in the commit or welcome message.
G.membTags	The set of MAC tags included in the proposal messages.

genSSK()

1 : $(\mathsf{svk}, \mathsf{ssk}) \leftarrow \mathsf{SIG.KeyGen}(\mathsf{pp}_{\mathsf{SIG}})$
2 : **return** $(\mathsf{svk}, \mathsf{ssk})$

genKP(id, svk, ssk)

1 : $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{CmGen}(\mathsf{pp}_{\mathsf{CmPKE}})$
2 : $\mathsf{sig} \leftarrow \mathsf{SIG.Sign}(\mathsf{pp}_{\mathsf{SIG}}, \mathsf{ssk}, (\mathsf{id}, \mathsf{ek}, \mathsf{svk}))$
3 : $\mathsf{kp} \leftarrow (\mathsf{id}, \mathsf{ek}, \mathsf{svk}, \mathsf{sig})$
4 : **return** $(\mathsf{kp}, \mathsf{dk})$

Figure 21: Key generation and verification algorithms.

Input (`Create`, svk)

1 : **req** $G = \bot \wedge \mathsf{id} = \mathsf{id}_{\text{creator}}$
2 : $G.\mathsf{gid} \leftarrow\!\!\$\ \{0,1\}^\kappa$; $G.\mathsf{joinerSecret} \leftarrow\!\!\$\ \{0,1\}^\kappa$
3 : $G.\mathsf{epoch} \leftarrow 0$
4 : $G.\mathsf{member}[*] \leftarrow \bot$; $G.\mathsf{memberHash} \leftarrow \bot$
5 : $G.\mathsf{confTransHash\text{-}w.o\text{-}`id_c'} \leftarrow \bot$
6 : $G.\mathsf{confTransHash} \leftarrow \bot$
7 : $G.\mathsf{certSvks}[*] \leftarrow \emptyset$
8 : $G.\mathsf{pendUpd}[*] \leftarrow \bot$; $G.\mathsf{pendCom}[*] \leftarrow \bot$
9 : $G.\mathsf{id} \leftarrow \mathsf{id}$
10 : **try** ssk $\leftarrow$ `*fetch-ssk-if-nec`(G, svk)
11 : $(\mathsf{kp}, \mathsf{dk}) \leftarrow \mathsf{genKP}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk})$
12 : $G \leftarrow$ `*assign-kp`$(G, \mathsf{id}, \mathsf{kp})$
13 : $G.\mathsf{member}[\mathsf{id}].\mathsf{dk} \leftarrow \mathsf{dk}$
14 : $G.\mathsf{ssk} \leftarrow \mathsf{ssk}$
15 : $G.\mathsf{memberHash} \leftarrow$ `*derive-member-hash`(G)
16 : $(G, \mathsf{confKey}) \leftarrow$ `*derive-epoch-keys`$(G, G.\mathsf{joinerSecret})$
17 : $\mathsf{confTag} \leftarrow$ `*gen-conf-tag`$(G, \mathsf{confKey})$
18 : $G \leftarrow$ `*set-interim-trans-hash`$(G, \mathsf{confTag})$

Input (`Propose`, 'upd'-svk)

1 : **req** $G \neq \bot$
2 : **try** ssk $\leftarrow$ `*fetch-ssk-if-nec`(G, svk)
3 : $(\mathsf{kp}, \mathsf{dk}) \leftarrow \mathsf{genKP}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk})$
4 : $P \leftarrow (\text{'upd'}, \mathsf{kp})$
5 : $p \leftarrow$ `*frame-prop`(G, P)
6 : $G.\mathsf{pendUpd}[p] \leftarrow (\mathsf{ssk}, \mathsf{dk})$
7 : $p^{\mathsf{ctxt}} \leftarrow$ `*enc-prop`$(G.\mathsf{encSecret}, p)$
8 : **return** p^{ctxt}

Input (`Propose`, 'add'-id_t-kp_t)

1 : **req** $G \neq \bot \wedge \mathsf{id}_t \notin G.\mathsf{memberIDs}()$
2 : **req** $\mathsf{kp}_t \neq \bot$
3 : Send $(\mathsf{has\text{-}kp}, \mathsf{id}_t, \mathsf{kp}_t)$ to $\mathcal{F}_{\text{KS}}$ and receive *ack*
4 : **req** *ack* / *ack* = true implies `*validate-kp`$(\mathsf{kp}_t, \mathsf{id}_t) = \mathsf{true}$
5 : $P \leftarrow (\text{'add'}, \mathsf{kp}_t)$
6 : $p \leftarrow$ `*frame-prop`(G, P)
7 : $p^{\mathsf{ctxt}} \leftarrow$ `*enc-prop`$(G.\mathsf{encSecret}, p)$
8 : **return** p^{ctxt}

Input (`Propose`, 'rem'-id_t)

1 : **req** $G \neq \bot \wedge \mathsf{id}_t \in G.\mathsf{memberIDs}()$
2 : $P \leftarrow (\text{'rem'}, \mathsf{id}_t)$
3 : $p \leftarrow$ `*frame-prop`(G, P)
4 : $p^{\mathsf{ctxt}} \leftarrow$ `*enc-prop`$(G.\mathsf{encSecret}, p)$
5 : **return** p^{ctxt}

Input (`Commit`, $\vec{p}^{\mathsf{ctxt}}$, svk)

1 : **req** $G \neq \bot$
2 : **try** $\vec{p} \leftarrow$ `*dec-and-sort-proposals`$(G.\mathsf{encSecret}, \vec{p}^{\mathsf{ctxt}})$
3 : $G' \leftarrow$ `*init-epoch`(G)
4 : **try** $(G', \mathit{upd}, \mathit{rem}, \mathit{add}) \leftarrow$ `*apply-props`$(G, G', \vec{p})$
5 : **req** $(*, \text{'rem'-id}) \notin \mathit{rem} \wedge (\mathsf{id}, *) \notin \mathit{upd}$
6 : $\mathsf{addedMem} \leftarrow \{\ \mathsf{id}_t \mid (*, \text{'add'-}\mathsf{id}_t\text{-}*) \in \mathit{add}\ \}$ / Recipients of the welcome message
7 : $\mathsf{receivers} \leftarrow G'.\mathsf{memberIDs}() \setminus \mathsf{addedMem}$ / Recipients of the new commit secret
8 : **try** $(G', \mathsf{comSecret}, \mathsf{kp}, \mathsf{ct}_0, \vec{\mathsf{ct}} = (\widehat{\mathsf{ct}}_{\mathsf{id}})_{\mathsf{id} \in \mathsf{receivers}}) \leftarrow$ `*rekey`$(G', \mathsf{receivers}, \mathsf{id}, \mathsf{svk})$
9 : $G' \leftarrow$ `*set-member-hash`(G')
10 : $\mathsf{propIDs} \leftarrow ()$
11 : **foreach** $p \in \vec{p}$ **do** $\mathsf{propIDs} \mathrel{+}\!\!\leftarrow H(p)$
12 : $C_0 \leftarrow (\mathsf{propIDs}, \mathsf{kp}, \mathsf{ct}_0)$
13 : $\mathsf{sig} \leftarrow$ `*sign-commit`(G, C_0)
14 : $G' \leftarrow$ `*set-conf-trans-hash`$(G, G', \mathsf{id}, C_0, \mathsf{sig})$
15 : $(G', \mathsf{confKey}, \mathsf{joinerSecret}) \leftarrow$ `*derive-keys`$(G, G', \mathsf{comSecret})$
16 : $\mathsf{confTag} \leftarrow$ `*gen-conf-tag`$(G', \mathsf{confKey})$
17 : $c_0 \leftarrow$ `*frame-commit`$(G, C_0, \mathsf{sig}, \mathsf{confTag})$
18 : $G' \leftarrow$ `*set-interim-trans-hash`$(G', \mathsf{confTag})$
19 : $\vec{c} \leftarrow \emptyset$
20 : **foreach** $\mathsf{id} \in G.\mathsf{memberIDs}()$ **do**
21 : **if** $\mathsf{id} \in \mathsf{receivers}$ **then** $\vec{c} \mathrel{+}\!\!\leftarrow (\mathsf{id}, \widehat{\mathsf{ct}}_{\mathsf{id}})$
22 : **else** $\vec{c} \mathrel{+}\!\!\leftarrow \widehat{c}_{\mathsf{id}} := (\mathsf{id}, \bot)$
23 : **if** $\mathit{add} \neq ()$ **then**
24 : $(G', w_0, \vec{w}) \leftarrow$ `*welcome-msg`$(G', \mathsf{addedMem}, \mathsf{joinerSecret}, \mathsf{confTag})$
25 : **else**
26 : $w_0 \leftarrow \bot$; $\vec{w} \leftarrow \emptyset$
27 : $G.\mathsf{pendCom}[c_0] \leftarrow (G', \vec{p}, \mathit{upd}, \mathit{rem}, \mathit{add})$
28 : / Encrypt messages
29 : $(c_0^{\mathsf{ctxt}}, \vec{c}^{\mathsf{ctxt}}) \leftarrow$ `*enc-commit`$(G.c_0, \vec{c})$
30 : $\vec{w}^{\mathsf{ctxt}} \leftarrow \emptyset$
31 : $\mathsf{welcomeSecret} \leftarrow \mathsf{HKDF.Expand}(\mathsf{joinerSecret}, \text{'wel'})$
32 : **foreach** $\widehat{w} \in \vec{w}$ **then**
33 : $\widehat{w}^{\mathsf{ctxt}} \leftarrow$ `*enc-welcome`$(\mathsf{welcomeSecret}, w_0, \widehat{w})$
34 : $\vec{w}^{\mathsf{ctxt}} \mathrel{+}\!\!\leftarrow \widehat{w}^{\mathsf{ctxt}}$
35 : **return** $(c_0^{\mathsf{ctxt}}, \vec{c}^{\mathsf{ctxt}}, \vec{w}^{\mathsf{ctxt}})$

Figure 22: Static metadata-hiding CGKA protocol Chained CmPKE$^{\mathsf{ctxt}}$: `Create`, `Propose`, and `Commit`. The major changes from [37] are highlighted in yellow.

Input $(\texttt{Process}, c_0^{\text{ctxt}}, \widehat{c}^{\text{ctxt}}, \vec{p}^{\text{ctxt}})$

1: **req** $G \neq \bot$
2: **try** $(c_0, \widehat{c}) \leftarrow$ *dec-commit$(G.\text{encSecret}, c_0^{\text{ctxt}}, \widehat{c}^{\text{ctxt}})$
3: **try** $\vec{p} \leftarrow$ *dec-and-sort-proposals$(G.\text{encSecret}, \vec{p}^{\text{ctxt}})$
4: $(\text{id}_c, C_0, \text{sig}, \text{confTag}) \leftarrow$ *unframe-commit(G, c_0)
5: **if** $\text{id}_c = \text{id}$ **then**
6: **parse** $(G', \vec{p}', upd, rem, add) \leftarrow G.\text{pendCom}[c_0]$
7: **req** $\vec{p} = \vec{p}'$
8: **return** $(\text{id}_c, upd \| rem \| add, G'.\text{memberIDsvks}())$
9: **parse** $(\text{propIDs}, \text{kp}_c, \text{ct}_0) \leftarrow C_0$
10: **parse** $(\text{id}', \widehat{\text{ct}}_{\text{id}'}) \leftarrow \widehat{c}'$
11: **req** $G.\text{id} = \text{id}'$
12: **for** $i \in 1, \ldots, |\vec{p}|$ **do**
13: **req** $\text{H}(\vec{p}[i]) = \text{propIDs}[i]$
14: $G' \leftarrow$ *init-epoch(G)
15: **try** $(G', upd, rem, add) \leftarrow$ *apply-props$(G, G', \vec{p})$
16: **req** $(*, \text{id}_c) \notin rem \wedge (\text{id}_c, *) \notin upd$
17: **if** $(*, \text{'rem'-id}) \in rem$ **then**
18: $G' \leftarrow \bot$
19: **return** $(\text{id}_c, upd \| rem \| add, \bot)$
20: **else**
21: $G' \leftarrow$ *set-conf-trans-hash$(G, G', \text{id}_c, C_0, \text{sig})$
22: $(G', \text{comSecret}) \leftarrow$ *apply-rekey$(G', \text{id}_c, \text{kp}_c, \text{ct}_0, \widehat{\text{ct}}_{\text{id}})$
23: $G' \leftarrow$ *set-member-hash(G')
24: $(G', \text{confKey}, \text{joinerSecret}) \leftarrow$ *derive-keys$(G, G', \text{comSecret})$
25: **req** *vrf-conf-tag$(G', \text{confKey}, \text{confTag})$
26: $G' \leftarrow$ *set-interim-trans-hash$(G', \text{confTag})$
27: **return** $(\text{id}_c, upd \| rem \| add, G'.\text{memberIDsvks}())$

Input $(\texttt{Join}, \widehat{w}^{\text{ctxt}})$

1: **req** $G = \bot$
2: $(w_0, \widehat{w}) \leftarrow$ *dec-welcome$(\widehat{w}^{\text{ctxt}})$
3: **parse** $(\text{ct}_0, \text{groupInfo}, \text{sig}) \leftarrow w_0$
4: **parse** $(\text{id}', \text{kphash}, \widehat{\text{ct}}_{\text{id}'}) \leftarrow \widehat{w}$
5: **req** $\text{id} = \text{id}'$
6: **try** $(G, \text{confTag}, \text{id}_c) \leftarrow$ *initialize-group$(G, \text{id}, \text{groupInfo})$
7: **req** $G.\text{confTransHash} = \text{H}(G.\text{confTransHash-w.o-'id}_c\text{'}, \text{id}_c)$
8: **req** $G.\text{interimTransHash} = \text{H}(G.\text{confTransHash}, \text{confTag})$
9: **req** $\text{SIG.Verify}(G.\text{member}[\text{id}_c].\text{svk}, \text{sig}, (\text{ct}_0, \widehat{\text{ct}}, \text{groupInfo}))$
10: **try** $G \leftarrow$ *vrf-group-state(G)
11: $G.\text{id} \leftarrow \text{id}$
12: $\text{svk} \leftarrow G.\text{member}[\text{id}].\text{svk}$
13: Send $(\texttt{get-ssk}, \text{svk})$ to $\mathcal{F}_{\text{AS}}$ and receive ssk
14: $G.\text{ssk} \leftarrow \text{ssk}$
15: Send $(\texttt{get-dks})$ to $\mathcal{F}_{\text{KS}}$ and receive kbs
16: $\text{joinerSecret} \leftarrow \bot$
17: **foreach** $(\text{kp}, \text{dk}) \in \text{kbs}$ **do**
18: **if** $\text{H}(\text{kp}) = \text{kphash}$ **then**
19: **req** $G.\text{member}[\text{id}].\text{kp}() = \text{kp}$
20: $G.\text{member}[\text{id}].\text{dk} \leftarrow \text{dk}$
21: $\text{joinerSecret} \leftarrow \text{mDec}(\text{dk}, \text{ct}_0, \widehat{\text{ct}})$
22: **req** $\text{joinerSecret} \neq \bot$
23: $(G, \text{confKey}) \leftarrow$ *derive-epoch-keys$(G, \text{joinerSecret})$
24: **req** *vrf-conf-tag$(G, \text{confKey}, \text{confTag})$
25: **return** $(\text{id}_c, G.\text{memberIDsvks}())$

Figure 23: Static metadata-hiding CGKA protocol Chained CmPKE$^{\text{ctxt}}$: `Process` and `Join`. The major changes from [37] are highlighted in yellow.

Input $(\texttt{Key})$

1: **req** $G \neq \bot$
2: $\text{k} \leftarrow G.\text{appSecret}$
3: **return** k

Input $(\texttt{Key}_{\text{mh}}, c_0)$

1: **req** $G \neq \bot$
2: $\text{k}_{\text{mh}} \leftarrow G.\text{metaKey}$
3: **return** k_{mh}

Input $(\texttt{NextKey}_{\text{mh}}, c_0)$

1: / Returns the pending k_{mh}'
2: / for the next epoch
3: **parse** $(G', *) \leftarrow G.\text{pendCom}[c_0]$
4: $\text{k}_{\text{mh}}' \leftarrow G'.\text{metaKey}$
5: **return** k_{mh}'

Figure 24: Static metadata-hiding CGKA protocol Chained CmPKE$^{\text{ctxt}}$: Retrieve group secret key. The major changes from [37] are highlighted in yellow. Note that $\texttt{Key}_{\text{mh}}$ and $\texttt{NextKey}_{\text{mh}}$ are used in the higher level metadata-hiding protocol.

*fetch-ssk-if-nec(G, svk$'$)

svk $\leftarrow$ G.member[G.id].svk
if svk $\neq$ svk$'$ **then**
  Send (get-ssk, svk) to $\mathcal{F}_{\text{AS}}$ and
    receive ssk
else
  ssk $\leftarrow$ G.ssk
return ssk

*validate-kp(G, kp, id)

parse (id$'$, ek, svk, sig) $\leftarrow$ kp
req id = id$'$
if svk $\notin$ G.certSvks[id] **then**
  Send (verify-cert, id$'$, svk) to $\mathcal{F}_{\text{AS}}$
    and receive *succ*
  req *succ*
  G.certSvks[id] $+\!\!\leftarrow$ svk
req SIG.Verify(pp$_{\text{SIG}}$, svk, sig, (id, ek, svk))
return G

*assign-kp(G, kp)

parse (id, ek, svk, sig) $\leftarrow$ kp
G.member[id].ek $\leftarrow$ ek
G.member[id].svk $\leftarrow$ svk
G.member[id].sig $\leftarrow$ sig
return G

Figure 25: Helper functions: Key material related.

*init-epoch(G)

G' $\leftarrow$ G.clone()
G'.epoch $\leftarrow$ G.epoch + 1
G'.pendUpd[$*$], G'.pendCom[$*$] $\leftarrow$ $\perp$
return G'

*rekey(G', receivers, id, svk)

try ssk $\leftarrow$ *fetch-ssk-if-nec(G', svk)
(kp, dk) $\leftarrow$ genKP(id, svk, ssk)
G' $\leftarrow$ *assign-kp(G', kp)
G'.ssk $\leftarrow$ ssk
G'.member[id].dk $\leftarrow$ dk
comSecret $\leftarrow\!\!\$$ $\{0,1\}^\kappa$
$\vec{\text{ek}}$ $\leftarrow$ (G.member[id$'$].ek)$_{\text{id}' \in \text{receivers}}$
$(\text{ct}_0, \vec{\text{ct}} = (\widehat{\text{ct}}_{\text{id}'})_{\text{id}' \in \text{receivers}}) \leftarrow$ mEnc(pp$_{\text{mPKE}}$, $\vec{\text{ek}}$, comSecret)
return (G', comSecret, kp, ct$_0$, $\vec{\text{ct}}$)

*apply-rekey(G', id$_c$, kp$_c$, ct$_0$, $\widehat{\text{ct}}$)

dk $\leftarrow$ G'.member[G'.id].dk
comSecret $\leftarrow$ mDec(dk, ct$_0$, $\widehat{\text{ct}}$)
try G' $\leftarrow$ *validate-kp(G', kp$_c$, id$_c$)
G' $\leftarrow$ *assign-kp(G', kp$_c$)
return (G', comSecret)

*apply-props(G, G', $\vec{p}$)

upd, *rem*, *add* $\leftarrow$ ()
foreach $p \in \vec{p}$ **do**
  try (id$_s$, P) $\leftarrow$ *unframe-prop(G, p)
  parse (type, val) $\leftarrow$ P
  if type = 'upd' **then**
    req id$_s$ $\in$ G.memberIDs()
    req (id$_s$, $*$) $\notin$ *upd* $\wedge$ *rem* = () $\wedge$ *add* = ()
    try G' $\leftarrow$ *validate-kp(G', val, id$_s$)
    G' $\leftarrow$ *assign-kp(G', val)
    if id$_s$ = G.id **then**
      parse (ssk, dk) $\leftarrow$ G.pendUpd[p]
      G'.ssk $\leftarrow$ ssk
      G'.member[G.id].dk $\leftarrow$ dk
    svk $\leftarrow$ G'.member[id$_s$].svk
    upd $+\!\!\leftarrow$ (id$_s$, 'upd'-svk)
  elseif type = 'rem' **then**
    parse id$_t$ $\leftarrow$ val
    req id$_t$ $\neq$ id$_s$ $\wedge$ id$_t$ $\in$ G.memberIDs()
    req (id$_t$, $*$) $\notin$ *upd* $\wedge$ *add* = ()
    G'.member[id$_t$] $\leftarrow$ $\perp$
    rem $+\!\!\leftarrow$ (id$_s$, 'rem'-id$_t$)
  elseif type = 'add' **then**
    (id$_t$, $*$, $*$, $*$) $\leftarrow$ val
    req id$_t$ $\notin$ G.memberIDs()
    try G' $\leftarrow$ *validate-kp(G', val, id$_t$)
    G' $\leftarrow$ *assign-kp(G', val)
    add $+\!\!\leftarrow$ (id$_s$, 'add'-val)
  else
    return $\perp$
return (G', *upd*, *rem*, *add*)

Figure 26: Helper functions: `Commit` and `Process` related.

*welcome-msg(G', addedMem, joinerSecret, confTag)

$\vec{ek} \leftarrow (G'.\text{member}[\text{id}_t].\text{ek})_{\text{id}_t \in \text{addedMem}}$ **do**
$(\text{ct}_0, \vec{ct} = (\widehat{ct}_{\text{id}_t})_{\text{id}_t \in \text{addedMem}}) \leftarrow \text{mEnc}(\text{pp}_{\text{mPKE}}, \vec{ek}, \text{joinerSecret})$
groupInfo $\leftarrow (G'.\text{gid}, G'.\text{epoch},$
  $G'.\text{memberPublicInfo}(), G'.\text{memberHash},$
  $G'.\text{confTransHash-w.o-'id}_c\text{'}, G'.\text{confTransHash},$
  $G'.\text{interimTransHash}, \text{confTag}, G'.\text{id})$
$\text{sig} \leftarrow \text{SIG.Sign}(\text{pp}_{\text{SIG}}, G'.\text{ssk}, (\text{ct}_0, \text{groupInfo}))$
$w_0 \leftarrow (\text{ct}_0, \text{groupInfo}, \text{sig})$
$\vec{w} \leftarrow \emptyset$
foreach $\text{id}_t \in$ addedMem **do**
  $\text{kphash}_t \leftarrow \text{H}(G'.\text{member}[\text{id}_t].\text{kp}())$
  $\vec{w} \mathrel{+}\!\!\leftarrow \widehat{w}_{\text{id}_t} = (\text{id}_t, \text{kphash}_t, \widehat{ct}_{\text{id}_t})$
return $(G', w_0, \vec{w})$

*initialize-group(G, id, groupInfo)

parse (gid, epoch, member, memberHash,
  confTransHash-w.o-'id_c', confTransHash,
  interimTransHash, confTag, id_c) $\leftarrow$ groupInfo
$(G.\text{gid}, G.\text{epoch}, G.\text{member}, G.\text{memberHash},$
  $G.\text{confTransHash-w.o-'id}_c\text{'}, G.\text{confTransHash},$
  $G.\text{interimTransHash}) \leftarrow (\text{gid}, \text{epoch}, \text{member},$
  memberHash, confTransHash-w.o-'id_c', confTransHash, interimTransHash)
$G.\text{certSvks}[*] \leftarrow \emptyset$
$G.\text{pendUpd}[*] \leftarrow \perp; G.\text{pendCom}[*] \leftarrow \perp$
$G.\text{id} \leftarrow \text{id}$
return $(G, \text{confTag}, \text{id}_c)$

*vrf-group-state(G)

req $G.\text{memberHash} =$ *derive-member-hash(G)
mem $\leftarrow G.\text{memberIDs}()$
foreach id $\in$ mem **do**
  kp $\leftarrow G.\text{member}[\text{id}].\text{kp}()$
  try $G \leftarrow$ *validate-kp(G, kp, id)
return G

Figure 27: Helper functions: Join related.

*gen-conf-tag(G, confKey)

return MAC.TagGen(confKey, G.confTransHash)

*vrf-conf-tag(G, confKey, confTag)

return MAC.TagVerify(confKey, confTag, G.confTransHash)

Figure 28: Helper function: Confirmation tag.

*set-member-hash(G)

$G.\text{memberHash} \leftarrow$ *derive-member-hash(G)
return G

*derive-member-hash(G)

KP $\leftarrow ()$; mem $\leftarrow G.\text{memberIDs}()$ / mem is sorted by dictionary order
foreach id $\in$ mem **do**
  KP $\mathrel{\#}\!\!\leftarrow G.\text{member}[\text{id}].\text{kp}()$
return H(KP)

*set-conf-trans-hash($G, G', \text{id}_c, C_0, \text{sig}$)

comCont $\leftarrow (G.\text{gid}, G.\text{epoch}, \text{'commit'}, C_0, \text{sig})$
$G'.\text{confTransHash-w.o-'id}_c\text{'} \leftarrow \text{H}(G.\text{interimTransHash}, \text{comCont})$
$G'.\text{confTransHash} \leftarrow \text{H}(G'.\text{confTransHash-w.o-'id}_c\text{'}, \text{id}_c)$
return G'

*set-interim-trans-hash(G', confTag)

$G'.\text{interimTransHash} \leftarrow \text{H}(G'.\text{confTransHash}, \text{confTag})$
return G'

Figure 29: Helper function: Member hash and transcript hash.

*derive-keys(G, G', comSecret)

$s \leftarrow \text{HKDF.Extract}(G.\text{initSecret}, \text{comSecret})$
joinerSecret $\leftarrow \text{HKDF.Expand}(s, \text{'joi'})$
$(G', \text{confKey}) \leftarrow$ *derive-epoch-keys(G', joinerSecret)
return $(G', \text{confKey}, \text{joinerSecret})$

*derive-epoch-keys(G', joinerSecret)

confKey $\leftarrow \text{HKDF.Expand}(\text{joinerSecret}, G'.\text{groupCont}() \| \text{'conf'})$
$G'.\text{appSecret} \leftarrow \text{HKDF.Expand}(\text{joinerSecret}, G'.\text{groupCont}() \| \text{'app'})$
$G'.\text{membKey} \leftarrow \text{HKDF.Expand}(\text{joinerSecret}, G'.\text{groupCont}() \| \text{'memb'})$
$G'.\text{encSecret} \leftarrow \text{HKDF.Expand}(\text{joinerSecret}, G'.\text{groupCont}() \| \text{'enc'})$
$G'.\text{metaKey} \leftarrow \text{HKDF.Expand}(\text{joinerSecret}, G'.\text{groupCont}() \| \text{'meta'})$
$G'.\text{initSecret} \leftarrow \text{HKDF.Expand}(\text{joinerSecret}, G'.\text{groupCont}() \| \text{'init'})$
return $(G', \text{confKey})$

Figure 30: Helper function: Key scheduling. The major changes from [37] are highlighted in yellow.

*frame-prop(G, P)

1 : propCont $\leftarrow$ (G.groupCont(), G.id, 'proposal', P)
2 : sig $\leftarrow$ SIG.Sign(pp_{SIG}, G.ssk, propCont)
3 : membTag $\leftarrow$ MAC.TagGen(G.membKey, (propCont, sig))
4 : $\overline{p} \leftarrow$ (G.id, P, sig, membTag)
5 : **return** (G.gid, G.epoch, 'proposal', $\overline{p}$)

*unframe-prop(G, p)

1 : **parse** (gid, epoch, contType, $\overline{p}$) $\leftarrow p$
2 : **parse** (id_s, P, sig, membTag) $\leftarrow \overline{p}$
3 : **req** contType = 'proposal' $\wedge$ gid = G.gid $\wedge$ epoch = G.epoch
4 : propCont $\leftarrow$ (G.groupCont(), id_s, 'proposal', P)
5 : **req** G.member[id_s] $\neq \perp$
$\wedge$ SIG.Verify(pp_{SIG}, G.member[id_s].svk, sig, propCont)
$\wedge$ MAC.TagVerify(G.membKey, membTag, (propCont, sig))
6 : **return** (id_s, P)

*sign-commit(G, C_0)

1 : comCont $\leftarrow$ (G.groupCont(), G.id, 'commit', C_0)
2 : sig $\leftarrow$ SIG.Sign(pp_{SIG}, G.ssk, comCont)
3 : **return** sig

*frame-commit$(G, C_0, \text{sig}, \text{confTag})$

1 : $\overline{c}_0 \leftarrow$ (G.id, C_0, sig, confTag)
2 : **return** (G.gid, G.epoch, 'commit', $\overline{c}_0$)

*unframe-commit(G, c_0)

1 : **parse** (G.gid, G.epoch, 'commit', $\text{CT}_{\overline{c}_0}$) $\leftarrow c_0$
2 : **parse** (id_c, C_0, sig, confTag) $\leftarrow \overline{c}_0$
3 : **req** contType = 'commit' $\wedge$ gid = G.gid $\wedge$ epoch = G.epoch
4 : comCont $\leftarrow$ (G.groupCont(), id_c, 'commit', C_0)
5 : $\text{svk}_c \leftarrow G$.member[$\text{id}_c$].svk
6 : **req** G.member[id_c] $\neq \perp \wedge$ SIG.Verify(pp_{SIG}, svk_c, sig, comCont)
7 : **return** (id_c, C_0, sig, confTag)

Figure 31: Helper function: Frame and unframe packets.

*enc-prop(encSecret, p)

1 : **parse** (gid, epoch, 'proposal', $\overline{p}$) $\leftarrow p$
2 : $\text{CT}_{\overline{p}} \leftarrow$ SKE.Enc(encSecret, $\overline{p}$)
3 : **return** $p^{\text{ctxt}} :=$ (gid, epoch, 'proposal', $\text{CT}_{\overline{p}}$)

*dec-prop(encSecret, p^{ctxt})

1 : **parse** (gid, epoch, 'proposal', $\text{CT}_{\overline{p}}$) $\leftarrow p^{\text{ctxt}}$
2 : $\overline{p} \leftarrow$ SKE.Dec(encSecret, $\text{CT}_{\overline{p}}$)
3 : **return** $p :=$ (gid, epoch, 'proposal', $\overline{p}$)

*enc-commit(encSecret, $c_0, \vec{c}$)

1 : **parse** (gid, epoch, 'commit', $\overline{c}_0$) $\leftarrow c_0$
2 : $\text{CT}_{\overline{c}_0} \leftarrow$ SKE.Enc(encSecret, $\overline{c}_0$)
3 : $\vec{c}^{\text{ctxt}} \leftarrow \emptyset$
4 : **foreach** $\widehat{c} \in \vec{c}$ **do**
5 : $\vec{c}^{\text{ctxt}} \mathrel{+}\!\!\leftarrow$ SKE.Enc(encSecret, $\widehat{c}$)
6 : $c_0^{\text{ctxt}} \leftarrow$ (G.gid, G.epoch, 'commit', $\text{CT}_{\overline{c}_0}$)
7 : **return** ($c_0^{\text{ctxt}}, \vec{c}^{\text{ctxt}}$)

*dec-commit(encSecret, $c_0^{\text{ctxt}}, \widehat{c}^{\text{ctxt}}$)

1 : **parse** (gid, epoch, 'commit', $\text{CT}_{\overline{c}_0}$) $\leftarrow c_0^{\text{ctxt}}$
2 : $\overline{c}_0 \leftarrow$ SKE.Dec(encSecret, $\text{CT}_{\overline{c}_0}$)
3 : $\widehat{c} \leftarrow$ SKE.Dec(encSecret, $\widehat{c}^{\text{ctxt}}$)
4 : $c_0 \leftarrow$ (gid, epoch, 'commit', $\overline{c}_0$)
5 : **return** ($c_0, \widehat{c}$)

*dec-and-sort-proposals(encSecret, $\vec{p}^{\text{ctxt}}$)

1 : $\vec{p}, \vec{p}_{\text{'rem'}}, \vec{p}_{\text{'upd'}}, \vec{p}_{\text{'add'}} \leftarrow ()$
2 : **foreach** $p^{\text{ctxt}} \in \vec{p}^{\text{ctxt}}$ **do**
3 : **try** $p \leftarrow$ *dec-prop(encSecret, p^{ctxt})
4 : **try** type $\leftarrow$ *extract-proposal-type(p)
5 : **if** type = 'upd' **then**
6 : $\vec{p}_{\text{'upd'}} \mathrel{+}\!\!\leftarrow p$
7 : **elseif** type = 'rem' **then**
8 : $\vec{p}_{\text{'rem'}} \mathrel{+}\!\!\leftarrow p$
9 : **elseif** type = 'add' **then**
10 : $\vec{p}_{\text{'add'}} \mathrel{+}\!\!\leftarrow p$
11 : **else return** $\perp$
12 : / Return sorted proposal list
13 : **return** $\vec{p}_{\text{'upd'}} \| \vec{p}_{\text{'rem'}} \| \vec{p}_{\text{'add'}}$

*extract-proposal-type(p)

1 : **parse** (gid, epoch, contType, $\overline{p}$) $\leftarrow p$
2 : **parse** (id_s, P, sig, membTag) $\leftarrow \overline{p}$
3 : **parse** (type, val) $\leftarrow P$
4 : **return** type

Figure 32: Helper functions unique to Chained CmPKE$^{\text{ctxt}}$: Encrypt and decrypt proposals and commits. The major changes from [37] are highlighted in yellow. *sort-proposals decrypts proposals and sort them following the order of MLS specification [14, Sec. 13.2.2].

*enc-welcome(welcomeSecret, w_0, $\widehat{w}$)

1: **parse** $(\mathsf{id}_t, \mathsf{kphash}_t, \widehat{\mathsf{ct}}) \leftarrow w_0$
2: **parse** $(\mathsf{ct}_0, \mathsf{groupInfo}, \mathsf{sig}) \leftarrow \widehat{w}$
3: $\mathsf{welcomeSecret}_{\mathsf{id}_t} \leftarrow \mathsf{HKDF.Expand}(\mathsf{welcomeSecret}, \mathsf{id}_t)$
4: $\mathsf{CT} \leftarrow \mathsf{SKE.Enc}(\mathsf{welcomeSecret}_{\mathsf{id}_t}, (\mathsf{groupInfo}, \mathsf{sig}))$
5: **return** $\widehat{w}^{\mathsf{ctxt}} := (\mathsf{id}_t, (\mathsf{kphash}_t, \mathsf{ct}_0, \widehat{\mathsf{ct}}), \mathsf{CT})$

*dec-welcome($\widehat{w}^{\mathsf{ctxt}}$)

1: **parse** $(\mathsf{id}_t, (\mathsf{kphash}, \mathsf{ct}_0, \widehat{\mathsf{ct}}), \mathsf{CT}) \leftarrow \widehat{w}^{\mathsf{ctxt}}$
2: Send (get-dks) to $\mathcal{F}_{\mathsf{KS}}$ and receive kbs
3: $\mathsf{joinerSecret}, \mathsf{kp}_{\mathsf{id}}, \mathsf{dk}_{\mathsf{id}} \leftarrow \perp$
4: **foreach** $(\mathsf{kp}, \mathsf{dk}) \in \mathsf{kbs}$ **do**
5: **if** $\mathsf{H}(\mathsf{kp}) = \mathsf{kphash}$ **then**
6: $(\mathsf{kp}_{\mathsf{id}}, \mathsf{dk}_{\mathsf{id}}) \leftarrow (\mathsf{kp}, \mathsf{dk})$
7: $\mathsf{joinerSecret} \leftarrow \mathsf{mDec}(\mathsf{dk}, \mathsf{ct}_0, \widehat{\mathsf{ct}})$
8: **break**
9: **req** $\mathsf{joinerSecret} \neq \perp$
10: $\mathsf{welcomeSecret} \leftarrow \mathsf{HKDF.Expand}(\mathsf{joinerSecret}, \text{'wel'})$
11: $\mathsf{welcomeSecret}_{\mathsf{id}_t} \leftarrow \mathsf{HKDF.Expand}(\mathsf{welcomeSecret}, \mathsf{id}_t)$
12: $(\mathsf{groupInfo}, \mathsf{sig}) \leftarrow \mathsf{SKE.Dec}(\mathsf{welcomeSecret}_{\mathsf{id}_t}, \mathsf{CT})$
13: $w_0 \leftarrow (\mathsf{ct}_0, \mathsf{groupInfo}, \mathsf{sig})$
14: $\widehat{w} \leftarrow (\mathsf{id}_t, \mathsf{kphash}_t, \widehat{\mathsf{ct}})$
15: **return** $(w_0, \widehat{w})$

Figure 33: Helper functions unique to Chained CmPKE$^{\mathsf{ctxt}}$: Encrypt and decrypt welcome messages. The major changes from [37] are highlighted in yellow.

C.2 Safety Predicates and Leakage Functions

As explained in App. B.2, to formally prove that Chained CmPKE$^{\mathsf{ctxt}}$ UC-realizes the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$, we must first specify the safety predicates **safe**, **mac-inj-allowed**, and **sig-inj-allowed**, and the leakage functions ***leak-create**, ***leak-prop**, ***leak-com**, ***leak-wel**, and ***leak-proc**.

These are formally defined in Figs. 34 and 35. The definition of the safety predicates **safe**, **mac-inj-allowed** and **sig-inj-allowed** is identical to those considered by Chained CmPKE [37, Fig. 28], modulo the syntactical difference in the definition of node pointers (see Sec. 3.2).

The leakage functions ***leak-create**, ***leak-prop**, ***leak-com**, ***leak-wel**, and ***leak-proc** are new to the definition of $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ and dictate the amount of static metadata leaked to the server. Since the concrete leakage information depends on the concrete choice of CGKA, our choice only captures the leakage of Chained CmPKE$^{\mathsf{ctxt}}$. Namely, other protocols such as the ciphertext variant of TreeKEM used in MLSCiphertext may require a slightly more complex leakage function. We provide some discussion in Sec. 7. Below, we explain the specific choice of our leakage functions in more detail.

***leak-create**(id, svk): This defines the information leaked when a new group is created. In the previous (non-metadata-hiding) Chained CmPKE, the simulator was given the identity of the party id creating the group and its corresponding signature verification key svk to simulate the group identifier gid. In other words, gid leaked information on (id, svk). In Chained CmPKE$^{\mathsf{ctxt}}$, the simulator is asked to simulate gid without giving any other information. This models the fact that when a new group is created, the server cannot guess who created it. As mentioned in the introduction, this could potentially open the door to a DoS attack on the server. One way to solve this would be to use standard anonymous credentials [26]. We leave such direction of research as an important future work.

***leak-prop**(id, act): This defines the information that is leaked from proposals p. In Chained CmPKE, the simulator was given the identity of the party id creating the proposal, the commit node Ptr[id] that locates id in the history graph, and the action act of the proposal to simulate the proposal p. Here, when there is no fork in the main group, then Ptr[id] essentially pinpoints the specific epoch of the group. In case there is a fork, then epoch alone is not sufficient to identify where party id is located, in which case we need the exact proposal node Ptr[id]. In other words, proposal p leaked the proposing party id and the action act, where note that p must always include Ptr[id] (which is (gid, epoch) essentially) for the server to deliver p to the appropriate member in a specific epoch.

In Chained CmPKE$^{\mathsf{ctxt}}$, we define ***leak-prop**(id, act) to only output Ptr[id] and the length of |id| and |act|. This models the fact that proposals p only leak the specific group by which p is supposed to be processed. Moreover, in case the size of each action act is different, this leaks the action type. However, considering that the addition or removal of a group member is noticeable from the server (since it changes the size of commit messages), hiding the type of action may not add as much security as one expects.

***leak-com**(id, $\vec{p}$, svk): This defines the information that is leaked from commits $(c_0, \vec{c} = (\widehat{c}_{\mathsf{id}'})_{\mathsf{id}'})$. In Chained CmPKE, the simulator was given the identity of the party id creating the commit; the commit node Ptr[id] that locates id in the history graph; the list of proposals $\vec{p}$ that is going to be committed; the new signing key svk of id; and the list of current members mem in the group to simulate the commit $(c_0, \vec{c})$. In other words, a commit leaks the committer identity id, its signing key svk, and the current list of members mem.

In Chained CmPKE$^{\mathsf{ctxt}}$, we define ***leak-com**(id, $\vec{p}$, svk) to only output (Ptr[id], |id|, $\vec{p}$, |svk|, |mem|). Note that similarly to proposals, Ptr[id] cannot be hidden. Moreover, $\vec{p}$ is guaranteed to hide the static metadata from above, we can provide this to the simulator. Finally, the size of the current member |mem| would necessarily leak from commit messages when using Chained CmPKE since $\vec{c}$ scales linearly with the group size. While we could apply some padding to hide the group size from the commit, the server can infer the group size when selective downloading is performed; if N distinct $\widehat{c}$ were downloaded out of $M \geq N$ commits (that includes the padded garbage commits), then the server can guess that the group size was N.

***leak-wel**(node-id$_{\mathsf{cur}}$, node-id$_{\mathsf{next}}$, id$_t$): This defines the information that is leaked from welcome messages $\widehat{w}$. In Chained CmPKE, the simulator was given the key package kp$_t$ of the added member id$_t$; the commit node of the next epoch node-id$_{\mathsf{next}}$; and all the information stored on the commit node Node[node-id$_{\mathsf{next}}$] to simulate the welcome message $\widehat{w}$. In other words, welcome messages leak the group identifier gid, the next epoch, the identity id of the party who created the welcome message, the size of the member of the next epoch, and so on.

In Chained CmPKE$^{\mathsf{ctxt}}$, we define ***leak-wel**(node-id$_{\mathsf{cur}}$, node-id$_{\mathsf{next}}$, id$_t$) to only output (kp$_t$, |gid|, |epoch|, |id|, |mem|). Here, we include kp$_t$ in clear since Chained CmPKE$^{\mathsf{ctxt}}$ (and MLSCiphertext) includes the hash of the key package of id$_t$ in the welcome message in the clear. Namely, in case id$_t$ creates many key packages, the simulator must know which kp$_t$ is used to simulate the welcome message.

We note that the reason why our protocol and MLSCiphertext includes a hash kphash$_t$ of kp$_t$ in the welcome message is for efficiency. The added party, which may have created many key packages for different groups, can check which key package to use by simply finding the key package that equals the hash value kphash$_t$. Otherwise, the party must go through all the decryption keys associated with each key package it has and try to see whether the welcome message can be decrypted correctly. We also note that id$_t$ is always leaked from the welcome message since otherwise, the server will not know the destination of the welcome message, i.e., it cannot deliver it.

***leak-proc**(id): This defines the information that is leaked from a party-dependent commit $(c_0, \widehat{c})$ and a list of proposals $\vec{p}$. In Chained CmPKE, the simulator was given the identity of the party id processing the commit and the commit node Ptr[id] that locates id in the history graph to decide whether $(c_0, \widehat{c}, \vec{p})$ should be processed correctly by a party id.

In Chained CmPKE$^{\mathsf{ctxt}}$, we define ***leak-proc**(id) to only output (Ptr[id], index$_{\mathsf{id}}$). As explained above, Ptr[id] cannot be hidden

since this information is required for id to retrieve the uploaded commit and proposals from the server. Moreover, since selective downloading is performed, $\widehat{c}$ must leak the position of the party id in the group (assuming that id can process $(c_0, \widehat{c})$ correctly).

Knowledge of party's secrets.

know(node-id, id) $\iff$
(a) id $\in$ Node[node-id].exp$\vee$
(b) ***secrets-injected**(node-id, id)$\vee$
(c) (Node[node-id].par $\neq \perp \wedge$ **know**(Node[node-id].par, id))
$\wedge \neg{}^*$**secrets-replaced**(node-id, id)$\vee$
(d) $\exists$node-id$'$: (Node[node-id$'$].par = node-id $\wedge$ **know**(node-id$'$, id)
$\wedge \neg{}^*$**secrets-replaced**(node-id$'$, id))

***secrets-injected**(node-id, id) $\iff$
(a) (Node[node-id].orig = id $\wedge$ Node[node-id].stat $\neq$ 'good')$\vee$
(b) $\exists p \in$ Node[node-id].prop with prop-id := PropID[p] :
(Prop[prop-id].act = 'upd'-$*\wedge$Prop[prop-id].orig = id
$\wedge$ Prop[prop-id].stat $\neq$ 'good')$\vee$
(c) $\exists p \in$ Node[node-id].prop with prop-id := PropID[p] :
(Prop[prop-id].act = 'add'-id-$*$-svk-$*\wedge$svk $\in$ ExposedSvk)

***secrets-replaced**(node-id, id) $\iff$
Node[node-id].orig = id$\vee$
$\exists p \in$ Node[node-id].prop with prop-id := PropID[p] :
Prop[prop-id].act $\in$ { 'add'-id-$*$, 'rem'-id }$\vee$
$\exists p \in$ Node[node-id].prop with prop-id := PropID[p] :
(Prop[prop-id].act = 'upd'-$*\wedge$Prop[prop-id].orig = id)

Knowledge of epoch secrets.

know(node-id, 'epoch') $\iff$ Node[node-id].exp $\neq \emptyset \vee$ ***can-traverse**(node-id)

***can-traverse**(node-id) $\iff$
(a) $\exists p \in$ Node[node-id].prop :
(Prop[PropID[p]].act = 'add'-id-$*$-svk-$*\wedge$svk $\in$ ExposedSvk)
(b) ***reused-welcome-key-leaks**(node-id)$\vee$
(c) Node[node-id].stat = 'bad'
$\wedge \exists p \in$ Node[node-id].prop : Prop[PropID[p]].act = 'add'-$*\vee$
(d) (Node[node-id].par = $\perp \vee$ **know**(Node[node-id].par, 'epoch'))$\wedge$
$\exists$(id, $*$) $\in$ Node[node-id].mem : **know**(node-id, id)

***reused-welcome-key-leaks**(node-id) $\iff$
$\exists$id, $p \in$ Node[node-id].prop : Prop[p].act = 'add'-id-$*\wedge$
$\exists$node-id$_d$: node-id$_d$ is a descendant of node-id $\wedge$ id $\in$ Node[node-id$_d$].exp$\wedge$
no node node-id$_h$ exists on node-id–node-id$_d$ path
s.t. ***secrets-replaced**(node-id$_h$, id) = true

Safe and can-inject.

safe(node-id) $\iff$ **know**(node-id, 'epoch')
sig-inj-allowed(node-id, id) $\iff$ Node[node-id].mem[id] $\in$ ExposedSvk
mac-inj-allowed(node-id) $\iff$ **know**(node-id, 'epoch')

Figure 34: The safety predicate for Chained CmPKE$^{\text{ctxt}}$. This is identical to [37, Fig. 28] modulo the syntactical difference in the definition of node pointers.

***leak-create**(id, svk)
1 : **if** flag$_{\text{contHide}}$ **then**
2 : **return** $\perp$
3 : **else**
4 : **return** (id, svk)

***leak-prop**(id, act)
1 : **if** flag$_{\text{contHide}} \wedge$ **safe**(Ptr[id]) **then**
2 : **return** (Ptr[id], |id|, |act|)
3 : **else**
4 : **return** (Ptr[id], id, act)

***leak-com**(id, $\vec{p}$, svk)
1 : mem $\leftarrow$ Node[Ptr[id]].mem
2 : **if** flag$_{\text{contHide}} \wedge$ **safe**(Ptr[id]) **then**
3 : **return** (Ptr[id], |id|, $\vec{p}$, |svk|, |mem|)
4 : **else**
5 : **return** (Ptr[id], id, $\vec{p}$, svk, mem)

***leak-proc**(id)
1 : index$_{\text{id}}$ $\leftarrow$ Node[Ptr[id]].indexOf(id)
2 : **if** flag$_{\text{contHide}} \wedge$ **safe**(Ptr[id]) **then**
3 : **return** (Ptr[id], index$_{\text{id}}$)
4 : **else**
5 : **return** (Ptr[id], id)

***leak-wel**(node-id$_{\text{cur}}$, node-id$_{\text{next}}$, id$_t$)
1 : gid $\leftarrow$ Node[node-id$_{\text{next}}$].gid
2 : epoch $\leftarrow$ Node[node-id$_{\text{next}}$].epoch
3 : id$_c$ $\leftarrow$ Node[node-id$_{\text{next}}$].orig
4 : mem $\leftarrow$ Node[node-id$_{\text{next}}$].mem
5 : **if** flag$_{\text{contHide}} \wedge$ **safe**(node-id$_{\text{next}}$) **then**
6 : **return** (kp$_t$, |gid|, |epoch|, |id$_c$|, |mem|)
7 : **else**
8 : **return** (kp$_t$, node-id$_{\text{next}}$, Node[node-id$_{\text{next}}$])

Figure 35: Leakage functions for Chained CmPKE$^{\text{ctxt}}$.

C.3 Security of Chained CmPKE$^{\mathsf{ctxt}}$

The following theorem establishes that Chained CmPKE$^{\mathsf{ctxt}}$ UC-realizes the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

Theorem C.1. *Assuming that* mPKE *is* IND-CCA *secure,* SIG *is* sEUF-CMA *secure, and* SKE *is* IND-CCA *secure and has key-committing property, our CGKA protocol Chained CmPKE*$^{\mathsf{ctxt}}$ *adaptively and securely realizes the ideal functionality* $\mathcal{F}_{CGKA}^{\mathsf{ctxt}}$ *in the* $(\mathcal{F}_{AS}, \mathcal{F}_{KS}, \mathcal{G}_{RO})$*-hybrid model.*

Here, $\mathcal{F}_{CGKA}^{\mathsf{ctxt}}$ *is defined with respect to the safety predicates and leakage functions in Figs. 34 and 35. Moreover, calls to the hash function* H*,* HKDF*, and* MAC *are replaced by calls to the global random oracle* $\mathcal{G}_{RO}$*.*

C.3.1 Proof Overview. The proof consists of a sequence of hybrids where we gradually modify the protocol Chained CmPKE$^{\mathsf{ctxt}}$ into the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$. As explained in Sec. 3.2, with some work, we can reuse a great part of the proof by Hashimoto et al. [36].

The key observation is that when the leakage functions ***leak-create**, ***leak-prop**, ***leak-com**, ***leak-wel** and ***leak-proc** are defined to leak all the static metadata (i.e., $\mathtt{flag}_{\mathsf{contHide}} = \mathtt{false}$), then $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ is *almost* identical to the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$ that does not hide the static metadata. The only difference is that the proposals and commits in $\mathcal{F}_{\mathrm{CGKA}}$ leak the static metadata, while $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ encrypts them. To this end, using the fact that the simulator $\mathcal{S}$ knows the encryption key (which follows from $\mathtt{flag}_{\mathsf{contHide}} = \mathtt{false}$), the simulator $\mathcal{S}$ for $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ will internally run the simulator $\widetilde{\mathcal{S}}$ for $\mathcal{F}_{\mathrm{CGKA}}$ defined by Hashimoto et al. [36].

In a bit more detail, our proof goes through the same Hybrids 1 to 6 as in [36, Theorem E.1]. At Hybrid 6, the application keys for the nodes with **safe** = `true` will become indistinguishable from random — this is the place where [36, Theorem E.1] ends. We then continue to complete the proof by further adding a new Hybrid 7, where $\mathtt{flag}_{\mathsf{contHide}}$ is switched to `true`. This hybrid is part of our security proof that takes care of the static metadata. In each Hybrid i for $i \in [6]$, we define the simulator $\mathcal{S}_i$ that internally runs $\widetilde{\mathcal{S}}_i$ defined in [36, Theorem E.1]. Informally, whenever $\widetilde{\mathcal{S}}_i$ takes as input a (non-static metadata-hiding) proposal, commit, or a welcome message, $\mathcal{S}_i$ first decrypts the (static metadata-hiding) proposal, commit, and welcome message before feeding it into $\widetilde{\mathcal{S}}_i$. On the other hand, if $\widetilde{\mathcal{S}}_i$ outputs a (non-static metadata-hiding) content, then $\mathcal{S}_i$ encrypts them before outputting them to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ or the environment $\mathcal{Z}$. Here, since $\mathtt{flag}_{\mathsf{contHide}} = \mathtt{false}$, we can assume $\mathcal{S}_i$ knows all the encryption/decryption key to perform the above procedure.

While the high-level idea of piggybacking on the prior proof of Hashimoto et al. [36] sounds straightforward, several technical issues make the above non-trivial. This non-triviality is caused by the difference of the *semantics* of the nodes in the history graph in our new ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ and the prior ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$. For instance, when the environment $\mathcal{Z}$ invokes id on input $(\mathtt{Commit}, \vec{p}, \mathsf{svk})$, the simulator in [36] (roughly) outputs $(ack, rt, c_0, \vec{c})$, where $rt \in \mathbb{N}$. Our simulator must reinterpret this and output an appropriate node-id such that the history graph defined with respect to c_0 remains consistent with the history graph defined with respect to node-id. In particular, the most non-trivial part of reusing the proof by Hashimoto et al. [36] is to check that the proof moves from Hybrid 2 to 3 in [36, Theorem E.1], which concerns the consistency of the history graph, translates to our setting.

We also note that while the proof of Hashimoto et al. [36] required an mPKE that is IND-CCA secure *with adaptive corruptions*, we only require a standard mPKE. As explained in Sec. 3.2, this is due to the added restriction on the adversary (see Fig. 14 and App. B.2), which is necessary for any natural static metadata-hiding CGKA in order not to trivially win the security game. Specifically, the exponential loss appearing in the proof of moving from Hybrid 5 to Hybrid 6 in [36] disappears by considering the restricted adversary defined in $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$. Moreover, while Hashimoto et al. [36] used a *committing* mPKE to construct Chained CmPKE, they mentioned in Footnote 5 that they are not required for the proof to go through. In this work, we thus rely on the more simpler mPKE.

Finally, we prove the indistinguishability of the real and ideal protocols assuming that the adversary *cannot* inject bad randomness. That is, RandCorr is always set to 'good' in the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$. To be more precise, our proof up till Hybrid 6 allows the adversary to inject bad randomness. We only restrict the adversary when moving to Hybrid 7 — this is where we explicitly use the group secret to perform symmetric key encryption to hide the static metadata. This restriction is also done in recent work by Alwen et al. [9], where they disallow the adversary to manipulate the randomness used for the symmetric key encryption. We leave security analysis of such types of attacks as future work.

C.3.2 Proof of Thm. C.1. We now provide the full proof of Thm. C.1.

Proof. We consider the following sequence of hybrids. While the environment $\mathcal{Z}$ interacts with Chained CmPKE$^{\mathsf{ctxt}}$ in Hybrid 1, it interacts with the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ in Hybrid 7. As explained above, Hybrids 1 to 6 are defined identically to [36], where the only difference is in the definition of the simulator $\mathcal{S}$ in each hybrid. Below, we first define all the hybrids and then explain how the simulators are defined.

Hybrid 1. This is the real world, where we make a syntactic change. We consider a simulator $\mathcal{S}_1$ that interacts with a dummy functionality $\mathcal{F}_{\mathrm{dummy}}$. $\mathcal{F}_{\mathrm{dummy}}$ sits between the environment $\mathcal{Z}$ and $\mathcal{S}_1$, and relays any message from $\mathcal{Z}$ to $\mathcal{S}_1$ without modifying them. $\mathcal{S}_1$ internally simulates the real-world parties and adversary $\mathcal{A}$ by using the messages sent from $\mathcal{F}_{\mathrm{dummy}}$. From $\mathcal{A}$'s point of view, $\mathcal{S}_1$ is the environment $\mathcal{Z}$.

Hybrid 2. This hybrid concerns the authentication service and key service. We replace $(\mathcal{F}_{\mathrm{AS}}, \mathcal{F}_{\mathrm{KS}})$ with these ideal version $(\mathcal{F}_{\mathrm{AS}}^{\mathrm{IW}}, \mathcal{F}_{\mathrm{KS}}^{\mathrm{IW}})$. Since these functions are not accessible by $\mathcal{Z}$, this modification is undetectable by $\mathcal{Z}$. Thus, the view of $\mathcal{Z}$ in Hybrid 1 and Hybrid 2 are identical.

Hybrid 3. This hybrid concerns the correctness and consistency guarantees. We replace $\mathcal{F}_{\mathrm{dummy}}$ with a variant of $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$, denoted as $\mathcal{F}_{\mathrm{CGKA},3}^{\mathsf{ctxt}}$, where **safe** (resp. **sig-inj-allowed** and **mac-inj-allowed**) always returns false (resp. true) and $\mathtt{flag}_{\mathsf{contHide}}$ is set to false. In other words, all application secrets are set by the simulator, injections are always allowed, and the confidentiality of messages is not considered. The simulator $\mathcal{S}_3$ is identical to $\mathcal{S}_2$.

Hybrid 4. This hybrid concerns the security of the signature scheme. We modify $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},3}$ to use the original **sig-inj-allowed** predicate, denoted as $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},4}$. $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},4}$ halts if a proposal, a commit, or a welcome message is injected when the sender's signing key is not exposed. The simulator $\mathcal{S}_4$ is identical to $\mathcal{S}_3$.

Hybrid 5. This hybrid concerns the security of the MAC scheme. We modify $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},4}$ to use the original **mac-inj-allowed** predicate, denoted as $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},5}$. $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},5}$ halts if a proposal or a commit are injected when the corresponding MAC key is not exposed. The simulator $\mathcal{S}_5$ is identical to $\mathcal{S}_4$.

Hybrid 6. This hybrid concerns the confidentiality of the application secrets. We modify $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},5}$ where it uses the original **safe** predicate, denoted as $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},6}$. The simulator $\mathcal{S}_6$ is identical to $\mathcal{S}_5$ except that it sets only those application secrets for which **safe** is false. (Note that this functionality roughly corresponds to the previous ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$ where the security of the static metadata is not considered.)

Hybrid 7. This hybrid concerns the confidentiality of proposal, commit and welcome messages. We modify $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},6}$ so that $\mathtt{flag}_{\mathsf{contHide}}$ is set to true, denoted as $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},7}$. The simulator $\mathcal{S}_7$ is identical to $\mathcal{S}_6$ except that it simulates the protocol executions with the leakage information defined by ***leak-create**, ***leak-prop**, ***leak-com**, ***leak-wel**, and ***leak-proc**. The functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},7}$ corresponds to the ideal functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

We show the indistinguishability of Hybrids 2 to 7 in Lems. C.2 and C.7. This completes the proof of the main theorem. □

C.3.3 From Hybrids 2 to 6: Proof of Lem. C.2. We first prove the indistinguishability of hybrid 2 to Hybrid 6.

Lemma C.2. *Hybrid 2 and Hybrid 6 are indistinguishable assuming the* IND-CCA *security of* mPKE, sEUF-CMA *security of* SIG, *and the key-committing property of* SKE.

Moreover, we assume the adversary can inject bad randomness throughout these hybrids, i.e., RandCorr *can be set to 'bad'.*

Proof. The proof consists of five parts. We first explain how the simulator simulates the protocols with the simulator defined in [36]; second, we explain how to prove the correspondence of the history graphs maintained by $\mathcal{F}_{\mathrm{CGKA},i}$ and $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},i}$, which is the goal of this proof; third, we explain the detailed description of the simulator; fourth, we prove the correspondence between the two history graphs; finally, we prove the supporting propositions to finish the proof of this lemma.

Part 1: Preparation. As explained in the overview, $\mathcal{S}_i$ internally executes the simulator $\widetilde{\mathcal{S}}_i$ presented in the proof by Hashimoto et al. [36]. To be more precise, $\mathcal{S}_i$ outsources the simulation of Chained CmPKE$^{\mathsf{ctxt}}$ to $\widetilde{\mathcal{S}}_i$ that simulates Chained CmPKE by appropriately modifying the inputs and outputs of $\tilde{\mathcal{S}}_i$.

To formalize the description of $\mathcal{S}_i$, we make one modification to $\widetilde{\mathcal{S}}_i$. Without loss of generality, while $\widetilde{\mathcal{S}}_i$ is internally simulating Chained CmPKE, we assume it executes and stores the following two secrets whenever the parties are invoked on a `Commit` or a `Join`:

- G.encSecret ← HKDF.Expand(joinerSecret, G.groupCont()‖ 'enc'), and
- welcomeSecret ← HKDF.Expand(joinerSecret, 'wel').

Here, the inputs to HKDF are states of the parties simulated by $\widetilde{\mathcal{S}}_i$. $\widetilde{\mathcal{S}}_i$ then outputs these stored secrets to $\mathcal{S}_i$. $\mathcal{S}_i$ uses these secrets to either encrypt proposal and commit messages output by $\widetilde{\mathcal{S}}_i$ or decrypt proposal, commit, and welcome messages sent by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},i}$. Note that these secret keys are never used in the original Chained CmPKE — these are encryption keys used only to secure the static metadata included in proposal, commit, and welcome messages of Chained CmPKE$^{\mathsf{ctxt}}$. It can be checked that the proof in [36] still holds even if we consider this slightly modified simulator $\widetilde{\mathcal{S}}_i$ as above since encSecret and welcomeSecret leak no information on joinerSecret in the global random oracle model. We also modify $\widetilde{\mathcal{S}}_i$ so that it outputs commit query (`Commit`, ...) and welcome query (`Welcome`, ...) separately like $\mathcal{S}_i$. That is, $\widetilde{\mathcal{S}}_i$ first receives (`Commit`, id, $\widetilde{\vec{p}}$, svk) and simulates the commit $\widetilde{c}_0$; then it receives (`Welcome`, kp_t, Ptr[id], $\widetilde{c}_0$) and simulates the welcome message, where kp_t is the receiver's key package, Ptr[id] is the committer's current node, and $\widetilde{c}_0$ is the new epoch associated with the welcome message. The above modification in the proof in [36] is without loss of generality since the information $\widetilde{\mathcal{S}}_i$ receives to simulate the welcome message corresponds exactly to the information required to invoke the simulated party id to output a welcome message.

Part 2: Goal of Proof. We are now ready to explain the description of $\mathcal{S}_i$ that internally executes $\widetilde{\mathcal{S}}_i$. Looking ahead, the main goal of the proof is to relate the history graph created within $\widetilde{\mathcal{S}}_i$ to those maintained by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},i}$. Then, since [36] proved that the history graph made within $\widetilde{\mathcal{S}}_i$ and those maintained by the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$ in Hybrid i (i.e., $\mathcal{F}_{\mathrm{CGKA},i}$) are identical, we will be able to indirectly prove that the history graph made within $\mathcal{S}_i$ and those maintained by the ideal functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},i}$ are identical as well. Indistinguishability between each adjacent hybrid then follows straightforwardly from the prior proof and the correspondence of the history graphs maintained by $\mathcal{F}_{\mathrm{CGKA},i}$ and $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},i}$.

Below, we put tilde on top, e.g., $\widetilde{\mathsf{Ptr}}[*]$, $\widetilde{\mathsf{Node}}[*]$, $\widetilde{\mathsf{Wel}}[*]$, to denote the components created by the history graph within $\widetilde{\mathcal{S}}_i$ (which is also known to $\mathcal{S}_i$). As explained above, the history graphs created within $\widetilde{\mathcal{S}}_i$ and maintained by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},i}$ are identical so we may use them interchangeably. We put tilde on proposals $\widetilde{p}$ and commits $\widetilde{c}_0$ to denote that they are *non-encrypted* variants used by $\widetilde{\mathcal{S}}_i$.

Using these terminologies, we can restate our goal as to create a one-to-one correspondence between the nodes $\widetilde{\mathsf{Node}}[\widetilde{c}]$, $\widetilde{\mathsf{Prop}}[\widetilde{p}]$, and $\widetilde{\mathsf{Wel}}[\widetilde{\vec{w}}]$ maintained by the history graph in $\mathcal{F}_{\mathrm{CGKA},i}$ and the nodes Node[node-id], Prop[prop-id], and Wel[node-id] maintained by the history graph in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},i}$. We will make the meaning of "one-to-one correspondence" more clear later, but we informally mean that the graph topology and the syntactical information stored in each node are consistent with each other. We assume $\mathcal{S}_i$ maintains lists L_p and L_c that (roughly) assign unique counters to proposal and commit nodes created within $\mathcal{S}_i$, respectively. This allows us to relate which node in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},i}$ relates to $\mathcal{F}_{\mathrm{CGKA},i}$.

Part 3: Description of Simulator $\mathcal{S}_i$. For a detailed motivation and description of $\widetilde{\mathcal{S}}_i$, we refer the readers to the original proof by Hashimoto et al. [36]. We only use their result proving that the history graphs created within $\widetilde{\mathcal{S}}_i$ and maintained by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},i}$ are identical.

Below, we explain how $\mathcal{S}_i$ answers each queries made by the ideal functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},i}$ for $i \in [3:6]$, where we omit the case $i = 2$ since it is already defined (see the definiton of Hybrids 1 and 2). Here, keep in mind that for $i \in [3:6]$ we have $\mathtt{flag}_{\mathsf{contHide}} = \mathtt{false}$, i.e., the leakage functions leak all the static metadata. Since the description of $\widetilde{\mathcal{S}}_i$ in [36] is identical for each $i \in [3:6]$, we omit the subscript i below for simplicity. In other words, we only need to check that the history graph maintained by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},3}$ has a one-to-one correspondence with $\mathcal{F}_{\mathrm{CGKA},3}$. As a result, the indistinguishability of Hybrids 2 to 6 then inherits from [36].

(1) Create query from $\mathsf{id}_{\mathsf{creator}}$. This concerns the case when $\mathcal{Z}$ queries $(\mathtt{Create}, \mathsf{svk})$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. If $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ outputs $(\mathtt{Create}, \mathsf{id}_{\mathsf{creator}}, \mathsf{svk})$ to $\mathcal{S}$, $\mathcal{S}$ invokes $\widetilde{\mathcal{S}}$ on the same input. From $\widetilde{\mathcal{S}}$'s point of view, $\mathcal{S}$ acts identically to the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$. $\widetilde{\mathcal{S}}$ simply runs the simulated party $\mathsf{id}_{\mathsf{creator}}$ on input $(\mathtt{Create}, \mathsf{svk})$.[30] $\mathcal{S}$ then samples a random group identifier $\mathsf{gid} \leftarrow\!\!\$\ \{0,1\}^\kappa$ and outputs it to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. This creates the node $\mathtt{Node}[0]$ within the history graph maintained by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. Finally, since $\widetilde{\mathcal{S}}$ creates a root node $\widetilde{\mathtt{Node}}[\mathsf{root}_0]$, $\mathcal{S}$ records $L_c[\mathsf{root}_0] \leftarrow 0$.

(2) Propose query from id. This concerns the case when $\mathcal{Z}$ queries $(\mathtt{Propose}, \mathsf{act})$ for some $\mathsf{act} \in \{$ 'upd'-svk, 'add'-id_t-kp_t, 'rem'-id_t $\}$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. If $\mathtt{Ptr}[\mathsf{id}] \neq \bot$, then $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ outputs $(\mathtt{Propose}, \text{node-id} := \mathtt{Ptr}[\mathsf{id}], \mathsf{id}, \mathsf{act})$ to $\mathcal{S}$, and $\mathcal{S}$ invokes $\widetilde{\mathcal{S}}$ on the same input. From $\widetilde{\mathcal{S}}$'s point of view, $\mathcal{S}$ acts identically to the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$. $\widetilde{\mathcal{S}}$ then runs the simulated party id on input $(\mathtt{Propose}, \mathsf{act})$. If $\mathtt{RandCorr}[\mathsf{id}] =$ 'bad' and $\mathsf{act} =$ 'upd'-svk, it asks $\mathcal{S}$ to provide the randomness to run party id. $\mathcal{S}$ then queries this request to its own adversary $\mathcal{A}$ to receive the adversarially chosen randomness. Here, recall that randomness is only used by $\widetilde{\mathcal{S}}$ to generate an update key package kp. If party id, internally simulated by $\widetilde{\mathcal{S}}$, returns $\bot$, then $\widetilde{\mathcal{S}}$ returns $(ack := \mathtt{false}, \bot)$ to $\mathcal{S}$. $\mathcal{S}$ then outputs $(ack := \mathtt{false}, \bot, \bot)$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.[31]

Otherwise, if id returns $\widetilde{p}$, then $\widetilde{\mathcal{S}}$ returns $(ack := \mathtt{true}, \widetilde{p})$. $\mathcal{S}$ then encrypts $\widetilde{p}$ as $p^{\mathsf{ctxt}} \leftarrow \mathtt{*enc\text{-}prop}(G.\mathsf{encSecret}, \widetilde{p})$ using G.encSecret of id (provided by $\widetilde{\mathcal{S}}$, see *Part 1*). It runs the encryption on the adversarially controlled randomness if $\mathtt{RandCorr}[\mathsf{id}] =$ 'bad'. There are two cases to consider.

Case 1. If $\widetilde{\mathtt{Prop}}[\widetilde{p}] = \bot$, i.e., it did not exist in the history graph maintained by $\widetilde{\mathcal{S}}$, then $\mathcal{S}$ outputs $(ack := \mathtt{true}, \bot, p^{\mathsf{ctxt}})$.

Case 2. If $\widetilde{\mathtt{Prop}}[\widetilde{p}] \neq \bot$, i.e., it exists in the history graph maintained by $\widetilde{\mathcal{S}}$, then $\mathcal{S}$ searches for any $p^{\mathsf{ctxt}\prime}$ such that $\widetilde{p} \leftarrow \mathtt{*dec\text{-}prop}(G.\mathsf{encSecret}, p^{\mathsf{ctxt}\prime})$ and $\text{prop-id} = \mathtt{PropID}[p^{\mathsf{ctxt}\prime}] \neq \bot$. It then chooses any such $p^{\mathsf{ctxt}\prime}$ and outputs $(ack := \mathtt{true}, \text{prop-id}, p^{\mathsf{ctxt}})$. By Proposition C.5, Item 1, which we prove later, such $p^{\mathsf{ctxt}\prime}$ exists and prop-id is unique regardless of the choice of $p^{\mathsf{ctxt}\prime}$.

If $\mathsf{act} =$ 'upd', $\widetilde{\mathcal{S}}$ outputs the updated key package kp[32]. $\mathcal{S}$ passes it to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

(3) Commit query from id. This concerns the case when $\mathcal{Z}$ queries $(\mathtt{Commit}, \vec{p}^{\mathsf{ctxt}}, \mathsf{svk})$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. If $\mathtt{Ptr}[\mathsf{id}] \neq \bot$, then $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ outputs $(\mathtt{Commit}, \text{node-id} := \mathtt{Ptr}[\mathsf{id}], \mathsf{id}, \vec{p}^{\mathsf{ctxt}}, \mathsf{svk})$ to $\mathcal{S}$. $\mathcal{S}$ then decrypts the proposals $\widetilde{\vec{p}} := \mathtt{*dec\text{-}and\text{-}sort\text{-}proposals}(G.\mathsf{encSecret}, \vec{p}^{\mathsf{ctxt}})$ using G.encSecret of id. If decryption fails, $\mathcal{S}$ outputs $(ack := \mathtt{false}, \bot, \bot, \bot)$. Otherwise, $\mathcal{S}$ invokes $\widetilde{\mathcal{S}}$ on input $(\mathtt{Commit}, \mathsf{id}, \widetilde{\vec{p}}, \mathsf{svk})$. $\widetilde{\mathcal{S}}$ then runs the simulated party id on input $(\mathtt{Commit}, \widetilde{\vec{p}}, \mathsf{svk})$, where it asks $\mathcal{S}$ to provide the randomness to run party id if $\mathtt{RandCorr}[\mathsf{id}] =$ 'bad'. $\mathcal{S}$ then queries this request to its own adversary $\mathcal{A}$ to receive the adversarially chosen randomness. If party id returns $\bot$, then $\widetilde{\mathcal{S}}$ outputs $(ack := \mathtt{false}, \bot, \bot\bot)$ to $\mathcal{S}$, and $\mathcal{S}$ outputs $(ack := \mathtt{false}, \bot, \bot, \bot)$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

Otherwise, if party id returns $(\widetilde{c}_0, \widetilde{\vec{c}}, \widetilde{w}_0, \widetilde{\vec{w}})$, $\widetilde{\mathcal{S}}$ decides what to output depending on the following checks (which are identical to the checks in [36]). We also explain what $\mathcal{S}$ does when given the output of $\widetilde{\mathcal{S}}$.

Case 1. If $\widetilde{\mathtt{Node}}[\widetilde{c}_0] = \bot$, $w_0 \neq \bot$, and if there exists some $rt' \in \mathbb{N}$ and $\widetilde{w}'_0$ such that $\widetilde{\mathtt{Wel}}[w'_0] = \mathsf{root}_{rt'}$ and $\widetilde{w}'_0$ includes the same confTag as $\widetilde{w}_0$, then $\widetilde{\mathcal{S}}$ chooses any such $(\widetilde{w}'_0, rt')$ (guaranteed to exist uniquely from [36, Proposition E.6]) and returns $(ack := \mathtt{true}, rt := rt' \neq \bot, \widetilde{c}_0, \widetilde{\vec{c}})$ to $\mathcal{S}$. $\mathcal{S}$ then encrypts the (non-encrypted) commit message as $(c_0^{\mathsf{ctxt}}, \vec{c}^{\mathsf{ctxt}}) \leftarrow \mathtt{*enc\text{-}commit}(G.\mathsf{encSecret}, \widetilde{c}_0, \widetilde{\vec{c}})$ using G.encSecret of id (provided by $\widetilde{\mathcal{S}}$, see *Part 1*). By Proposition C.6, Item 3 (which we prove later), there exists a unique node-id such that $\mathtt{Node}[\text{node-id}] = \widetilde{\mathtt{Node}}[\mathsf{root}_{rt}]$. That is, the syntactical information stored on the node, e.g., gid, epoch, prop, orig, and so on, are identical. $\mathcal{S}$ then outputs $(ack := \mathtt{true}, \text{node-id}, c_0^{\mathsf{ctxt}}, \vec{c}^{\mathsf{ctxt}})$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. Afterwards, when $\mathcal{S}$ is invoked on $(\mathtt{Welcome}, \mathsf{kp}_t, \text{node-id} := \mathtt{Ptr}[\mathsf{id}], \widetilde{c}_0)$ by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$, it invokes $\widetilde{\mathcal{S}}$ on the same input. $\widetilde{\mathcal{S}}$ then returns $(ack := \mathtt{true}, \widetilde{w}_0, \widetilde{\widehat{w}})$ to $\mathcal{S}$, where $\widetilde{\widehat{w}}$ is the welcome message for added member $\mathsf{id}_t \in \mathsf{mem}$ with kp_t. Finally, $\mathcal{S}$ encrypts $(\widetilde{w}_0, \widetilde{\widehat{w}})$ as $\widehat{w}^{\mathsf{ctxt}}$ using G.welcomeSecret of id and returns $(ack := \mathtt{true}, \widehat{w}^{\mathsf{ctxt}})$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

Case 2. Otherwise, if $\widetilde{\mathtt{Node}}[\widetilde{c}_0] = \bot$ and either $\widetilde{w}_0 = \bot$ or there does not exist $\widetilde{w}'_0$ such that $\widetilde{\mathtt{Wel}}[\widetilde{w}'_0] = \mathsf{root}_{rt'}$ for some $rt' \in \mathbb{N}$ and $\widetilde{w}'_0$ includes the same confTag as $\widetilde{w}_0$, then $\widetilde{\mathcal{S}}$ returns $(ack := \mathtt{true}, rt := \bot, \widetilde{c}_0, \widetilde{\vec{c}})$ to $\mathcal{S}$. $\mathcal{S}$ then encrypts the commit as above and returns $(ack := \mathtt{true}, \bot, c_0^{\mathsf{ctxt}}, \vec{c}^{\mathsf{ctxt}})$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. The welcome message is handled similarly to Case 1.

Case 3. Finally, if $\widetilde{\mathtt{Node}}[\widetilde{c}_0] \neq \bot$, then $\widetilde{\mathcal{S}}$ returns $(ack := \mathtt{true}, rt := \bot, \widetilde{c}_0, \widetilde{\vec{c}})$ to $\mathcal{S}$. $\mathcal{S}$ then searches for any $c_0^{\mathsf{ctxt}\prime}$ such that $\widetilde{c}_0 \leftarrow \mathtt{*dec\text{-}commit}_0(G.\mathsf{encSecret}, c_0^{\mathsf{ctxt}\prime})$ and $\text{node-id} = \mathtt{NodeID}[c_0^{\mathsf{ctxt}\prime}] \neq \bot$ (see Proposition C.6 for the definition of $\mathtt{*dec\text{-}commit}_0$).

[30] We note that in the previous definition of $\mathcal{F}_{\mathrm{CGKA}}$, the group identifies gid was omitted. However, this can be amended to $\mathcal{F}_{\mathrm{CGKA}}$ without loss of generality.

[31] Recall that due to our modification in the definition of the key service, $\widetilde{\mathcal{S}}$ does not need to output svk_t (see App. B.1.2).

[32] Recall that due to our modification in the definition of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$, $\widetilde{\mathcal{S}}$ needs to output kp if $\mathsf{act} =$ 'upd'. However, this can be amended to $\widetilde{\mathcal{S}}$ without loss of generality.

It then chooses any such $c_0^{\mathsf{ctxt}\prime}$ and outputs (ack := true, node-id, $c^{\mathsf{ctxt}}, \vec{c}^{\mathsf{ctxt}}$). By Proposition C.6, Item 1 (which we prove later), such $c_0^{\mathsf{ctxt}\prime}$ exists and node-id is unique regardless of the choice of $c_0^{\mathsf{ctxt}\prime}$.

If act = 'upd', $\widetilde{\mathcal{S}}$ outputs the committer's updated key package kp, and $\mathcal{S}$ passes it to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ accordingly.

Finally, when $\mathcal{F}_{\mathrm{CGKA}}$ queries (Propose, Ptr[id], p^{ctxt}) to $\mathcal{S}$ during the *fill-prop check, it must first decrypt p^{ctxt} to obtain the (non-encrypted) proposal $\widetilde{p}$. We use [36, Proposition E.8] that establishes that any member at the same node $\widetilde{\mathsf{Ptr}}$[id] stores the same secret key G.encSecret. Namely, $\mathcal{S}$ retrieves the unique G.encSecret assigned to the node $\widetilde{\mathsf{Ptr}}$[id] to decrypt p^{ctxt}. If $\widetilde{\mathsf{Prop}}[\widetilde{p}] = \perp$, then $\mathcal{S}$ invokes $\widetilde{\mathcal{S}}$ on input (Propose, $\widetilde{p}$) and receives (orig, act)[33]. It then outputs ($\perp$, orig, act) to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$. Otherwise, if $\widetilde{\mathsf{Prop}}[\widetilde{p}] \neq \perp$, then $\mathcal{S}$ searches for any $p^{\mathsf{ctxt}\prime}$ such that $\widetilde{p} \leftarrow$ *dec-prop(G.encSecret, $p^{\mathsf{ctxt}\prime}$) and prop-id = PropID[$p^{\mathsf{ctxt}\prime}$] $\neq \perp$. It then choses any such $p^{\mathsf{ctxt}\prime}$ and outputs (ack := true, prop-id, p^{ctxt}). By Proposition C.5, Item 1, such $p^{\mathsf{ctxt}\prime}$ exists and prop-id is unique regardless of the choice of $p^{\mathsf{ctxt}\prime}$. It retrieves (orig, act) $\leftarrow$ ($\widetilde{\mathsf{Prop}}[\widetilde{p}]$.orig, $\widetilde{\mathsf{Prop}}[\widetilde{p}]$.act) and returns (prop-id, orig, act) to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

(4) Process query from id**.** This concerns the case when $\mathcal{Z}$ queries (Process, $c_0^{\mathsf{ctxt}}, \widehat{c}^{\mathsf{ctxt}}, \vec{p}^{\mathsf{ctxt}}$) to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$. If Ptr[id] $\neq \perp$, then $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ outputs (Process, node-id, id, $c_0^{\mathsf{ctxt}}, \widehat{c}^{\mathsf{ctxt}}, \vec{p}^{\mathsf{ctxt}}$) to $\mathcal{S}$. $\mathcal{S}$ then decrypts the input messages as $(\widetilde{c}_0, \widetilde{\widehat{c}})$:= *dec-commit(G.encSecret, $c_0^{\mathsf{ctxt}}, \widehat{c}^{\mathsf{ctxt}}$) and $\vec{\widetilde{p}}$:= *dec-and-sort-proposals(G.encSecret, $\vec{p}^{\mathsf{ctxt}}$) as in the real protocol. If decryption fails, $\mathcal{S}$ outputs (ack := false, $\perp, \perp, \perp$). Otherwise, $\mathcal{S}$ invokes $\widetilde{\mathcal{S}}$ on input (Process, $\widetilde{c}_0, \widetilde{\widehat{c}}, \vec{\widetilde{p}}$). $\widetilde{\mathcal{S}}$ then (deterministically) runs the simulated party id on input (Process, id, $\widetilde{c}_0, \widetilde{\widehat{c}}, \vec{\widetilde{p}}$). If party id returns $\perp$, then $\widetilde{\mathcal{S}}$ outputs (ack := false, $\perp, \perp, \perp$) to $\mathcal{S}$. $\mathcal{S}$ then outputs (ack := false, $\perp, \perp, \perp$) to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

Otherwise, if party id returns (id_c, $upd \| rem \| add$), $\widetilde{\mathcal{S}}$ decides what to output depending on the following checks (which are the checks identical to [36]). We also explain what $\mathcal{S}$ does when given the output of $\widetilde{\mathcal{S}}$. Note that *fill-prop queries from $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ to $\mathcal{S}$ are answered exactly as in commit queries described above.

Case 1. If $\widetilde{\mathsf{Node}}[\widetilde{c}_0] = \perp$ and if there exists $\widetilde{w}_0$ that includes the same confTag as $\widetilde{c}_0$ such that $\widetilde{\mathsf{Wel}}[\widetilde{w}_0] = \mathsf{root}_{rt'}$ for some $rt' \in \mathbb{N}$, then $\widetilde{\mathcal{S}}$ chooses any such $(\widetilde{w}_0, rt')$ (guaranteed to exist uniquely from [36, Proposition E.6]) and returns (ack := true, $rt := rt', \perp, \perp$) to $\mathcal{S}$. By Proposition C.6, Item 3, there exists a unique node-id such that Node[node-id] = $\widetilde{\mathsf{Node}}[\mathsf{root}_{rt}]$. $\mathcal{S}$ then outputs (ack := true, node-id, $\perp, \perp$) to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

Case 2. If $\widetilde{\mathsf{Node}}[\widetilde{c}_0] = \perp$ and no such $\widetilde{w}_0$ exists, then $\widetilde{\mathcal{S}}$ retrieves the associating long-term public key svk_c of id_c (which is guaranteed to exist when process succeeds in the real protocol) and returns (ack := true, $\perp$, orig$'$:= id_c, svk$'$:= svk_c) to $\mathcal{S}$. $\mathcal{S}$ then outputs (ack := true, $\perp$, orig$'$, svk$'$) to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

Case 3. Finally, if $\widetilde{\mathsf{Node}}[\widetilde{c}_0] \neq \perp$, then $\widetilde{\mathcal{S}}$ simply returns (ack := true, $\perp, \perp, \perp$) to $\mathcal{S}$. $\mathcal{S}$ then searches for any $c_0^{\mathsf{ctxt}\prime}$ such that $\widetilde{c}_0 \leftarrow$ *dec-commit$_0$(G.encSecret, $c_0^{\mathsf{ctxt}\prime}$) and node-id = NodeID[$c_0^{\mathsf{ctxt}\prime}$] $\neq \perp$ (see Proposition C.6 for the definition of *dec-commit$_0$). It then choses any such $c_0^{\mathsf{ctxt}\prime}$ and outputs (ack := true, $\perp, \perp, \perp$). By Proposition C.6, Item 1, such $c_0^{\mathsf{ctxt}\prime}$ exists and node-id is unique regardless of the choice of $c_0^{\mathsf{ctxt}\prime}$.

(5) Join query from id**.** This concerns the case when $\mathcal{Z}$ queries (Join, $\widehat{w}^{\mathsf{ctxt}}$) to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$. If Ptr[id] $= \perp$, then $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ outputs (Join, id, $\widehat{w}^{\mathsf{ctxt}}$) to $\mathcal{S}$. $\mathcal{S}$ then decrypts the input messages as $(\widetilde{w}_0, \widetilde{\widehat{w}})$:= *dec-welcome($\widehat{w}^{\mathsf{ctxt}}$) as in the real protocol. If decryption fails, $\mathcal{S}$ outputs (ack := false, $\perp, \perp, \perp, \perp, \perp$). Otherwise, $\mathcal{S}$ invokes $\widetilde{\mathcal{S}}$ on input (Join, id, $\widetilde{w}_0, \widetilde{\widehat{w}}$). $\widetilde{\mathcal{S}}$ then (deterministically) runs the simulated party id on input (Join, $\widetilde{w}_0, \widetilde{\widehat{w}}$). If party id returns $\perp$, then $\widetilde{\mathcal{S}}$ outputs (ack := false, $\perp, \perp, \perp$) to $\mathcal{S}$. $\mathcal{S}$ then outputs (ack := false, $\perp, \perp, \perp, \perp, \perp$) to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

Otherwise, if party id returns (id_c, mem), $\widetilde{\mathcal{S}}$ decides what to output depending on the following checks (which are the checks identical to [36]). We also explain what $\mathcal{S}$ does when given the output of $\widetilde{\mathcal{S}}$.

Case 1. If $\widetilde{\mathsf{Wel}}[\widetilde{w}_0] \neq \perp$, $\widetilde{\mathcal{S}}$ returns (ack := true, $\perp, \perp, \perp$) to $\mathcal{S}$. $\mathcal{S}$ then returns (ack := true, $\perp, \perp, \perp, \perp, \perp$) to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

Case 2. Otherwise, $\widetilde{\mathcal{S}}$ checks if there exists a non-root $\widetilde{c}_0$ such that $\widetilde{\mathsf{Node}}[\widetilde{c}_0] \neq \perp$ and $\widetilde{c}_0$ includes the same confTag as the one included in w_0. If such $\widetilde{c}_0$ exists (guaranteed to be unique by [36]), then $\widetilde{\mathcal{S}}$ returns (ack := true, $\widetilde{c}_0' := \widetilde{c}_0, \perp, \perp$) to $\mathcal{S}$. $\mathcal{S}$ then retrieves G.encSecret stored on the node $\widetilde{\mathsf{Node}}[\widetilde{c}_0]$ and searches for any $c_0^{\mathsf{ctxt}\prime}$ such that $\widetilde{c}_0 \leftarrow$ *dec-commit$_0$(G.encSecret, $c_0^{\mathsf{ctxt}\prime}$) and node-id = NodeID[$c_0^{\mathsf{ctxt}\prime}$] $\neq \perp$ (see Proposition C.6 for the definition of *dec-commit$_0$). It then choses any such $c_0^{\mathsf{ctxt}\prime}$ and outputs (ack := true, node-id, $\perp, \perp, \perp, \perp$) to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

Case 3. Otherwise, if no such $\widetilde{c}_0$ exists, then $\widetilde{\mathcal{S}}$ further checks if there exists $\widetilde{w}_0'$ such that $\widetilde{\mathsf{Wel}}[\widetilde{w}_0'] \neq \perp$ that includes the same confTag as the one included in $\widetilde{w}_0$. If so, $\widetilde{\mathcal{S}}$ chooses any such $\widetilde{w}_0'$ and returns (ack := true, $\widetilde{c}_0' := \widetilde{\mathsf{Wel}}[\widetilde{w}_0'], \perp, \perp$). Here, [36, Proposition E.6] establishes that the value of $\widetilde{\mathsf{Wel}}[\widetilde{w}_0']$ is guaranteed to be the same root value, i.e., root_{rt} for some $rt \in \mathbb{N}$, for all such $\widetilde{w}_0'$. Then, by Proposition C.6, Item 3, there exists a unique node-id such that Node[node-id] = $\widetilde{\mathsf{Node}}[\mathsf{root}_{rt}]$. $\mathcal{S}$ then outputs (ack := true, node-id, $\perp, \perp, \perp, \perp$) to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

Case 4. Finally, if no such $\widetilde{c}_0$ or $\widetilde{w}_0'$ exist, then $\widetilde{\mathcal{S}}$ returns (ack := true, $\perp$, orig$'$:= id_c, mem$'$:= mem) to $\mathcal{S}$. This corresponds to the case $\widetilde{\mathsf{Wel}}[\widetilde{w}_0]$ is initialized by root_{rt} for a new $rt \in \mathbb{N}$, whose value is known to $\widetilde{\mathcal{S}}$. $\mathcal{S}$ then outputs (ack := true, $\perp$, gid$'$, epoch$'$, orig$'$, mem$'$) to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$, where gid$'$ and epoch$'$ are included in $\widetilde{w}_0'$ in the clear.

(6) Key query from id**.** This concerns the case when $\mathcal{Z}$ queries Key to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$. If Ptr[id] $\neq \perp$ and Node[Ptr[id]].key $= \perp$, then $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ outputs (Key, id) to $\mathcal{S}$. (Recall that **safe** is always set to false in this hybrid.) $\mathcal{S}$ invokes $\widetilde{\mathcal{S}}$ on input (Key, id) and receives the group secret key k. $\mathcal{S}$ returns k to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

Part 4: Correspondence Between the Two History Graphs. Before proving the supporting Propositions C.5 and C.6, we prove that the

[33] Without loss of generality, we assume $\widetilde{\mathcal{S}}$ returns act $\in$ { 'upd'-kp, 'add'-id$_t$-kp$_t$, 'rem'-id$_t$ }

history graph created within $\widetilde{\mathcal{S}}$ (and hence the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$) has a "one-to-one" correspondence between the history graph maintained by $\mathcal{F}^{\mathrm{ctxt}}_{\mathrm{CGKA}}$. To formalize this, we first define what it means for the two history graphs to be *isomorphic*.

Definition C.3. Let HG and $\widetilde{\mathsf{HG}}$ be the history graphs maintained by $\mathcal{F}_{\mathrm{CGKA}}$ and $\mathcal{F}^{\mathrm{ctxt}}_{\mathrm{CGKA}}$, respectively. We say the two history graphs are *isomorphic* if the following holds:

- Let $S_p := \{\,\mathsf{prop\text{-}id}\,\}_{\mathsf{prop\text{-}id}}$ and $\widetilde{S}_p := \{\,p\,\}_p$ be the sets of all proposal nodes in HG and $\widetilde{\mathsf{HG}}$, respectively. Then, there is a bijection f_p between these two sets such that $\mathsf{Prop}[\mathsf{prop\text{-}id}]$ and $\widetilde{\mathsf{Prop}}[f_p(\mathsf{prop\text{-}id})]$ store the same values (modulo the syntactical difference that .par stores node-id or c_0/root_{rt}).
- Let $S_c := \{\,\mathsf{node\text{-}id}\,\}_{\mathsf{node\text{-}id}}$ and $\widetilde{S}_c := \{\,c_0\,\}_{c_0} \cup \{\,\mathsf{root}_{rt}\,\}_{rt}$ be the sets of all commit nodes in HG and $\widetilde{\mathsf{HG}}$, respectively. Then, there is a bijection f_c between these two sets such that $\mathsf{Node}[\mathsf{node\text{-}id}]$ and $\widetilde{\mathsf{Node}}[f_c(\mathsf{node\text{-}id})]$ store the same values (modulo the syntactical difference that .par stores node-id or c_0/root_{rt} and .prop stores the encrypted or non-encrypted proposal).

The following lemma is the key lemma to prove Lem. C.2. Since isomorphic graphs agree on the same syntactical information, this also implies that the nodes for which the predicate **safe** is true are identical. Notably, all the proofs used to move from Hybrids 2 to 6 in Hashimoto et al. [36] with respect to $\mathcal{F}_{\mathrm{CGKA}}$ carryovers to those of $\mathcal{F}^{\mathrm{ctxt}}_{\mathrm{CGKA}}$.

Lemma C.4. *The history graph* HG *maintained by* $\mathcal{F}_{CGKA}$ *and* $\widetilde{\mathsf{HG}}$ *maintained by* $\mathcal{F}^{\mathrm{ctxt}}_{CGKA}$ *are isomorphic.*

Proof of Lem. C.4. We prove by induction. When the history graphs are both empty, then they are isomorphic with the two bijections f_p and f_c defined to be the zero function. Now, assume the history graphs HG and $\widetilde{\mathsf{HG}}$ are currently isomorphic, i.e., there exists bijections f_p and f_c between the propose and commit nodes. We check that the isomorphism is maintained even after a `Propose`, a `Commit`, a `Process`, or a `Join` query is performed. We omit queries to `Create` and key as they trivially maintain the isomorphism.

Consistency of isomorphism after `Propose`. Let us consider *Case 1* in **(2)**, i.e., $\widetilde{\mathsf{Prop}}[\widetilde{p}] = \bot$. By Proposition C.5, Item 2, we have $\mathsf{PropID}[p^{\mathrm{ctxt}}] = \bot$. Thus, both $\mathcal{F}_{\mathrm{CGKA}}$ and $\mathcal{F}^{\mathrm{ctxt}}_{\mathrm{CGKA}}$ creates a new proposal node. By assumption on the induction and how `*create-prop` is defined, we have isomorphism even after `Propose`. In particular, letting propCtr be the current stored by $\mathcal{F}^{\mathrm{ctxt}}_{\mathrm{CGKA}}$, the new bijection f'_p extends the domain and range of f_p by adding $f'_p(\mathsf{propCtr}) = \widetilde{p}$.

Next, let us consider *Case 2* in **(2)**, i.e., $\widetilde{\mathsf{Prop}}[\widetilde{p}] \neq \bot$. In this case $\mathcal{F}_{\mathrm{CGKA}}$ and $\mathcal{F}^{\mathrm{ctxt}}_{\mathrm{CGKA}}$ performs a consistency check `*consistent-prop`. Since no new proposal node is created, we only need to show that the output of this consistency check is identical for both ideal functionalities. There are two cases: $\mathsf{PropID}[p^{\mathrm{ctxt}}] = \bot$ or $\mathsf{PropID}[p^{\mathrm{ctxt}}] = \mathsf{prop\text{-}id'}$ for some $\mathsf{prop\text{-}id'} \neq \bot$. In the former case, by Proposition C.5, Item 1, $\mathcal{S}$ finds $p^{\mathrm{ctxt}\prime} \neq p^{\mathrm{ctxt}}$ and outputs $\mathsf{prop\text{-}id} = \mathsf{PropID}[p^{\mathrm{ctxt}\prime}]$. $\mathcal{F}^{\mathrm{ctxt}}_{\mathrm{CGKA}}$ performs the consistency check using this prop-id. In the latter case, $\mathcal{F}^{\mathrm{ctxt}}_{\mathrm{CGKA}}$ uses the existing $\mathsf{prop\text{-}id'} = \mathsf{PropID}[p^{\mathrm{ctxt}}]$ and ignores prop-id output by $\mathcal{S}$. By Proposition C.5, Item 1, the values of prop-id and prop-id′ are identical. We thus only need to focus on prop-id′.

Recall $\mathsf{prop\text{-}id'} = \mathsf{PropID}[p^{\mathrm{ctxt}}]$ is created either during `Propose` or `*fill-prop`. Due to the key-committing property of SKE, it must be created by a node $\widetilde{\mathsf{Ptr}}[\mathsf{id'}]$ that has the same $G.\mathsf{encSecret}$ as $\widetilde{\mathsf{Ptr}}[\mathsf{id}]$. Using [36, Proposition E.8], we must have $\widetilde{\mathsf{Ptr}}[\mathsf{id'}] = \widetilde{\mathsf{Ptr}}[\mathsf{id}]$. In other words, prop-id′ was created by the node $\mathsf{Ptr}[\mathsf{id}]$. Thus, by the assumption on the induction, the consistency check by $\mathcal{F}_{\mathrm{CGKA}}$ using $\widetilde{p}$ and those by $\mathcal{F}^{\mathrm{ctxt}}_{\mathrm{CGKA}}$ using prop-id′ are identical.

Consistency of isomorphism after `Commit`. Let us consider *Case 1* in **(3)**. This is the case where $\mathcal{S}$ assigns a detached root to c_0^{ctxt}. By Proposition C.6, Item 3, there exists a unique node-id such that $\mathsf{Node}[\mathsf{node\text{-}id}] = \widetilde{\mathsf{Node}}[\mathsf{root}_{rt}]$. Thus, both $\mathcal{F}_{\mathrm{CGKA}}$ and $\mathcal{F}^{\mathrm{ctxt}}_{\mathrm{CGKA}}$ perform the same consistency check and the attach procedure. At the end of `Commit`, the new bijection f'_c is identical to f_c except that $f'_c(\mathsf{node\text{-}id}) = \widetilde{c}_0$ rather than $f_c(\mathsf{node\text{-}id}) = \mathsf{root}_{rt}$.

Next, let us consider *Case 2* in **(3)**. This is the case where $\mathcal{S}$ assigns c_0^{ctxt} to a new node. By Proposition C.5, Item 2, we have $\mathsf{PropID}[p^{\mathrm{ctxt}}] = \bot$. Thus, both $\mathcal{F}_{\mathrm{CGKA}}$ and $\mathcal{F}^{\mathrm{ctxt}}_{\mathrm{CGKA}}$ create a new commit node. By assumption on the induction and how `*create-child` is defined, we have isomorphism even after `Commit`. In particular, letting nodeCtr be the current stored by $\mathcal{F}^{\mathrm{ctxt}}_{\mathrm{CGKA}}$, the new bijection f'_c extends the domain and range of f_c by adding $f'_c(\mathsf{nodeCtr}) = \widetilde{c}_0$.

Finally, let us consider *Case 3* in **(3)**. In this case $\mathcal{F}_{\mathrm{CGKA}}$ and $\mathcal{F}^{\mathrm{ctxt}}_{\mathrm{CGKA}}$ performs a consistency check `*consistent-com`. Since no new commit node is created, we only need to show that the output of this consistency check is identical for both ideal functionalities. Similarly to the argument made in `Propose`, there are two cases to consider: $\mathsf{NodeID}[c_0^{\mathrm{ctxt}}] = \bot$ or $\mathsf{NodeID}[c_0^{\mathrm{ctxt}}] = \mathsf{node\text{-}id'}$ for some $\mathsf{node\text{-}id'} \neq \bot$. In the former case, by Proposition C.6, Item 1, $\mathcal{S}$ finds $c_0^{\mathrm{ctxt}\prime} \neq c^{\mathrm{ctxt}}$ and outputs $\mathsf{node\text{-}id} = \mathsf{NodeID}[c_0^{\mathrm{ctxt}\prime}]$. $\mathcal{F}^{\mathrm{ctxt}}_{\mathrm{CGKA}}$ performs the consistency check using this node-id. In the latter case, $\mathcal{F}^{\mathrm{ctxt}}_{\mathrm{CGKA}}$ uses the existing $\mathsf{node\text{-}id'} = \mathsf{NodeID}[c_0^{\mathrm{ctxt}}]$ and ignores node-id output by $\mathcal{S}$. By Proposition C.6, Item 1, the values of node-id and node-id′ are identical. We thus only need to focus on node-id′.

Recall $\mathsf{node\text{-}id'} = \mathsf{NodeID}[c_0^{\mathrm{ctxt}}]$ is created either during `Commit` or `Process`. Following an almost exact argument made for PropID above, we can show node-id′ was created by the node $\mathsf{Ptr}[\mathsf{id}]$. Thus, by the assumption on the induction, the consistency check by $\mathcal{F}_{\mathrm{CGKA}}$ using $\widetilde{c}$ and those by $\mathcal{F}^{\mathrm{ctxt}}_{\mathrm{CGKA}}$ using node-id′ are identical.

Consistency of isomorphism after `Process`. All three cases considered in `Process` are covered by the cases considered in `Commit`. Namely, $\mathcal{S}$ chooses to either attach a detached root to c_0^{ctxt}, create a new commit node, or attach the commit to an already existing node.

Isomorphism after `Join`. First, we show that if $\widetilde{\mathsf{Wel}}[\widetilde{w}_0] = \bot$, then $\mathsf{Wel}[\mathsf{id}, \widehat{w}] = \bot$. Assume $\mathsf{Wel}[\mathsf{id}, \widehat{w}] \neq \bot$. Since $\mathsf{Wel}[\mathsf{id}, \widehat{w}]$ is only created during a `Commit`, this implies that there exists $\widetilde{w}'_0 \neq \widetilde{w}_0$ such that $\widetilde{\mathsf{Wel}}[\widetilde{w}'_0] \neq \bot$. This further implies that $\widehat{w}$ decrypts to two different values $\widetilde{w}'_0$ and $\widetilde{w}_0$, which contradicts the committing property of the SKE. Therefore, we have $\mathsf{Wel}[\mathsf{id}, \widehat{w}] = \bot$ when $\widetilde{\mathsf{Wel}}[\widetilde{w}_0] = \bot$.

With this in mind, *Cases 1* and *2* in **(5)** do not alter the propose or commit nodes of the history graphs maintained by both $\mathcal{F}_{\mathrm{CGKA}}$ and

$\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. These cases correspond to attaching a welcome message to an existing node on the history graph. Moreover, in *Case 3* in **(5)**, as we already established that $\mathsf{Node}[\text{node-id}] = \widetilde{\mathsf{Node}}[\mathsf{root}_{rt}] \neq \bot$. Therefor, this also does not alter the history graph.

The final *Case 4* in **(5)** corresponds to the case where a new detached root is created within $\mathcal{F}_{\mathrm{CGKA}}$. By Proposition C.6, Item 2, $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ also creates a new detached root. Since $\star$`create-root` is called on the same input, it is clear that the newly created history graphs remain isomorphic. Namely, the new bijection f'_c extends the domain and range of f_c by adding $f'_c(\mathsf{nodeCtr}) = rt$, where rt and nodeCtr are the counters used to add a new commit node in $\mathcal{F}_{\mathrm{CGKA}}$ and $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$, respectively. □

Part 5: Proving the Supporting Propositions. To finish the proof of Lem. C.2, we finally prove Propositions C.5 and C.6. Below, we rely on [36, Proposition E.8] that establishes that any member at the same node $\widetilde{\mathsf{Ptr}}[\mathsf{id}]$ stores the same secret key $G.\mathsf{encSecret}$.

Proposition C.5. *We have the following:*

1. *If* $\widetilde{\mathsf{Prop}}[\widetilde{p}] \neq \bot$, *then there exists some* p^{ctxt} *such that* $\widetilde{p} \leftarrow \star\mathtt{dec\text{-}prop}(G.\mathsf{encSecret}, p^{\mathsf{ctxt}})$ *and* $\mathsf{PropID}[p^{\mathsf{ctxt}}] \neq \bot$, *where* $G.\mathsf{encSecret}$ *is the secret key maintained in the node* $\widetilde{\mathsf{Prop}}[\widetilde{p}]$. *Moreover, for all such* p^{ctxt}, *the value of* $\mathsf{PropID}[p^{\mathsf{ctxt}}]$ *is identical.*
2. *If* $\widetilde{\mathsf{Prop}}[\widetilde{p}] = \bot$, *then does not exist any* p^{ctxt} *and* encSecret *such that* $\widetilde{p} \leftarrow \star\mathtt{dec\text{-}prop}(\mathsf{encSecret}, p^{\mathsf{ctxt}})$ *and* $\mathsf{PropID}[p^{\mathsf{ctxt}}] \neq \bot$.

Proof. Let us first consider Item 1. $\widetilde{\mathsf{Prop}}[\widetilde{p}]$ is only created during a `Propose` or a $\star$`fill-prop`. If it was created during a `Propose`, then by definition such p^{ctxt} exists. If it was created during a $\star$`fill-prop`, then $\mathcal{S}$ was given p^{ctxt} such that $\widetilde{p} \leftarrow \star\mathtt{dec\text{-}prop}(G.\mathsf{encSecret}, p^{\mathsf{ctxt}})$. At the end of $\star$`fill-prop`, $\mathsf{PropID}[p^{\mathsf{ctxt}}]$ is created. Moreover, due to the key-committing property of the SKE, the proposal node assigned to such proposals p^{ctxt} are identical.

Next, let us consider Item 2. Due to the key-committing property of the SKE, if such a p^{ctxt} existed, then we must have $\widetilde{\mathsf{Prop}}[\widetilde{p}] \neq \bot$. Thus, Item 2 follows by taking the contrapositive. □

Proposition C.6. *Let us define the function* $\star\mathtt{dec\text{-}commit}_0$ *such that given* c_0^{ctxt} *and* encSecret, *it parses* $(\mathsf{gid}, \mathsf{epoch}, \text{`commit'}, \mathsf{CT}_{\widetilde{c}_0}) \leftarrow c_0^{\mathsf{ctxt}}$, *runs* $\widetilde{c}_0 \leftarrow \mathsf{SKE.Dec}(\mathsf{encSecret}, \mathsf{CT}_{\widetilde{c}_0})$, *and outputs* $\widetilde{c}_0 := (\mathsf{gid}, \mathsf{epoch}, \text{`commit'}, \widetilde{c}_0)$.

Then, we have the following:

1. *If* $\widetilde{\mathsf{Node}}[\widetilde{c}_0] \neq \bot$, *then there exists some* c_0^{ctxt} *such that* $\widetilde{c}_0 \leftarrow \star\mathtt{dec\text{-}commit}_0(G.\mathsf{encSecret}, c_0^{\mathsf{ctxt}})$ *and* $\mathsf{NodeID}[c_0^{\mathsf{ctxt}}] \neq \bot$, *where* $G.\mathsf{encSecret}$ *is the secret key maintained in the node* $\widetilde{\mathsf{Node}}[\widetilde{c}_0]$. *Moreover, for all such* c^{ctxt}, *the value of* $\mathsf{NodeID}[c_0^{\mathsf{ctxt}}]$ *is identical.*
2. *If* $\widetilde{\mathsf{Node}}[\widetilde{c}_0] = \bot$, *then there does not exist any* c_0^{ctxt} *and* encSecret *such that* $\widetilde{c}_0 \leftarrow \star\mathtt{dec\text{-}commit}_0(\mathsf{encSecret}, c_0^{\mathsf{ctxt}})$ *and* $\mathsf{NodeID}[c_0^{\mathsf{ctxt}}] \neq \bot$.
3. *If* $\widetilde{\mathsf{Node}}[\widetilde{c}_0] = \bot$ *and there exists* $\widetilde{w}_0$ *that includes the same* $\mathsf{confTag}$ *as* $\widetilde{c}_0$ *such that* $\widetilde{\mathsf{Wel}}[\widetilde{w}_0] = \mathsf{root}_{rt}$ *for some* $rt \in \mathbb{N}$, *then there exists a unique* node-id *such that* $\mathsf{Node}[\text{node-id}] = \widetilde{\mathsf{Node}}[\mathsf{root}_{rt}]$. *That is, the syntactical information stored on the node, e.g.,* gid, epoch, prop, orig, *and so on, are identical.*

Proof. Let us first consider Item 1. $\widetilde{\mathsf{Node}}[\widetilde{c}]$ is only created during a `Commit` or a `Process`. More concretely, $\widetilde{\mathsf{Node}}[\widetilde{c}_0]$ is only created if $\widetilde{\mathcal{S}}$ assigns a detached root to $\widetilde{c}_0$ or a new commit node during a `Commit` or a `Process`. In either cases, a corresponding c_0^{ctxt} is output by $\mathcal{S}$ and $\mathsf{NodeID}[c_0^{\mathsf{ctxt}}]$ is generated within $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. Moreover, due to the key-committing property of the SKE, the commit node assigned to such commits c_0^{ctxt} are identical.

Next, let us consider Item 2. Due to the key-committing property of the SKE, if such a c^{ctxt} existed, then we must have $\widetilde{\mathsf{Node}}[\widetilde{c}_0] \neq \bot$. Thus, Item 2 follows by taking the contrapositive.

Finally, let us consider Item 3. Such $\widetilde{w}_0$ is generated when $\widetilde{\mathcal{S}}$ creates a new root detached root during a `Join` (corresponding to *Case 4* in **(5)**). Since $\mathcal{S}$ creates a new commit node with the same syntactical information that $\widetilde{\mathcal{S}}$ used, Item 3 follows. □

This concludes the proof of Lem. C.2. □

C.3.4 From Hybrid 6 to 7: Proof of Lem. C.7. In this section, we prove the indistinguishability of remaining hybrid 6 and hybrid 7. As explained in the overview, we assume the adversary *cannot* inject bad randomness throughout these hybrids, i.e., RandCorr is always 'good'.

Also, we slightly modify the description of $\mathcal{S}_6$. We assume $\mathcal{S}_6$ executes the code of $\widetilde{\mathcal{S}}_6$ by itself instead of invoking $\widetilde{\mathcal{S}}_6$.

To prove Lem. C.7, we first formally define the behavior of simulator $\mathcal{S}_7$ in Hybrid 7; then we analyze the simulation provided by $\mathcal{S}_7$ provides an indistinguishable view to $\mathcal{Z}$ as in Hybrid 6.

Part 1: Description of the Simulator $\mathcal{S}_7$. We first explain the description of $\mathcal{S}_7$. For simulation, $\mathcal{S}_7$ stores group encryption key encSecret for epoch node-id in the list $L_{\mathsf{encSecret}}$.

$\mathcal{S}_7$ simulates group states G (which includes e.g., membership list and group secret keys) of node-id depending on the status of the parent epoch of node-id denoted by node-id_p.

Case (1). If $\mathbf{safe}(\text{node-id}_p) = \mathtt{false}$ when node-id is created, $\mathcal{S}_7$ knows the group state of node-id_p. Thus, $\mathcal{S}_7$ generates the group state of node-id from it following the protocol as in $\mathcal{S}_6$.

Case (2). If $\mathbf{safe}(\text{node-id}_p) = \mathtt{true}$ when node-id is created, $\mathcal{S}_7$ defers the generation of group state of node-id to the following timing. Note that due to the restricted environment, these are the only subcase of Case (2), and only one of the following cases occurs for each node-id.

- **Case (2-1) safe**(node-id) = `true` **when proposal/commit messages are generated/processed at** node-id**:** $\mathcal{S}_7$ sets $G.\mathsf{gid} \leftarrow \mathsf{Node}[\text{node-id}].\mathsf{gid}$ and $G.\mathsf{epoch} \leftarrow \mathsf{Node}[\text{node-id}].\mathsf{epoch}+1$. Also, it chooses a random encSecret and stores $L_{\mathsf{encSecret}}[\text{node-id}] \leftarrow \mathsf{encSecret}$. In this case, it does not generate the other group state.
- **Case (2-2) Party** id **at** node-id **is corrupted (i.e., safe**(node-id) **becomes** `false`**) before proposal/commit messages are generated/processed at** node-id**:** When $\mathcal{A}$ corrupts id at node-id, $\mathcal{S}_7$ receives the following information from the functionality.
  - id's current mPKE secret key dk (stored in CurrDK array)
  - id's current signature signing key ssk (stored in SSK array)
  - The epoch pointer node-id = $\mathsf{Ptr}[\mathsf{id}]$ (i.e., id's current node in the history graph)

- The information stored in Node[node-id]

Using this information, $\mathcal{S}_7$ simulates the group states. It first initializes the group state G as follows:

- G.gid ← Node[node-id].gid
- G.epoch ← Node[node-id].epoch
- G.member ← Node[node-id].mem
- G.joinerSecret $\leftarrow_{\$} \{0,1\}^\kappa$
- G.confTransHash-w.o-'id_c' $\leftarrow_{\$} \{0,1\}^\kappa$
- G.confTransHash ← H(G.confTransHash-w.o-'id_c', id_c), where id_c := Node[node-id].orig
- G.certSvks[$*$], G.pendUpd[$*$], G.pendCom[$*$] ← $\emptyset$
- G.id ← id
- G.ssk ← ssk
- G.member[id].dk ← dk

Then, $\mathcal{S}_7$ executes the following functions in order to generate group secrets and hash values.

(1) G.memberHash ← `*derive-member-hash`(G)
(2) (G, confKey) ← `*derive-epoch-keys`(G, G.joinerSecret)
(3) confTag ← `*gen-conf-tag`(G, confKey)
(4) G ← `*set-interim-trans-hash`(G, confTag)

Since the previous epoch is secure and due to the restricted environment, $\mathcal{A}$ cannot corrupt the past secure epochs (cf. lines 9-10 in Fig. 14). Thus, $\mathcal{A}$ cannot distinguish the simulated group state from a real one since $\mathcal{A}$ cannot check whether the simulated group states are consistent for the previous epoch. Finally, $\mathcal{S}_7$ stores $L_{\mathsf{encSecret}}$[node-id] ← G.encSecret.

$\mathcal{S}_7$ answers each query made by the ideal functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},7}$ as in $\mathcal{S}_6$ except for the differences shown below. We only describe $\mathcal{S}_7$ when **safe** is true for the epoch $\mathcal{Z}$ queries. This is because if **safe** is false, $\mathcal{S}_7$ knows the corresponding group state (see above) and obtains the same information $\mathcal{S}_6$ receives from $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},7}$; thus $\mathcal{S}_7$ can simulate the protocol messages as $\mathcal{S}_6$ does.

Simulation on input (**`Create`**)**.** $\mathcal{S}_7$ chooses gid at random and returns it to the functionality. Note that $\mathcal{S}_7$ does not generate other group information in this timing (see above for when and how group state is generated). h$(1-\delta)$-correct **Simulation on input** (**`Propose`**, node-id, |id|, |act|)**.** Let encSecret := $L_{\mathsf{encSecret}}$[node-id]. $\mathcal{S}_7$ computes $\mathsf{CT}_{\overline{p}} \leftarrow \mathsf{SKE.Enc}(\mathsf{encSecret}, 0^{\ell_{\overline{p}}})$ and sets p^{ctxt} := (gid, epoch, 'proposal', $\mathsf{CT}_{\overline{p}}$), where gid and epoch is the group identity and the current epoch number of node-id.[34] It returns (ack := `true`, prop-id := $\perp, p^{\mathsf{ctxt}}$) to the functionality Note that $\ell_{\overline{p}}$, the length of $\overline{p}$, can be computed from the received information and public parameters (which determine the length of the hash value, mPKE ciphertext, signature, etc.).

Simulation on input (**`Commit`**, node-id, |id|, $\vec{p}^{\mathsf{ctxt}}$, |svk|, |mem|)**.** For each $p^{\mathsf{ctxt}} \in \vec{p}^{\mathsf{ctxt}}$, $\mathcal{S}_7$ checks whether it satisfies the following conditions.

- PropID[p^{ctxt}] ≠ node-id, i.e., the received proposal was issued at a different epoch. We call the event $\mathsf{E}_{\text{rej-1}}$.
- PropID[p^{ctxt}] = $\perp$, i.e., the received proposal was generated by the adversary. We call the event $\mathsf{E}_{\text{rej-2}}$.

If one of the above condition holds for some $p^{\mathsf{ctxt}} \in \vec{p}^{\mathsf{ctxt}}$, $\mathcal{S}_7$ outputs ack := `false`.

Otherwise[35], $\mathcal{S}_7$ generates $\mathsf{CT}_{\overline{c}_0} \leftarrow \mathsf{SKE.Enc}(\mathsf{encSecret}, 0^{\ell_{\overline{c}_0}})$ and $\widehat{c}_i^{\mathsf{ctxt}} \leftarrow \mathsf{SKE.Enc}(\mathsf{encSecret}, 0^{\ell_{\widehat{c}}})$ for encSecret := $L_{\mathsf{encSecret}}$[node-id] and each $i \in$ [|mem|]. Then, it sets c_0^{ctxt} := (gid, epoch, 'commit', $\mathsf{CT}_{\overline{c}_0}$) and $\vec{c}^{\mathsf{ctxt}} := \{\widehat{c}_i^{\mathsf{ctxt}}\}_{i\in[|\mathsf{mem}|]}$, and returns ($ack$:= `true`, node-id := $\perp$, $c_0^{\mathsf{ctxt}}, \vec{c}^{\mathsf{ctxt}}$) to the functionality, where gid and epoch is the group information of the epoch node-id. $\mathcal{S}_7$ stores $L_{\widehat{c}^{\mathsf{ctxt}}}$[node-id] ← $\vec{c}^{\mathsf{ctxt}}$. Note that both the length of $\overline{c}_0$ and $\widehat{c}$ ($\ell_{\overline{c}_0}$ and $\ell_{\widehat{c}}$) can be computed from the received information and public parameters.

Simulation on input (**`Welcome`**, kp_t, |gid|, |epoch|, |id|, |mem|)**.** $\mathcal{S}_7$ receives this message if **safe** is true for the next epoch, where new members will join. Let ek_t be the mPKE encryption key in kp_t. $\mathcal{S}_7$ computes the following:

- ct ← $\mathsf{mEnc}(\mathsf{ek}_t, 0)$
- $\mathsf{welcomeSecret}_{\mathsf{id}_t} \leftarrow_{\$} \{0,1\}^\kappa$
- $\mathsf{CT} \leftarrow \mathsf{SKE.Enc}(\mathsf{welcomeSecret}_{\mathsf{id}_t}, 0^{\ell_{(\mathsf{groupInfo,sig})}})$.

Note that $\ell_{(\mathsf{groupInfo,sig})}$ can be computed from the received leakage information and public system parameters. $\mathcal{S}_7$ sets $\widehat{w}^{\mathsf{ctxt}}$:= (id_t, (H(kp_t), ct), CT) and returns (ack := `true`, $\widehat{w}^{\mathsf{ctxt}}$) to the functionality. $\mathcal{S}_7$ also stores $L_{\widehat{w}^{\mathsf{ctxt}}}$ +← (id_t, H(kp_t), ct, CT).

Simulation on input (**`Process`**, node-id, index, $c_0^{\mathsf{ctxt}}, \widehat{c}^{\mathsf{ctxt}}, \vec{p}^{\mathsf{ctxt}}$)**.** $\mathcal{S}_7$ first checks the following conditions and outputs ack := `false` if one of the conditions holds.

- PropID[p^{ctxt}] = node-id′, NodeID[c_0^{ctxt}] = node-id′ or $\widehat{c}^{\mathsf{ctxt}} \in$ Node[node-id′].vcom such that node-id′ ≠ node-id, i.e., the received proposal/commit message was generated at a different epoch. We call the event $\mathsf{E}_{\text{rej-1}}$.
- PropID[p^{ctxt}] = $\perp$, NodeID[c_0^{ctxt}] = $\perp$ or $\widehat{c}^{\mathsf{ctxt}} \notin$ Node[node-id′].vcom for all node-id′, i.e., the received proposal/commit was generated by the adversary. We call the event $\mathsf{E}_{\text{rej-2}}$.

Otherwise[36], $\mathcal{S}_7$ returns (ack := `true`, $\perp, \perp, \perp$).

Simulation on input (**`Join`**, id, $\widehat{w}^{\mathsf{ctxt}}$)**.** Let ($\mathsf{id}_t$, kphash_t, ct_t, CT_t) := $\widehat{w}^{\mathsf{ctxt}}$. (In the following, we assume id_t = id and id succeeds to fetch kp_t such that $\mathsf{kphash}_t = \mathsf{h}(\mathsf{kp}_t)$; otherwise both $\mathcal{S}_6$ and $\mathcal{S}_7$ return ack := `false`.) If $\widehat{w}^{\mathsf{ctxt}}$ is generated by an honest committer (i.e., `*succeed-wel` returns `true`), $\mathcal{S}_7$ returns (ack := `true`, $\perp, \perp, \perp, \perp, \perp$). Else, $\mathcal{S}_7$ processes the welcome message as in $\mathcal{S}_6$ (i.e., following protocol description).

Part 2: Indistinguishability of Two Hybrids. We then prove the following lemma.

Lemma C.7. *Hybrid 6 and Hybrid 7 are indistinguishable assuming the INDCCA security of* mPKE *and the* IND-CCA *security of* SKE.

Proof. To show Lem. C.7, we consider the following sub-hybrids between Hybrid 6 and Hybrid 7.

Hybrid 6-0 := Hybrid 6. This is identical to Hybrid 6. We use the functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},6}$, and the simulator $\mathcal{S}_{6\text{-}0} := \mathcal{S}_6$.

[34] The simulator always knows gid and epoch for each node-id.

[35] This implies $\vec{p}^{\mathsf{ctxt}}$ only contains proposals such that PropID[p^{ctxt}] = node-id and thus `*succeed-com` returns true.

[36] This implies ($c_0^{\mathsf{ctxt}}, \widehat{c}^{\mathsf{ctxt}}, \vec{p}^{\mathsf{ctxt}}$) is honestly issued at node-id and thus `*succeed-proc` returns `true`

Hybrid 6-1 In this hybrid, the simulator $\mathcal{S}_{6\text{-}1}$ is defined exactly as $\mathcal{S}_{6\text{-}0}$ except that it always outputs $ack := \texttt{false}$ to the functionality when the event $\mathsf{E}_{\text{rej-1}}$ occurs.

Hybrid 6-2 In this hybrid, the simulator $\mathcal{S}_{6\text{-}2}$ is defined exactly as $\mathcal{S}_{6\text{-}1}$ except that it replaces encSecret and $\text{welcomeSecret}_{\text{id}_t}$ with a random value if **safe** is true.

Hybrid 6-3. In this hybrid, the simulator $\mathcal{S}_{6\text{-}3}$ is defined exactly as $\mathcal{S}_{6\text{-}2}$ except that it always outputs $ack := \texttt{false}$ when the event $\mathsf{E}_{\text{rej-2}}$ occurs.

Hybrid 6-4. In this hybrid, the simulator $\mathcal{S}_{6\text{-}4}$ is defined exactly as $\mathcal{S}_{6\text{-}3}$ except that it replaces a mPKE ciphertext in welcome messages with a ciphertext of 0 if **safe** is true for the next epoch where new members join with the welcome messages.

Hybrid 6-5. In this hybrid, the simulator $\mathcal{S}_{6\text{-}5}$ is defined exactly as $\mathcal{S}_{6\text{-}4}$ except that it replaces SKE ciphertexts in welcome messages with encryption of zero-string if **safe** is true for the epoch where new members join with the welcome messages.

Hybrid 6-6. In this hybrid, the simulator $\mathcal{S}_{6\text{-}6}$ is defined exactly as $\mathcal{S}_{6\text{-}5}$ except that it replaces SKE ciphertexts in proposal and commit messages with encryption of zero-string if **safe** is true for the epoch where these messages are issued. Note that $\mathcal{S}_{6\text{-}6}$ is identical to $\mathcal{S}_7$.

Hybrid 6-7 = Hybrid 7. We replace the functionality $\mathcal{F}^{\text{ctxt}}_{\text{CGKA},6}$ with the functionality $\mathcal{F}^{\text{ctxt}}_{\text{CGKA},7}$. The simulator $\mathcal{S}_7$ is defined by the above description. Since $\mathcal{S}_7$ has simulated the protocol with the information given from $\mathcal{F}^{\text{ctxt}}_{\text{CGKA},7}$, Hybrid 6-6 and Hybrid 6-7 are identical.

From Lems. C.8 to C.11, C.13 and C.15 provided below, Hybrid 6-0 and Hybrid 6-6 are indistinguishable. Therefore, we conclude that Hybrid 6 and Hybrid 7 are indistinguishable. □

C.3.5 From Hybrid 6-0 to 6-1: Proof of Lem. C.8.

Lemma C.8. *Hybrid 6-0 and Hybrid 6-1 are indistinguishable assuming the key-committing property of* SKE.

Proof. The difference between $\mathcal{S}_{6\text{-}0}$ and $\mathcal{S}_{6\text{-}1}$ is $\mathcal{S}_{6\text{-}1}$ always outputs $ack := \texttt{false}$ when the event $\mathsf{E}_{\text{rej-1}}$ occurs. In other words, if $\mathcal{S}_{6\text{-}0}$ outputs $ack := \texttt{true}$ when $\mathsf{E}_{\text{rej-1}}$ occurs, $\mathcal{Z}$ can distinguish the two hybrids. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the key-committing property of SKE.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}^{\text{IW}}_{\text{AS}}$ and $\mathcal{F}^{\text{IW}}_{\text{KS}}$, and the protocol execution of each party as in $\mathcal{S}_{6\text{-}0}$. Assume $\mathsf{E}_{\text{rej-1}}$ occurs for some SKE ciphertext CT and $\mathcal{Z}$ distinguish the two hybrids when CT is processed. By the condition of $\mathsf{E}_{\text{rej-1}}$, there exists the encryption key k (encSecret or $\text{welcomeSecret}_{\text{id}_t}$) used to generate CT and the other encryption key $k' \neq k$ that correctly decrypts CT. This means CT can be correctly decrypted with both k and k', which implies $\mathcal{B}$ can break the key-committing property of SKE. This contradicts the assumption that SKE has key-committing property. Therefore, both $\mathcal{S}_{6\text{-}0}$ and $\mathcal{S}_{6\text{-}1}$ always outputs $ack := \texttt{false}$ when $\mathsf{E}_{\text{rej-1}}$ occurs. □

C.3.6 From Hybrid 6-1 to 6-2: Proof of Lem. C.9.

Lemma C.9. *Hybrid 6-1 and Hybrid 6-2 are indistinguishable assuming* mPKE *is Chained CmPKE conforming GSD secure.*

Proof. The proof is identical to the proof in [37, Lemma E.29]. To prove the indistinguishability of the two hybrids, we gradually replace each encSecret and $\text{welcomeSecret}_{\text{id}_t}$ with a random value if **safe** is true when a propose/commit/welcome message is created. Similar to the proof of randomness of appSecret provided in [37, Lemma E.29], we can show that, if $\mathcal{Z}$ can distinguish whether encSecret or $\text{welcomeSecret}_{\text{id}}$ are real or random, it can be used to break the Chained CmPKE conforming GSD security of mPKE. Note that the value of **safe** is fixed when a propose/commit/welcome message is created at the epoch (cf. `*mark-content-hiden-epoch`), and the adversary is restricted from colluding to change this value (cf. line 11 in Fig. 14). Thus, we can construct a reduction $\mathcal{B}$ as in [37, Lemma E.29]. Therefore, if mPKE is Chained CmPKE conforming GSD secure, Hybrid 6-1 and Hybrid 6-2 are indistinguishable. □

C.3.7 From Hybrid 6-2 to 6-3: Proof of Lem. C.10.

Lemma C.10. *Hybrid 6-2 and Hybrid 6-3 are indistinguishable assuming* SKE *is the* IND-CCA *secure.*

Proof. The difference between $\mathcal{S}_{6\text{-}2}$ and $\mathcal{S}_{6\text{-}3}$ is $\mathcal{S}_{6\text{-}3}$ always outputs $ack := \texttt{false}$ when the event $\mathsf{E}_{\text{rej-2}}$ occurs[37]. In other words, $\mathcal{Z}$ distinguishes the two hybrids if the adversary produces malicious ciphertexts that can be decrypted correctly. Such a malicious ciphertext may contain (1) a malicious plain message (i.e., the message generated by the adversary) or (2) an honest plain message (i.e., the message generated by an honest party at the same epoch). For the former case, we already proved that the adversary cannot produce acceptable messages without knowing MAC key (cf. Hybrid 5). Thus, the previous simulator also outputs $ack := \texttt{false}$ in the former case. Therefore, we care about the latter case. We show that, if $\mathcal{Z}$ can distinguish the two hybrids when the adversary produces a malicious ciphertext that contains an honestly generated plain message, then we can construct an adversary $\mathcal{B}$ that breaks the IND-CCA security of SKE.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}^{\text{IW}}_{\text{AS}}$ and $\mathcal{F}^{\text{IW}}_{\text{KS}}$, and the protocol execution of each party as in $\mathcal{S}_{6\text{-}2}$ except for encrypting messages. At the beginning of the game, $\mathcal{B}$ chooses $i \in [Q]$ at random (where Q is the total number of honestly generated epochs) and hopes that $\mathcal{Z}$ distinguishes the hybrids when the event occurs in the i-th honest epoch. That is, we assume that, when a party processes some ciphertext CT with the i-th honest encryption key, $\mathcal{S}_{6\text{-}2}$ outputs $ack := \texttt{true}$, but $\mathcal{S}_{6\text{-}3}$ outputs $ack := \texttt{false}$. ($\mathcal{B}$ succeeds to guess such an epoch with probability $1/Q$.) $\mathcal{B}$ embed the challenge SKE key of the IND-CCA game to the i-th encSecret. Note that if **safe** is true for the corresponding epoch, the encryption key is chosen at random due to the modification we made in Hybrid 6-2; thus the challenge key can be embedded. In addition, the adversary cannot corrupt the challenge key after the key is used for encryption since the adversary's corruption is restricted (cf. `*mark-content-hiden-epoch` function). When $\mathcal{B}$ encrypts messages with the i-th encSecret (i.e., the challenge key), it uses the encryption oracle $\mathcal{LR}$ by setting M_0 as the actual message and M_1 as a random message with the same length. $\mathcal{B}$ uses its decryption

[37] The event $\mathsf{E}_{\text{rej-2}}$ only occurs in epochs where **safe** is true.

oracle when it wants to decrypt a ciphertext[38]. If the decryption result is identical to the message M_b, $\mathcal{B}$ outputs the bit b to the challenger of the IND-CCA game. Else, the decryption result is different from the messages sent to the encryption oracle, it outputs $ack := \texttt{false}$. Note that for other keys, $\mathcal{B}$ simulates protocol as in the previous hybrid.

We can verify that $\mathcal{B}$ wins the IND-CCA game if the adversary can produce a ciphertext that contains an honestly generated message. Note that the messages M_{1-b} are information-theoretically hidden from the adversary (and $\mathcal{Z}$). Hence, if $\mathcal{Z}$ can distinguish the two hybrids, $\mathcal{B}$ can break the IND-CCA security of SKE. This contradicts the assumption that SKE is the IND-CCA secure. Therefore, $\mathcal{Z}$ cannot distinguish the two hybrids. In other words, both $\mathcal{S}_{6\text{-}2}$ and $\mathcal{S}_{6\text{-}3}$ outputs $ack = \texttt{false}$ when $\mathsf{E}_{\text{rej-2}}$ occurs. □

C.3.8 From Hybrid 6-3 to 6-4: Proof of Lem. C.11.

Lemma C.11. *Hybrid 6-3 and Hybrid 6-4 are indistinguishable assuming* mPKE *is* IND-CCA *secure.*

Proof. We assume $\mathcal{Z}$ creates at most W welcome messages by `Commit` query. To show Lem. C.11, we consider the following sub-hybrids between Hybrid 6-3 and Hybrid 6-4.

Hybrid 6-3-0 := Hybrid 6-3. This is identical to Hybrid 6-3. The simulator $\mathcal{S}_{6\text{-}3\text{-}0} = \mathcal{S}_{6\text{-}3}$ generates protocol messages following the protocol procedures.

Hybrid 6-3-*i*. i runs through $[W]$. The simulator $\mathcal{S}_{6\text{-}3\text{-}i}$ is defined exactly as $\mathcal{S}_{6\text{-}3\text{-}(i-1)}$ except that, when it simulates the i-th welcome message, if the **safe** predicates is true for the next epoch the welcome message is associated with, it encrypts zero-strings instead of the joiner secret with the new member's mPKE encryption key. Note that we count welcome messages in the order the simulator creates. We show in Lem. C.12 that Hybrid 6-3-$(i-1)$ and Hybrid 6-3-i are indistinguishable.

Hybrid 6-4. This is identical to Hybrid 6-3-W. In this hybrid, mPKE ciphertexts in welcome messages issued for secure epochs (i.e., **safe** is true) are replaced with the encryption of zero-string.

The indistinguishability between Hybrid 6-3-0 and Hybrid 6-3-Q is derived by applying Lem. C.12 for all $i \in [W]$. Therefore, we conclude that Hybrid 6-3 and Hybrid 6-3 are indistinguishable. □

C.3.9 Proof of Lem. C.12: From Hybrid 6-3-$(i-1)$ to 6-3-i.

Lemma C.12. *Hybrid 6-3-$(i-1)$ and Hybrid 6-3-i are indistinguishable assuming* mPKE *is* IND-CCA *secure*[39].

Proof. The difference between $\mathcal{S}_{6\text{-}3\text{-}(i-1)}$ and $\mathcal{S}_{6\text{-}3\text{-}i}$ is, when generating the i-th welcome message, if the next epoch is secure (i.e., **safe** is true for the next epoch), $\mathcal{S}_{6\text{-}3\text{-}i}$ replaces the mPKE ciphertext in the i-th welcome message with a ciphertext of 0 encrypted with the new member's mPKE encryption key (the simulator receives the new member's key package from the functionality). We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks IND-CCA security of mPKE. We first explain the description of $\mathcal{B}$ and then evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution as in $\mathcal{S}_{6\text{-}3\text{-}(i-1)}$ except for generating the i-th welcome message. Let ek^* be the challenge mPKE encryption key provided by the IND-CCA game. Observer that honest key packages are generated on `register-kp` queries to $\mathcal{F}_{\text{KS}}^{\text{IW}}$. We assume $\mathcal{Z}$ issues at most Q `register-kp` queries. At the beginning of the game, $\mathcal{B}$ chooses an index $J \in [Q]$ at random, embeds the challenge key in the J-th `register-kp` query, and hopes that the J-th key package will be used to add the i-th welcome message. ($\mathcal{B}$ succeeds to guess with probability $1/Q$.) For other `register-kp` queries, $\mathcal{B}$ generates key packages following the description of $\mathcal{F}_{\text{KS}}^{\text{IW}}$. If $\mathcal{B}$ decrypts a ciphertext with the challenge decryption key, $\mathcal{B}$ uses the decryption oracle provided by the IND-CCA game.

Assume the i-th welcome message is created with the J-th key package and **safe** is true for the new epoch where the i-th welcome message is created. Let joinerSecret be the corresponding joiner secret. $\mathcal{B}$ outputs $\mathsf{M}_0 :=$ joinerSecret and $\mathsf{M}_1 := 0$ to IND-CCA game challenger and receives the challenge ciphertext $(\mathsf{ct}_0^*, \widehat{\mathsf{ct}}^*)$. $\mathcal{B}$ uses it as the mPKE ciphertext in the i-th welcome message. Note that, since **safe** is true, the adversary has not corrupted the challenge key, and it is restricted from colluding to compute the challenge key (cf. `*mark-content-hiden-epoch` function). Thus, $\mathcal{B}$ never corrupts the challenge key.

We finally analyze $\mathcal{B}$'s advantage. If the challenger returns the ciphertext of the joiner secret, $\mathcal{Z}$'s view is identical to Hybrid 6-3-$(i-1)$; else, i.e., the challenger returns the ciphertext of 0, $\mathcal{Z}$'s view is identical to Hybrid 6-3-i. Hence, if $\mathcal{Z}$ distinguishes Hybrid 6-3-$(i-1)$ and Hybrid Hybrid 6-3-i with non-negligible probability, $\mathcal{B}$ wins the IND-CCA game with non-negligible probability by using $\mathcal{Z}$'s output. This contradicts the assumption that mPKE is IND-CCA secure. Therefore, Hybrid 6-3-$(i-1)$ and Hybrid Hybrid 6-3-i are indistinguishable. □

C.3.10 From Hybrid 6-4 to 6-5: Proof of Lem. C.13.

Lemma C.13. *Hybrid 6-4 and Hybrid 6-5 are indistinguishable assuming* SKE *is the* IND-CCA *secure.*

Proof. We assume $\mathcal{Z}$ creates at most W welcome messages by `Commit` query. To show Lem. C.13, we consider the following sub-hybrids between Hybrid 6-4 and Hybrid 6-5.

Hybrid 6-4-0 := Hybrid 6-4. This is identical to Hybrid 6-4. The simulator $\mathcal{S}_{6\text{-}4\text{-}0} = \mathcal{S}_{6\text{-}4}$ generates protocol messages following the protocol procedures.

Hybrid 6-4-*i*. i runs through $[W]$. The simulator $\mathcal{S}_{6\text{-}4\text{-}i}$ is defined exactly as $\mathcal{S}_{6\text{-}4\text{-}(i-1)}$ except that, when it simulates the i-th welcome message, if **safe** is true for the next epoch the welcome message is associated with, it generates a ciphertext as $\mathsf{CT} \leftarrow \mathsf{SKE.Enc}(\mathsf{welcomeSecret}_{\mathsf{id}_t}, 0^{\ell_{(\mathsf{groupInfo,sig})}})$. Note that we count welcome messages in the order the simulator creates. We show in Lem. C.14 that Hybrid 6-4-$(i-1)$ and Hybrid 6-4-i are indistinguishable.

Hybrid 6-3. This is identical to Hybrid 6-2-W. In this hybrid, SKE ciphertext in welcome messages issued for secure epochs (i.e., **safe** is true) is replaced with a random string.

[38] By definition of the simulator, $\mathcal{B}$ only sends ciphertexts that are not produced by the encryption oracle.

[39] We use the IND-CCA game with $N = 1$ (cf. Fig. 8).

The indistinguishability between Hybrid 6-4-0 and Hybrid 6-4-Q is derived by applying Lem. C.14 for all $i \in [W]$. Therefore, we conclude that Hybrid 6-4 and Hybrid 6-3 are indistinguishable. □

C.3.11 Proof of Lem. C.14: From Hybrid 6-4-($i-1$) to 6-4-i.

Lemma C.14. *Hybrid 6-4-($i-1$) and Hybrid 6-4-i are indistinguishable assuming* SKE *is the* IND-CCA *secure.*

Proof. The difference between $\mathcal{S}_{\text{6-4-}(i-1)}$ and $\mathcal{S}_{\text{6-4-}i}$ is, when generating the i-th welcome message, if the next epoch is secure (i.e., **safe** is true for the next epoch), $\mathcal{S}_{\text{6-4-}i}$ replaces the SKE ciphertext in the i-th welcome message with an encryption of zero-string. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the IND-CCA security of SKE. We first explain the description of $\mathcal{B}$ and then evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution as in $\mathcal{S}_{\text{6-4-}(i-1)}$ except for generation of SKE ciphertext in the i-th welcome message if **safe** is true for the new epoch corresponding to the i-th welcome message. $\mathcal{B}$ embed the challenge SKE key of the IND-CCA game to $\text{welcomeSecret}_{\text{id}_t}$, which is used to encrypt the i-th welcome message. If **safe** is true for the corresponding epoch, welcomeSecret is chosen at random due to the modification we made in Hybrid 6-1; thus the challenge key can be embedded. In addition, the adversary cannot corrupt the challenge key after messages are encrypted with the key since the adversary's corruption is restricted (cf. `*mark-content-hiden-epoch` function).

When $\mathcal{B}$ encrypts the contents M in the i-th welcome message, it queries $\mathsf{CT} := \mathcal{LR}(\mathsf{M}_0 := (\text{groupInfo}, \text{sig}), \mathsf{M}_1 := 0^{\ell_{(\text{groupInfo,sig})}})$ and uses the challenge ciphertext CT as the ciphertext in the i-th welcome message. When $\mathcal{B}$ decrypts ciphertexts different from challenge ciphertexts with the challenge key, it uses its decryption oracle.

We finally analyze $\mathcal{B}$'s advantage. If the oracle $\mathcal{LR}$ returns a ciphertext of the real contents M_0, $\mathcal{Z}$'s view is identical to Hybrid 6-4-($i-1$); else, i.e., the oracle $\mathcal{LR}$ returns a ciphertext of zero-string, $\mathcal{Z}$'s view is identical to Hybrid 6-4-i. Hence, if $\mathcal{Z}$ distinguishes Hybrid 6-4-($i-1$) and Hybrid Hybrid 6-4-i with non-negligible probability, $\mathcal{B}$ breaks the IND-CCA security of SKE with non-negligible probability by using $\mathcal{Z}$'s output. This contradicts the assumption that SKE is IND-CCA secure. Therefore, Hybrid 6-4-($i-1$) and Hybrid Hybrid 6-4-i are indistinguishable. □

C.3.12 From Hybrid 6-5 to 6-6: Proof of Lem. C.15.

Lemma C.15. *Hybrid 6-5 and Hybrid 6-6 are indistinguishable assuming* SKE *is* IND-CCA *secure.*

Proof. We assume $\mathcal{Z}$ creates at most Q epochs (i.e., commit nodes). To show Lem. C.15, we consider the following sub-hybrids between Hybrid 6-5 and Hybrid 6-6.

Hybrid 6-5-0 := Hybrid 6-5. This is identical to Hybrid 6-5. We use the functionality $\mathcal{F}_{\text{CGKA},6}^{\text{ctxt}}$, and the simulator $\mathcal{S}_{\text{6-5}} = \mathcal{S}_{\text{6-5-0}}$.

Hybrid 6-5-i. i runs through $[Q]$. The simulator $\mathcal{S}_{\text{6-5-}i}$ is defined exactly as $\mathcal{S}_{\text{6-5-}(i-1)}$ except that, when a party issues proposal or commit messages at the i-th epoch, if **safe** is true for the epoch, it encrypts a zero-string instead of the real contents in the non-encrypted proposal/commit message. Note that we count epochs in the order in which `Propose` or `Commit` is called. We show in Lem. C.16 that Hybrid 6-5-($i-1$) and Hybrid 6-5-i are indistinguishable.

Hybrid 6-6. This is identical to Hybrid 6-5-Q. In this hybrid, ciphertexts in proposal and commit messages issued at secure epochs (epochs such that **safe** is true) are replaced with random bit-strings.

The indistinguishability between Hybrid 6-5-0 and Hybrid 6-5-Q is derived by applying Lem. C.16 for all $i \in [Q]$. Therefore, we conclude that Hybrid 6-5 and Hybrid 6-6 are indistinguishable. □

C.3.13 Proof of Lem. C.16: From Hybrid 6-5-($i-1$) to 6-5-i.

Lemma C.16. *Hybrid 6-5-($i-1$) and Hybrid 6-5-i are indistinguishable assuming* SKE *is* IND-CCA *secure.*

Proof. The difference between $\mathcal{S}_{\text{6-5-}(i-1)}$ and $\mathcal{S}_{\text{6-5-}i}$ is, when generating proposal or commit messages at the i-th epoch such that **safe** is true, $\mathcal{S}_{\text{6-5-}i}$ encrypts a zero-string instead of the real contents. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the IND-CCA security of SKE. We first explain the description of $\mathcal{B}$. Then we evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution of each party as in $\mathcal{S}_{\text{6-5-}(i-1)}$ except for encryption of proposal and commit messages generated at the i-th epoch if **safe** is true. $\mathcal{B}$ embed the challenge SKE key of the IND-CCA game to encSecret of the i-th epoch (if **safe** is true for the i-th epoch, encSecret is chosen at random due to the modification we made in Hybrid 6-2; thus the challenge key can be embedded). In addition, the adversary cannot corrupt the challenge key after messages are encrypted with the key since the adversary's corruption is restricted (cf. `*mark-content-hiden-epoch` function). When $\mathcal{B}$ encrypts contents M in a proposal or commit message at the i-th epoch, it queries $\mathsf{CT} := \mathcal{LR}(\mathsf{M}_0 := \mathsf{M}, \mathsf{M}_1 := 0^{|\mathsf{M}|})$. Note that due to the modification we made in Hybrid 6-3, $\mathcal{B}$ can reject any messages which do not generate with the i-th encryption key or generated by the adversary. Thus it does not need to use the decryption oracle.

We finally analyze $\mathcal{B}$'s advantage. If the encryption oracle $\mathcal{LR}$ returns ciphertexts of M, $\mathcal{Z}$'s view is identical to Hybrid 6-5-($i-1$); else, i.e., the $\mathcal{LR}$ oracle returns ciphertexts of zero-string, $\mathcal{Z}$'s view is identical to Hybrid 6-5-i. In addition, the i-th encryption secret is hidden from the adversary. Hence, if $\mathcal{Z}$ distinguishes Hybrid 6-5-($i-1$) and Hybrid Hybrid 6-5-i with non-negligible probability, $\mathcal{B}$ wins the IND-CCA game with non-negligible probability by using $\mathcal{Z}$'s output. This contradicts the assumption that SKE is IND-CCA secure. Therefore, Hybrid 6-5-($i-1$) and Hybrid Hybrid 6-5-i are indistinguishable. □

D METADATA-HIDING CGKA: DEFINITION

In this section, we propose a UC security model capturing the security of the 1st, 2nd, and 3rd layers (i.e., group secret keys, static and *dynamic* metadata) by defining a new ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$.

D.1 An Overview of $\mathcal{F}^{\mathsf{mh}}_{\mathbf{CGKA}}$

The ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ is formally defined in Figs. 36 to 38, 40 and 41, along with the helper functions in Fig. 39 to aid the readability. As it can be checked, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ shares a large portion of its code with $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. This is because $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ by definition also models an ideal functionality of a CGKA securing the 1st & 2nd layers. For better readability, we outsourced the description of $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ that is non-essential to the dynamic metadata to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. This should not be misunderstood as $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ making oracle calls to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ — the former simply reuses the codes of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

While $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ allows the adversary to corrupt the parties (and the server) as in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$, we do not model adversarial controlled randomness (i.e., RandCorr is always set to 'good'). As the first work to formally capture the dynamic metadata layer, we opted for simplicity and better readability. We leave it as future work to allow such types of attacks. We now explain the ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ in more detail.

States. Similarly to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ maintains a history graph. As a custom, we assume one honest group is created by the designated party $\mathsf{id}_{\mathsf{creator}}$. Such group is assigned to the main root node-id = 0. Since $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ does not allow the adversary to manipulate the party's randomness, RandCorr[id] cannot be switched to 'bad'.

Other than the history graph, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ also maintains three databases: PropDB[$*, *$], ComDB[$*, *$], and WelDB[$*$]. The proposal database PropDB[gid, epoch] stores proposals issued at (gid, epoch). The welcome database WelDB[id] stores a welcome message sent to id. Since $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ processes a join query only when Ptr[id] = $\perp$ (i.e., a party id is not assigned to any group), WelDB[id] stores only one welcome message for each id, and overwrites the old one if a new welcome message is published to id. Finally, the commit database ComDB[gid, epoch] stores the status of the group gid at epoch. Depending on the status of the group, it takes one of the following five values:

- ComDB[gid, epoch] = $\perp$: It indicates that the epoch has not been initialized yet.
- ComDB[gid, epoch] = ($\top$, node-id): It indicates that the epochhas been initialized by an honest party located at the commit node node-id (of the history graph), and no commit has been issued at epoch.
- ComDB[gid, epoch] = (($c_0, \vec{c}$), node-id): It indicates that the epoch has been initialized by an honest party located at the commit node node-id, and a commit ($c_0, \vec{c}$) has been issued at epoch.
- ComDB[gid, epoch] = ($\top$, 'adv'): It indicates that the epoch has been initialized by an adversary, and no commit has been issued at epoch.
- ComDB[gid, epoch] = (($c_0, \vec{c}$), 'adv'):It indicates that the epoch has been initialized by an adversary, and a commit ($c_0, \vec{c}$) has been issued at epoch.

Looking ahead, (assuming the server Sv is honest) ComDB[gid, epoch] is initialized with $\perp$. If some party at epoch − 1 creates a commit, ComDB[gid, epoch] is initialized to ($\top, *$), where $*$ depends on whether the party was honest or corrupt. Once the proposals in at epoch, i.e., PropDB[gid, epoch], are committed, then ComDB[gid, epoch] stores that commit and this freezes the epoch, and initializes a new ComDB[gid, epoch + 1] = $\perp$. This models the fact that a commit is generated only once per epoch when the server is honest. We refer the readers to see Sec. 4.2 for a pictorial example.

Interfaces. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ offers similar interfaces to the parties as $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. A party can create a group, create proposal, commit, or welcome messages, process these messages, and obtain group secret keys. In addition to these functionalities, it also offers the functionalities of publishing and fetching messages, which models the message delivery between a (possibly malicious) party and the server. To formally capture this, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ makes the existence of the server explicit — recall that the server was implicit in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ and abstracted away as a malicious network that injects arbitrary message. Specifically $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ models an *honest-but-curious* server. When the server is corrupted by the adversary, we end up with the same functionality as $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

In the following, we explain each of the functionalities in mode detail.

D.2 Functions Used by Legitimate Parties

Create and register group (See Fig. 36). The designated party $\mathsf{id}_{\mathsf{creator}}$ first creates the group as in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ (cf. *Group creation* in App. B.2). $\mathsf{id}_{\mathsf{creator}}$ then registers the group to the server via the subroutine RegisterGroup. $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ first checks that the party $\mathsf{id}_{\mathsf{creator}}$ is located at epoch = 0 (i.e., Ptr[$\mathsf{id}_{\mathsf{creator}}$] = 0). Then, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ informs the adversary that a new group with (gid, 0) is being registered.

- If the server is honest (i.e., ServerStat = 'good'), the ideal server checks that the group with gid has not been registered yet. If so, the server initializes the group as ComDB[gid, epoch] ← ($\top$, 0)[40] and returns *accept* = true to the party. Else, the registration is rejected and the party receives *accept* = false. The ideal server outputs (*accept*, gid, epoch = 0) (to the environment $\mathcal{Z}$). Since parties are assumed to access the server via a client-anonymous authenticated channel, the server never outputs the accessing party's identity.
- If the server is malicious, the adversary specifies the protocol result *accept*′, and it is outputted to id. This means the malicious server can decide the protocol result arbitrarily.

Create and publish proposals (See Fig. 36). When a party id create a proposal, $\mathcal{Z}$ invokes $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ on input (Propose, act). $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ first creates a proposal message p as in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ (cf. *Creating proposals* in App. B.2). Then, id publishes p to the server via the subroutine PublishProposal.

- If the server is honest (i.e., ServerStat = 'good'), $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ gives (gid, epoch, p) to the adversary, where (gid, epoch) is the destination group identity and epoch. The party accesses the server via the client-anonymous authenticated channel, and therefore, the adversary does not receive the party's identity. The adversary reports whether the protocol succeeds or not by setting *accept*′ to true or false. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ then determines the party and server's output as follows.

[40] The RegisterGroup subroutine only occurs when the main group (which is assigned to node-id = 0) is registered.

- If ComDB[gid, epoch] = ($\top$, node-id) and node-id = Ptr[id], the ideal party always outputs *accept* := true. This models correctness: if a party holds the same group state used to initialize node-id, the server must accept the published proposal.
- If ComDB[gid, epoch] = ($\top$, 'adv') and *accept*′ = true, the ideal party outputs *accept* := true. This models the fact that when the destination (gid, epoch) is initialized by the adversary, we let the adversary decide if the ideal server should accept the published proposal or not. Looking at the example from Sec. 4.2, if ComDB[gid, epoch] was initialized with a fake group statement gvk, then the adversary can deliberately make the publish proposal fail.
- Otherwise, the ideal party outputs *accept* := false. This case occurs if *accept*′ = false (i.e., the adversary decides to reject), ComDB[gid, epoch] ≠ ($\top$, $*$) (i.e., the destination has not been initialized or a commit has been issued), or ComDB[gid, epoch] = ($\top$, node-id) but node-id ≠ Ptr[id] (i.e., the party is located in a different commit node).

The ideal server outputs the destination (gid, epoch) and the published proposal p to $\mathcal{Z}$. Since parties access the server via a client-anonymous authenticated channel, the server never outputs the accessing party's identity.

- If the server is malicious, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ gives Ptr[id] to the adversary, where Ptr[id] = node-id is the current commit node on the history graph, which id is located at. The reason why we don't give (gid, epoch) as above is that when the server is malicious, it can arbitrary fork the group. In such a case, (gid, epoch) is not enough to identify the location of the party. Note that in the real protocol, the malicious server gets to learn which fork a party is in since the party will try to access the server using the group state. Hence, Ptr[id] correctly models what the malicious server learns in the real world. Finally, the adversary specifies the protocol result *accept*′, and it is outputted to id. This means the malicious server can decide the protocol result arbitrarily.

Create and publish commits (See Fig. 36). When a party id wants to issue a commit message, $\mathcal{Z}$ invokes $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ on input (Commit, svk). The description composes of two parts.

Fetch proposals. id must first fetch the list of proposals from the server via the subroutine FetchProposals.

- If the server is honest (i.e., ServerStat = 'good'), $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ gives (gid, epoch) to the adversary, where (gid, epoch) is the destination. Since the party accesses to the server via a client-anonymous authenticated channel, and therefore the server does not receive the party's identity. The adversary reports whether the protocol succeeds or not by setting *accept*′ to true or false, and specifies the proposals $\vec{p}'$. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ then determines the party and server's output as follows.
  - If ComDB[gid, epoch] = ($\top$, node-id) and node-id = Ptr[id], the ideal party always outputs *accept* := true and $\vec{p}$:= PropDB[gid, epoch]. This models correctnes: if a party holds the same group state used to initialize node-id, the server must accept and returns the proposals stored on the database.
  - If ComDB[gid, epoch] = ($\top$, 'adv') and *accept*′ = true, the ideal party outputs *accept* := true and $\vec{p}$:= PropDB[gid, epoch]. This models the fact that when the destination (gid, epoch) is initialized by the adversary, we let the adversary decide if the ideal server should return the stored proposals or not.
  - Otherwise, the ideal party outputs *accept* := false. This case occurs if *accept*′ = false (i.e., the adversary decides to reject) or ComDB[gid, epoch] = $\perp$ (i.e., the destination has not been initialized).

  In all three cases, the ideal server outputs the destination (gid, epoch) to $\mathcal{Z}$. Since parties access the server via a client-anonymous authenticated channel, the server never outputs the accessing party's identity.
- If the server is malicious, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ gives Ptr[id] to the adversary. The adversary specifies the protocol result *accept*′ and the proposals $\vec{p}'$, and they ate outputted to id. This means the malicious server can return an arbitrary message to the parties. This is consistent with $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ which allows an adversary to inject malicious proposals.

If id successfully fetches the proposals (i.e., id receives *accept* = true), then id creates a corresponding commit ($c_0, \vec{c}$) and welcome messages $\vec{w} = \{\widehat{w}\}$ as in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ (cf. *Committing to proposals* in App. B.2).

Publish commit and welcome messages. id finally publishes the commit and welcome messages to the server. We first describe how it publishes a commit. Before publishing the commit message, the party permutes $\vec{c}$ to $\vec{c}_{\mathsf{perm}}$ using the function *permute-commit. This effectively makes the party's identity and index of $\vec{c}$ unlikable. Note that the *permute-commit function uses a random permutation (if **safe** is true for the id's current epoch) or a permutation the adversary chooses (if **safe** is false). This means if **safe** is true the permuted index will look random from the adversary.

- If the server is honest (i.e., ServerStat = 'good'), $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ gives (gid, epoch, $c_0, \vec{c}_{\mathsf{perm}}$) to the adversary. The party accesses the server via a client-anonymous authenticated channel, and therefore, the adversary does not receive the party's identity. The adversary reports whether the protocol succeeds or not by setting *accept*′ to true or false. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ then determines the party and server's output as follows.
  - If ComDB[gid, epoch] = ($\top$, node-id) and node-id = Ptr[id], the ideal party always outputs *accept* := true. This models correctness: if a party holds the same group state used to initialize node-id, the server must accept the published commit message.
  - If ComDB[gid, epoch] = ($\top$, 'adv') and *accept*′ = true, the ideal party outputs *accept* := true. This models the fact that if the destination (gid, epoch) is initialized by the adversary, we let the adversary decide if the ideal server accepts the published commit.
  - Otherwise, the ideal party outputs *accept* := false. This case occurs if *accept*′ = false (i.e., the adversary decides to reject), ComDB[gid, epoch] ≠ ($\top$, $*$) (i.e., the destination has not been initialized or a commit message has been issued), or ComDB[gid, epoch] = ($\top$, node-id) but

node-id ≠ Ptr[id] (i.e., the party is located in a different commit node).

In all three cases, the ideal server outputs the destination (gid, epoch) and the published commit $(c_0, \vec{c}_{\mathsf{perm}})$ to $\mathcal{Z}$. Since parties access the server via a client-anonymous authenticated channel, the server never outputs the accessing party's identity.

- If the server is malicious, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ gives $(\mathsf{Ptr}[\mathsf{id}], c_0, \vec{c}_{\mathsf{perm}})$ to the adversary. The adversary specifies the protocol result *accept*′, and it is outputted to id. This means the malicious server can decide the protocol result arbitrarily.

Finally, id publishes the welcome messages $\widehat{w} \in \vec{w}$ via the subroutine `PublishWelcome`. To hide the relationship among welcome messages, id publish $\widehat{w}$ separately. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ informs the adversary of publishing a welcome message by giving $(\mathsf{id}_t, \widehat{w})$, where id_t is the intended recipient of $\widehat{w}$. Note that the party accesses the server via a client-anonymous authenticated channel, and therefore the adversary does not receive the party's identity. The adversary reports whether the protocol succeeds or not by setting *accept*′ to true or false. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ then determines the party and server's output as follows.

- If the server is honest (i.e., `ServerStat` = 'good'), $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ stores $\mathsf{WelDB}[\mathsf{id}_t] \leftarrow \widehat{w}$ and outputs *accept* := `true` to id. The ideal server outputs $(\mathsf{id}_t, \widehat{w})$. This models correctness: the honest server always accepts the welcome message and does not know who sent it.
- If the server is malicious, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ outputs *accept*′ specified by the adversary. This means the malicious server can decide the protocol result arbitrarily.

Fetch and process commits (See Fig. 36) To process, party id needs to fetch the commit and proposals from the server via the subroutine `FetchCommit`. Before fetching a commit and proposals, the party permutes its index to $\mathsf{index}_{\widehat{c}}$ by the function `*permuted-commit-index` to make the party's identity and index of $\vec{c}$ unlikable. Here, the function uses a random permutation (if **safe** is true for the id's current epoch) or a permutation the adversary chooses (if **safe** is false). This models the fact that if **safe** is true, then the permuted index seems random to the adversary.

- If the server is honest (i.e., `ServerStat` = 'good'), $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ gives $(\mathsf{gid}, \mathsf{epoch}, \mathsf{index}_{\widehat{c}})$ to the adversary, where $\mathsf{index}_{\widehat{c}}$ is the permuted id's index. Note that the party accesses to the server via a client-anonymous authenticated channel, and therefore, the adversary does not receive the party's identity. The adversary reports whether the protocol succeeds or not by setting *accept*′ to true or false, and specifies the commit message $(c_0', \widehat{c}', \vec{p}')$. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ then determines the party and server's output as follows.
  - If $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = ((c_0', \vec{c}'), \text{node-id})$ and node-id = Ptr[id], the ideal party always outputs *accept* := `true` and the commit message $(c_0, \widehat{c}, \vec{p}) := (c_0', \vec{c}'[\mathsf{index}_{\widehat{c}}], \mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}])$. This models correctness: if a party holds the same group state used to initialize node-id, the server must accept and returns the stored commit (which was received during protocol `PublishCommit`)
  - If $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = ((c_0', \vec{c}'), \text{'adv'})$ and *accept*′ = `true`, the ideal party outputs *accept* := `true` and the commit $(c_0, \widehat{c}, \vec{p}) := (c_0', \vec{c}'[\mathsf{index}_{\widehat{c}}], \mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}])$. This models the fact that if the destination (gid, epoch) was initialized by the adversary, the adversary gets to decide if the ideal server accepts.
  - Otherwise, the ideal party outputs *accept* := `false`. This case occurs if *accept*′ = `false` (i.e., the adversary decides to reject), $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] \neq ((c_0', \vec{c}'), *)$ (i.e., the destination has not been initialized or a commit message has not been issued), or $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \text{node-id})$ but node-id ≠ Ptr[id] (i.e., the party is located in a different commit node).

  In all three cases, the ideal server outputs $(\mathsf{gid}, \mathsf{epoch}, \mathsf{index}_{\widehat{c}})$ to $\mathcal{Z}$.
- If the server is malicious, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ gives $(\mathsf{Ptr}[\mathsf{id}], \mathsf{index}_{\widehat{c}})$ to the adversary. The adversary specifies the protocol result *accept*′ and the commit $(c_0', \widehat{c}', \vec{p}')$, and it is outputted to id. This means the malicious server can return an arbitrary message to the parties. This is consistent with $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ which allows an adversary to inject malicious commits and proposals.

Finally, if id successfully fetches a commit and a list of proposals (i.e., id receives *accept* = `true`), id then processes them as in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ (cf. *Processing commits* in App. B.2).

Join a group (See Fig. 36). To join a group, party id fetches a welcome message from the server via the subroutine `FetchWelcome`. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ first informs the adversary that id fetches a welcome message. In this case, the server and the adversary lean who is accessing. The adversary reports whether the protocol succeeds or not by setting *accept*′ to true or false, and specifies the welcome message $\widehat{w}'$. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ then determines the party and server's output as follows.

- If the server is honest (i.e., `ServerStat` = 'good'), $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ returns to id the welcome message $\widehat{w} := \mathsf{WelDB}[\mathsf{id}]$ stored in its database or *accept* = `false` if $\mathsf{WelDB}[\mathsf{id}] = \perp$. This models correctness: the honest server must return the message received during the subroutine `PublishWelcome`. The ideal server outputs the accessing party's identity id, which models the fact that any concrete protocol allows the server to know who was fetching the welcome message.
- If the server is malicious, the specified welcome message $\widehat{w}'$ is outputted to id. This models that the malicious server can send an arbitrary message to parties.

Finally, if id successfully fetches a welcome message (i.e., id receives *accept* = `true`), $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ then processes the welcome message as in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ (cf. *Joining a group* in App. B.2).

Group keys (See Fig. 36). Parties can fetch the current group secret via the `Key` query. The returned group secret k is random if the protocol guarantees its confidentiality (identified by the **safe** predicate). Otherwise, k is set by the adversary. Unlike $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ only provides an interface to retrieve one group secret k. That is, it does not have the interface to retrieve the group metadata secret k_{mh} and the next key function. This is because, those were special keys only used to secure the dynamic metadata (see App. E), and

in particular, $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ only requires one group secret k that will be used to exchange the actual message.

Corruption. Similar to $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$, the adversary can obtain party id's internal states via the (exposed, id) query. To prevent the so-called commitment problem, $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ fixes the value of the safety predicate **safe** when a new group or epoch is initialized. This is controlled by the function `*mark-next-db-initialized-epoch` in `Create` and `Commit`. This is because the group secret generated at a specific commit node is explicitly used in the real protocol and we must restrict the adversary to not corrupting these nodes in order not to trivially win the security game. In addition, the adversary can corrupt the server via the `CorruptServer` query, and take over the role of the server. For simplicity, we assume once the server becomes malicious, it remains malicious and will never become honest. Finally, as mentioned in the overview, we do not consider adversary-controlled randomness in the current model.

D.3 Functions Used by the Adversary

$\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ offers the publish and fetch message interfaces to the adversary so that it can impersonate an honest party to a server. Similar to the case an honest party accesses the sever, $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ defines the ideal function when the adversary (malicious party) accesses the server. Since the party is malicious, $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ never requires correctness. In contrast, $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ defines the security requirements: It defines the conditions under which the adversary can access the server. In case the conditions do not hold, the functionality models the fact that the adversary cannot access the server. Note that when both party and server are malicious, $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ does nothing since the adversary can perform all the protocols by itself.

Publish or fetch proposal or commit messages by the adversary. To publish a proposal or commit, the adversary sends `PublishProposalAdv` or `PublishCommitAdv` to $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$. $\mathcal{F}^{\text{ctxt}}_{\text{CGKA}}$ determines the server's output as follows.

- If ComDB[gid, epoch] = ($\top$, node-id) and *accept*′ = `true`, $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ checks whether authenticity is guaranteed for the honestly generated epoch node-id by the safety predicate **adv-access-allowed**(node-id). If the predicate returns `false`, then $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ halts. This models the fact that the adversary can publish messages for an honestly initialized epoch only when adversarial access is allowed. If access is allowed, the server stores the published message from the adversary in its database.
- If ComDB[gid, epoch] = ($\top$, 'adv') and *accept*′ = `true`, the ideal server outputs *accept* := `true`. This models the fact that if the destination (gid, epoch) was initialized by the adversary, the adversary gets to decide whether the ideal server accepts.
- Otherwise, the ideal server outputs *accept* := `false`. This case occurs if *accept*′ = `false` (i.e., the adversary decides to reject) or ComDB[gid, epoch] $\neq$ ($\top$, $*$) (i.e., the destination has not been initialized or a commit message has been issued).

To fetch proposals, the adversary sends `FetchProposalsAdv` to $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$. $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ determines the server's output as follows.

- If ComDB[gid, epoch] = ($*$, node-id) and *accept*′ = `true`, $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ checks whether authenticity is guaranteed for the honestly generated epoch node-id by the safety predicate **adv-access-allowed**(node-id).If the predicate returns `false`, then $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ halts. This models the fact that the adversary can fetch proposals from an honestly initialized epoch only when adversarial access is allowed. If the access is allowed, the adversary obtains $\vec{p}$:= PropDB[gid, epoch].
- If ComDB[gid, epoch] = ($*$, 'adv') and *accept*′ = `true`, the ideal server outputs *accept* := `true`. This models the fact that if the destination (gid, epoch) was initialized by the adversary, the adversary can decide to accept.
- Otherwise, the ideal party outputs *accept* := `false`. This case occurs if *accept*′ = `false` (i.e., the adversary decides to reject) or ComDB[gid, epoch] = $\perp$ (i.e., the destination has not been initialized).

Finally, to publish a proposal or a commit the adversary sends (`FetchCommitAdv`, gid, epoch, index) to $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$. $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ determines the server's output as follows.

- If ComDB[gid, epoch] = ((c_0', $\vec{c}'$), node-id) and *accept*′ = `true`, $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ checks whether authenticity is guaranteed for the honestly generated epoch node-id by the safety predicate **adv-access-allowed**(node-id). If If the predicate returns `false`, then $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ halts. This models the fact that the adversary can fetch messages for an honestly initialized epoch only when adversarial access is allowed. If access is allowed, the adversary obtains the stored (c_0, $\widehat{c}$, $\vec{p}$) := (c_0', $\vec{c}'$[index$_{\widehat{c}}$], PropDB[gid, epoch]).
- If ComDB[gid, epoch] = ((c_0', $\vec{c}'$), 'adv') and *accept*′ = `true`, the adversary obtains the stored (c_0, $\widehat{c}$, $\vec{p}$) := (c_0', $\vec{c}'$[index$_{\widehat{c}}$], PropDB[gid, epoch]). This models the fact that if the destination (gid, epoch) was initialized by the adversary, the adversary can decide to accept.
- Otherwise, the ideal party outputs *accept* := `false`. This case occurs if *accept*′ = `false` (i.e., the adversary decides to reject) or ComDB[gid, epoch] $\neq$ ((c_0', $\vec{c}'$), $*$) (i.e., the destination has not been initialized or a commit message has not been issued).

Publish or fetch welcome messages by the adversary. To publish or fetch messages, the adversary sends `PublishWelcome` or `FetchWelcome` to $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$. In this case, $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ determines the server's output as in the case an honest party accesses the server. The protocol succession means the adversary succeeds to publish or fetch a welcome message.

Initialization

/ Line 1 to 7 below are identical to selective downloading $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$
Ptr[∗], Prop[∗], Node[∗], Wel[∗] ← ⊥
propCtr, nodeCtr ← 1
$\mathtt{flag}_{\mathsf{selDL}}$ = true, DesignatedCom[∗] ← ⊥
$\mathtt{flag}_{\mathsf{contHide}}$ = true
PropID[∗], NodeID[∗] ← ⊥
RandCorr[∗] ← 'good'
/ Initialize below for dynamic metadata-hiding
$\mathtt{flag}_{\mathsf{dbinit}}$ = true
PropDB[∗, ∗], ComDB[∗, ∗], WelDB ← ⊥
ServerStat ← 'good'

Input from the party $\mathsf{id}_{\mathsf{creator}}$

Input (Create, svk)

req Ptr[$\mathsf{id}_{\mathsf{creator}}$] = ⊥
/ Run identical code as Input (Create, svk) of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$,
/ excluding the final **return** (gid, 0, mem) line
⟪Insert (Create, svk) of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$⟫
req RegisterGroup(gid, 0)
∗mark-next-db-initialized-epoch(Ptr[$\mathsf{id}_{\mathsf{creator}}$])

Subroutine RegisterGroup(gid, epoch)

channelType ← ('anon', ⊥, Sv)
req Ptr[id] = 0
Send (channelType, RegisterGroup, gid, 0) to $\mathcal{S}$ and
  receive $accept'$
/ If Sv is good, then accept if ComDB[gid, ∗] is empty
if ServerStat = 'good' **then**
  if ComDB[gid, ∗] = ⊥ **then**
    $accept$ ← true
    / Main group is assigned to node-id = 0. ComDB[gid, 0] ← (⊤, 0)
  else
    $accept$ ← false
  Send (channelType, RegisterGroup, ($accept$, gid, 0)) to Sv
/ If Sv is corrupt, then let $\mathcal{S}$ decide if Sv accepts
else
  $accept$ ← $accept'$
return $accept$

Input Key

/ Run identical code as Input (Key) of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$,
/ including the final **return** Node[Ptr[id]].key line
⟪Inset (Key) of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$⟫

Inputs from a party id

Input (Propose, act), act ∈ { 'upd'-svk, 'add'-id_t, 'rem'-id_t }

req Ptr[id] ≠ ⊥
gid ← Node[Ptr[id]].gid; epoch ← Node[Ptr[id]].epoch
/ Run identical code as Input (Propose, act) of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$,
/ excluding the final **return** p line
⟪Insert (Propose, act) of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$⟫
req PublishProposal(gid, epoch, p)
return p

Input (Commit, svk)

req Ptr[id] ≠ ⊥
gid ← Node[Ptr[id]].gid; epoch ← Node[Ptr[id]].epoch
($accept$, $\vec{p}$) ← FetchProposals(gid, epoch)
req $accept$
/ Run identical code as Input (Commit, $\vec{p}$, svk) of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$,
/ excluding the final **return** (c_0, $\vec{c}$, $\vec{w}$ = { $\widehat{w}$ }) line
⟪Insert (Commit, $\vec{p}$, svk) of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$⟫
/ Permute the order of party sensitive commitment $\vec{c}$
$\vec{c}_{\mathsf{perm}}$ ← ∗permute-commit(Ptr[id], $\vec{c}$)
try PublishCommit(gid, epoch, c_0, $\vec{c}_{\mathsf{perm}}$)
foreach $\widehat{w}$ ∈ $\vec{w}$ **do**
  parse (id_t, ∗) ← $\widehat{w}$
  try PublishWelcome(id_t, $\widehat{w}$)
∗mark-next-db-initialized-epoch(NodeID[c_0])
return (c_0, $\vec{c}_{\mathsf{perm}}$, $\vec{w}$)

Input Process

req Ptr[id] ≠ ⊥
gid ← Node[Ptr[id]].gid; epoch ← Node[Ptr[id]].epoch
/ Permuted index of party id required for selective downloading.
$\mathsf{index}_{\widehat{c}}$ ← ∗permuted-commit-index(Ptr[id], id)
($accept$, c_0, $\widehat{c}$, $\vec{p}$) ← FetchCommit(gid, epoch, $\mathsf{index}_{\widehat{c}}$)
req $accept$
/ Run identical code as Input (Process, c_0, $\widehat{c}$, $\vec{p}$) of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$,
/ including the final **return** ∗output-proc(node-id′) line
⟪Insert (Process, c_0, $\widehat{c}$, $\vec{p}$) of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$⟫

Input Join

req Ptr[id] = ⊥
($accept$, $\widehat{w}$) ← FetchWelcome(id)
req $accept$
/ Run identical code as Input (Join, $\widehat{w}$) of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$,
/ including the final **return** ∗output-join(node-id) line
⟪Insert (Join, $\widehat{w}$) of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$⟫

Figure 36: The ideal MH-CGKA functionality $\mathcal{F}^{\mathsf{mh}}_{\mathbf{CGKA}}$: `Create`, `Propose`, `Commit`, `Process`, and `Join` interface for honest parties. For better readability, we outsource the lines identical to $\mathcal{F}^{\mathsf{ctxt}}_{\mathbf{CGKA}}$ to Fig. 13. Note that the `Create` function is invoked only once by the previously designated party $\mathsf{id}_{\mathsf{creator}}$. (Only one $\mathsf{id}_{\mathsf{creator}}$ exists.)

Subroutine PublishProposal(gid, epoch, p)

channelType ← ('anon', ⊥, Sv) / Connect to the server via anonymous channel.
if ServerStat = 'good' **then**
  Send (channelType, PublishProposal, gid, epoch, p) to $\mathcal{S}$ and receive $accept'$
  / If the party assigned to gid-epoch accesses, the server must accept.
  if ComDB[gid, epoch] = (⊤, node-id) ∧ node-id = Ptr[id] **then**
    PropDB[gid, epoch] +← p
    $accept$ ← true
  / If gid-epoch was initialized by the adversary, $\mathcal{S}$ decide to accept or reject.
  elseif ComDB[gid, epoch] = (⊤, 'adv') ∧ $accept'$ **then**
    PropDB[gid, epoch] +← p
    $accept$ ← true
  / Otherwise, the server must reject.
  else
    $accept$ ← false
  Send (channelType, PublishProposal, ($accept$, gid, epoch, p)) to Sv
/ If Sv is corrupt, then $\mathcal{S}$ decides if Sv accepts.
else
  Send (channelType, PublishProposal, Ptr[id], p) to $\mathcal{S}$ and receive $accept'$
  $accept$ ← $accept'$
return $accept$

Subroutine FetchProposals(gid, epoch)

channelType ← ('anon', ⊥, Sv) / Connect to the server via anonymous channel.
if ServerStat = 'good' **then**
  Send (channelType, FetchProposals, gid, epoch) to $\mathcal{S}$ and receive $accept'$
  / If the party assigned to gid-epoch accesses, the server must accept.
  if ComDB[gid, epoch] = (∗, node-id) ∧ node-id = Ptr[id] **then**
    ($accept$, $\vec{p}$) ← (true, PropDB[gid, epoch])
  / If gid-epoch was initialized by the adversary, $\mathcal{S}$ decide to accept or reject.
  elseif ComDB[gid, epoch] = (∗, 'adv') ∧ $accept'$ **then**
    ($accept$, $\vec{p}$) ← (true, PropDB[gid, epoch])
  / Otherwise, the server must reject.
  else
    ($accept$, $\vec{p}$) ← (false, ⊥)
  Send (channelType, FetchProposals, ($accept$, gid, epoch)) to Sv
/ If Sv is corrupt, then $\mathcal{S}$ decides what Sv returns to id.
else
  Send (channelType, FetchProposals, Ptr[id]) to $\mathcal{S}$ and receive ($accept'$, $\vec{p}'$)
  ($accept$, $\vec{p}$) ← ($accept'$, $\vec{p}'$)
return ($accept$, $\vec{p}$)

Subroutine PublishCommit(gid, epoch, c_0, $\vec{c}_{\mathsf{perm}}$)

channelType ← ('anon', ⊥, Sv) / Connect to the server via anonymous channel.
if ServerStat = 'good' **then**
  Send (channelType, PublishCommit, gid, epoch, c_0, $\vec{c}_{\mathsf{perm}}$) to $\mathcal{S}$ and receive $accept'$
  / If the party uses the state assigned to gid-epoch, the server must accept.
  if ComDB[gid, epoch] = (⊤, node-id) ∧ node-id = Ptr[id] **then**
    ComDB[gid, epoch] ← ((c_0, $\vec{c}_{\mathsf{perm}}$), node-id)
    / If an honest party initialize the next epoch, the corresponding node is marked.
    ComDB[gid, epoch + 1] ← (⊤, NodeID[c_0])
    $accept$ ← true
  / If gid-epoch was initialized by the adversary, $\mathcal{S}$ decide to accept or reject.
  elseif ComDB[gid, epoch] = (⊤, 'adv') ∧ $accept'$ **then**
    ComDB[gid, epoch] ← ((c_0, $\vec{c}_{\mathsf{perm}}$), 'adv')
    / If an honest party initialize the next epoch, the corresponding node is marked.
    ComDB[gid, epoch + 1] ← (⊤, NodeID[c_0])
    $accept$ ← true
  / Otherwise, the server must reject.
  else
    $accept$ ← false
  Send (channelType, PublishCommit, ($accept$, gid, epoch, c_0, $\vec{c}_{\mathsf{perm}}$)) to Sv
/ If Sv is corrupt, then $\mathcal{S}$ decides if Sv accepts.
else
  Send (channelType, PublishCommit, gid, epoch, c_0, $\vec{c}_{\mathsf{perm}}$) to $\mathcal{S}$ and receive $accept'$
  $accept$ ← $accept'$
return $accept$

Subroutine FetchCommit(gid, epoch, index$_{\widehat{c}}$)

channelType ← ('anon', ⊥, Sv) / Connect to the server via anonymous channel.
if ServerStat = 'good' **then**
  Send (channelType, FetchCommit, gid, epoch, index$_{\widehat{c}}$) to $\mathcal{S}$ and receive $accept'$
  / If the party assigned to gid-epoch accesses, the server must accept.
  if ComDB[gid, epoch] = ((c_0', $\vec{c}'$), node-id) ∧ node-id = Ptr[id] **then**
    ($accept$, c_0, $\widehat{c}$, $\vec{p}$) ← (true, c_0', $\vec{c}'$[index$_{\widehat{c}}$], PropDB[gid, epoch])
  / If gid-epoch was initialized by the adversary, $\mathcal{S}$ decide to accept or reject.
  elseif ComDB[gid, epoch] = ((c_0', $\vec{c}'$), 'adv') ∧ $accept'$ **then**
    ($accept$, c_0, $\widehat{c}$, $\vec{p}$) ← (true, c_0', $\vec{c}'$[index$_{\widehat{c}}$], PropDB[gid, epoch])
  / Otherwise, the server must reject.
  else
    ($accept$, c_0, $\widehat{c}$, $\vec{p}$) ← (false, ⊥, ⊥, ⊥)
  Send (channelType, FetchCommit, ($accept$, gid, epoch, index$_{\widehat{c}}$) to Sv
else
  Send (channelType, FetchCommit, Ptr[id], index$_{\widehat{c}}$) to $\mathcal{S}$ and receive ($accept'$, c_0', $\widehat{c}'$, $\vec{p}'$)
  ($accept$, c_0, $\widehat{c}$, $\vec{p}$) ← ($accept'$, c_0', $\widehat{c}'$, $\vec{p}'$)
return ($accept$, c_0, $\widehat{c}$, $\vec{p}$)

Figure 37: The ideal metadata-hiding CGKA functionality $\mathcal{F}^{\mathsf{mh}}_{\mathsf{CGKA}}$: subroutines for publish and fetch proposal and commit messages. They are used in `Propose`, `Commit` and `Process` interface shown in Fig. 36. If $\vec{c} = \bot$, we define $\vec{c}$[index] = ⊥ for any index index**.**

Input from a party id **or adversary** $\mathcal{S}$

Input `PublishWelcome`$(\mathsf{id}_t, \widehat{w})$

/ Connect to the server via anonymous channel.
channelType $\leftarrow$ ('anon', $\perp$, Sv)
Send (channelType, `PublishWelcome`, id_t, $\widehat{w}$) to $\mathcal{S}$ and receive $accept'$
/ If Sv is honest, then store $\widehat{w}$ in WelDB
if `ServerStat` = 'good' **then**
  $accept \leftarrow$ `true`
  WelDB[id_t] $\leftarrow \widehat{w}$
  Send (channelType, `PublishWelcome`, ($accept$, id_t, $\widehat{w}$)) to Sv
/ If Sv is corrupt, then $\mathcal{S}$ decides if Sv accepts.
else
  $accept \leftarrow accept'$
/ If invoked by $\mathcal{S}$, then no output is required
if `ServerStat` = 'good' $\vee$ invoked by party id **then**
  return $accept$

Input `FetchWelcome`(id)

/ Connect to the server via authenticated channel.
channelType $\leftarrow$ ('auth', id, Sv)
Send (channelType, `FetchWelcome`) to $\mathcal{S}$ and receive ($accept'$, $\widehat{w}'$)
if `ServerStat` = 'good' **then**
  if WelDB[id] $\neq \perp$ **then**
    ($accept$, $\widehat{w}$) $\leftarrow$ (`true`, WelDB[id])
  else
    ($accept$, $\widehat{w}$) $\leftarrow$ (`false`, $\perp$)
  Send (channelType, `FetchWelcome`, ($accept$, id)) to Sv
/ If Sv is corrupt, then let $\mathcal{S}$ decide Sv's output
else
  ($accept$, $\widehat{w}$) $\leftarrow$ ($accept'$, $\widehat{w}'$)
/ If invoked by $\mathcal{S}$, then no output is required
if `ServerStat` = 'good' $\vee$ invoked by party id
  return ($accept$, $\widehat{w}$)

Figure 38: The ideal metadata-hiding CGKA functionality $\mathcal{F}^{\mathsf{mh}}_{\mathbf{CGKA}}$: Subroutines for publish and fetch welcome messages. They are used in `Commit` and `Join` interface shown in Fig. 36.

`*permuted-commit-index`(node-id, id)

assert Node[node-id] $\neq \perp$
/ If $\mathtt{flag}_{\mathsf{selDL}}$ is false, there is no need to permute index.
if $\neg\mathtt{flag}_{\mathsf{selDL}}$ **then**
  return $\perp$
mem $\leftarrow$ Node[node-id].mem
assert id $\in$ mem
index $\leftarrow$ Node[node-id].index(id)
if Node[node-id].perm $= \perp$ **then**
  `*set-index-permutation`(node-id)
$\phi \leftarrow$ Node[node-id].perm
return ϕ(index)

`*permute-commit`(node-id, $\vec{c}$)

assert Node[node-id] $\neq \perp$
/ If $\mathtt{flag}_{\mathsf{selDL}}$ is false, no party-dependent commitment can exist.
if $\neg\mathtt{flag}_{\mathsf{selDL}}$ **then**
  assert $\vec{c} = \perp$
  return $\perp$
if Node[node-id].perm $= \perp$ **then**
  `*set-index-permutation`(node-id)
$\phi \leftarrow$ Node[node-id].perm
$\vec{c}_{\mathsf{perm}} \leftarrow ()$
for index $= 1, \ldots, |\vec{c}|$ **do**
  $\vec{c}_{\mathsf{perm}} \mathbin{+\!\!\leftarrow} \vec{c}[\phi(\text{index})]$
return $\vec{c}_{\mathsf{perm}}$

`*set-index-permutation`(node-id)

mem $\leftarrow$ Node[node-id].mem
/ If the input node is not corrupted,
/ sample a random permutation over $S_{|\mathsf{mem}|}$.
if random-index(node-id) **then**
  $\phi \leftarrow\!\!\$\ S_{|\mathsf{mem}|}$
  Node[node-id].perm $\leftarrow \phi$
else
  Send (`Permutation`, node-id) to $\mathcal{S}$ and receive ϕ
  assert $\phi \in S_{|\mathsf{mem}|}$
  Node[node-id].perm $\leftarrow \phi$

Figure 39: Helper function: Permute indices. $S_{|\mathsf{mem}|}$ is the set of permutations on the range $[|\mathsf{mem}|]$.

Input from the adversary $\mathcal{S}$

Input (PublishProposalAdv, gid, epoch, p)

channelType ← ('anon', ⊥, Sv)
Send (channelType, PublishProposalAdv, gid, epoch, p) to $\mathcal{S}$ and receive *accept*′
/ The honest server accepts if $\mathcal{S}$ decides to accept and ComDB[gid, epoch] = (⊤, ∗).
if ServerStat = 'good' **then**
  if ComDB[gid, epoch] = (⊤, node-id) ∧ *accept*′ **then**
    / If gid-epoch was initialized by an honest party, $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ checks authenticity is guaranteed.
    assert adv-access-allowed(node-id)
    PropDB[gid, epoch] ↞ p
    accept ← true
  elseif ComDB[gid, epoch] = (⊤, 'adv') ∧ *accept*′ **then**
    PropDB[gid, epoch] ↞ p
    accept ← true
  else
    accept ← false
  Send (channelType, PublishProposal, (*accept*, gid, epoch, p)) to Sv
  return *accept*

Input (FetchProposalsAdv, gid, epoch)

channelType ← ('anon', ⊥, Sv)
Send (channelType, FetchProposalsAdv, gid, epoch) to $\mathcal{S}$ and receive *accept*′
/ The honest server accepts if $\mathcal{S}$ decides to accept and ComDB[gid, epoch] ≠ ⊥.
if ServerStat = 'good' **then**
  elseif ComDB[gid, epoch] = (∗, node-id) ∧ *accept*′ **then**
    / If gid-epoch was initialized by an honest party, $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ checks authenticity is guaranteed.
    assert adv-access-allowed(node-id)
    (*accept*, $\vec{p}$) ← (true, PropDB[gid, epoch])
  elseif ComDB[gid, epoch] = (∗, 'adv') ∧ *accept*′ **then**
    (*accept*, $\vec{p}$) ← (true, PropDB[gid, epoch])
  else
    (*accept*, $\vec{p}$) ← (false, ⊥)
  Send (channelType, FetchProposals, (*accept*, gid, epoch) to Sv
  return (*accept*, $\vec{p}$)

Input (PublishCommitAdv, gid, epoch, $c_0, \vec{c}$)

channelType ← ('anon', ⊥, Sv)
Send (channelType, PublishCommitAdv, gid, epoch, $c_0, \vec{c}$) to $\mathcal{S}$ and receive *accept*'
/ The honest server accepts if $\mathcal{S}$ decides to accept and ComDB[gid, epoch] = (⊤, ∗).
if ServerStat = 'good' **then**
  ComDB[gid, epoch]
  elseif ComDB[gid, epoch] = (⊤, node-id) ∧ *accept*′ **then**
    / If gid-epoch was initialized by an honest party, $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ checks authenticity is guaranteed.
    assert adv-access-allowed(node-id)
    ComDB[gid, epoch] ← (($c_0, \vec{c}$), node-id)
    / If injection succeeds, the next epoch is marked as adversarial initialized.
    ComDB[gid, epoch + 1] ← (⊤, 'adv')
  elseif ComDB[gid, epoch] = (⊤, 'adv') ∧ *accept*′ **then**
    ComDB[gid, epoch] ← (($c_0, \vec{c}$), 'adv')
    / If injection succeeds, the next epoch is marked as adversarial initialized.
    ComDB[gid, epoch + 1] ← (⊤, 'adv')
  else
    accept ← false
  Send (channelType, PublishCommit, (*accept*, gid, epoch, $c_0, \vec{c}$)) to Sv
  return *accept*

Input (FetchCommitAdv, gid, epoch, index)

channelType ← ('anon', ⊥, Sv)
Send (channelType, FetchCommitAdv, gid, epoch, index) to $\mathcal{S}$ and receive *accept*′
/ The honest server accepts if $\mathcal{S}$ decides to accept and ComDB[gid, epoch] = (($c_0', \vec{c}'$), ∗).
if ServerStat = 'good' **then**
  if ComDB[gid, epoch] = (($c_0', \vec{c}'$), node-id) ∧ *accept*′ **then**
    / If gid-epoch was initialized by an honest party, $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$ checks authenticity is guaranteed.
    assert adv-access-allowed(node-id)
    (*accept*, $c_0, \widehat{c}, \vec{p}$) ← (true, $c_0', \vec{c}'$[index], PropDB[gid, epoch])
  if ComDB[gid, epoch] = (($c_0', \vec{c}'$), 'adv') ∧ *accept*′ **then**
    (*accept*, $c_0, \widehat{c}, \vec{p}$) ← (true, $c_0', \vec{c}'$[index], PropDB[gid, epoch])
  else
    (*accept*, $c_0, \widehat{c}, \vec{p}$) ← (false, ⊥, ⊥, ⊥)
  Send (channelType, PublishProposal, (*accept*, gid, epoch, index) to Sv
  return (*accept*, $c_0, \widehat{c}, \vec{p}$)

Figure 40: The ideal metadata-hiding CGKA functionality $\mathcal{F}^{\text{mh}}_{\text{CGKA}}$: Interface for the adversary $\mathcal{S}$. If $\vec{c} = \bot$, $\vec{c}$[index] is defined to be ⊥ for any index**.**

Input (Expose, id)

1 : **if** Ptr[id] ≠ ⊥ **then**
2 : Node[Ptr[id]].exp +← id
3 : *update-stat-after-exp(id) / Pending secrets are marked as exposed.
4 : svk ← Node[Ptr[id]].mem[id]
5 : Send (exposed, id, svk) to $\mathcal{F}_{\mathrm{AS}}$
6 : Send (Ptr[id], Node[Ptr[id]]) to $\mathcal{S}$ / All information stored in Node[Ptr[id]] is sent to $\mathcal{S}$.
7 : Send (exposed, id) to $\mathcal{F}_{\mathrm{KS}}$
8 : **restrict** ∀node-id :
9 : **if** Node[node-id].chall = true **then safe**(node-id) = true
10 : **if** Node[node-id].conthide = true **then safe**(node-id) = true
11 : **if** Node[node-id].dbinit = true **then safe**(node-id) = true

Input CorruptServer

1 : / Once corrupted, the server remains corrupted.
2 : ServerStat ← 'adv'

*mark-next-db-initialized-epoch(node-id)

1 : **if safe**(node-id) **then**
2 : Node[node-id].dbinit ← true
3 : **else**
4 : Node[node-id].dbinit ← false

Figure 41: The metadata-hiding CGKA functionality $\mathcal{F}^{\mathsf{mh}}_{\mathbf{CGKA}}$: Corruptions from the adversary $\mathcal{S}$. The difference between those of $\mathcal{F}^{\mathsf{ctxt}}_{\mathbf{CGKA}}$ is highlighted in yellow. $\mathcal{S}$ can call CorruptServer only once.

E METADATA-HIDING CGKA: CONSTRUCTION AND SECURITY PROOF

E.1 Constructing the Wrapper Protocol W^{mh}

In this section, we propose a metadata CGKA W^{mh} and prove that it UC-realizes $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ in the $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$-hybrid model. We call W^{mh} as a *wrapper* protocol since it works as a wrapper around any static metadata-hiding CGKA that UC-realizes $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$, and turns it into full metadata-hiding CGKA. In particular, the sole functionality of W^{mh} is to take care of how the proposals, commits, and welcome messages are uploaded and downloaded from the server in a dynamic metadata-hiding manner. The construction is very simple and can be implemented only from a standard signature scheme.

G.gid	The identifier of the group.
G.epoch	The current epoch number.
G.mem	A list of (id, svk)-pair. The list is sorted in lexicographic order by ids.
(G.gsk, G.gvk)	The group signature key used to authenticate group membership to the server.
G.permKey	The PRP key used to permute member index in membership list.
G.indexOf(id)	Returns the index of id in the list G.mem.

Table 9: The party's protocol state and helper method.

PropDB[$*$, $*$]	It stores a list of proposal message p issued at (gid, epoch).
ComDB[$*$, $*$]	It stores a group signature key gvk used to authenticate membership on (gid, epoch) and possibly a commit message $(c_0, \vec{c})$ to move to (gid, epoch).
WelDB[$*$]	It stores a welcome message $\widehat{w}$ for id.

Table 10: The server's protocol state.

E.1.1 Protocol States. Each party holds a group state G. It consists of the variables listed in Tab. 9. The G.mem list stores the group member's identity and its signature key. The list is sorted in lexicographic order by the party's identity. Parties can fetch the index of their identities in G.mem via the method G.indexOf($*$).

The server keeps three databases: PropDB[$*$, $*$], ComDB[$*$, $*$], and WelDB[$*$]. PropDB[$*$, $*$] and ComDB[$*$, $*$] both use (gid, epoch) as indices to store proposals and commits, respectively. We refer the readers to Sec. 4.2 for a pictorial example, where we merge the PropDB and ComDB into one database in the figures.

- PropDB[gid, epoch] = $\vec{p}$: The proposal database stores the list of proposals $\vec{p}$ created at (gid, epoch). These proposals, once committed, are used to move any party at epoch to the next epoch$'$ = epoch + 1, where the group state is updated accordingly to the proposals.
- ComDB[gid, epoch] = (gvk, $c_0, \vec{c}$) or (gvk, $\perp, \perp$): The commit database stores a group signing key (also called a group statement) gvk. This will be used by a group member at epoch to anonymously prove that they are indeed a group member. ComDB[gid, epoch] is initialized with (gvk, $\perp, \perp$) and is later updated to (gvk, $c_0, \vec{c}$) when some party creates a commit $(c_0, \vec{c})$ with the proposals stored in PropDB[gid, epoch]. Once ComDB[gid, epoch] has a commit stored, then no other commits can be made at this epoch.
- WelDB[id] = $\widehat{w}$: It stores a welcome message for id. We assume WelDB[id] stores only one welcome message for id. This is due to the fact that previous CGKA UC-security models assume a party can join at most one group (i.e., Ptr[id] identifies the unique group that id is a member of). In our protocol, when the server receives a new welcome message (id, $\widehat{w}$), it overwrites it as WelDB[id] := $\widehat{w}$. We emphasize that this restriction is required only to prove UC-security,

and functionality-wise, WelDB[id] can store as many welcome messages as it needs.

E.1.2 Protocol Algorithms. The main interface and the associated helper functions are depicted in Fig. 42. The protocol W^{mh} is defined in the $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$-hybrid model and internally calls $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ to create a group, generate and process protocol messages. To register groups and publish or fetch protocol messages, it uses the subroutines depicted in Figs. 43 to 49.

Group Creation. A group can be created by the group creator $\mathsf{id}_{\mathsf{creator}}$ by invoking $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ on input (Create, svk). The group creator first invokes $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ with the same input. Then, it runs the helper function *init-states to initialize the group state of the wrapper protocol. Note that *init-states internally queries ($\mathsf{Key}_{\mathsf{mh}}$) to $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ to use the group secret k_{mh}. Finally, it executes the group registration protocol RegisterGroup shown in Fig. 43. The group creator sends the new group's group identity gid, epoch counter epoch, and group signature key gvk via a client-anonymous authenticated channel. Upon receiving the group creation message, the server checks that a group with the same identity does not exist and the epoch counter is equal to 0. If the check passes, the server stores (gvk, $\perp$, $\perp$) in ComDB[gid, 0]. Finally, the server notifies the party of the success or failure of the group creation. The group creator checks the protocol result. If group creation fails, it unwinds all state changes and outputs $\perp$.

Proposal. A party located at (gid, epoch) can generate a proposal message p by invoking $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ on input (Propose, act). Then, it executes the publish proposal protocol PublishProposal shown in Fig. 44, where party accesses the server via a client-anonymous authenticated channel. The party and the server perform a challenge-response type *membership authentication protocol*, that allows a party to anonymously prove that it is a valid group member for (gid, epoch). In more detail, upon receiving the PublishProposal proposal message, the server chooses a random challenge message $\mathsf{ch} \leftarrow \{0,1\}^{\kappa}$ and sends it to the party. The party then signs the challenge with its group signing key G.ssk and sends the signature σ along with the destination (gid, epoch) and the proposal p. The server checks that ComDB[gid, epoch] = (gvk, $\perp$, $\perp$) and the signature σ is valid with respect to gvk. If the check passes, the server stores the proposal p in PropDB[gid, epoch]. Note that the server rejects the proposal if ComDB[gid, epoch] contains a commit since this indicates that a new epoch has been created. We later show in App. E.2 that the above membership authentication protocol can be made non-interactive if we allow the server to perform an additional simple check on the database.

Commit. A party located at (gid, epoch) can perform a commit by invoking $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ on input (Commit, svk). The party first executes the subroutine FetchProposals shown in Fig. 45 to download the list of proposals created in (gid, epoch). FetchProposals consists of a challenge-response type membership authentication protocol almost identical to PublishProposal explained above. Once the server accepts, it is convinced that the calling anonymous party is indeed a valid group member at epoch so it sends the list of proposals $\vec{p}$ stored in the database PropDB[gid, epoch].

Once the party succeeds in fetching the proposals $\vec{p}$, it generates a commit and welcome messages $(c_0, \vec{c}, \vec{w})$ by invoking $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ on input (Commit, svk, $\vec{p}$). The list $\vec{c}$ is then randomly permuted to $\vec{c}_{\mathsf{perm}}$ by *permute-commit. This procedure allows for a shuffle of the order of the member for each new epoch and makes the selective downloading performed during FetchCommit unlinkable between different epochs. The party then publishes $(c_0, \vec{c}_{\mathsf{perm}}, \vec{w})$ by performing two different uploads. It first executes PublishCommit in Fig. 46 to publish the commit $(c_0, \vec{c}_{\mathsf{perm}})$. The party accesses the server via a client-anonymous authenticated channel and performs the authentication protocol with the server similar to PublishProposal. After generating the response signature σ, the party further generates a group signature key $(\mathsf{gvk}', \mathsf{gsk}')$ for the next epoch. This is generated from the group secret $\mathsf{k}_{\mathsf{mh}}{}'$ at the next epoch, which can be obtained by querying $(\mathsf{NextKey}_{\mathsf{mh}}, c_0)$ to $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$. The party finally sends $(\sigma, \mathsf{gid}, \mathsf{epoch}, c_0, \vec{c}_{\mathsf{perm}}, \mathsf{gvk}')$ to the server. If the signature σ is a valid signature with respect to the group signing key stored in ComDB[gid, epoch] = (gvk, $\perp$, $\perp$), then it updates the database by ComDB[gid, epoch] $\leftarrow$ (gvk, $c_0, \vec{c}_{\mathsf{perm}}$). In case a commit was already stored, the server rejects the commit. Moreover, the server initializes a new entry in the database as ComDB[gid, epoch+1] $\leftarrow$ (gvk′, $\perp$, $\perp$). This creates a new epoch to which the parties can upload new proposals. The server returns whether it succeeded or not to the party.

If the party succeeds to publish the commit and welcome messages exit (i.e., $\vec{w} \neq \emptyset$), it then further executes PublishWelcome shown in Fig. 48 for each $\widehat{w} \in \vec{w}$. The party accesses the server via a client-anonymized authenticated channel and uploads $(\mathsf{id}_t, \widehat{w})$, where id_t is extracted from $\widehat{w}$. The server stores $\widehat{w}$ in WelDB[id_t]. Here, there is no membership authentication protocol.

Process. A party located at (gid, epoch) can try to process the current commit and proposals by invoking $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ on input (Process). The party first executes the FetchCommit subroutine shown in Fig. 47 to download a commit and their associating proposals. The FetchCommit protocol is similar to the FetchProposals protocol. The party and server engage in the membership authentication protocol explained above. Notably, the party sends an index that specifies the index of the party-dependent commit the party wants to download. Since the party sends an epoch-dependent permuted index using the function *permuted-commit-index, the index from different epochs remain unlinkable. If the server accepts the party, it returns the list of proposals $\vec{p}$ stored on the database PropDB[gid, epoch], the party independent commit c_0, and the party dependent commit $\widehat{c} := \vec{c}_{\mathsf{perm}}[\mathsf{index}]$ stored on the database ComDB[gid, epoch].

Upon fetching the content $(c_0, \widehat{c}, \vec{p})$, the party processes them by invoking $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ on input $(c_0, \widehat{c}, \vec{p})$. Finally, it also updates the internal state by calling *update-states function.

Join. A party can join a group by invoking $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ on input (Join). The party first executes the FetchWelcome subroutine shown in Fig. 49 to download a welcome message. The party accesses the server via a standard authenticated channel (i.e., the party discloses its identity id). This is necessary for the server to identify which welcome message to provide. The server either returns the welcome message $\widehat{w}$ designated to id or notifies that there is no welcome message. Once the party receives the welcome message $\widehat{w} \neq \perp$, it processes $\widehat{w}$ by invoking $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ on input (Join, $\widehat{w}$). Finally, it setups the initial state by calling *init-states function.

E.2 Making Membership Authentication Protocol Non-Interactive

We discuss two simple ways to make the membership authentication protocol ran to publish and fetch the contents from the server non-interactive.

Recall that when the group member tries to access the server, they always engage in a membership authentication protocol to prove anonymously to the server that they are indeed a valid group member. Our W^{mh} protocol realized this membership authentication protocol in an *interactive* challenge-response type manner using a digital signature scheme. The server sends a challenge and asks the group member to sign the challenge. Assuming that the same challenge is never reused, this allowed the server to be convinced that the communicating party indeed has a group signing key.

We can remove this interaction between the server in some scenarios by asking the server to perform an extra check on the database. Namely, during the membership authentication protocol run during the protocols `PublishProposal` and `PublishCommit`, we allow the parties to sign on to the content they wish to upload to the server, rather than being provided a challenge from the server. For instance, in the protocol `PublishProposal`, the party id signs the message $(\mathsf{gid}, \mathsf{epoch}, p)$. The server checks if the signature is valid with respect to the group signing key at $(\mathsf{gid}, \mathsf{epoch})$, and additionally checks if such p is included in the proposal database $\mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}]$. If all check passes, it updates the database with p. Notice that the server did not need to check if p was in the database in the previous interactive protocol. This non-interactive variant securely realizes the desired functionality since due to the EUF-CMA security of the signature scheme, a non-group member cannot upload any proposals that haven't been signed. The only possible attack would be to resend the observed signature-proposal pair to the server. However, since the server is modified to never accept the same proposal, this attack fails. The same idea can be used to make the protocol `PublishCommit` non-interactive.

Unfortunately, it is not clear how to make the protocols `FetchProposals` and `FetchCommit` non-interactive using the above idea. This is because the party does not have any contents to sign when it is performing a fetch/download. A potential idea is to weaken our UC-security model to allow non-group members to fetch/download the contents from the server, while still disallowing them to publish/upload any contents on the server. In particular, the protocols `FetchProposals` and `FetchCommit` will simply consist of a party querying the server $(\mathsf{gid}, \mathsf{epoch})$, and the server responding with the proposals and commits on the database without checking membership. This seems like a reasonable compromise considering that the contents uploaded on the server can be provided to an adversary without compromising the security of the group — this follows since the contents are commits and proposals generated from $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$, which by definition secures the group secret key and static metadata. One possible issue is that without any authentication on the fetch/download, it will allow an adversary to learn if a group with the group identifier gid exists. Our interactive protocol does not leak such information since it will output *accept* = `false` when authentication fails.

In summary, by reasonably weakening the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ in a meaningful way, we can make the wrapper protocol W^{mh} fully non-interactive. It remains an interesting future work to investigate whether this weakening has any practical impact on the protocol.

E.3 Security of the Wrapper Protocol W^{mh}

The following theorem proves that the wrapper protocol W^{mh} UC-realizes the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ in the $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$-hybrid model.

Theorem E.1. *Assuming that* SIG′ *is* EUF-CMA *secure,* PRF *is a secure pseudorandom function, and* PRP *is a secure pseudorandom permutation, the protocol W^{mh} UC-realizes the ideal functionality $\mathcal{F}_{CGKA}^{\mathsf{mh}}$ in the $\mathcal{F}_{CGKA}^{\mathsf{ctxt}}$-hybrid model, where the safety predicates and leakage functions for $\mathcal{F}_{CGKA}^{\mathsf{mh}}$ are defined in Figs. 34, 35 and 50.*

Proof. We consider the following sequence of hybrids. While the environment $\mathcal{Z}$ interacts with W^{mh} in Hybrid 1, it interacts with the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ in Hybrid 6. Below, we first define all the hybrids and then explain how the simulators are defined.

Hybrid 1. This is the real-world execution of the protocol, where we make a syntactic change. We consider a simulator $\mathcal{S}_1$ that interacts with a dummy functionality $\mathcal{F}_{\mathsf{dummy}}$. $\mathcal{F}_{\mathsf{dummy}}$ sits between the environment $\mathcal{Z}$ and $\mathcal{S}_1$, and simply routs all messages without any modification. $\mathcal{S}_1$ internally simulates the real-world parties and adversary $\mathcal{A}$ by routing all the messages sent from $\mathcal{F}_{\mathsf{dummy}}$; from $\mathcal{A}$'s point of view, $\mathcal{S}_1$ is the environment $\mathcal{Z}$.

Hybrid 2. In this hybrid, we replace the dummy functionality $\mathcal{F}_{\mathsf{dummy}}$ by the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ except that we replace the functions used within `Create`, `Propose`, `Commit`, `Process`, and `Join` by those defined in Figs. 51 to 54. We call this modified ideal functionality $\mathcal{F}_{\mathrm{CGKA},2}^{\mathsf{mh}}$. In words, $\mathcal{F}_{\mathrm{CGKA},2}^{\mathsf{mh}}$ includes all the descriptions of the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ for static metadata-hiding CGKA and outsources any other checks performed by the wrapper protocol W^{mh} to the simulator $\mathcal{S}_2$. Namely, these correspond to the party-server interaction. In this hybrid, all consistency and security regarding the static metadata are guaranteed. The description of $\mathcal{S}_2$ is provided in Lem. E.2.

Hybrid 3. In this hybrid, we add all the missing consistency checks regarding the wrapper protocol W^{mh} of the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ into $\mathcal{F}_{\mathrm{CGKA},2}^{\mathsf{mh}}$. We cal this ideal functionality $\mathcal{F}_{\mathrm{CGKA},3}^{\mathsf{mh}}$. More precisely, $\mathcal{F}_{\mathrm{CGKA},3}^{\mathsf{mh}}$ is identical to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ except that **random-index** (resp. **adv-access-allowed**) always returns false (resp. true). It only takes care of the correctness guarantees and does not guarantee any security properties regarding the dynamic metadata. Simulator $\mathcal{S}_3$ is defined identically to $\mathcal{S}_2$.

Hybrid 4. In this hybrid, we change how the randomness used to derive the permutation key and group signature key are generated. The simulator $\mathcal{S}_4$ is identical to $\mathcal{S}_3$ except that, rather than generating $\mathsf{permKey} \leftarrow \mathsf{PRF}(\mathsf{k}_{\mathsf{mh}}, \text{'perm'})$ and $\mathsf{authKey} \leftarrow \mathsf{PRF}(\mathsf{k}_{\mathsf{mh}}, \text{'auth'})$ (which occurs during `Create` or `Commit`), if **safe** is true for that epoch, then it samples

a random permKey and authKey. In this hybrid, we use $\mathcal{F}_{\mathrm{CGKA},4} := \mathcal{F}_{\mathrm{CGKA},3}$.

Hybrid 5. In this hybrid, we add the missing security guarantee on the randomness of the party's index. Namely, we modify $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},4}$ to use the original **random-index** predicate, denoted as $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},5}$. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},5}$ permutes the indices with a random permutation if predicate **random-index** is true. Simulator $\mathcal{S}_5$ is identical to $\mathcal{S}_4$.

Hybrid 6. In this hybrid, we add the missing security guarantee when an adversary tries to access the honest server. Namely, we modify $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},5}$ to use the original predicate **adv-access-allowed**, denoted as $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},6}$. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},6}$ halts if an adversary succeeds to fetch or publish a proposal or commit message without knowing the corresponding group secret key. Simulator $\mathcal{S}_6$ is identical to $\mathcal{S}_5$. At this point, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},6}$ is identical to the ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$.

We show indistinguishability of Hybrids 1 to 6 in Lems. E.2 to E.4, E.6 and E.8. This completes the proof of the main theorem. □

E.3.1 From Hybrid 1 to 2: Lem. E.2.

Lemma E.2. *Hybrid 1 and Hybrid 2 are perfectly indistinguishable.*

Proof. We first provide the description of $\mathcal{S}_2$. Whenever $\mathcal{S}_2$ is invoked on an input from $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ called within $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},2}$,[41] it simply relays the same input to the adversary $\mathcal{A}$. Since W^{mh} is constructed in the $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$-hybrid model, these inputs are exactly what $\mathcal{A}$ was provided in Hybrid 1. $\mathcal{S}_2$ returns to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ whatever provided by $\mathcal{A}$. It remains to describe how $\mathcal{S}_2$ answers `RegisterGroup`, and publish and fetch queries sent from $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},2}$. To this end, we first make a detour and explain how k_{mh} is set in Hybrids 1 and 2.

In Hybrid 1, k_{mh} for a commit node node-id is created when $\mathcal{S}_1$ invokes some party id with Ptr[id] = node-id on a `Create` or a `Commit`. When id queries `Create`, id performs a query $\mathsf{Key}_{\mathsf{mh}}$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ within `*init-states`; when id queries `Commit`, id performs a query $\mathsf{NextKey}_{\mathsf{mh}}$ within `PublishCommit`. If **safe**(node-id) = true, then $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ samples a random key k_{mh}. Otherwise, $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ asks $\mathcal{A}$ to provide the key. These are the only two places where a new k_{mh} is set — $\mathsf{Key}_{\mathsf{mh}}$ and $\mathsf{NextKey}_{\mathsf{mh}}$ are also queried during `Propose`, `Process`, and `Join` but k_{mh} will already be defined.

In Hybrid 2, however, $\mathcal{S}_2$ can no longer directly invoke a party id when the static metadata is hidden. For example, when $\mathcal{Z}$ invokes party $\mathsf{id}_{\mathsf{creator}}$ on a `Create` while ***leak-create** is activated (i.e., the predicate **safe** is true), then $\mathcal{S}_2$ is only provided with $(|\mathsf{id}_{\mathsf{creator}}|, |\mathsf{svk}|)$ from $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},2}$. Without knowing $\mathsf{id}_{\mathsf{creator}}$, $\mathcal{S}_2$ cannot invoke $\mathsf{id}_{\mathsf{creator}}$ to execute `*init-states`. This in particular means that it cannot set k_{mh} like $\mathcal{S}_1$ did above by invoking $\mathsf{Key}_{\mathsf{mh}}$ or $\mathsf{NextKey}_{\mathsf{mh}}$ of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

With this in mind, we now finish the description of $\mathcal{S}_2$. $\mathcal{S}_2$ maintains a list $L_{\mathsf{k}_{\mathsf{mh}}}$ to store the key k_{mh} generated at node node-id, i.e., $L_{\mathsf{k}_{\mathsf{mh}}}[\text{node-id}] = \mathsf{k}_{\mathsf{mh}}$. Notice that when $\mathcal{S}_2$ needs to simulate a `RegisterGroup`, or a fetch or publish query, it is given either (gid, epoch) if the server Sv is honest and Ptr[id] if Sv is malicious. Since there is no fork in the group when Sv is honest, (gid, epoch) uniquely defines node-id. Notably, even if $\mathcal{S}_2$ does not know who the calling party id is, it knows which group and epoch (or Ptr[id] in case of a fork) id is included in, and thus, knows which $\mathsf{k}_{\mathsf{mh}} = L_{\mathsf{k}_{\mathsf{mh}}}[\text{node-id}]$ to use if it exists.

As explained above, when $\mathcal{Z}$ invokes some party to perform `Propose`, `Process`, or `Join`, k_{mh} is already defined. Thus, $\mathcal{S}_2$ simply uses $\mathsf{k}_{\mathsf{mh}} = L_{\mathsf{k}_{\mathsf{mh}}}[\text{node-id}]$ to perform the same simulation as $\mathcal{S}_1$, where recall id is not used during a publish or fetch protocol. When the environment $\mathcal{Z}$ invokes $\mathsf{id}_{\mathsf{creator}}$ on (`Create`, svk), this is when a new k_{mh} is generated, i.e., $L_{\mathsf{k}_{\mathsf{mh}}}[\mathsf{Ptr}[\mathsf{id}_{\mathsf{creator}}]]$ is still undefined. $\mathcal{S}_2$ checks if **safe**(0) = false — which it can do since it can simulate an identical semantics of the history graph maintained within $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},2}$ — and asks $\mathcal{A}$ for k_{mh}. This is identical to what $\mathcal{S}_1$ did. Otherwise, $\mathcal{S}_2$ samples a random k_{mh} on its own. In either case, $\mathcal{S}_2$ stores $L_{\mathsf{k}_{\mathsf{mh}}}[0] \leftarrow \mathsf{k}_{\mathsf{mh}}$ and uses k_{mh} to perform the `RegisterGroup` protocol. The only difference between the previous hybrid is that $\mathcal{S}_2$ samples k_{mh} when **safe**(0) = true, rather than letting $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ sampling it. However, since `*mark-next-db-initialized-epoch`(0) is called and due to the restriction on $\mathcal{A}$, $\mathcal{A}$ cannot compromise k_{mh} after `Create` was invoked. Therefore, from the view of $\mathcal{Z}$, k_{mh} is distributed identically in both hybrids. Thus, the simulation of `Create` is perfectly indistinguishable from Hybrid 1. The case when the environment $\mathcal{Z}$ invokes id on (`Commit`, svk) is proven analogously to (`Create`, svk). This completes the proof. □

E.3.2 From Hybrid 2 to 3: Lem. E.3.

Lemma E.3. *Hybrid 2 and Hybrid 3 are indistinguishable assuming the correctness of* SIG′.

Proof. The only difference between Hybrids 2 and 3 is that $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},3}$ in Hybrid 3 mandates correctness of the `RegisterGroup`, and publish and fetch protocols when the server Sv is honest. For instance, in Hybrid 2, during the `PublishProposal` protocol, $\mathcal{S}_2$ simulated the interaction between the party id and the honest Sv using the key k_{mh} stored in the list $L_{\mathsf{k}_{\mathsf{mh}}}$. $\mathcal{S}_2$ then returned *accept′* output by Sv to $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},2}$, and in particular, this notified the environment $\mathcal{Z}$ that $\mathcal{S}$ output *accept′*. In particular, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},2}$ did not model any notion of correctness of the `PublishProposal` protocol.

On the other hand, in Hybrid 3, if ComDB[gid, epoch] = (⊤, node-id) and node-id = Ptr[id] (i.e., the group statement gvk is created honestly and id holds the corresponding signing key gsk), then $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},3}$ always sends *accept* = true to $\mathcal{Z}$. This models the fact that if the new epoch (node-id) is initialized by an honest party, then any member assigned on the same commit node (i.e., node-id = Ptr[id]) can upload a proposal. It is clear that this holds assuming that SIG′ is correct, i.e., a properly generated signature is always accepted. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},3}$ always sends *accept* = false to $\mathcal{Z}$ if (1) ComDB[gid, epoch] ≠ (⊤, node-id) or (2) ComDB[gid, epoch] = (⊤, node-id) and node-id ≠ Ptr[id]. For case (1), we can verify that the real server performs the same check as ComDB[gid, epoch] is equal to (gvk, ⊥, ⊥) or not, and thus $\mathcal{S}_2$ returned *accept′* = false. For case (2), if $\mathcal{S}_2$ returned *accept′* = true in case (2), we can construct an adversary that breaks the EUF-CMA security of SIG′ by using such $\mathcal{S}_2$. Thus $\mathcal{S}_2$ returned *accept′* = false assuming

[41] Note that this is not strictly true since $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ is not defined within $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},2}$. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},2}$ merely has most of the codes included in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. We use this informal wording for simplicity.

the EUF-CMA security of SIG′. Therefore, the view to $\mathcal{Z}$ remain identical.

All other RegisterGroup, and publish and fetch protocols in Hybrid 3 can be checked to behave identically to those of Hybrid 2 conditioned on SIG′ being correct and EUF-CMA secure. □

E.3.3 From Hybrid 3 to 4: Lem. E.4.

Lemma E.4. *Hybrid 3 and Hybrid 4 are indistinguishable assuming PRF is a secure pseudorandom function.*

Proof. We assume $\mathcal{Z}$ creates at most Q epochs (i.e., commit nodes node-id in the history graph). To show Lem. E.4, we consider the following sub-hybrids between Hybrid 3 and Hybrid 4.

Hybrid 3-0 := Hybrid 3. This is identical to Hybrid 3. We use the functionality $\mathcal{F}_{\text{CGKA},3}$, and the simulator $\mathcal{S}_{3\text{-}0} := \mathcal{S}_3$. When simulating protocol, $\mathcal{S}_{3\text{-}0}$ uses the group key k_{mh} stored in the list $L_{\mathsf{k}_{\mathsf{mh}}}$ as PRF key.

Hybrid 3-*i*. *i* runs through $[Q]$. The simulator $\mathcal{S}_{3\text{-}i}$ is defined exactly as $\mathcal{S}_{3\text{-}(i-1)}$ except that when $\mathcal{S}_{3\text{-}(i-1)}$ generates a random k_{mh} for the i-th epoch where **safe** is true, it chooses a random authKey and permKey instead of deriving them from PRF and k_{mh}. Note that we count epochs (i.e., node-id) in the order in which Create or Commit is invoked. We show in Lem. E.5 that Hybrid 3-$(i-1)$ and Hybrid 3-i are indistinguishable.

Hybrid 3-Q := Hybrid 4. In this hybrid, all authKey and permKey generated at epochs such that **safe** is true are chosen at random.

The indistinguishability between Hybrid 3 and Hybrid 4 is established by applying the following Lem. E.5 for all $i \in [Q]$.

Lemma E.5. *Hybrid 3-$(i-1)$ and Hybrid 3-i are indistinguishable assuming PRF is a secure pseudo-random function.*

Proof. The difference between Hybrid 3-$(i-1)$ and Hybrid 3-i is whether authKey and permKey at the i-th epoch (i.e., node-id) are chosen uniformly at random if **safe** is true. (If **safe** is false for the i-th epoch, two hybrids proceed identically.) Due to the modification we made in Hybrid 2, the group key k_{mh} is chosen uniformly at random. Moreover, due to the restriction on the adversary, k_{mh} generated when **safe** was true cannot be corrupted by the adversary. Thus, by the pseudorandomness of the PRF, the output of the PRF is indistinguishable from a random string for the input labels 'perm' and 'auth'. This implies Hybrid 3-$(i-1)$ and Hybrid 3-i are indistinguishable. More formally, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the pseudorandomness of the PRF. We first explain the description of $\mathcal{B}$; then we evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the protocol executions between a party and the server as in $\mathcal{S}_{3\text{-}(i-1)}$, except for the evaluation of the PRF. To generate authKey and permKey, $\mathcal{B}$ queries authKey := $\mathcal{F}$('auth') and permKey := $\mathcal{F}$('perm') to its oracle $\mathcal{F}$.

We evaluate the success probability of $\mathcal{B}$. If the oracle $\mathcal{F}$ evaluates a pseudo-random function, then $\mathcal{Z}$'s view is identical to Hybrid 3-$(i-1)$. If $\mathcal{F}$ evaluates a truly random function, then $\mathcal{Z}$'s view is identical to Hybrid 3-i. Hence, if $\mathcal{Z}$ distinguishes Hybrid 3-$(i-1)$ and Hybrid 3-i with non-negligible probability, $\mathcal{B}$ breaks the pseudorandomness of the PRF with non-negligible probability. This contradicts the assumption that PRF is secure. Therefore, Hybrid 3-$(i-1)$ and Hybrid 3-i are indistinguishable. □

□

E.3.4 From Hybrid 4 to 5: Lem. E.6. Hybrid 5 concerns the randomness of the access indix of group members. If **random-index** is true, then the ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\text{CGKA},5}$ randomizes the commit vector $\vec{c}$ and the index of each party in the group on behalf of the simulator. We prove in Lem. E.6 that assuming that PRP is secure, Hybrid 4 and Hybrid 5 are indistinguishable.

Lemma E.6. *Hybrid 4 and Hybrid 5 are indistinguishable assuming PRP is a secure pseudorandom permutation.*

Proof. We assume $\mathcal{Z}$ creates at most Q epochs (i.e., commit nodes node-id in the history graph). To show Lem. E.6, we consider the following sub-hybrids between Hybrid 4 and Hybrid 5.

Hybrid 4-0 := Hybrid 4. This is identical to Hybrid 4. We use the functionality $\mathcal{F}_{\text{CGKA},4}$, and the simulator $\mathcal{S}_{4\text{-}0} := \mathcal{S}_4$. In this hybrid, $\mathcal{F}_{\text{CGKA},4}$ permutes the indices index and commit vectors $\vec{c}$ with a permutation chosen by $\mathcal{S}_{4\text{-}0}$, and $\mathcal{S}_{4\text{-}0}$ simulates the PublishCommit and FetchCommit protocols using the indices and commit messages provided from $\mathcal{F}_{\text{CGKA},4\text{-}i}$.

Hybrid 4-*i*. *i* runs through $[Q]$. We use the functionality $\mathcal{F}_{\text{CGKA},4\text{-}i}$ which is defined exactly as $\mathcal{F}_{\text{CGKA},4\text{-}(i-1)}$ except that the indices index and commit vectors $\vec{c}$ are permutated with a truly random permutation if **random-index** is true for the i-th epoch. The simulator $\mathcal{S}_{4\text{-}i}$ is defined exactly as $\mathcal{S}_{4\text{-}(i-1)}$. Note that we count epochs (i.e., node-id) in the order in which Create or Commit is invoked. We show in Lem. E.7 that Hybrid 4-$(i-1)$ and Hybrid 4-i are indistinguishable.

Hybrid 4-Q := Hybrid 5. We use the functionality $\mathcal{F}_{\text{CGKA},4\text{-}Q}$ which permutes the indices index and commit vectors $\vec{c}$ with a random permutation for all epochs where **random-index** is true. $\mathcal{F}_{\text{CGKA},4\text{-}Q}$ is identical to $\mathcal{F}_{\text{CGKA},5}$.

Indistinguishability between Hybrid 4 and Hybrid 5 is established by applying the following Lem. E.7 for all $i \in [Q]$.

Lemma E.7. *Hybrid 4-$(i-1)$ and Hybrid 4-i are indistinguishable assuming PRP is a secure pseudorandom permutation.*

Proof. The difference between Hybrid 4-$(i-1)$ and Hybrid 4-i is whether index and $\vec{c}$ issued at the i-th epoch where **random-index** is true are permuted by a truly random permutation. (If **random-index** is false for the i-th epoch, two hybrids proceed identically.) Due to the modification we made in Hybrid 4, the i-th permutation key permKey is chosen uniformly at random if **random-index** (=**safe**) is true. Thus, by the pseudorandomness of the PRP, the output of the PRP is indistinguishable from that of a truly random permutation. This implies Hybrid 4-$(i-1)$ and Hybrid 4-i are indistinguishable. More formally, if $\mathcal{Z}$ can distinguish the two hybrids, then we can construct an adversary $\mathcal{B}$ that breaks the pseudorandomness of the PRP. We first explain the description of $\mathcal{B}$; then we evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the protocol executions between a party and the server as in $\mathcal{S}_{4\text{-}(i-1)}$, except for the evaluation of PRP at the i-th epoch. Rather than sampling a permutation key permKey for

the i-th epoch to evaluate the permutation, $\mathcal{B}$ queries its challenge oracle to compute $\mathsf{index}' \leftarrow \mathcal{P}(\mathsf{index})$.

We evaluate the success probability of $\mathcal{B}$. If the oracle $\mathcal{P}$ evaluates a pseudorandom permutation, then $\mathcal{Z}$'s view is identical to Hybrid 4-$(i-1)$. Otherwise, if $\mathcal{P}$ evaluates a truly random permutation, then $\mathcal{Z}$'s view is identical to Hybrid 4-i. Hence, if $\mathcal{Z}$ distinguishes Hybrid 4-$(i-1)$ and Hybrid 4-i with non-negligible probability, $\mathcal{B}$ breaks the pseudorandomness of the PRP with non-negligible probability. This contradicts the assumption that PRP is secure. Therefore, Hybrid 4-$(i-1)$ and Hybrid 4-i are indistinguishable. □

□

E.3.5 From Hybrid 5 to 6: Lem. E.8. Hybrid 6 concerns the group membership check performed by the honest server. When the server is honest (i.e., `ServerStat` = 'good') and **adv-access-allowed** is false (i.e., group secrets are not compromised), the functionality $\mathcal{F}^{\mathsf{mh}}_{\mathsf{CGKA},6}$ halts if an adversary succeeds to publish or fetch a proposal or commit messages. That is an adversary that does not know the group secret should not be able to publish or fetch contents from the honest server. We prove in Lem. E.8 that if $\mathcal{Z}$ can distinguish the two hybrids, then we can construct an adversary that breaks the EUF-CMA security of SIG′. In other words, assuming that SIG' is EUF-CMA secure, Hybrid 5 and Hybrid 6 are indistinguishable. Concretely, we show the following.

Lemma E.8. *Hybrid 5 and Hybrid 6 are indistinguishable assuming* SIG′ *is* EUF-CMA *secure.*

Proof. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the EUF-CMA security of SIG′. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ extracts a valid signature forgery using $\mathcal{Z}$; we then show the validity of the forged signature and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the protocol executions between a party and the server as in $\mathcal{S}_6$, except for the signature generation during the publish and fetch protocols. At the beginning of the game, $\mathcal{B}$ chooses an index $i \in [Q]$ at random, where Q is the total number of epochs (i.e., commit nodes node-id) that $\mathcal{Z}$ creates. $\mathcal{B}$ aborts if **adv-access-allowed** is `true` for the i-th epoch, and otherwise embeds the challenge signing key svk^* in the i-th group signing key gvk. Here, if **adv-access-allowed** is false (i.e, **safe** is true) for the i-th epoch, the group signing key is generated with a uniformly random authKey due to the modification we made in Hybrid 4. Therefore, $\mathcal{B}$ perfectly simulates svk^* as in Hybrid 5. Moreover, note that once **safe** is true, the restricted adversary cannot compromise the i-th group signing key. Whenever an honest party at the i-th epoch publishes or fetches proposal/commit messages, $\mathcal{B}$ uses the signing oracle to sign the challenge message sent from the server. For the other epochs, $\mathcal{B}$ generates gvk and signs as in the previous hybrid. If the adversary succeeds in making the server accept at execution of a publish and fetch proposal/commit protocols in the i-th epoch, $\mathcal{B}$ retrieves the pair of challenge message and response signature (ch, σ) from the successful transcript, and submits (ch, σ) as the forgery.

Let us analyze the success probability of $\mathcal{B}$. First, by noticing the only way that $\mathcal{Z}$ can distinguish the two hybrids is by triggering the **assert** condition on **adv-access-allowed**, there must exist at least one epoch for which the adversary succeeds in the protocol `PublishProposalAdv`, where **adv-access-allowed** is false. Since the choice $i \leftarrow\!\!\$\ [Q]$ of $\mathcal{B}$ is information-theoretically hidden from $\mathcal{Z}$ and the adversary, the guess made by $\mathcal{B}$ is correct with probability at least $1/Q$. Conditioning on the guess being correct, (ch, σ) is a valid message and signature pair for gvk. Moreover, since the server is honest and the challenge message space is exponentially large, the server never picks the same challenge message ch. Therefore, (ch, σ) is a pair of message and signature that $\mathcal{B}$ did not query to the signing oracle and constitutes a valid forgery. In summary, if $\mathcal{Z}$ distinguishes the two hybrids with non-negligible probability ϵ, $\mathcal{B}$ wins the game with probability at least ϵ/Q, which is also non-negligible. This contradicts the assumption that SIG′ is EUF-CMA secure. Thus, Hybrid 5 and Hybrid 6 are indistinguishable. □

Input (Create, svk)

1 : **req** $G = \bot$
2 : **try** query (Create, svk) to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$
3 : $G \leftarrow$ *init-states(gid, 0, { ($\text{id}_{\text{creator}}$, svk) })
4 : **req** RegisterGroup(gid, 0)

Input (Propose, act), act ∈ { 'upd'-svk, 'add'-id_t-kp_t, 'rem'-id_t }

1 : **req** $G \neq \bot$
2 : **try** $p \leftarrow$ query (Propose, act) to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$
3 : **req** PublishProposal(G.gid, G.epoch, p)
4 : **return** p

Input (Commit, svk)

1 : **req** $G \neq \bot$
2 : $(accept, \vec{p}) \leftarrow$ FetchProposals(G.gid, G.epoch)
3 : **req** *accept*
4 : **try** $(c_0, \vec{c}, \vec{w}) \leftarrow$ query (Commit, svk, $\vec{p}$) to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$
5 : $\vec{c}_{\text{perm}} \leftarrow$ *permute-commit($G, \vec{c}$)
6 : **try** PublishCommit(G.gid, G.epoch, $c_0, \vec{c}_{\text{perm}}$)
7 : **foreach** $\widehat{w} \in \vec{w}$ **do**
8 : **parse** $(\text{id}_t, *) \leftarrow \widehat{w}$
9 : **try** PublishWelcome($\text{id}_t, \widehat{w}$)
10 : **return** $(c_0, \vec{c}_{\text{perm}}, \vec{w})$

Input Process

1 : **req** $G \neq \bot$
2 : $\text{index}_{\widehat{c}} \leftarrow$ *permuted-commit-index(G, id)
3 : **try** $(c_0, \widehat{c}, \vec{p}) \leftarrow$ FetchCommit(G.gid, G.epoch, $\text{index}_{\widehat{c}}$)
4 : **try** $(\text{id}_c, \text{propSem}, \text{mem}) \leftarrow$ query (Process, $c_0, \widehat{c}, \vec{p}$) to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$
5 : $G' \leftarrow$ *update-states(G, mem)
6 : **return** $(\text{id}_c, \text{propSem}, \text{mem})$

Input (Key)

1 : **req** $G \neq \bot$
2 : **try** k $\leftarrow$ query (Key) to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$
3 : **return** k

Input Join

1 : **req** $G = \bot$
2 : **try** $\widehat{w} \leftarrow$ FetchWelcome(id)
3 : **try** $(\text{id}_c, \text{gid}, \text{epoch}, \text{mem}) \leftarrow$ query (Join, $\widehat{w}$) to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$
4 : $G \leftarrow$ *init-states(gid, epoch, mem)
5 : **return** $(\text{id}_c, \text{gid}, \text{epoch}, \text{mem})$

*init-states(gid, epoch, mem)

1 : G.gid $\leftarrow$ gid; G.epoch $\leftarrow$ epoch; G.mem $\leftarrow$ mem
2 : $\text{k}_{\text{mh}} \leftarrow$ query Key_{mh} to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$
3 : G.permKey $\leftarrow$ PRF(k_{mh}, 'perm')
4 : authKey $\leftarrow$ PRF(k_{mh}, 'auth')
5 : $(G.\text{gvk}, G.\text{gsk}) \leftarrow \text{SIG}'.\text{KeyGen}(\text{pp}_{\text{SIG}'}; \text{authKey})$
6 : **return** G

*update-states(G, mem)

1 : G'.gid $\leftarrow$ G.gid; G'.epoch $\leftarrow$ G.epoch + 1
2 : G'.mem $\leftarrow$ mem
3 : $\text{k}_{\text{mh}} \leftarrow$ query Key_{mh} to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$
4 : G'.permKey $\leftarrow$ PRF(k_{mh}, 'perm')
5 : authKey $\leftarrow$ PRF(k_{mh}, 'auth')
6 : $(G'.\text{gvk}, G'.\text{gsk}) \leftarrow \text{SIG}'.\text{KeyGen}(\text{pp}_{\text{SIG}'}; \text{authKey})$
7 : **return** G'

*permute-commit($G, \vec{c}$)

1 : $\vec{c}_{\text{perm}} \leftarrow ()$
2 : **for** index = $1, \ldots, |\vec{c}|$ **do**
3 : $\vec{c}_{\text{perm}} \,{+}{\leftarrow}\, \vec{c}[\text{PRP}(G.\text{permKey}, \text{index})]$
4 : **return** $\vec{c}_{\text{perm}}$

*permuted-commit-index(G, id)

1 : index $\leftarrow$ G.indexOf(id)
2 : **return** PRP(G.permKey, index)

Figure 42: Metadata-hiding CGKA protocol W^{mh} in the $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$-hybrid model: Create, Propose, Commit, Process, Join, Key, and some helper functions.

Subroutine RegisterGroup(gid, 0)

Party id		**Server** Sv

gvk ← G.gvk	('anon', ⊥, Sv), RegisterGroup, gid, 0, gvk →	**if** ComDB[gid, ∗] = ⊥ **then**
		  ComDB[gid, 0] ← (gvk, ⊥, ⊥)
		  accept ← true
		else
		  accept ← false
		return (*accept*, gid, 0)

return *accept*	← *accept*	

Figure 43: Subroutines for metadata-hiding CGKA protocol W^{mh} in the $\mathcal{F}^{\mathsf{ctxt}}_{\mathbf{CGKA}}$-hybrid model: Register a new group and initialize group states for the current epoch. Party id **and the server** Sv **are connected via a client-anonymized authenticated channel.**

Subroutine PublishProposal(gid, epoch, p)

Party id		**Server** Sv
	('anon', ⊥, Sv), PublishProposal →	
	← ch	ch ←$ $\{0,1\}^\kappa$

(gid, epoch) ← (G.gid, G.epoch)
$\sigma \leftarrow \mathsf{SIG}'.\mathsf{Sign}(\mathsf{pp}_{\mathsf{SIG}'}, G.\mathsf{gsk}, \mathsf{ch})$
	σ, gid, epoch, p →	**if** ComDB[gid, epoch] = (gvk, ⊥, ⊥)
		  ∧ $\mathsf{SIG}'.\mathsf{Verify}(\mathsf{pp}_{\mathsf{SIG}'}, \mathsf{gvk}, \sigma, \mathsf{ch})$ **then**
		  PropDB[gid, epoch] ∥← p
		  accept ← true
		else
		  accept ← false
		return (*accept*, gid, epoch, p)

return *accept*	← *accept*	

Figure 44: Subroutines for metadata-hiding CGKA protocol W^{mh} in the $\mathcal{F}^{\mathsf{ctxt}}_{\mathbf{CGKA}}$-hybrid model: Publish proposal messages. Party id **and the server** Sv **are connected via a client-anonymized authenticated channel.**

Subroutine FetchProposals(gid, epoch)

Party id		**Server** Sv
	('anon', ⊥, Sv), FetchProposals →	
	← ch	ch ←$ $\{0,1\}^\kappa$

$\sigma \leftarrow \mathsf{SIG}'.\mathsf{Sign}(\mathsf{pp}_{\mathsf{SIG}}, G.\mathsf{gsk}, \mathsf{ch})$	σ, gid, epoch →	**if** ComDB[gid, epoch] = (gvk, ⊥, ⊥)
		  ∧ $\mathsf{SIG}'.\mathsf{Verify}(\mathsf{pp}_{\mathsf{SIG}'}, \mathsf{gvk}, \sigma, \mathsf{ch}) = \top$ **then**
		  (*accept*, $\vec{p}$) ← (true, PropDB[gid, epoch])
		else
		  (*accept*, $\vec{p}$) ← (false, ⊥)
		return (*accept*, gid, epoch)

return (*accept*, $\vec{p}$)	← *accept*, $\vec{p}$	

Figure 45: Subroutines for metadata-hiding CGKA protocol W^{mh} in the $\mathcal{F}^{\mathsf{ctxt}}_{\mathbf{CGKA}}$-hybrid model: Fetch proposal messages. Party id **and the server** Sv **are connected via a client-anonymized authenticated channel.**

Subroutine PublishCommit(gid, epoch, $c_0, \vec{c}$)

Party id		**Server** Sv
	('anon', $\perp$, Sv), PublishCommit $\longrightarrow$	
	$\longleftarrow$ ch	ch $\leftarrow\$ \{0,1\}^\kappa$
(gid, epoch) $\leftarrow$ (G.gid, G.epoch)	σ, gid, epoch, gvk′, c_0, $\vec{c}_{\text{perm}}$ $\longrightarrow$	**if** ComDB[gid, epoch] = (gvk, $\perp$, $\perp$)
$\sigma \leftarrow$ SIG′.Sign($\text{pp}_{\text{SIG}'}$, G.gsk, ch)		$\wedge$ SIG′.Verify($\text{pp}_{\text{SIG}'}$, gvk, σ, ch) = $\top$ **then**
/ Generate the next epoch's group statement.		ComDB[gid, epoch] $\leftarrow$ (gvk, c_0, $\vec{c}_{\text{perm}}$)
$\text{k}_{\text{mh}}' \leftarrow$ query ($\text{NextKey}_{\text{mh}}$, c_0) to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$		ComDB[gid, epoch + 1] $\leftarrow$ (gvk′, $\perp$, $\perp$)
authKey′ $\leftarrow$ PRF(k_{mh}', 'auth')		*accept* $\leftarrow$ true
(gvk′, gsk′) $\leftarrow$ SIG′.KeyGen($\text{pp}_{\text{SIG}'}$; authKey′)		**else**
		accept $\leftarrow$ false
		return (*accept*, gid, epoch, c_0, $\vec{c}_{\text{perm}}$)
return *accept*	$\longleftarrow$ *accept*	

Figure 46: Subroutines for metadata-hiding CGKA protocol W^{mh} in the $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$-hybrid model: Publish commit messages. Party id **and the server** Sv **are connected via a client-anonymized authenticated channel.**

Subroutine FetchCommit(gid, epoch, index)

Party id		**Server** Sv
	('anon', $\perp$, Sv), FetchCommit $\longrightarrow$	
	$\longleftarrow$ ch	ch $\leftarrow\$ \{0,1\}^\kappa$
$\sigma \leftarrow$ SIG′.Sign($\text{pp}_{\text{SIG}'}$, G.gsk, ch)	σ, gid, epoch, index $\longrightarrow$	**if** ComDB[gid, epoch] = (gvk, c_0, $\vec{c}_{\text{perm}}$)
		$\wedge$ SIG′.Verify($\text{pp}_{\text{SIG}'}$, gvk, σ, ch) = $\top$ **then**
		($accept$, $\vec{p}$) $\leftarrow$ (true, PropDB[gid, epoch])
		$\widehat{c} \leftarrow \vec{c}_{\text{perm}}$[index]
		else
		($accept$, c_0, $\widehat{c}$, $\vec{p}$) $\leftarrow$ (false, $\perp$, $\perp$, $\perp$)
		return (*accept*, gid, epoch, index)
return ($accept$, c_0, $\widehat{c}$, $\vec{p}$)	$\longleftarrow$ $accept$, c_0, $\widehat{c}$, $\vec{p}$	

Figure 47: Subroutines for metadata-hiding CGKA protocol W^{mh} in the $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$-hybrid model: Fetch commit messages. Party id **and the server** Sv **are connected via a client-anonymized authenticated channel.**

Subroutine PublishWelcome(id_t, $\widehat{w}$)

Party id		**Server** Sv
	('anon', $\perp$, Sv), PublishWelcome, id_t, $\widehat{w}$ $\longrightarrow$	WelDB[id_t] $\leftarrow$ $\widehat{w}$
		accept $\leftarrow$ true
		return ($accept$, id_t, $\widehat{w}$)
return *accept*	$\longleftarrow$ *accept*	

Figure 48: Subroutines for metadata-hiding CGKA protocol W^{mh} in the $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$-hybrid model: Publish and fetch welcome messages. Party id **and the server** Sv **are connected via a client-anonymized authenticated channel.**

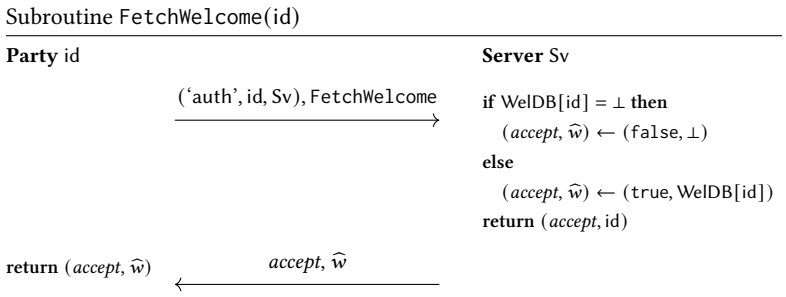


Figure 49: Subroutines for metadata-hiding CGKA protocol W^{mh} in the $\mathcal{F}_{\mathbf{CGKA}}^{\mathsf{ctxt}}$-hybrid model: Fetch welcome messages. Party id **and the server** Sv **are connected via an authenticated channel. Note that the channel does not need to be anonymized.**

Random index and group authentication.

random-index$(c) \iff$ **safe**(c)

adv-access-allowed$(c) \iff \neg$**safe**(c)

Figure 50: Additional safety predicates unique to the wrapper protocol W^{mh}. The other safety predicates used to implicitly define the underlying ideal functionality $\mathcal{F}_{\mathbf{CGKA}}^{\mathsf{ctxt}}$ are identical to those provided in App. C.2.

Subroutine `RegisterGroup`(gid, epoch)

channelType ← ('anon', ⊥, Sv) / Connect to the server via anonymous channel
req epoch = 0
Send (channelType, `RegisterGroup`, gid, 0) to $\mathcal{S}$ and receive *accept*
if `ServerStat` = 'good' **then**
  Send (*accept*, gid, 0) to Sv
return *accept*

Subroutine `PublishProposal`(gid, epoch, p)

channelType ← ('anon', ⊥, Sv) / Connect to the server via anonymous channel
if `ServerStat` = 'good' **then**
  Send (channelType, `PublishProposal`, gid, epoch, p) to $\mathcal{S}$ and
    receive *accept*
  Send (*accept*, gid, epoch, p) to Sv
else
  Send (channelType, `PublishProposal`, Ptr[id], p) to $\mathcal{S}$ and
    receive *accept*
return *accept*

Subroutine `FetchProposals`(gid, epoch)

channelType ← ('anon', ⊥, Sv) / Connect to the server via anonymous channel
if `ServerStat` = 'good' **then**
  Send (channelType, `FetchProposals`, gid, epoch) to $\mathcal{S}$ and
    receive (*accept*, $\vec{p}$)
  Send (*accept*, gid, epoch) to Sv
else
  Send (channelType, `FetchProposals`, Ptr[id]) to $\mathcal{S}$ and
    receive (*accept*, $\vec{p}$)
return (*accept*, $\vec{p}$)

Subroutine `PublishCommit`(gid, epoch, $c_0, \vec{c}_{\text{perm}}$)

channelType ← ('anon', ⊥, Sv) / Connect to the server via anonymous channel
if `ServerStat` = 'good' **then**
  Send (channelType, `PublishCommit`, gid, epoch, $c_0, \vec{c}_{\text{perm}}$) to $\mathcal{S}$ and
    receive *accept*
  Send (*accept*, gid, epoch, $c_0, \vec{c}_{\text{perm}}$) to Sv
else
  Send (channelType, `PublishCommit`, Ptr[id], $c_0, \vec{c}_{\text{perm}}$) to $\mathcal{S}$ and
    receive *accept*
return *accept*

Subroutine `FetchCommit`(gid, epoch, $\text{index}_{\widehat{c}}$)

channelType ← ('anon', ⊥, Sv) / Connect to the server via anonymous channel
if `ServerStat` = 'good' **then**
  Send (channelType, `FetchCommit`, gid, epoch, $\text{index}_{\widehat{c}}$) to $\mathcal{S}$ and
    receive (*accept*, $c_0, \widehat{c}, \vec{p}$)
  Send (*accept*, gid, epoch, $\text{index}_{\widehat{c}}$) to Sv
else
  Send (channelType, `FetchCommit`, Ptr[id], $\text{index}_{\widehat{c}}$) to $\mathcal{S}$ and
    receive (*accept*, $c_0, \widehat{c}, \vec{p}$)
return (*accept*, $c_0, \widehat{c}, \vec{p}$)

Figure 51: Subroutines for publish and fetch proposal and commit messages used in Hybrid 2.

Input from a party id **or adversary** $\mathcal{S}$
Input `PublishWelcome`($\text{id}_t, \widehat{w}$)

/ Connect to the server via anonymous channel
channelType ← ('anon', ⊥, Sv)
Send (channelType, `PublishWelcome`, $\text{id}_t, \widehat{w}$) to $\mathcal{S}$ and
  receive *accept*
if `ServerStat` = 'good' **then**
  Send (*accept*, $\text{id}_t, \widehat{w}$) to Sv
return *accept*

Input `FetchWelcome`(id)

/ Connect to the server via authenticated channel
channelType ← ('auth', id, Sv)
Send (channelType, `FetchWelcome`) to $\mathcal{S}$ and receive (*accept*, $\widehat{w}$)
if `ServerStat` = 'good' **then**
  Send (*accept*, id) to Sv
return (*accept*, $\widehat{w}$)

Figure 52: Subroutines for publish and fetch welcome messages used in Hybrid 2.

*permuted-commit-index(node-id, id)

1 : **assert** Node[node-id] $\neq \bot$
2 : / If $\mathtt{flag}_{\mathsf{selDL}}$ is false, there is no need to permute index.
3 : **if** $\neg\mathtt{flag}_{\mathsf{selDL}}$ **then**
4 : **return** $\bot$
5 : mem $\leftarrow$ Node[node-id].mem
6 : **assert** id $\in$ mem
7 : Send (*permuted-commit-index, Ptr[id], id) to $\mathcal{S}$ and receive $\mathsf{index}_{\widehat{c}}$
8 : **return** $\mathsf{index}_{\widehat{c}}$

*permute-commit(node-id, $\vec{c}$)

1 : **assert** Node[node-id] $\neq \bot$
2 : / If $\mathtt{flag}_{\mathsf{selDL}}$ is false, no party-dependent commitment can exist.
3 : **if** $\neg\mathtt{flag}_{\mathsf{selDL}}$ **then**
4 : **assert** $\vec{c} = \bot$
5 : **return** $\bot$
6 : Send (*permute-commit, Ptr[id], $\vec{c}$) to $\mathcal{S}$ and receive $\vec{c}_{\mathsf{perm}}$
7 : **return** $\vec{c}_{\mathsf{perm}}$

Figure 53: Helper functions: permute functions used in Hybrid 2.

Input from the adversary $\mathcal{S}$

Input (PublishProposalAdv, gid, epoch, p)

1 : channelType $\leftarrow$ ('anon', $\bot$, Sv)
2 : Send (channelType, PublishProposalAdv, gid, epoch, p) to $\mathcal{S}$ and receive *accept*
3 : **if** ServerStat = 'good' **then**
4 : Send (channelType, PublishProposal, (*accept*, gid, epoch, p)) to Sv
5 : **return** *accept*

Input (FetchProposalsAdv, gid, epoch)

1 : channelType $\leftarrow$ ('anon', $\bot$, Sv)
2 : Send (channelType, FetchProposalsAdv, gid, epoch) to $\mathcal{S}$ and receive (*accept*, $\vec{p}$)
3 : **if** ServerStat = 'good' **then**
4 : **return** (*accept*, $\vec{p}$)

Input (PublishCommitAdv, gid, epoch, $c_0, \vec{c}$)

1 : channelType $\leftarrow$ ('anon', $\bot$, Sv)
2 : Send (channelType, PublishCommitAdv, gid, epoch, $c_0, \vec{c}$) to $\mathcal{S}$ and receive *accept*
3 : **if** ServerStat = 'good' **then**
4 : **return** *accept*

Input (FetchCommitAdv, gid, epoch, index)

1 : channelType $\leftarrow$ ('anon', $\bot$, Sv)
2 : Send (channelType, FetchCommitAdv, gid, epoch, index) to $\mathcal{S}$ and receive (*accept*, $c_0, \widehat{c}, \vec{p}$)
3 : **if** ServerStat = 'good' **then**
4 : **return** (*accept*, $c_0, \widehat{c}, \vec{p}$)

Figure 54: Publish functions used by the adversary in Hybrid 2.

Contents