End-to-End Encrypted Zoom Meetings: Proving Security and Strengthening Liveness[⋆] [⋆⋆]

Yevgeniy Dodis[1] [⋆⋆⋆], Daniel Jost[1] [⋆⋆⋆], Balachandar Kesavan[2], and Antonio Marcedone[2]

[1] New York University {dodis, daniel.jost}@cs.nyu.edu
[2] Zoom Video Communications {balachandar.kesavan, antonio.marcedone}@zoom.us

Abstract. In May 2020, Zoom Video Communications, Inc. (Zoom) announced a multi-step plan to comprehensively support end-to-end encrypted (E2EE) group video calls and subsequently rolled out basic E2EE support to customers in October 2020. In this work we provide the first formal security analysis of Zoom's E2EE protocol, and also lay foundation to the general problem of E2EE group video communication.

We observe that the vast security literature analyzing asynchronous messaging does not translate well to synchronous video calls. Namely, while strong forms of forward secrecy and post compromise security are less important for (typically short-lived) video calls, various *liveness* properties become crucial. For example, mandating that participants quickly learn of updates to the meeting roster and key, media streams being displayed are recent, and banned participants promptly lose any access to the meeting. Our main results are as follows:

1. Propose a new notion of *leader-based continuous group key agreement with liveness*, which accurately captures the E2EE properties specific to the synchronous communication scenario.
2. Prove security of the core of Zoom's E2EE meetings protocol in the above well-defined model.
3. Propose ways to strengthen Zoom's liveness properties by simple modifications to the original protocol, which subsequently influenced updates implemented in production.

[⋆] This is the full version of the article with the same title published in the Proceedings of EUROCRYPT 2023, Springer, © IACR 2023.

[⋆⋆] In this version, we clarified that the protocol deployed by Zoom since v5.13 of the client has some differences with respect to the abstraction presented in the paper, and corrected some inaccuracies in the theorem statements and protocol description with respect to the original proceedings version that do not fundamentally change our results.

[⋆⋆⋆] Research conducted while contracting for Zoom

Table of Contents

1 Introduction

Group video communication tools have gained immense popularity both in personal and professional settings. They were instrumental in bringing people closer together at a time when travel and in-person interaction were severely limited by the COVID-19 pandemic. Zoom Video Communications, Inc. (Zoom) is one of the leading providers of video communications with millions of active users, and aims to distinguish itself not just in ease-of-use and richness of features, but also by offering strong security and privacy capabilities.

Historically, Zoom meetings have been encrypted in transit between the clients and the Zoom servers. This allows Zoom to offer features that require the server to access meeting streams, such as live transcription and the ability to join a meeting by dialing a phone number through the telephony network. In May 2020, Zoom announced a multistep plan to comprehensively support end-to-end (E2E) encrypted group video calls [47] and rolled out basic E2EE support to the public in October 2020 [33]. E2EE protects the privacy of attendees against any compromise to Zoom's infrastructure/keys.

Zoom has also published a whitepaper [11] describing its protocol, design goals, and methodology for E2EE meetings. The whitepaper explains how the protocol is run as part of a group call and provides intuition on the threat model and security. Subsequent academic work has performed an initial analysis of the protocol [29], emphasizing a number of potential attacks at the boundary of the threat model outlined in the whitepaper. However, this security analysis is far from comprehensive and does not include any formal security definitions or theorems.

Group video calls. E2EE group video calls have not gained any major scrutiny from the academic community. This stands in stark contrast to related fields such as secure text messaging, where the ubiquitously used Signal protocol [35] has received significant attention [2,17,10]. For secure group messaging, the Internet Engineering Task Force (IETF) has even launched the Messaging Layer Security (MLS) working group [8] with mutual support from industry and academia, resulting in a number of analyses [3,4,5].

A defining feature of any group video call that distinguishes it from the asynchronous nature of text messaging is that video calls happen in real-time with all participants online at the same time. This suggests that protocols could achieve strong *liveness properties* generally deemed to be intrinsically unattainable in messaging. First, an attacker should not be able to arbitrarily delay communication. For example, if Alice sends a video stream at time t, then Bob should not accept it at a time significantly later than t. Depending on the type of content, such delays may pose a significant threat; for instance, if the message is an instruction to buy or sell a certain stock, then the ability to delay it might allow an attacker to front-run the transaction. Second, if the meeting host decides on a certain management action, such as adding or removing parties, then an attacker must not be able to delay or prevent those decisions from taking effect. These liveness properties are new and not demanded in the (asynchronous) group messaging setting, in which the network attacker can simply pretend that the initiating party is offline, without any of the other parties being able to detect the attack.

Goals of this work. In this work we aim to analyze the core of Zoom's E2EE meetings protocol[1]. We follow the approach successfully used to analyze (group) messaging protocols and single out the key agreement using the abstraction of a so-called *continuous group key agreement (CGKA)* protocol [3], albeit with weaker forward secrecy (FS) and post-compromise security (PCS) properties than for secure messaging, as explained below. The CGKA abstraction establishes a sequence of shared symmetric keys, accounting for the need to re-key when parties join or leave the meeting (even without strong FS/PCS). The current key — known exactly to the current members of the meeting — can then be used with authenticated encryption with associated data (AEAD) to achieve secure communication.

To provide the first formal security analysis of Zoom's E2EE protocol, our main objectives are, thus, to:

1. Propose a CGKA definition that takes Zoom's unique aspects into account and captures the liveness properties made possible by the online assumption.

[1] We analyze the Zoom E2EE meetings protocol as deployed in the Zoom client version 5.12. In this paper, we refer to this version as the *current* protocol/scheme.

2. Provide an analysis of the core of Zoom's E2EE protocol in the above well-defined model.

To the best of our knowledge, Zoom is the only E2EE group video protocol that aims to provide stringent liveness properties. We believe our work is the first in the realm of CGKA to formalize and analyze such assurances. As part of this process, we observed that Zoom's liveness assurances could be strengthened and thus, we set out to:

3. Propose tangible strengthenings to Zoom's liveness properties, via two simple modifications to the protocol which offer different tradeoffs between efficiency and security. Zoom has evaluated these modifications and successfully deployed a variant of one of them in production (in version 5.13 of the Zoom client).

1.1 Contributions

Definition. We formally define a *leader-based continuous group key agreement with liveness* (LL-CGKA), which encompasses all the desired security properties of Zoom's core E2EE meetings protocol in a single security game[1]. In general terms, an LL-CGKA protocol requires the following properties:

- At each stage of the meeting, the shared symmetric key is only known to the set of current participants as decided by the current meeting host.
- All participants have a consistent view of the set of current meeting participants (as displayed in the UI) as well as of the key.
- Changes to the group, decided by the meeting host, are applied within a bounded (and short) amount of time; otherwise, participants drop out of the meeting.

Attacker model. We consider a powerful adversary that has control over the evolution of the group, fully controls the network and Zoom's server infrastructure, and can passively corrupt any parties, thereby obtaining their current state. We remark, however, that most of our guarantees hold only when the current meeting leader and participants execute the protocol honestly, and any active attackers previously in the meeting have been removed.

FS and PCS. Due to the short-lived nature of video calls, our CGKA notion differs from those in realm of secure messaging by requiring neither strong forward secrecy nor post-compromise security guarantees within a single meeting. An attacker compromising a party's state in an ongoing meeting may learn both past and future content of said meeting. We do, however, require the following properties: first, corrupting a party must not reveal any of the meeting's content before the party has been admitted or after it has been removed by the meeting host. Second, compromising a party after a meeting has ended must not compromise the meeting in any form (weak FS). Third, even if a party's long-term secret have been leaked, this party can still securely join meetings, maintaining confidentiality as long as the adversary does not act as an active meddler-in-the-middle.

Modularization. One of the contributions of this work is to distill out basic building blocks of Zoom's protocol, which could be instantiated differently in pursuit of improved efficiency or, e.g. to achieve post-quantum security. To this end, we consider the intermediate *continuous multi-recipient key encapsulation (cmKEM)* abstraction from which we then build the aforementioned LL-CGKA notion. Put simply, the former naturally captures that in Zoom's protocol a designated party (the meeting host) chooses the symmetric key and distributes it to all the meeting participants. The latter abstraction then models the core of Zoom's E2EE meetings protocol, including the unique liveness properties.

Finally, we discuss how Zoom's overall protocol is built on top of the LL-CGKA protocol, considering audio and video encryption. In particular, we relate the respective confidentiality, authenticity and liveness assurances to those of the LL-CGKA notion.

We remark that the above modularization follows Zoom's whitepaper [11] version 4.0, with the cmKEM notion roughly corresponding to Sections 7.6.2 – 7.6.6, the LL-CGKA notion to Section 7.6.7, and video stream encryption discussed in Sections 7.2 and 7.11, among others.

Liveness. One of the main novelties of Zoom's E2EE protocol is its focus on liveness properties. They assure that whenever the host adds or removes a participant, the action cannot be withheld by an adversary for any extended period of time. That is, if for instance the host removes a member from the group, such as when removing a candidate at the end of an interview so that the hiring panel can reach a decision, that member must no longer be able to decrypt meeting contents even if they manage to compromise Zoom's cloud infrastructure or exert significant control over the network.

In this work, we present a simple time-based model that allows us to formalize and analyze those liveness properties. Our model balances simplicity and generality by assuming that parties have access to local clocks that all run at the same speed, but are otherwise not assumed to be synchronized. We then formalize liveness as follows: whenever a participant is in a given state at time t, then the meeting host has been in the same state recently, i.e., at some time $t' \geq t - \Delta$ where Δ is some protocol-dependent liveness slack. Turned around, whenever the host moves on to a new state (e.g., by changing the group roster) then all participants must also move on within time Δ (or else drop from the meeting).

While the protocol we analyze[1] achieves good liveness properties, these assurances degrade in the number of host changes. As part of this paper we propose two potential improvements. First, we propose a modification that strictly improves on the liveness and yields bounds independent of the number of host changes. This comes at the cost of increased communication by making the protocol more interactive. As an alternative, we propose a strengthening that does not incur any communication overhead and improves on Zoom's properties if parties have well-synchronized clocks; we believe this to be the common case for modern devices. After testing, Zoom implemented a variant of the first option, which was deployed in version 5.13 of the Zoom client.

1.2 Related Work

We have already commented above on the relationship of this work to the areas of secure messaging.

Group video calls. There are numerous solutions for group video calls. The vast majority offers transport layer encryption, with some of them [7,16,11,44,45] offering E2EE group calls, and others offering this feature only for two-party calls [27,35]. While some of the solutions do offer intuitive security descriptions in the form of a whitepaper, such as Wire [45], Cisco [16], and WhatsApp [44], to the best of our knowledge only Cisco WebEx enjoys formal security claims, as it is directly built on top of the IETF MLS draft [8,3,4,5].

Liveness. The terms liveness, liveliness, and aliveness are frequently used to describe various *authentication properties* of key agreement protocols, e.g., in [34] (and many subsequent works). Those properties, roughly speaking, guarantee that if one party completes a run of the protocol, then its peer at some point also has run the same protocol. (Slightly stronger variants exist.) As such most of those definitions not only have no direct relation to *physical time* but also are typically not enforced on an ongoing basis, contrary to our liveness definition. Further, in the context of E2EE group messaging, some work previously used the liveness as synonymous to correctness [40] — with no direct relation to actions having to occur in a timely manner.

However, using timing is not new in the design and analysis of cryptographic protocols. Some such works (e.g., [38,23,30]) use timing assumptions to improve efficiency (or overcome impossibility results) for problems which do not inherently require timing assumptions. Other works (e.g., [22,41,9,12]) use various forms of "moderately hard function" to achieve different cryptographic properties which critically rely on the notion of time. The type of liveness used in this work is much more closely related to more traditional distributed computing literature (e.g., [21,24]) on consensus and, more recently, blockchain protocols (e.g., [26,37]). However, the existence of a unique meeting leader, coupled with the online assumption, makes Zoom's protocol (and our security model) much more lightweight. Finally, the use of heartbeats to ensure liveness is similar to the heartbeat extension of the TLS protocols [42].

Related Notions. The cmKEM notion is an extension of multi-recipient Key Encapsulation (mKEM) [43,39,46] to the setting of dynamically changing groups. Zoom's scheme is based around the authenticated public-key encryption[2] scheme from the `libsodium` library [20]. It is very similar to one of the early authenticated public-key encryption schemes formally analyzed by An [6] (and simpler then the recently analyzed HPKE standard [1]).

The LL-CGKA notion is further related to Dynamic Group Key Agreement with an extensive body of literature, notable examples including [14,28,31]. Similar to CGKA, the Dynamic GKA notion supports changes to group membership during a session and, in fact, in terms of FS and PCS guarantees those notions resemble our LL-CGKA notion more closely than most prior CGKA variants. In contrast to CGKA, Dynamic GKA schemes are designed for an interactive setting, i.e., typically require all parties to contribute to any one operation via interactive rounds, and / or rely on a trusted group manager. (In contrast to LL-CGKA the group manager is, however, static and cannot be replaced mid-session.) Further, we note that while group video calls in principle can tolerate interactive protocols, such as [31], requiring several parties to contribute to each operation can be nevertheless problematic, as for example parties can unexpectedly drop out. Furthermore, we believe this simplifies extending our notion for a more advanced group video call protocol, compared to a Dynamic GKA based one. Closely related to Dynamic GKA are further Multicast Encryption, e.g., [36], and the line of work on Logical Key Hierarchies, e.g., [15].

Another related notion to both cmKEM and LL-CGKA is Multi-Stage Authenticated Key Exchange [25]. Several variants, each with slightly different guarantees, have been considered and the notion has, e.g., been used to analyze the Double Ratchet protocol [18]. In contrast to CGKA, Multi-Stage AKE has exclusively been applied to the two-party setting.

1.3 Organization

In Section 2, we formalize the cmKEM notion, in which one participant in a meeting generates and distributes key material to the remaining participants, and describe Zoom's implementation of this building block. In Section 3, we define the LL-CGKA notion, which strengthens the cmKEM definition with properties such as liveness and group consistency, and present an abstraction of Zoom's LL-CGKA protocol[1]. In Section 4, we point out limitations in Zoom protocol's liveness guarantees, and propose two improved protocols, one of which influenced an update to Zoom's protocol. Finally, in Section 5, we comment on the confidentiality, authenticity, and liveness guarantees of the meeting stream contents itself.

Appendix A describes the syntax, primitives, and cryptographic assumptions used in this paper. Appendix B describes the PKI used by Zoom for long-term key verification. Appendices C, D and E provide more details and formal security proofs about the cmKEM notion, the LL-CGKA notion, and our proposed liveness improvements, respectively.

2 Continuous Multi-Recipient KEM

Zoom's protocol works by having a designated party distribute shared symmetric key material to all the participants upon each change to the group. We abstract this as a *Continuous Multi-Recipient Key Encapsulation* (cmKEM) scheme that allows the designated party to encapsulate a stream of shared symmetric keys to a dynamically evolving set of recipients. This results in a sequence of independent and uniformly random *key*s, each only known to the authorized parties. We number the states (i.e., keys) using two counters: the *epoch* and a sub-epoch called *period*.[3]

In the following, we call the designated party *leader*.[4] We ignore policy aspects and assume that the leader is told whom to add or remove.

[2] Authenticated public-key encryption schemes are often also referred to as *signcryption schemes*. The latter term is however more commonly used to denote schemes satisfying insider security rather than outsider security, as achieved by `libsodium`'s scheme.

[3] Looking ahead, rotating the period instead of the full epoch during group additions is more efficient. Zoom's protocol currently does not take advantage of period rotations, but we capture and analyze this option since it is being considered as a future optimization.

[4] Typically the leader coincides with the meeting host, but if, e.g., the host is on a low-bandwidth connection, these concepts can be decoupled.

The cmKEM notion distinguishes long-term identities and *ephemeral users*. Each long-term identity id is assumed to have an associated public key ipk. A party id can then create one or more ephemeral users, identified by uid, each linked to a specific *meeting*. That is, each meeting will consist of a group of ephemeral uids that just exist for the duration of that meeting. Roughly speaking, in Zoom, each long-term identity id corresponds to a device; if a user logs into multiple devices, each will have its own long-term key material. Note that a device can be part of the same meeting under different ephemeral identities over time, e.g., after leaving the meeting and then rejoining it.

To cope with the leader suddenly losing connection, leader switches are initiated by the (untrusted) server without any hand-off. As a result, a user uid can be asked at any point of time to become the new leader of a meeting, with any given set of participants, as long as they are associated with the same meeting. To simplify notation, we introduce the notion of a *session* that denotes a segment of meeting between leader changes.

2.1 Syntax

For simplicity, we define the clients' cmKEM algorithms to be non-interactive, making all the interaction explicit by having multiple algorithms. User algorithms moreover have implicit access to a PKI as described in the next section. The server aids the protocol execution by performing explicit message routing.

Definition 1. *A cmKEM scheme consists of the following algorithms. For ease of presentation, the client state* st *is assumed to expose the current key* $\mathsf{st.k} \in \mathcal{K}$*, epoch* $\mathsf{st.e}$*, and period* $\mathsf{st.p}$*, while the server state* pub *is assumed to expose a mapping* $\mathsf{pub.E}[\cdot]$ *from meetings to the meeting's current epoch.*

User management:

- $(\mathsf{st}, \mathsf{uid}, \mathsf{sig}) \leftarrow \mathsf{CreateUser}(\mathsf{id}, \mathsf{meetingId})$ *creates an ephemeral user belonging to* id *and the meeting* $\mathsf{meetingId}$*. It outputs the initial state* st*, the user's identity* uid*, and credential* sig *binding* uid *to* id*.*
- $(\mathsf{id}, \mathsf{ipk}) \leftarrow \mathsf{Identity}(\mathsf{uid})$ *and* $\mathsf{meetingId} \leftarrow \mathsf{Meeting}(\mathsf{uid})$ *deterministically compute* uid*'s long-term information, and associated meeting respectively.*

Session management:

- $(\mathsf{st}', \mathsf{M}) \leftarrow \mathsf{StartSession}(\mathsf{st}, \{(\mathsf{uid}_i, \mathsf{ad}_i, \mathsf{sig}_i)\}_{i \in [n]}, \mathsf{e}')$ *instructs the user to start a new session with the given members. For each member, credential* sig_i *as well as associated data* ad_i *(which needs to match with the user's respective value when joining) are provided. The algorithm takes an optional argument* e' *indicating the starting epoch for the session. The welcome message* M *is to be distributed to the other group members by the server.*
- $\mathsf{st}' \leftarrow \mathsf{JoinSession}(\mathsf{st}, \mathsf{uid}_{\mathsf{lead}}, \mathsf{sig}_{\mathsf{lead}}, \mathsf{m}, \mathsf{ad})$ *makes the user join the leader's session using their share* m *of the welcome message.*

Group and management (leader):

- $(\mathsf{st}', \mathsf{M}) \leftarrow \mathsf{Add}\big(\mathsf{st}, \{(\mathsf{uid}_i, \mathsf{ad}_i, \mathsf{sig}_i)\}_{i \in [n]}, \mathsf{newEpoch}\big)$ *adds the users* uid_1 *to* uid_n *to the group. The boolean flag* $\mathsf{newEpoch}$ *indicates whether this action should create a new epoch or period. This algorithm can also be called without any* uid *tuple as input in order to introduce a new key without changing the set of participants.*
- $(\mathsf{st}', \mathsf{M}) \leftarrow \mathsf{Remove}(\mathsf{st}, \{\mathsf{uid}_i\}_{i \in [n]})$ *removes the users* uid_1 *to* uid_n *from the group.*

Message processing (non-leaders):

- $\mathsf{st}' \leftarrow \mathsf{Process}(\mathsf{st}, \mathsf{m})$ *lets a participant advance to the next epoch or period.*

Message passing (server):

- $\mathsf{pub} \leftarrow \mathsf{InitSplitState}()$ *generates an initial public server state.*
- $(\mathsf{pub}, \{(\mathsf{uid}_i, \mathsf{m}_i)\}_{i \in [n]}) \leftarrow \mathsf{Split}(\mathsf{pub}, \mathsf{M})$ *deterministically splits* M *into shares* m_i *for each recipient.*

Correctness is formally defined in Appendix C.2.

2.2 PKI

Each ephemeral user identity uid is bound to its long-term identity id via a signed credential. To this end, id has a long-term signing key isk. In order to prevent meddler-in-the-middle (MITM) attacks, other parties must authenticate the respective long-term public key ipk. For the sake of our analysis, we assume a simple (long-term) public-key infrastructure (PKI). The PKI provides to each long-term identity id their respective private signing key isk while allowing all other users to verify that the respective public verification key ipk belongs to id.

Zoom currently does not have any such PKI but relies on the host reading out a *meeting leader security code* — a digest of ipk — that all participants then compare to ensure they have the host's correct key. Authentication crucially depends on the leader visually recognizing participants and vice versa. Formalizing the exact guarantees given by this process is outside the scope of this work — specifically because the authenticity is only established during and not before a meeting, and because it relies on non-cryptographic assumptions such as the host recognizing participants' faces.

In the future, Zoom plans to build a PKI based on key transparency and external identity providers; analysis thereof is left for future work. We refer to Appendix B for a more in-depth discussion on how Zoom currently verifies public keys, as well as their ongoing efforts for improving user authentication.

2.3 Security Definition

The security notion for the cmKEM primitive encompasses all the desired security properties in a single game. We next describe its the high-level workings, with the full formal definition presented in Appendix C.3.

Game overview. The attacker has full control over the evolution of the group and the network. We now sketch the various oracles the adversary may call. First, the adversary can *create a user* for a provided long-term identity id and meeting $\mathsf{meetingId}$. The game ensures that the generated user id uid is unique.

The adversary can then instruct $\mathsf{uid}_{\mathsf{lead}}$ to *start a session* for a provided list of participants and their respective credentials. Afterwards, they can instruct a user uid to *join* $\mathsf{uid}_{\mathsf{lead}}$*'s session* using a welcome message m of the adversary's choice. The leader can also be instructed to *add or remove members*. In the former case, it is up to the adversary to specify whether this should initiate a new epoch or period. Finally, the adversary can get a participant uid to *process an arbitrary message* m, though the protocol might of course reject malicious messages.

The game ensures that additions and removals only succeed if the adversary does not try to add existing members or remove nonexistent ones. Additionally, the leader must not remove themselves from the group; instead, another party could assume the role of the leader and exclude the old leader from the group. The game keeps track of, for each leader's epoch and period, the leader's view of the session state, which consists of the meeting key and participant roster. Throughout the execution, the game then ensures consistency of the parties' view with their leader's respective view, which we discuss below.

The attacker can passively corrupt long-term identities, which reveals (a) the secret states of all still active associated ephemeral identities and (b) the long-term identity's signing key from the PKI.

Key confidentiality. The adversary must not be able to distinguish the keys produced by the cmKEM scheme from random ones. To this end, the adversary may try to guess a bit b by challenging a state's key (identified by the leader, epoch, and period) to either receive the real key (if $b = 0$) or a uniform random one (if $b = 1$). Additionally, the game allows the adversary to instead request the actual key, irrespective of the bit b, which may be useful since the adversary cannot win if it corrupts any party who knows any of the challenged keys.

The game needs to rule out trivial wins stemming from the adversary being able to compute certain keys themselves after passively corrupting parties. Since Zoom's scheme neither encompasses forward secrecy (FS) nor post-compromise security (PCS) within a session, this has to be reflected in our notion. In short, corrupting a user potentially reveals the key for all epochs and periods where he has been a member of a given session. However, keys must remain secure in the following situations:

- A user must never know keys from before being added to, or after having been removed from the group. Hence, the confidentiality of those keys must not be affected by compromising the given user.
- Corrupting a device after a session has ended, i.e., after the respective ephemeral identity has been deleted, must not affect the session's confidentiality.[5]
- Corrupting a long-term identity id (and thus learned isk) must not affect the security of future sessions involving an honestly generated ephemeral identity uid for id. (The adversary might of course impersonate id by creating a valid ephemeral user uid' instead, which would compromise the session's security.)

Consistency properties. Parties must agree on the key for each epoch and period within a given session. That is, no two parties should ever output conflicting keys, unless after an active attack in which the adversary uses either the leader's or the receiving party's leaked state to tamper with the messages.[6] Consistency, moreover, takes into account at which point in time parties can reach a given state. Our notion distinguishes between epochs and periods, among others, due to those properties differing. Participants must only move to an epoch once their leader has arrived there, while for periods, we allow participants to run ahead and thus reach periods that formally are not supposed to exist. (Still, parties must agree on the keys for those spurious periods.)

Finally, consistency must hold even if the adversary tampers with, reorders, or replaces messages — as long as the involved parties are honest. Due to the leader-based nature of the cmKEM primitive, a malicious leader however could always break consistency by simply sending inconsistent messages to the respective parties. To formalize outsider security, we thus simply deem attacks enabled by corrupting one of the involved parties trivial and no longer enforce consistency properties for a user uid once either uid or their leader $\mathsf{uid}_{\mathsf{lead}}$ has been corrupted.

Member authentication. For many operations such as adding users to an existing session or instructing a user to join another session, the adversary is allowed to provide the respective user identifiers. Our security notion ensures that the adversary cannot impersonate long-term identities unless they have been corrupted, i.e., the adversary cannot inject an ephemeral user uid unless the associated long-term identity id has been corrupted.

2.4 Zoom's Scheme

Zoom's cmKEM scheme uses point-to-point encryption to communicate fresh keys to the participants — i.e., it does not leverage any efficiency gains from sending the same message to multiple recipients. It is based on Diffie-Hellman key exchange over a cyclic group $\mathbb{G} = \langle g \rangle$ with a fixed generator g. The identifier uid mainly consists of a Diffie-Hellman public key $\mathsf{upk} \in \mathbb{G}$ alongside the contextual data of the meeting identifier, the user's long-term identity id, the user's long-term public key ipk, and a signature under the user's long-term signing key isk binding it all together. The respective secret key $\mathsf{usk} := \mathsf{DLog}_g(\mathsf{upk})$ is stored as part of the protocol's state. See Fig. 1 for a formal description of the scheme.

For each epoch, the leader samples a new *seed*, from which the sequence of period keys are derived by iteratively applying a PRG to derive the key and seed for the next period of that epoch[3]. (Observe that this construction is forward-secure.) When removing parties, the leader initiates the next epoch and communicates the new seed to all remaining participants as described below. They then all derive the first key and the seed for the second key using the PRG. Analogously, to add participants with $\mathsf{newEpoch} = \mathtt{true}$, the leader communicates the seed to all participants. More efficiently, however, when adding participants with $\mathsf{newEpoch} = \mathtt{false}$, the leader only sends the seed for the next key to the freshly joined parties and instructs the others to just ratchet forward.

To send a seed to a party, the leader first derives for each recipient a shared symmetric key from a Diffie-Hellman element computed from its own secret key usk and the recipient's public key upk'. The scheme

[5] As in TLS, we require FS on the granularity of sessions.

[6] This formalizes an outsider notion actually achieved by Zoom. Stronger protocols could tolerate leaking the recipient's state.

Protocol cmKEM Client Protocol

```
User management

Algorithm: CreateUser(id, meetingId)
  $\mathsf{usk} \leftarrow\!\!\$\ \{0, 1, \ldots, |\mathbb{G}| - 1\}$; $\mathsf{upk} \leftarrow g^{\mathsf{usk}}$
  $(\mathsf{isk}, \mathsf{ipk}) \leftarrow \mathsf{PKI.get\text{-}sk}(\mathsf{id})$ // id's long-term keys
  $\mathsf{me} \leftarrow (\mathsf{meetingId}, \mathsf{id}, \mathsf{ipk}, \mathsf{upk})$
  $\mathsf{sig} \leftarrow \mathsf{Sig.Sign}(\mathsf{isk}, \text{‘}\mathsf{EncryptionKeyAnnouncement}', \mathsf{me})$
  $K[\cdot], \mathsf{uid}_{\mathsf{lead}}, G, \mathsf{k}, \mathsf{e}, \mathsf{p}, \mathsf{seed} \leftarrow \bot$
  return $(\mathsf{me}, \mathsf{sig})$

Algorithm: Meeting(uid)
  parse $(\mathsf{meetingId}, \mathsf{id}, \mathsf{ipk}, \mathsf{upk}) \leftarrow \mathsf{uid}$
  return meetingId

Algorithm: Identity(uid)
  parse $(\mathsf{meetingId}, \mathsf{id}, \mathsf{ipk}, \mathsf{upk}) \leftarrow \mathsf{uid}$
  return $(\mathsf{id}, \mathsf{ipk})$

Session management

Algorithm: StartSession($\{(\mathsf{uid}_i, \mathsf{ad}_i, \mathsf{sig}_i)\}_{i \in [n]}, \mathsf{e}'$)
  if $\mathsf{e}' = \bot$ then
    if $\mathsf{e} = \bot$ then $\mathsf{e}' \leftarrow 1$
    else $\mathsf{e}' \leftarrow \mathsf{e} + 1$
  else
    req $\mathsf{e}' > \mathsf{e} \vee \mathsf{e} = \bot$
  $G \leftarrow \{\mathsf{uid}_1, \ldots, \mathsf{uid}_n\}$
  req $\mathsf{me} \neq \bot \wedge \mathsf{me} \notin G$
  $\mathsf{AD}[\cdot] \leftarrow \bot$
  for $i \in [n]$ do
    req *verify-user($\mathsf{uid}_i, \mathsf{sig}_i$)
    $\mathsf{AD}[\mathsf{uid}_i] \leftarrow \mathsf{ad}_i$
  $\mathsf{uid}_{\mathsf{lead}} \leftarrow \mathsf{me}$
  $\mathsf{e} \leftarrow \mathsf{e}'$
  $\mathsf{p} \leftarrow 0$
  $\mathsf{seed} \leftarrow \mathsf{PRG.Init}(1^\kappa)$
  $\mathsf{C} \leftarrow$ *encrypt-seed($G, \mathsf{AD}$)
  $\mathsf{M} \leftarrow (\mathsf{meetingId}, \mathsf{e}, G, \mathsf{C})$
  $(\mathsf{seed}, \mathsf{k}) \leftarrow \mathsf{PRG.Eval}(\mathsf{seed})$
  return M

Algorithm: JoinSession($\mathsf{uid}'_{\mathsf{lead}}, \mathsf{sig}'_{\mathsf{lead}}, \mathsf{m}, \mathsf{ad}$)
  req $\mathsf{me} \neq \bot \wedge \mathsf{uid}'_{\mathsf{lead}} \neq \mathsf{me} \wedge \mathsf{uid}'_{\mathsf{lead}} \neq \mathsf{uid}_{\mathsf{lead}}$
  req *verify-user($\mathsf{uid}'_{\mathsf{lead}}, \mathsf{sig}'_{\mathsf{lead}}$)
  $\mathsf{uid}_{\mathsf{lead}} \leftarrow \mathsf{uid}'_{\mathsf{lead}}$
  parse $(\text{‘}\mathsf{epoch}', \mathsf{c}) \leftarrow \mathsf{m}$
  $(\mathsf{e}', \mathsf{p}', \mathsf{seed}') \leftarrow$ *decrypt-seed($\mathsf{c}, \mathsf{uid}'_{\mathsf{lead}}, \mathsf{ad}$)
  if $\mathsf{e} \neq \bot$ then
    req $\mathsf{e}' > \mathsf{e}$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{e}', \mathsf{p}')$
  $(\mathsf{seed}, \mathsf{k}) \leftarrow \mathsf{PRG.Eval}(\mathsf{seed}')$
```

```
Group and key management (leader)

Algorithm: Add($\{(\mathsf{uid}_i, \mathsf{ad}_i, \mathsf{sig}_i)\}_{i \in [n]}, \mathsf{newEpoch}$)
  req $\mathsf{me} \neq \bot \wedge \mathsf{uid}_{\mathsf{lead}} = \mathsf{me}$
  $\mathsf{AD}[\cdot] \leftarrow \bot$
  for $i \in [n]$ do
    req $\mathsf{uid}_i \neq \mathsf{me} \wedge \mathsf{uid}_i \notin G \wedge$ *verify-user($\mathsf{uid}_i, \mathsf{sig}_i$)
    $G \leftarrow G \cup \{\mathsf{uid}_i\}$
    $\mathsf{AD}[\mathsf{uid}_i] \leftarrow \mathsf{ad}_i$
  if newEpoch then
    $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{e} + 1, 0)$
    $\mathsf{seed} \leftarrow \mathsf{PRG.Init}(1^\kappa)$
    $G' \leftarrow G$
  else
    $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{e}, \mathsf{p} + 1)$
    $G' \leftarrow \{\mathsf{uid}_1, \ldots, \mathsf{uid}_n\}$
  $\mathsf{C} \leftarrow$ *encrypt-seed($G', \mathsf{AD}$)
  $(\mathsf{seed}, \mathsf{k}) \leftarrow \mathsf{PRG.Eval}(\mathsf{seed})$
  return $\mathsf{M} \leftarrow (\mathsf{meetingId}, \mathsf{e}, G, \mathsf{C})$

Algorithm: Remove($\{\mathsf{uid}_i\}_{i \in [n]}$)
  req $\mathsf{me} \neq \bot \wedge \mathsf{uid}_{\mathsf{lead}} = \mathsf{me}$
  for $i \in [n]$ do req $\mathsf{uid}_i \in G$
  $G \leftarrow G \setminus \{\mathsf{uid}_1, \ldots, \mathsf{uid}_n\}$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{e} + 1, 0)$
  $\mathsf{seed} \leftarrow \mathsf{PRG.Init}(1^\kappa)$
  $\mathsf{AD}[\cdot] \leftarrow \bot$
  $\mathsf{C} \leftarrow$ *encrypt-seed($G, \mathsf{AD}$)
  $(\mathsf{seed}, \mathsf{k}) \leftarrow \mathsf{PRG.Eval}(\mathsf{seed})$
  return $\mathsf{M} \leftarrow (\mathsf{meetingId}, \mathsf{e}, G, \mathsf{C})$

Message processing (participants)

Algorithm: Process(m)
  req $\mathsf{me} \neq \bot \wedge \mathsf{uid}_{\mathsf{lead}} \neq \bot \wedge \mathsf{uid}_{\mathsf{lead}} \neq \mathsf{me}$
  if $\mathsf{m} = (\text{‘}\mathsf{epoch}', \mathsf{c})$ then
    $(\mathsf{e}', \mathsf{p}', \mathsf{seed}') \leftarrow$ *decrypt-seed($\mathsf{c}, \mathsf{uid}_{\mathsf{lead}}, \bot$)
    req $(\mathsf{e}', \mathsf{p}') = (\mathsf{e} + 1, 0)$
    $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{e}', \mathsf{p}')$
    $(\mathsf{seed}, \mathsf{k}) \leftarrow \mathsf{PRG.Eval}(\mathsf{seed}')$
  else if $\mathsf{m} = \text{‘}\mathsf{period}'$ then
    $\mathsf{p} \leftarrow \mathsf{p} + 1$
    $(\mathsf{seed}, \mathsf{k}) \leftarrow \mathsf{PRG.Eval}(\mathsf{seed})$
```

```
Helper: *encrypt-seed($G', \mathsf{AD}$)
  $\mathsf{C}[\cdot] \leftarrow \bot$
  for $\mathsf{uid}' \in G'$ do
    parse $(\cdot, \cdot, \cdot, \mathsf{upk}') \leftarrow \mathsf{uid}'$
    $\mathsf{nonce} \leftarrow\!\!\$\ \mathsf{AEAD}.\mathcal{N}$
    if $K[\mathsf{uid}'] = \bot$ then
      $K[\mathsf{uid}'] \leftarrow \mathsf{HKDF}((\mathsf{upk}')^{\mathsf{usk}}, \text{‘}\mathsf{KeyMeetingSeed}')$
    $\mathsf{ad}' \leftarrow (\mathsf{meetingId}, \mathsf{me}, \mathsf{AD}[\mathsf{uid}'])$
    $\mathsf{ad}'' \leftarrow \mathsf{Hash}(\text{‘}\mathsf{EncryptionKeyMeetingSeed}' \parallel \mathsf{Hash}(\mathsf{ad}'))$
    $\mathsf{c}' \leftarrow \mathsf{AEAD.Enc}(K[\mathsf{uid}'], \mathsf{nonce}, (\mathsf{e}, \mathsf{p}, \mathsf{seed}), \mathsf{ad}'')$
    $\mathsf{C}[\mathsf{uid}'] \leftarrow (\mathsf{c}', \mathsf{nonce})$
  return C
```

```
Helper: *decrypt-seed($\mathsf{c}, \mathsf{uid}_{\mathsf{lead}}, \mathsf{ad}$)
  parse $(\cdot, \mathsf{id}', \cdot, \mathsf{upk}') \leftarrow \mathsf{uid}_{\mathsf{lead}}$
  if $K[\mathsf{uid}_{\mathsf{lead}}] = \bot$ then
    $K[\mathsf{uid}_{\mathsf{lead}}] \leftarrow \mathsf{HKDF}((\mathsf{upk}')^{\mathsf{usk}}, \text{‘}\mathsf{KeyMeetingSeed}')$
  $\mathsf{ad}' \leftarrow (\mathsf{meetingId}, \mathsf{id}', \mathsf{ad})$
  $\mathsf{ad}'' \leftarrow \mathsf{Hash}(\text{‘}\mathsf{EncryptionKeyMeetingSeed}') \parallel \mathsf{Hash}(\mathsf{ad}'))$
  parse $(\mathsf{c}', \mathsf{nonce}) \leftarrow \mathsf{c}$
  parse $(\mathsf{e}', \mathsf{p}', \mathsf{seed}') \leftarrow \mathsf{AEAD.Dec}(K[\mathsf{uid}_{\mathsf{lead}}], \mathsf{nonce}, \mathsf{c}', \mathsf{ad}'')$
  return $(\mathsf{e}', \mathsf{p}', \mathsf{seed}')$

Helper: *verify-user($\mathsf{uid}', \mathsf{sig}'$)
  parse $(\mathsf{meetingId}', \mathsf{id}', \mathsf{ipk}', \mathsf{upk}') \leftarrow \mathsf{uid}'$
  return $\mathsf{meetingId}' = \mathsf{meetingId}$
    $\wedge$ $\mathsf{Sig.Verify}(\mathsf{ipk}', \text{‘}\mathsf{EncryptionKeyAnnouncement}', \mathsf{uid}', \mathsf{sig}')$
    $\wedge$ $\mathsf{PKI.verify\text{-}pk}(\mathsf{id}', \mathsf{ipk}')$
```

Fig. 1: The client protocol of Zoom's cmKEM scheme. The protocol implicitly maintains a state st which includes $\mathsf{me}, \mathsf{usk}, \mathsf{isk}, \mathsf{uid}, K, \mathsf{uid}_{\mathsf{lead}}, G, \mathsf{seed}$ as well as the exposed key $\mathsf{st.k}$, epoch $\mathsf{st.e}$, and period $\mathsf{st.p}$.

Protocol cmKEM Server Protocol

```
Algorithm: Init()
    $E[\cdot] \leftarrow \bot$

Algorithm: Split(M)
    parse $(meetingId, e', G, C) \leftarrow M$
    req $e' > E[meetingId] \vee E[meetingId] = \bot$
    $E[meetingId] \leftarrow e'$
    $ms[\cdot] \leftarrow \bot$
    for $uid \in G$ do
        if $C[uid] \neq \bot$ then
            $ms[uid] \leftarrow$ ('epoch', $C[uid']$)
        else
            $ms[uid] \leftarrow$ 'period'
    return ms
```

Fig. 2: The server protocol of the cmKEM scheme. The protocol implicitly maintains a state pub that just contains the mapping from meetings to their respective latest epoch number received pub.E[meetingId].

uses HKDF for this derivation, which for the purpose of the security analysis we model as a random oracle. For efficiency reasons, this key is cached as part of the sender's secret state and reused for future messages to or from the same party. The seed is then encrypted using a nonce-based AEAD with a random nonce that is transmitted as part of the resulting ciphertext. The associated data contains the meeting and sender identifiers, as well a fixed context string.

Server protocol. The Split protocol works by delivering the respective AEAD-ciphertext to each party and sending a special "ratchet period" message to parties for which no such ciphertext is specified. In terms of state, the server simply stores the latest epoch for each meeting meetingId, which is exposed as pub.E[meetingId]. For simplicity, we model that the message $\mathsf{M} = (\mathsf{meetingId}, \mathsf{e}, G, \mathsf{C})$ sent to the server includes the meeting identifier, the current epoch, as well as set of recipients. Each recipients uid' for which C contains a share obtains $\mathsf{m} = (\text{'epoch'}, \mathsf{C}[\mathsf{uid}'])$, while for other recipients the server delivers $\mathsf{m} = \text{'period'}$. The scheme is depicted in Fig. 2.

Security. The following theorem establishes the security of the scheme.

Theorem 1. *Zoom's cmKEM scheme is secure according to the outlined definition under the Gap-DH assumption, if the AEAD scheme is secure,* Hash *collision resistant, the signature scheme is EUF-CMA secure, the PRG satisfies the standard indistinguishability from random notion, and* HKDF *is modeled as a random oracle.*

A full proof is presented in Appendix C.4. In short, based on the security of Gap Diffie-Hellman, we can first switch to a hybrid where we use independently generated symmetric keys as opposed to the outputs of the DH operation (between the leader and each participant), programming the random oracle to make things look consistent on corruption. Then, we can argue that each of the adversary's winning conditions in the game cannot be triggered based on the unforgeability of the signature scheme (credentials cannot be forged), the authenticity of the AEAD (malicious keys cannot be injected), and the confidentiality of the AEAD (encrypted keys cannot be distinguished from encryptions of random messages).

According to the whitepaper [11], Zoom's scheme performs Diffie-Hellman over Curve25519.[7] We note that the Gap-DH assumption (rather than, e.g., CDH) appears to be rather intrinsic to this kind of simple Diffie-Hellman based protocol and has been assumed for Curve25519 before [10,1]. Moreover, Zoom uses XChaCha20Poly1305 with 192-bit nonces as the nonce-based AEAD scheme, and HKDF for both the key-derivation as well as the PRG. (We model the latter use as a PRG to clarify the exact required security properties.) Finally, for a signature scheme, Zoom uses EdDSA over Ed25519 satisfying EUF-CMA security [13].

[7] Technically, Curve25519 breaks the abstraction of cyclic groups we use (for simplicity) to present our scheme. We refer to the analysis of the HPKE standard [1] for an extended discussion and the formalization of *nominal groups* with the respective Gap-DH assumption. Their results directly apply to our construction.

3 Leader-based GCKA with Liveness

We now abstract the core of Zoom's E2EE meetings protocol[1] as a *leader-based continuous group key agreement with liveness* (LL-CGKA) scheme. On a high level, the primitive works similarly to the previously introduced cmKEM one, with the following differences: (1) participants are aware of the group roster and in particular only use keys for which they know the roster, (2) as a result participants can no longer run ahead of their leader in terms of the period, and (3) liveness is enforced.

Liveness. To achieve liveness, the LL-CGKA primitive is time based. More concretely (1) algorithms can depend on time and (2) in addition to event-based actions (e.g., reacting to an incoming packet), there are also time-driven actions. We make the following (simplifying) assumptions:

- Each party has a *local clock*, and all clocks run at *the same speed* (constant drift).
- Local algorithms complete instantaneously, i.e., no time elapses between invocation and completion. As a consequence, the algorithms simply take the party's current time as an input argument.

We remark that the vast majority of Zoom meetings last only a couple of hours, which limits any significant amount of clock drift in practice and thus justifies the first assumption.

3.1 Syntax

The algorithms of a LL-CGKA scheme closely follow the ones of a cmKEM scheme, with two major differences. First, client algorithms take the current *local time* as input. Second, there are *clock ticking* algorithms that allow to specify clock-driven actions, i.e., actions that happen at a certain time rather upon receiving a message.

As in cmKEM, the server performs message routing. Additionally, it hands out the current public[8] group state to newly joining parties, freeing the leader from maintaining additional state.

Definition 2. *A LL-CGKA scheme consists of the algorithms described in the following, where, for ease of presentation, the client state* ust *is assumed to expose the following fields:*

- *The user's current epoch* $\mathsf{ust.e}$ *and period* $\mathsf{ust.p}$.
- *For each epoch and period a key* $\mathsf{ust.k[e,p]}$ *(or $\perp$ if not known yet). It is assumed that operations do not change keys once they are defined.*
- *The user's current view on the group* $\mathsf{ust}.G$.

User management:

- $(\mathsf{ust}, \mathsf{uid}, \mathsf{sig}) \leftarrow \mathsf{CreateUser}(\mathsf{time}, \mathsf{id}, \mathsf{meetingId})$ *creates an ephemeral user for the given identity and meeting. Outputs the user's initial state* ust*, their identity* uid*, and credential* sig *binding* uid *to* id.
- $\mathsf{id} \leftarrow \mathsf{Identity}(\mathsf{uid})$ *and* $\mathsf{meetingId} \leftarrow \mathsf{Meeting}(\mathsf{uid})$ *are deterministic algorithms that return the ephemeral user's long-term identity and meeting, respectively.*
- $\mathsf{ust}' \leftarrow \mathsf{CatchUp}(\mathsf{ust}, \mathsf{time}, \mathsf{grpPub})$ *prepares the user for joining the group by processing the current public group state* grpPub *provided by the sever.*

Leader's algorithms:

- $(\mathsf{ust}', \mathsf{M}) \leftarrow \mathsf{Lead}(\mathsf{ust}, \mathsf{time}, \{(\mathsf{uid}_i, \mathsf{sig}_i)\}_{i \in [n]})$ *instructs the user to become the new group leader with the specified participants. Outputs a message to be split and distributed to the other group members.*
- $(\mathsf{ust}', \mathsf{M}) \leftarrow \mathsf{Add}(\mathsf{ust}, \mathsf{time}, \{(\mathsf{uid}_i, \mathsf{sig}_i)\}_{i \in [n]})$ *is used to add users* uid_1 *to* uid_n *to the group.*
- $(\mathsf{ust}', \mathsf{M}) \leftarrow \mathsf{Remove}(\mathsf{ust}, \mathsf{time}, \{\mathsf{uid}_i\}_{i \in [n]})$ *is used to remove users* uid_1 *to* uid_n *from the group.*
- $(\mathsf{ust}', \mathsf{M}) \leftarrow \mathsf{LeaderTick}(\mathsf{ust}, \mathsf{time})$ *is executed on each clock tick by the leader. Outputs the leader's updated state and an optional messages* M.

[8] By public, we mean known to the (untrusted) Zoom server; i.e., the current roster, but not any keys.

Participants' algorithms:

- $\mathsf{ust}' \leftarrow \mathsf{Follow}(\mathsf{ust}, \mathsf{time}, \mathsf{m}, \mathsf{uid}'_{\mathsf{lead}}, \mathsf{sig}'_{\mathsf{lead}})$ *instructs the user to treat* $\mathsf{uid}'_{\mathsf{lead}}$ *as the new leader. Expects the first message share* m *from the new leader.*
- $\mathsf{ust}' \leftarrow \mathsf{Process}(\mathsf{ust}, \mathsf{time}, \mathsf{m})$ *is used by participants to process any incoming message* m*.*
- $(\mathsf{ust}', \mathsf{alive}, \mathsf{sig}') \leftarrow \mathsf{ParticipantTick}(\mathsf{ust}, \mathsf{time})$ *is executed by a participant on each clock tick. The flag* alive *indicates whether the participant is still in the meeting or dropped out (for a violation of liveness) and optionally updates the credential (for the server) with* $\mathsf{sig}' = \bot$ *denoting no update.*

Server's algorithms:

- $\mathsf{pub} \leftarrow \mathsf{Init}()$ *generates an initial server state.*
- $\big(\mathsf{pub}', \{(\mathsf{uid}_i, \mathsf{m}_i)\}_{i\in[n]}\big) \leftarrow \mathsf{Split}(\mathsf{pub}, \mathsf{M})$ *is a deterministic algorithm that takes message* M *and splits out each user* uid*'s share* m*.*
- $\mathsf{grpPub} \leftarrow \mathsf{GroupState}(\mathsf{pub}, \mathsf{meetingId})$ *is a deterministic algorithm that returns the public group state.*

We discuss correctness, and in particular how it is affected by the liveness properties, in Appendix D.4.

Meeting Flow. Let us briefly discuss how Zoom uses the above defined LL-CGKA abstraction to orchestrate a meeting. Parties first generate a per-meeting ephemeral identity using CreateUser and communicate it to the Zoom server. Before any party can start or join the meeting, the server hands them the most recent public group state (using GroupState) that the party processes using CatchUp.

To start a meeting, Zoom then instructs the initial host to invoke the Lead algorithm. Later, the server can instruct the leader to modify the set of meeting participants using Add and Remove. All messages sent by the leader to current participants are mediated through the server, which uses Split to compute the share of the message that each participant needs.

Participants join the meeting by invoking Follow with their respective share of the message generated by the leader's corresponding Lead (or Add) invocation. They also use Process when receiving further messages from the same leader. Observe that, in some cases, the Follow invocation might not be enough for participants to know about the group roster and the latest key. Instead, it might take up to an additional message generated by LeaderTick for the participant to fully join the meeting.

Analogously, to switch leaders, the new one is instructed to invoke Lead and all other participants are instructed to invoke Follow again. Note that it is not required for the new leader to already be part of the group — the CatchUp algorithm can directly be followed by Lead (instead of Follow) to immediately start as the new leader.

3.2 Security Definition

Overall, the game follows closely the one of the cmKEM primitive outlined in Section 2.3. In the following we discuss the key aspects and highlight the differences to the cmKEM game. We refer to Appendix D.1 for a formal definition.

Clocks. The security game maintains a global clock time. Each honestly created user uid maintains a local clock that is specified as an offset to the global one; that is, all local clocks run at the same speed. (For our analysis, we do not make use of the fact that two users uid and uid' belonging to the same long-term identity id, i.e. a device, typically would have the same local clock. Trivially, our results also hold for this special case.)

The adversary chooses each user's offset and drives the global clock, i.e., decides whenever the clock is supposed to advance by a tick. Those ticks model an abstract discrete unit of time, which can be thought of as milliseconds or nanoseconds, roughly corresponding to the precision of clocks used by the various parties. Whenever the adversary ticks the global clock, each party's local clock thus also advances, and their respective procedures LeaderTick or ParticipantTick are invoked, depending on whether the party is currently a leader or not.

Liveness. An important objective of the LL-CGKA primitive is to ensure liveness: all participants must either keep up with the current meeting's state or drop out of the meeting. This is formalized as follows: whenever ParticipantTick indicates that uid is still alive, then the participant's state must not be too outdated, which in turn is defined as that the participant's current leader *must have been in the same state recently*. How recently, exactly, is a parameter of our security definition we call the *liveness slack*; we introduce the concrete slack achieved by Zoom's protocol as part of its description below.

Observe that this formalization essentially means that whenever the leader makes a change to the group by either adding or removing parties (resulting in an epoch or period change), this change cannot be withheld by a malicious server. We thus call this property *key liveness* and briefly discuss *content liveness* in Section 5.

Confidentiality. Group key confidentiality is formalized analogously as in the cmKEM scheme. That is, upon a challenge, the security game outputs either the real or an independent uniform random key depending on a bit b. We remark that the game only allows to challenge keys for epochs and periods that are to be used in the higher-level application, omitting those that are skipped by the LPL mechanism and hence never output. This simplifies the notion as, in contrast to the cmKEM security notion, each challengeable key has a well-defined group roster associated.

Consistency, authenticity, and no-merging. The game ensures both *key consistency* and *group consistency*, meaning that for a given honest (and uncompromised) leader, epoch, and period, all (uncompromised) participants agree on a key and group roster. However, a malicious server could cause the group to split by assigning different leaders to different partitions of the group, in which case those partitions will no longer agree on the key. Furthermore, two parties, say Alice and Bob, in different partitions might both believe to have a third party Charlie in their group, or even believe to be in the same group with the other party – that is, group rosters output by different partitions are not guaranteed to be disjoint. However, the game ensures that after such an aforementioned (inherent) splitting attack, the various partitions cannot be re-merged into a consistent state, which makes the attack easier to detect.

Remark 1 (Insider security). This work focuses on outsider security, and only formalizes limited insider security guarantees. For instance, whenever the adversary performs a trivial injection, enabled by, e.g. a corruption of the leader, most security properties are (temporarily) disabled. On the other hand, we do formalize that confidentiality and authenticity recover after switching from a malicious leader to an honest one. While Zoom's protocol does not aim to provide strong guarantees in the presence of malicious insiders (as full insider security would, e.g. require asymmetric authentication for video data), a more comprehensive analysis of the properties it does achieve would nevertheless be interesting.

3.3 Zoom's Scheme

We now describe Zoom's LL-CGKA scheme. On a high-level, the protocol enhances the cmKEM scheme by having the leader broadcast the group membership to all participants, as well as regular *heartbeat messages* that help guarantee liveness. A formal description is presented in Fig. 3. Additional details can be found in Appendix D.2.

Leader Participant List. For the participants to learn the group roster, the session leader broadcasts the so-called *leader participant list (LPL)* tabulating the members. The LPL is, for bandwidth efficiency, represented as a linked list of differential updates containing the set of added and removed participants since the last LPL. Each message also references the leader's current epoch e and period p. For efficiency reasons, an LPL message is not sent on every single change to the group roster, but on regular intervals instead. (It is skipped if no change to the group roster has been done in the meantime.) To ensure that parties know to whom they speak to, the scheme only proceeds to epochs and periods that have been certified by an LPL message. Re-keying is nevertheless performed eagerly, potentially leading to unused keys.

The protocol furthermore relies on the LPL to communicate the group roster to newly joining parties. To avoid having new parties process the entire history of LPL messages, thus increasing the server's storage

Protocol Zoom's Client LL-CGKA

User management

```
Algorithm: CreateUser(time, id, meetingId)
  $(\mathsf{st}, \mathsf{me}, \mathsf{sig}) \leftarrow \mathrm{cmKEM}.\mathsf{CreateUser}(\mathsf{id}, \mathsf{meetingId})$
  $(\mathsf{isk}, \cdot) \leftarrow \mathsf{PKI.get\text{-}sk}(\mathsf{id})$
  $\mathsf{lastHb} \leftarrow \mathsf{time}$
  $\mathsf{uid}_{\mathsf{lead}} \leftarrow \bot$
  $G \leftarrow \varnothing$
  $\mathsf{e}, \mathsf{p}, \mathsf{e}_{\mathsf{next}}, \mathsf{p}_{\mathsf{next}}, \mathsf{v}, \mathsf{t} \leftarrow 0$
  $\mathsf{e}_{\mathsf{pub}} \leftarrow \bot$
  $\mathsf{k}[\cdot,\cdot] \leftarrow \bot$
  $\delta[\cdot] \leftarrow \infty$
  $\mathsf{lplHash}, \mathsf{hbHash} \leftarrow \bot$
  return $(\mathsf{me}, \mathsf{sig})$

Algorithm: Identity(uid)
  return $\mathrm{cmKEM}.\mathsf{Identity}(\mathsf{uid})$

Algorithm: Meeting(uid)
  return $\mathrm{cmKEM}.\mathsf{Meeting}(\mathsf{uid})$

Algorithm: CatchUp(time, grpPub)
  req $\mathsf{uid}_{\mathsf{lead}} = \bot$
  parse $(\mathsf{e}_{\mathsf{pub}}, \mathsf{lpls}', \mathsf{hb}') \leftarrow \mathsf{grpPub}$
  // Process LPLs
  while $\mathsf{lpls}' \neq \langle\rangle$ do
      $\mathsf{lpl}' \leftarrow \mathsf{lpls}'.\mathsf{deq}()$
      try $*\texttt{receive-LPL}(\mathsf{lpl}')$
  // Store Heartbeat (no verification)
  $\mathsf{hbHash} \leftarrow \mathsf{Hash}(\mathsf{hb}')$
```

Participants' algorithms

```
Algorithm: Follow(time, $\mathsf{m}'$, $\mathsf{uid}'_{\mathsf{lead}}$, $\mathsf{sig}'_{\mathsf{lead}}$)
  req $\mathsf{uid}'_{\mathsf{lead}} \neq \bot \wedge \mathsf{uid}'_{\mathsf{lead}} \neq \mathsf{me}$
  parse $(\mathsf{m}'_K, \mathsf{lpl}', \mathsf{hb}') \leftarrow \mathsf{m}'$
  if $\mathsf{uid}_{\mathsf{lead}} \neq \bot$ then
      req $\mathsf{lpl}' \neq \bot \wedge \mathsf{hb}' \neq \bot$
  try $\mathsf{st} \leftarrow \mathrm{cmKEM}.\mathsf{JoinSession}(\mathsf{st}, \mathsf{uid}'_{\mathsf{lead}}, \mathsf{sig}'_{\mathsf{lead}}, \mathsf{m}'_K, \bot)$
  $\mathsf{Key}[\mathsf{st.e}, \mathsf{st.p}] \leftarrow \mathsf{st.k}$
  $\mathsf{uid}_{\mathsf{lead}} \leftarrow \mathsf{uid}'_{\mathsf{lead}}$
  $\mathsf{e}_{\mathsf{pub}} \leftarrow \bot$
  if $\mathsf{lpl}' \neq \bot$ then
      try $*\texttt{receive-LPL}(\mathsf{lpl}')$
      req $\mathsf{me} \in G \wedge \mathsf{hb}' \neq \bot \wedge (\mathsf{e}_{\mathsf{next}}, \mathsf{p}_{\mathsf{next}}) = (\mathsf{st.e}, \mathsf{st.p})$
  if $\mathsf{hb}' \neq \bot$ then
      try $*\texttt{receive-heartbeat}(\mathsf{hb}')$

Algorithm: Process(time, $\mathsf{m}'$)
  req $\mathsf{uid}_{\mathsf{lead}} \neq \bot \wedge \mathsf{uid}_{\mathsf{lead}} \neq \mathsf{me}$
  parse $(\mathsf{m}'_K, \mathsf{lpl}', \mathsf{hb}') \leftarrow \mathsf{m}'$
  if $\mathsf{m}'_K \neq \bot$ then
      try $\mathsf{st} \leftarrow \mathrm{cmKEM}.\mathsf{Process}(\mathsf{st}, \mathsf{m}'_K)$
      $\mathsf{Key}[\mathsf{st.e}, \mathsf{st.p}] \leftarrow \mathsf{st.k}$
  if $\mathsf{lpl}' \neq \bot$ then
      try $*\texttt{receive-LPL}(\mathsf{lpl}')$
      req $\mathsf{me} \in G \wedge \mathsf{hb}' \neq \bot$
  if $\mathsf{hb}' \neq \bot$ then
      try $*\texttt{receive-heartbeat}(\mathsf{hb}')$
```

Leader's algorithms

```
Algorithm: Lead(time, $\{(\mathsf{uid}_i, \mathsf{sig}_i)\}_{i\in[n]}$)
  if $\mathsf{e}_{\mathsf{pub}} \neq \bot$ then $\mathsf{e}' \leftarrow \mathsf{e}_{\mathsf{pub}} + 1$
  else $\mathsf{e}' \leftarrow \bot$
  try $(\mathsf{st}, \mathsf{M}_K) \leftarrow \mathrm{cmKEM}.\mathsf{StartSession}(\mathsf{st}, \{(\mathsf{uid}_i, \bot, \mathsf{sig}_i)\}_{i\in[n]}, \mathsf{e}')$
  $\mathsf{uid}_{\mathsf{lead}} \leftarrow \mathsf{me}$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{st.e}, \mathsf{st.p})$
  $\mathsf{e}_{\mathsf{pub}} \leftarrow \bot$
  $G \leftarrow \{\mathsf{uid}_1, \ldots, \mathsf{uid}_n\} \cup \{\mathsf{me}\}$
  $\mathsf{Added}, \mathsf{Removed} \leftarrow \varnothing$
  $\mathsf{numLplLinks} \leftarrow \mathsf{max\text{-}links}$
  $\mathsf{lpl} \leftarrow *\texttt{send-LPL}()$
  $\mathsf{hb} \leftarrow *\texttt{send-heartbeat}()$
  $\mathsf{M} \leftarrow (\mathsf{me}, \mathsf{M}_K, \mathsf{lpl}, \mathsf{hb})$
  return $\mathsf{M}$

Algorithm: Add(time, $\{(\mathsf{uid}_i, \mathsf{sig}_i)\}_{i\in[n]}$)
  req $\mathsf{uid}_{\mathsf{lead}} = \mathsf{me}$
  for $i \in [n]$ do
      req $\mathsf{uid}_i \notin G$
      $G \leftarrow G \cup \{\mathsf{uid}_i\}$
      $\mathsf{Added} \leftarrow \mathsf{Added} \cup \{\mathsf{uid}_i\}$
  if $\mathsf{p} \geq \mathsf{p}_{\mathsf{MAX}}$ then
      try $(\mathsf{st}, \mathsf{M}_K) \leftarrow \mathrm{cmKEM}.\mathsf{Add}(\mathsf{st}, \{(\mathsf{uid}_i, \bot, \mathsf{sig}_i)\}_{i\in[n]}, \texttt{true})$
      $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{st.e}, \mathsf{st.p})$
  else
      try $(\mathsf{st}, \mathsf{M}_K) \leftarrow \mathrm{cmKEM}.\mathsf{Add}(\mathsf{st}, \{(\mathsf{uid}_i, \bot, \mathsf{sig}_i)\}_{i\in[n]}, \texttt{false})$
  $\mathsf{M} \leftarrow (\mathsf{me}, \mathsf{M}_K, \bot, \bot)$
  return $\mathsf{M}$

Algorithm: Remove(time, $\{\mathsf{uid}_i\}_{i\in[n]}$)
  req $\mathsf{uid}_{\mathsf{lead}} = \mathsf{me}$
  for $i \in [n]$ do
      req $\mathsf{uid}_i \in G \wedge \mathsf{uid}_i \neq \mathsf{me}$
  $G \leftarrow G \setminus \{\mathsf{uid}_1, \ldots, \mathsf{uid}_n\}$
  $\mathsf{Removed} \leftarrow \mathsf{Removed} \cup \{\mathsf{uid}_1, \ldots, \mathsf{uid}_n\}$
  $\mathsf{Added} \leftarrow \mathsf{Added} \setminus \{\mathsf{uid}_1, \ldots, \mathsf{uid}_n\}$
  try $(\mathsf{st}, \mathsf{M}_K) \leftarrow \mathrm{cmKEM}.\mathsf{Remove}(\mathsf{st}, \{\mathsf{uid}_i\}_{i\in[n]})$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{st.e}, \mathsf{st.p})$
  $\mathsf{M} \leftarrow (\mathsf{me}, \mathsf{M}_K, \bot, \bot)$
  return $\mathsf{M}$
```

Time driven

```
Algorithm: LeaderTick(time)
  req $\mathsf{uid}_{\mathsf{lead}} = \mathsf{me}$
  $\mathsf{lpl}, \mathsf{hb} \leftarrow \bot$
  if $\mathsf{time} - \mathsf{lastHb} \geq \Delta_{\mathsf{LPL}}$ then
      if $\mathsf{Added} \neq \varnothing \vee \mathsf{Removed} \neq \varnothing$ then
          $\mathsf{lpl} \leftarrow *\texttt{send-LPL}()$
          $\mathsf{hb} \leftarrow *\texttt{send-heartbeat}()$
      else if $\mathsf{time} - \mathsf{lastHb} \geq \Delta_{\mathsf{heartbeat}}$ then
          $\mathsf{hb} \leftarrow *\texttt{send-heartbeat}()$
  $\mathsf{M} \leftarrow (\mathsf{me}, \bot, \mathsf{lpl}, \mathsf{hb})$
  return $\mathsf{M}$

Algorithm: ParticipantTick(time)
  req $\mathsf{uid}_{\mathsf{lead}} \neq \bot \wedge \mathsf{uid}_{\mathsf{lead}} \neq \mathsf{me}$
  $\mathsf{alive} \leftarrow (\mathsf{time} - \mathsf{lastHb} \leq \Delta_{\mathsf{live}})$
  return $(\mathsf{alive}, \bot)$ // no updated credential
```

Fig. 3: The client part of Zoom's overall LL-CGKA scheme. The description omits the state ust, which is input and output by most algorithms as specified in Section 3.1, and implicitly updated. ust is described in detail in Appendix D.2.

Protocol Zoom's Client LL-CGKA Helpers

```
Helper: *send-LPL()
  $\mathsf{v} \gets \mathsf{v} + 1$
  if $\mathsf{numLplLinks} \geq \mathsf{max\text{-}links}$ then
    $\mathsf{lpl} \gets (\mathsf{me}, \mathsf{v}, \mathtt{true}, \perp, G, \perp, \mathsf{e}, \mathsf{p})$
    $\mathsf{numLplLinks} \gets 1$
  else
    $\mathsf{lpl} \gets (\mathsf{me}, \mathsf{v}, \mathtt{false}, \mathsf{lplHash}, \mathsf{Added}, \mathsf{Removed}, \mathsf{e}, \mathsf{p})$
    $\mathsf{numLplLinks} \gets \mathsf{numLplLinks} + 1$
  $\mathsf{lplHash} \gets \mathsf{Hash}(\mathsf{lpl})$
  $(\mathsf{Added}, \mathsf{Removed}) \gets \varnothing$
  return $\mathsf{lpl}$

Helper: *receive-LPL($\mathsf{lpl}'$)
  parse $(\mathsf{uid}'_{\mathsf{lead}}, \mathsf{v}', \mathsf{coalesced}', \mathsf{lplHash}',$
              $\mathsf{Added}', \mathsf{Removed}', \mathsf{e}', \mathsf{p}') \gets \mathsf{lpl}'$
  req $\mathsf{uid}'_{\mathsf{lead}} = \mathsf{uid}_{\mathsf{lead}}$
  req $\mathsf{v} = \perp \vee \mathsf{v}' = \mathsf{v} + 1$
  $\mathsf{v} \gets \mathsf{v}'$
  if $\neg\mathsf{coalesced}$ then
    req $\mathsf{lplHash}' = \mathsf{lplHash} \wedge \mathsf{lplHash} \neq \perp$
    req $(\mathsf{Removed}' \setminus G) = \varnothing$
    $G \gets G \setminus \mathsf{Removed}'$
    req $(\mathsf{Added}' \cap G) = \varnothing$
    $G \gets G \cup \mathsf{Added}'$
  else
    $G \gets \mathsf{Added}'$
  $\mathsf{lplHash} \gets \mathsf{Hash}(\mathsf{lpl}')$
  $(\mathsf{e}_{\mathsf{next}}, \mathsf{p}_{\mathsf{next}}) \gets (\mathsf{e}', \mathsf{p}')$
```

```
Helper: *send-heartbeat()
  $\mathsf{t} \gets \mathsf{t} + 1$
  $\mathsf{sig}_{\mathsf{hb}} \gets \mathsf{Sig.Sign}(\mathsf{isk}, \text{`}\mathsf{LeaderParticipantList}\text{'},$
              $(\mathsf{me}, \mathsf{t}, \mathsf{hbHash}, \mathsf{time}, \mathsf{v}, \mathsf{lplHash}, \mathsf{e}, \mathsf{p}))$
  $\mathsf{hb} \gets (\mathsf{t}, \mathsf{time}, \mathsf{sig}_{\mathsf{hb}})$
  $\mathsf{hbHash} \gets \mathsf{Hash}(\mathsf{hb})$, $\mathsf{lastHb} \gets \mathsf{time}$
  return $\mathsf{hb}$

Helper: *receive-heartbeat($\mathsf{hb}$)
  parse $(\mathsf{t}', \mathsf{time}', \mathsf{sig}'_{\mathsf{hb}}) \gets \mathsf{hb}$
  req $\mathsf{t} = \perp \vee \mathsf{t}' = \mathsf{t} + 1$
  $\mathsf{t} \gets \mathsf{t}'$
  $(\cdot, \mathsf{ipk}') \gets \mathrm{cmKEM}.\mathsf{Identity}(\mathsf{uid}_{\mathsf{lead}})$
  req $\mathsf{Sig.Verify}(\mathsf{ipk}', \text{`}\mathsf{LeaderParticipantList}\text{'},$
              $(\mathsf{uid}_{\mathsf{lead}}, \mathsf{t}, \mathsf{hbHash}, \mathsf{time}', \mathsf{v}, \mathsf{lplHash}, \mathsf{e}_{\mathsf{next}}, \mathsf{p}_{\mathsf{next}}), \mathsf{sig}'_{\mathsf{hb}})$
  req $\mathsf{Key}[\mathsf{e}_{\mathsf{next}}, \mathsf{p}_{\mathsf{next}}] \neq \perp$
  $(\mathsf{e}, \mathsf{p}) \gets (\mathsf{e}_{\mathsf{next}}, \mathsf{p}_{\mathsf{next}})$
  *update-drift($\mathsf{time}'$)
  *update-liveness($\mathsf{time}'$)
  $\mathsf{hbHash} \gets \mathsf{Hash}(\mathsf{hb})$
  $\mathsf{t} \gets \mathsf{t}'$

Helper: *update-drift($\mathsf{time}'$)
  $\delta[\mathsf{uid}_{\mathsf{lead}}] \gets \min(\delta[\mathsf{uid}_{\mathsf{lead}}], \mathsf{time} - \mathsf{time}')$

Helper: *update-liveness($\mathsf{time}'$)
  $\mathsf{lastHb} \gets \mathsf{time}' + \delta[\mathsf{uid}_{\mathsf{lead}}]$
```

Fig. 4: Helper algorithms for the client part of Zoom's overall LL-CGKA scheme.

requirement, the leader will from time to time use a special *coalesced* LPL message encoding the entire group.[9] A joining party therefore needs all the links up to and including the latest coalesced message only. The frequency of coalesced messages is determined by the parameter $\mathsf{max\text{-}links}$.

Heartbeats. The LPL messages are unauthenticated. To authenticate them, the leader broadcasts a signature (of a hash) thereof under the leader's long-term identity key. Those signatures moreover form another hash chain, with each signature including the hash of the previous one to ensure the continuity of the meeting. That is, while certain splitting attacks — where a malicious server might tell subgroups to accept different leaders — are unavoidable, those diverging meetings cannot later be merged.

The leader broadcasts one such signature at least at a fixed interval $\Delta_{\mathsf{heartbeat}}$, even when no LPL has been sent for lack of any change to the group membership. Since they are regularly sent, these signatures are called *heartbeat messages* and double as a mechanism to ensure liveness. To this end, the signature additionally includes the latest epoch e, and period p. Hence, if an attacker attempts to withhold either key rotations or updates to the membership, causing a participant to be stuck in an old state, they would need to withhold the heartbeat message as well. As a countermeasure, participants drop out from the meeting if they have not received a heartbeat message for a certain amount of time.

For this mechanism to not abruptly end meetings (despite potential network hiccups), participants do not expect to receive the heartbeats in perfectly regular intervals. Rather, each heartbeat itself contains a timestamp time' (the sending time) whose state it certifies. Receiving this heartbeat then prolongs the liveness of the receiver until time $\mathsf{time}' + \delta + \Delta_{\mathsf{live}}$, when the party will drop out if no further heartbeat has been received. Here, δ denotes an upper bound on the clock drift (between the participant and their respective leader) and Δ_{live} denotes a protocol parameter. As a best effort to prevent this from happening, the server will elect a new leader whenever the current one struggles to upload heartbeats.

[9] In the deployed version of the protocol, the coalesced LPL also includes a list of all participants who were in the meeting at some point in the past but have since left. This additional information is displayed in the client's user interface, but is not modeled in this work.

The protocol estimates the upper bound on the clock drift δ as follows: Upon receiving the first heartbeat with timestamp t' at local time t from a given leader, the protocol simply assumes that $t - t'$ is the drift, i.e., that the heartbeat has been delivered instantaneously. Clearly, $t' + \delta = t$ is an upper bound on the effective sending time. Upon receiving a subsequent heartbeat, the party corrects the drift to $t - t'$ whenever this is smaller, and otherwise keeps it unchanged. Hence, if the network delay, and thus the interval between received heartbeat, increases (e.g., due to a network attacker) then each subsequently received heartbeat extends liveness by a smaller amount, until the party eventually drops out. Conversely, if the network delay decreases, the drift estimates and, hence, the liveness assurances improve.

Evolving the group. The protocol uses the cmKEM scheme to rotate keys whenever the group membership changes. The participants, however, do not immediately transition to the new epoch or period upon receiving such a cmKEM message. Rather, they just store the new key. Only once the group membership is known via receiving a corresponding LPL message and heartbeat, they transition to the new epoch and period and advertise the respective key for content encryption to the higher-level protocol. For membership changes containing only additions, the protocol avoids overly frequent epoch changes by rotating the period instead, however, limiting the number of consecutive periods to a fixed number $\mathsf{p}_{\mathsf{MAX}}$.

Joining a meeting. To join a meeting, a party needs to learn the latest key and group roster in an authenticated manner. The former is communicated via a cmKEM message and the latter via the sequence of LPL messages starting with the latest coalesced one. Authentication of the LPL is achieved by verifying the latest heartbeat message that certifies the final LPL message, as well as epoch and period numbers. (The previous links are implicitly authenticated due to the links forming a hash chain.)

Leader changes. A newly elected leader continues the meeting by starting a new cmKEM session and generating a coalesced LPL message and a heartbeat, which the server then distributes to the other participants. The new leader will continue the relevant counters (i.e., e, p, and t) and hash chains where the old leader left off, such that they uniquely identify a meeting state. The server is responsible to ensure that the party has the latest state the moment it becomes the new leader.

Note that the new leader obtains the group roster from the server rather than deducing it from the previous LPL messages. Otherwise, they might inadvertently revert some of the previous leader's final changes to the group, if for instance the previous leader added or removed a party on the cmKEM level but did not manage to broadcast a corresponding LPL message before dropping out. Users are shown a warning on every leader change, and are advised to manually check whether the group roster displayed in their client is expected.

The server scheme. The messages the leader uploads consist of up to three components, a cmKEM message, a LPL and a heartbeat message. If the message contains a cmKEM message, then the server splits this using the respective cmKEM algorithm and forwards the respective share alongside the LPL and heartbeat (if present) to the users. Otherwise, the server forwards the LPL and heartbeat messages to the last known roster, as derived from the cmKEM messages. See Appendix D.2 for details.

Security. Security is summarized in our main result below, with a more detailed proof given in Appendix D.3.

Theorem 2. *Zoom's LL-CGKA scheme is secure with the liveness slack of P being at most*

$$\min(n \cdot \Delta_{\mathsf{live}}, t_{\mathsf{now}} - t_{\mathsf{joined}}) + \Delta_{\mathsf{live}},$$

where t_{now} denotes the current time, t_{joined} the time P joined the meeting, and n denotes the number of distinct leaders P has encountered so far. Liveness holds if all those leaders have followed the protocol, while all other properties hold as long as the current leader is honest.

Proof (Sketch). Confidentiality and key consistency follow directly from the underlying cmKEM scheme which is used to distribute the group keys. While the LL-CGKA notion mandates slightly stronger properties, those additional assurances relate directly to members only transitioning to subsequent periods if their leader initiated this. This is ensured by parties only transitioning to a new state once a heartbeat certified it, leveraging the unforgeability of the employed signature scheme. Similarly, group consistency — i.e., authenticity of each participant's view on the group roster — is ensured by the combined LPL and heartbeat mechanism, with the LPL distributing the group and the heartbeat authenticating the LPL. Additionally, the hash links of the heartbeat messages yields the no-merging property after a group-splitting attack.

Finally, observe that liveness slack is directly linked to the accuracy of each party's estimate on the clock drift with their respective leader: If the estimate were precise, then each party would have a liveness slack of at most Δ_{live} since they would know exactly when the last heartbeat they received has been sent allowing them to drop out Δ_{live} after. Further, the estimate only degrades by at most Δ_{live} with each leader change — the maximum interval between receiving the old leader's last heartbeat and the new leader's first one. □

The above theorem relies on the underlying cmKEM scheme being secure according to the respective definition, the signature scheme being EUF-CMA secure, and the hash function being collision resistant. According to the whitepaper [11], Zoom's instantiation uses SHA256 and EdDSA (as provided by `libsodium`) for the hash function and signature algorithm, respectively, satisfying those requirements [19,13].

Concrete parameters. At the time of writing, Zoom uses $\Delta_{\mathsf{live}} = 100s$, $\Delta_{\mathsf{heartbeat}} = 10s$, $\Delta_{\mathsf{LPL}} = 2s$, and $\mathsf{max\text{-}links} = 20$, respectively. Moreover, $\mathsf{p}_{\mathsf{MAX}} = 0$, i.e., Zoom always ratchets the full epoch instead of the period[3].

4 Improved Liveness

4.1 Limitations of Zoom's Protocol

For a typical meeting with a single (honest) host that stays online for the duration of the entire meeting — and thus is the leader for the entire meeting — Zoom's current scheme[1] provides strong liveness properties. Indeed, to the best of our knowledge, Zoom is the only E2EE group video protocol that provides any such liveness assurance. As highlighted by Theorem 2, however, there two distinct aspects with respect to which the assurances could be further improved:

1. Zoom's current liveness assurance degrade in the number of meeting leaders encountered. This is sub-optimal for a protocol such as Zoom where the (untrusted) server can initiate leader changes.[10]
2. While all other security properties, such as key confidentiality and authenticity, recover after removing a malicious party from the meeting, liveness does not.[11]

We particularly stress that both aspects are not merely deficiencies of our analysis. Concrete (though contrived) attacks exist, even if they could be mitigated by countermeasures relying on the end user, such as user-interface warnings.

Lemma 1. *Even with all honest participants, the liveness properties of Zoom's LL-CGKA scheme degrade in the number of leader changes, assuming an all powerful malicious server carefully orchestrating the meeting.*

Proof. Consider a meeting with parties $P_1, P_2, \ldots, P_n$, as well as a designated party P^*. All parties, unbeknownst to each other, have precisely synchronized clocks. The party P_1 is the one to start the meeting and act as its initial leader. When adding the parties $P_2, \ldots, P_n$ to the meeting, the network adversary delivers the respective messages immediately. That is, the moment those parties create their ephemeral user identities $\mathsf{uid}_2, \ldots, \mathsf{uid}_n$, party P_1 is immediately instructed to add them to the meeting using Add producing M, and the respective shares obtained by Split are handed to the parties to execute Follow without any delay. (To

[10] This is currently remedied by the client showing a warning upon each leader change, since the leader-authentication codes anyway require to repeat the authentication process in this event. With the introduction of the advanced PKI replacing the leader-authentication codes, Zoom might however consider dropping those warnings.

[11] Note that Zoom does not aim to provide strong guarantees *while* a malicious insider is part of the meeting. Yet, removing a malicious party should ideally reestablish security without the need to restart the entire meeting.

this end, assume that the heartbeat interval perfectly aligns with the moment all those parties join.) This results in each of those parties estimating their drift to be 0, i.e., $\delta_{\mathsf{uid}_j}[\mathsf{uid}_1] = 0$ for $j \in \{2, \ldots, n\}$.

In contrast, when party P^* joins the meeting, their respective ephemeral identity uid^* is still handed immediately to P_1, but the respective response delayed by Δ_{live}. Assuming P^* created their identity at time t and got the LL-CGKA message at time $t + \Delta_{\mathsf{live}}$, but with timestamp t, then P^* assumes that their clock runs ahead by Δ_{live}, i.e., $\delta_{\mathsf{uid}^*}[\mathsf{uid}_1] = \Delta_{\mathsf{live}}$. All subsequent heartbeats from P_1 are then delivered to P^* with a delay of Δ_{live}. As a result, if P_1 sends a further heartbeat at time t', P^* will set $\mathsf{lastHb} \leftarrow t' + \Delta_{\mathsf{live}}$ and therefore extend the time until they drop out until $t' + 2\Delta_{\mathsf{live}}$ (instead of the optimal $t' + \Delta_{\mathsf{live}}$).

Next, consider P_1 sending their last heartbeat (which is delivered to all parties as previously described) at a time t_2 after P^* joined the meeting, and immediately afterwards the party P_2 becoming the leader (still at time t_2). Again, the messages derived from the output of Lead are distributed to $P_3, \ldots, P_n$ without delay, again resulting in $\delta_{\mathsf{uid}_j}[\mathsf{uid}_2] = 0$. For P^*, on the other hand, P_1's last heartbeat is delivered at time $t_2 + \Delta_{\mathsf{live}}$, extending liveness until $t_2 + 2\Delta_{\mathsf{live}}$. The adversary now takes advantage of this fact by delaying the first message from P_2 as well as all subsequent ones by $2\Delta_{\mathsf{live}}$. This process can then be repeated with sequentially switching leaders to $P_3, P_4, \ldots, P_n$, leading to a liveness slack of $(n+1)\Delta_{\mathsf{live}}$. □

Lemma 2. *If parties join a meeting that currently has a malicious leader colluding with a party with extensive control over Zoom's server infrastructure, then the liveness assurance can be arbitrarily broken even after all malicious parties have been removed from the meeting (and an honest leader has taken over).*

Proof. Consider a malicious insider attacker P_M starting a meeting. Moreover, assume that there are two honest parties P_A and P_B, where first P_A wants to join and at a later point P_B wants to join. Assume that all have perfectly synchronized clocks. In the meeting, attacker first adds P_A to the group, without any delay, i.e., such that $\delta_{\mathsf{uid}_A}[\mathsf{uid}_M] = 0$. At time t, right when P_B is about to join (e.g., once P_B advertised their ephemeral uid_B) the malicious insider does the following:

1. P_M creates k heartbeat messages $\mathsf{t}+1, \mathsf{t}+2, \ldots, \mathsf{t}+k$ (when t denotes the number of heartbeats created so far) for which they pretend to be normally spaced out by $\Delta_{\mathsf{heartbeat}}$ with respect to the included timestamps.
2. P_M then adds P_B to the meeting in state $\mathsf{t}+k$, i.e., the first heartbeat signing over the LPL containing uid_B is with counter $\mathsf{t}+k+1$.

The attacker controlling Zoom's server infrastructure now delivers those messages as follows:

1. Immediately deliver the welcoming message, including the $(\mathsf{t}+k+1)$-th heartbeat, at time t to P_B. As a result P_B will set $\delta_{\mathsf{uid}_B}[\mathsf{uid}_M] = -k \cdot \Delta_{\mathsf{heartbeat}}$, since to P_B it looks like the clock of P_M simply runs ahead.
2. Immediately make P_B the new leader at time t.
3. Deliver all the k intermediate heartbeats to P_A at the regular interval $\Delta_{\mathsf{heartbeat}}$. At time $t + k \cdot \Delta_{\mathsf{heartbeat}}$ first deliver the messages corresponding to P_B joining and then, immediately afterwards, the first message from the new leader P_B.

It is easy to see that P_A does not drop out as they get heartbeats exactly as if the meeting would progress normally. More concretely, to P_A it looks like a perfectly normal meeting in which P_B joins at time $t + k \cdot \Delta_{\mathsf{heartbeat}}$. At the end, P_A will still accept the message from P_B, thinking that the clock of P_B must run ahead. □

As a result, we now propose two alternative strengthened liveness protocols.

4.2 Additional Interaction

As a first proposal we suggest adding additional interaction in the form of sporadic messages from each participant.

Protocol Client LL-CGKA (Improvement 1)

```
User management

Algorithm: CreateUser(time, id, meetingId)
    (st, me, sig′) ← cmKEM.CreateUser(id, meetingId)
    nonce ←$ $\mathcal{N}$
    nonce′ ← ⊥
    lastNonce ← time
    sig ← (sig′, nonce)
    (isk, ·) ← PKI.get-sk(id)
    lastHb ← time
    $\mathsf{uid}_{\mathsf{lead}}$ ← ⊥
    $G$ ← ∅
    e, p, $\mathsf{e}_{\mathsf{next}}$, $\mathsf{p}_{\mathsf{next}}$, v, t ← 0
    $\mathsf{e}_{\mathsf{pub}}$ ← ⊥
    k[·, ·] ← ⊥
    δ[·] ← ∞
    lplHash, hbHash ← ⊥
    return (me, sig)

Participants' algorithms

Algorithm: Follow(time, m′, $\mathsf{uid}'_{\mathsf{lead}}$, $\mathsf{sig}'_{\mathsf{lead}}$)
    req $\mathsf{uid}_{\mathsf{lead}} \neq \bot \wedge \mathsf{uid}'_{\mathsf{lead}} \neq$ me
    parse ($\mathsf{m}'_K$, lpl′, hb′) ← m′
    if $\mathsf{uid}_{\mathsf{lead}} \neq \bot$ then
        req lpl′ ≠ ⊥ ∧ hb′ ≠ ⊥
    st′ ← cmKEM.JoinSession(st, $\mathsf{uid}'_{\mathsf{lead}}$, $\mathsf{sig}'_{\mathsf{lead}}$, $\mathsf{m}'_K$, nonce)
    if st′ = ⊥ then // joining failed
        // try previous nonce
        try st ← cmKEM.JoinSession(st, $\mathsf{uid}'_{\mathsf{lead}}$, $\mathsf{sig}'_{\mathsf{lead}}$, $\mathsf{m}'_K$, nonce′)
    else
        st ← st′
    Key[st.e, st.p] ← st.k
    $\mathsf{uid}_{\mathsf{lead}}$ ← $\mathsf{uid}'_{\mathsf{lead}}$
    $\mathsf{e}_{\mathsf{pub}}$ ← ⊥
    if lpl′ ≠ ⊥ then
        try *receive-LPL(lpl′)
        req me ∈ $G$ ∧ hb′ ≠ ⊥ ∧ ($\mathsf{e}_{\mathsf{next}}$, $\mathsf{p}_{\mathsf{next}}$) = (st.e, st.p)
    if hb′ ≠ ⊥ then
        try *receive-heartbeat(hb′)

Algorithm: ParticipantTick(time)
    req $\mathsf{uid}_{\mathsf{lead}} \neq \bot \wedge \mathsf{uid}_{\mathsf{lead}} \neq$ me
    alive ← (time − lastHb ≤ $\Delta_{\mathsf{live}}$)
    if time − lastNonce ≥ $\Delta_{\mathsf{nonce}}$ then
        nonce′ ← nonce
        nonce ←$ $\mathcal{N}$
        lastNonce ← time
        sig ← (sig′, nonce)
        return (alive, sig)
    else
        return (alive, ⊥)

Leader's algorithms

Algorithm: Lead(time, $\{(\mathsf{uid}_i, \mathsf{sig}_i)\}_{i\in[n]}$)
    if $\mathsf{e}_{\mathsf{pub}} \neq \bot$ then e′ ← $\mathsf{e}_{\mathsf{pub}}$ + 1
    else e′ ← ⊥
    for $i \in [n]$ do
        parse ($\mathsf{sig}'_i$, $\mathsf{nonce}_i$) ← $\mathsf{sig}_i$
    try (st, $\mathsf{M}_K$) ← cmKEM.StartSession(st, $\{(\mathsf{uid}_i, \boxed{\mathsf{nonce}_i, \mathsf{sig}'_i})\}_{i\in[n]}$, e′)
    $\mathsf{uid}_{\mathsf{lead}}$ ← me
    (e, p) ← (st.e, st.p)
    $\mathsf{e}_{\mathsf{pub}}$ ← ⊥
    $G$ ← $\{\mathsf{uid}_1, \ldots, \mathsf{uid}_n\} \cup \{\mathsf{me}\}$
    Added, Removed ← ∅
    numLplLinks ← max-links
    lpl ← *send-LPL()
    hb ← *send-heartbeat()
    M ← (me, $\mathsf{M}_K$, lpl, hb)
    return M

Algorithm: Add(time, $\{(\mathsf{uid}_i, \mathsf{sig}_i)\}_{i\in[n]}$)
    req $\mathsf{uid}_{\mathsf{lead}}$ = me
    for $i \in [n]$ do
        req $\mathsf{uid}_i \notin G$
        parse ($\mathsf{sig}'_i$, $\mathsf{nonce}_i$) ← $\mathsf{sig}_i$
    $G$ ← $G \cup \{\mathsf{uid}_1, \ldots, \mathsf{uid}_n\}$
    Added ← Added ∪ $\{\mathsf{uid}_1, \ldots, \mathsf{uid}_n\}$
    if p ≥ $\mathsf{p}_{\mathsf{MAX}}$ then
        try (st, $\mathsf{M}_K$) ← cmKEM.Add(st, $\{(\mathsf{uid}_i, \boxed{\mathsf{nonce}_i, \mathsf{sig}'_i})\}_{i\in[n]}$, true)
        (e, p) ← (st.e, st.p)
    else
        try (st, $\mathsf{M}_K$) ← cmKEM.Add(st, $\{(\mathsf{uid}_i, \boxed{\mathsf{nonce}_i, \mathsf{sig}'_i})\}_{i\in[n]}$, false)
    return (me, $\mathsf{M}_K$, ⊥, ⊥)
```

Fig. 5: The proposed changes with respect to Zoom's scheme from Fig. 3.

The protocol. Concretely, our enhancement to Zoom's protocol is as follows: First, each party generates an unpredictable nonce nonce (from some nonce space $\mathcal{N}$, e.g., 192-bit strings) with frequency Δ_{nonce}. These nonces are seen as part of a party's credential and hence ParticipantTick outputs a new credential whenever the nonce is updated. (In practice one would of course only upload the nonce, not the entire credential, each time.)

Whenever a new leader is elected, they get each participants' latest nonce from the server. We encode this as part of the credentials sig_i for the Lead algorithm, which can now be thought as a sort of time-based credentials. The new leader then uses those nonces as associated data for the cmKEM primitive (which for Zoom's instantiation means it is used as associated data for the authenticated PKE). The same mechanism is used for adding new members to the group.

Each party as part of the Follow algorithm provides their current nonce as associated data to $\mathsf{JoinSession}$, thus verifying that the new leader used the correct one. To prevent race conditions, parties moreover store their second latest nonce nonce' and try with that one if $\mathsf{JoinSession}$ initially fails. See Fig. 5 for a formal description of the changes with respect to Zoom's current scheme from Fig. 3.

Security. Our proposal improves the liveness properties twofold. First, the liveness slack no longer degrades in the number of leader changes. Second, liveness now holds even if a *past leader* has been corrupted as long as the current leader is honest.

We now state the resulting theorem. A more formal version thereof and a proof can be found in Appendix E.1.

Theorem 3. *The modified LL-CGKA scheme from Fig. 5 is secure with the liveness slack of P being at most*

$$\min\bigl(t_{\mathsf{now}} - t_{\mathsf{joined}}, 2 \cdot \mathit{\Delta}_{\mathsf{nonce}} + \mathit{\Delta}_{\mathsf{live}}\bigr) + \mathit{\Delta}_{\mathsf{live}},$$

where t_{now} denotes the current time, t_{joined} the time P joined the meeting. In contrast to Theorem 2, liveness holds if the current leader is honest (as apposed to all leaders encountered so far), analogous to all other properties. Additionally, as long as all n leaders encountered so far have been honest, the liveness slack of P is also at most $(n+1) \cdot \mathit{\Delta}_{\mathsf{live}}$.

Remark 2 (Zoom's adaptation.). Zoom implemented a variant of this proposal in the Zoom client since version 5.13. The most significant difference is that, while our analysis treats the nonce interval $\mathit{\Delta}_{\mathsf{nonce}}$ as a constant, in Zoom's implementation for efficiency reasons $\mathit{\Delta}_{\mathsf{nonce}}$ scales quadratically with the number of current meeting participants. It is easy to see that in our proposed protocol sending fresh nonces more frequently only improves security — hence Theorem 3 can be seen as worst-case analysis with $\mathit{\Delta}_{\mathsf{nonce}}$ set to the maximal encountered during a meeting.

4.3 Leveraging Clock Synchronicity

In this section, we explore an alternative approach towards mitigating the degradation of liveness properties during leader changes. Concretely, we propose to leverage pre-existing clock synchronicity to achieve better liveness without having to introduce additional communication. For E2EE protocols, however, it is undesirable to simply assume synchronized clocks since this, for all practical purposes, implies assuming a trusted reference clock (such as a time server) and introduces additional friction for users on misconfigured devices (for example, with the wrong timezone settings.).

Unfortunately, even detecting whether clocks are synchronized is non-trivial. For instance, consider the interaction between a participant P and a leader L depicted in Fig. 6: in one situation, L's clock is in sync while in the other situation L's clock runs ahead — yet the scenarios look completely indistinguishable to both P and L. As such, we propose the following hybrid strategy:

- For correctness, i.e., functionality of the scheme, assume clocks to be properly synchronized. After all, Zoom is usually run on modern devices such as laptop computers or smartphones that generally do have well synchronized clocks. An honest Zoom server could moreover detect erroneous time settings and instruct the client to re-synchronize their clock (either by displaying a warning, or by doing it automatically with a somewhat trusted external server).
- For security, well-synchronized clocks should yield tight liveness assurances, while worst-case liveness should degrade to the current[1] protocol's properties.

The protocol. We now discuss our proposed mechanism. In a nutshell, our proposed improvement works by each party P maintaining not just an upper bound $\delta_P[L]$ on the clock drift with their current leader L (which in this protocol we rename as $\delta_P^{\mathsf{max}}[L]$ for clarity), but also a lower bound $\delta_P^{\mathsf{min}}[L]$, such that $\delta_P^{\mathsf{min}}[L] \leq \mathsf{offset}_{L\to P} \leq \delta_P^{\mathsf{max}}[L]$. As in the current protocol[1], these bounds are derived from simple causality observations, can gradually improve over the course of the execution, and in turn are used to adjust the timestamp indicated as part of the heartbeat messages and therefore decide when to drop out. See Fig. 7 for a formal description of the proposed modifications with respect to Zoom's current scheme.

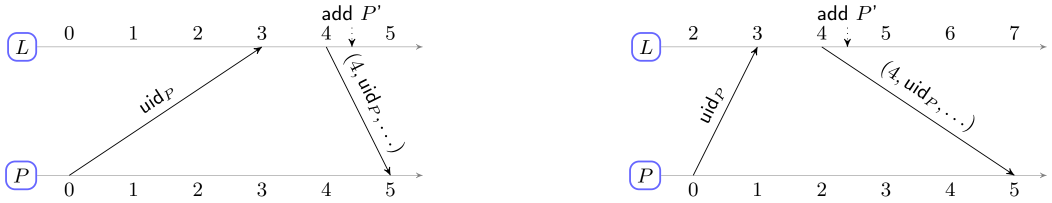

Fig. 6: The leader L's clock running ahead (right) negatively affects liveness as the addition of P' can be withhold longer from P.

Protocol Client LL-CGKA (Improvement 2)

```
Algorithm: CreateUser(time, id, meetingId)
  $(\mathsf{st}, \mathsf{me}, \mathsf{sig}) \leftarrow \mathrm{cmKEM}.\mathsf{CreateUser}(\mathsf{id}, \mathsf{meetingId})$
  $(\mathsf{isk}, \cdot) \leftarrow \mathsf{PKI.get\text{-}sk}(\mathsf{id})$
  $\mathsf{lastHb} \leftarrow \mathsf{time}$
  $\boxed{\mathsf{lastHb}^{\mathsf{min}}, \mathsf{lastHb}^{\mathsf{max}} \leftarrow \bot}$
  $\mathsf{uid}_{\mathsf{lead}} \leftarrow \bot$
  $G \leftarrow \varnothing$
  $\mathsf{e}, \mathsf{p}, \mathsf{e}_{\mathsf{next}}, \mathsf{p}_{\mathsf{next}}, \mathsf{v}, \mathsf{t} \leftarrow 0$
  $\mathsf{e}_{\mathsf{pub}} \leftarrow \bot$
  $\mathsf{k}[\cdot,\cdot] \leftarrow \bot$
  $\boxed{\begin{array}{l} \delta^{\mathsf{min}}[\cdot] \leftarrow -\infty \\ \delta^{\mathsf{max}}[\cdot] \leftarrow +\infty \end{array}}$
  $\mathsf{lplHash}, \mathsf{hbHash} \leftarrow \bot$
  return $(\mathsf{me}, \mathsf{sig})$

Algorithm: CatchUp(ust, time, grpPub)
  req $\mathsf{uid}_{\mathsf{lead}} = \bot$
  parse $(\mathsf{e}_{\mathsf{pub}}, \mathsf{lpls}', \mathsf{hb}') \leftarrow \mathsf{grpPub}$
  // Process LPLs
  while $\mathsf{lpls} \neq \langle\rangle$ do
      $\mathsf{lpl}' \leftarrow \mathsf{lpls}'.\mathsf{deq}()$
      try *receive-LPL($\mathsf{lpl}'$)
  // Store Heartbeat (no verification)
  $\mathsf{hbHash} \leftarrow \mathsf{Hash}(\mathsf{hb}')$
  $\boxed{\mathsf{lastHb}^{\mathsf{max}} \leftarrow \mathsf{time}}$

Helper: *send-heartbeat()
  $\mathsf{t} \leftarrow \mathsf{t} + 1$
  $\boxed{\mathsf{elapsed} \leftarrow \mathsf{time} - \mathsf{lastHb}^{\mathsf{max}}}$
  $\mathsf{sig}_{\mathsf{hb}} \leftarrow \mathsf{Sig.Sign}(\mathsf{isk}, \text{‘}\mathsf{LeaderParticipantList}\text{’},$
      $(\mathsf{me}, \mathsf{t}, \mathsf{hbHash}, \mathsf{time}, \boxed{\mathsf{elapsed}}, \mathsf{v}, \mathsf{lplHash}, \mathsf{e}, \mathsf{p}))$
  $\mathsf{hb} \leftarrow (\mathsf{t}, \mathsf{time}, \boxed{\mathsf{elapsed}}, \mathsf{sig}_{\mathsf{hb}})$
  $\mathsf{hbHash} \leftarrow \mathsf{Hash}(\mathsf{hb})$
  $\mathsf{lastHb}, \boxed{\mathsf{lastHb}^{\mathsf{min}}, \mathsf{lastHb}^{\mathsf{max}}} \leftarrow \mathsf{time}$
  return hb

Helper: *receive-heartbeat(hb)
  parse $(\mathsf{t}', \mathsf{time}', \boxed{\mathsf{elapsed}'}, \mathsf{sig}'_{\mathsf{hb}}) \leftarrow \mathsf{hb}$
  req $\mathsf{t} = \bot \vee \mathsf{t}' = \mathsf{t} + 1$
  $\mathsf{t} \leftarrow \mathsf{t}'$
  $(\cdot, \mathsf{ipk}') \leftarrow \mathrm{cmKEM}.\mathsf{Identity}(\mathsf{uid}_{\mathsf{lead}})$
  req $\mathsf{Sig.Verify}(\mathsf{ipk}', \text{‘}\mathsf{LeaderParticipantList}\text{’},$
                  $(\mathsf{uid}_{\mathsf{lead}}, \mathsf{t}, \mathsf{hbHash}, \mathsf{time}', \boxed{\mathsf{elapsed}'}, \mathsf{v}, \mathsf{lplHash}, \mathsf{e}_{\mathsf{next}}, \mathsf{p}_{\mathsf{next}}), \mathsf{sig}'_{\mathsf{hb}})$
  req $\mathsf{Key}[\mathsf{e}_{\mathsf{next}}, \mathsf{p}_{\mathsf{next}}] \neq \bot$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{e}_{\mathsf{next}}, \mathsf{p}_{\mathsf{next}})$
  *update-drift($\mathsf{time}', \mathsf{elapsed}'$)
  *update-liveness($\mathsf{time}'$)
  $\mathsf{hbHash} \leftarrow \mathsf{Hash}(\mathsf{hb})$
  $\mathsf{t} \leftarrow \mathsf{t}'$

Helper: *update-drift(time', elapsed')
  // Upon receiving hb with timestamp time'
  if $\mathsf{lastHb}^{\mathsf{min}} \neq \bot$ then
      $\mathsf{earliest} \leftarrow \mathsf{lastHb}^{\mathsf{min}} + \mathsf{elapsed}'$
  else
      $\mathsf{earliest} \leftarrow \mathsf{lastHb}$ // time of creating uid
  $\delta^{\mathsf{max}}[\mathsf{uid}_{\mathsf{lead}}] \leftarrow \min(\delta^{\mathsf{max}}[\mathsf{uid}_{\mathsf{lead}}], \mathsf{time} - \mathsf{time}')$
  $\delta^{\mathsf{min}}[\mathsf{uid}_{\mathsf{lead}}] \leftarrow \max(\delta^{\mathsf{min}}[\mathsf{uid}_{\mathsf{lead}}], \mathsf{earliest} - \mathsf{time}')$

Helper: *update-liveness(time')
  // Updates time of last received heartbeat
  if $\delta^{\mathsf{max}}[\mathsf{uid}_{\mathsf{lead}}] < 0$ then
      // Leader's clock definitively behind
      $\mathsf{lastHb} \leftarrow \mathsf{time}' + \delta^{\mathsf{max}}[\mathsf{uid}_{\mathsf{lead}}]$
  else if $\delta^{\mathsf{min}}[\mathsf{uid}_{\mathsf{lead}}] > 0$ then
      // Leader's clock definitively ahead
      $\mathsf{lastHb} \leftarrow \mathsf{time}' + \delta^{\mathsf{min}}[\mathsf{uid}_{\mathsf{lead}}]$
  else
      // Unsure
      $\mathsf{lastHb} \leftarrow \mathsf{time}'$
  $\mathsf{lastHb}^{\mathsf{max}} \leftarrow \mathsf{time}' + \delta^{\mathsf{max}}[\mathsf{uid}_{\mathsf{lead}}]$
  $\mathsf{lastHb}^{\mathsf{min}} \leftarrow \mathsf{time}' + \delta^{\mathsf{min}}[\mathsf{uid}_{\mathsf{lead}}]$
```

Fig. 7: The proposed changes with respect to Zoom's scheme from Fig. 3.

Deriving bounds. To this end, consider the case that P receives a heartbeat with timestamp t_L (according to L's clock) at time t_{now} (according to P's clock). As in Zoom's current protocol, P clearly knows that the heartbeat has not been sent after t_{now}, i.e. $t_L + \mathsf{offset}_{L \to P} \leq t_{\mathsf{now}}$. Furthermore, assume that (for whatever reason) P knows that this heartbeat has been sent definitively not before t_{earliest}. P can use this to deduce the following bounds:

$$t_{\mathsf{earliest}} - t_L \leq \delta_P^{\mathsf{min}}[L] \qquad \text{and} \qquad \delta_P^{\mathsf{max}}[L] \leq t_{\mathsf{now}} - t_L.$$

P will only update a bound if it improves the current one. (At the beginning, the protocol initializes them to $\delta_P^{\mathsf{min}}[L] = -\infty$ and $\delta_P^{\mathsf{max}}[L] = +\infty$.)

In our protocol, P will have a meaningful such lower bound t_{earliest} in the following two situations:

- **Upon joining the meeting:** When P joins the meeting, the first heartbeat they get will sign over an LPL containing their freshly generated ephemeral key. Hence, that heartbeat must have been sent after the time t_{joined} when P generated the key.
- **Upon receiving the first heartbeat from a new leader L':** The protocol works by having P deduce a lower bound on when the last heartbeat from the old leader was sent, and the new leader L' indicating as part of the heartbeat a lower bound $\mathsf{elapsed}$ on the *elapsed duration* between the last heartbeat of the old leader L and their first one. Hence, upon receiving the first heartbeat from L', P can use $\mathsf{time}_L + \delta_P^{\mathsf{min}}[L] + \mathsf{elapsed}$ as a lower bound on the sending time.
  Observe that the new leader L' can compute a lower bound $\mathsf{elapsed}$ based on the last heartbeat from L as follows: If L' has already been part of the meeting, it can leverage their own bound $\delta_{L'}^{\mathsf{max}}[L]$ to deduce the upper bound $\mathsf{time}_L + \delta_{L'}^{\mathsf{max}}[L]$ on the prior heartbeat's sending time. Otherwise, L' can use the time they got the last heartbeat from the server as part of $\mathsf{CatchUp}$ yielding at least some (very conservative) bound.

For subsequent heartbeats of the same leader, P only updates the upper bound (if tighter than the previous one).

Correcting the drift. We then modify the "conversion" of timestamp that P performs accordingly. That is, whenever P receives a heartbeat with timestamp time'_L, in Zoom's protocol P knows that this has been sent no later than $\mathsf{time}'_P := \mathsf{time}'_L + \delta_P[L]$ and conservatively delays dropping out until $\mathsf{time}'_P + \Delta_{\mathsf{live}}$. Unfortunately, after a number of leader changes the tightness of the bound $\delta_P[L]$ degrades (i.e. the difference between $\delta_P[L]$ and the actual $\mathsf{offset}_{L\to P}$ can become quite large). In other words, Zoom's protocol favors correctness over liveness. Instead, this improved protocol adjusts the received timestamp if and only if the leader's clock is surely behind or ahead, respectively:

$$\mathsf{time}'_P := \begin{cases} \mathsf{time}'_L + \delta_P^{\mathsf{min}}[L] & \text{if } \delta_P^{\mathsf{min}}[L] > 0, \\ \mathsf{time}'_L + \delta_P^{\mathsf{max}}[L] & \text{if } \delta_P^{\mathsf{max}}[L] < 0, \\ \mathsf{time}'_L & \text{otherwise.} \end{cases}$$

This results in each participant potentially dropping out earlier than they would in Zoom's protocol (as the drift adjustment here is bounded by the one in Zoom's protocol), allowing us to prove tighter liveness guarantees. At the same time, if clocks are synchronized the protocol will not make any drift adjustments and thus participants won't prematurely drop out, ensuring a smooth meeting experience (i.e. provable correctness conditions, formalized in Appendix E.4).

Security. We now quantify the strengthened liveness properties of this scheme. A more formal version thereof and a proof is given in Appendix E.3.

Theorem 4. *The modified LL-CGKA scheme from Fig. 7 is secure with the following improved liveness slack*

$$\min\bigl(|\mathsf{offset}_{L\to P}|,\ n\cdot\Delta_{\mathsf{live}},\ t_{\mathsf{now}} - t_{\mathsf{joined}}\bigr) + \Delta_{\mathsf{live}},$$

where $\mathsf{offset}_{L\to P}$ denotes the clock drift between P and their respective leader L, t_{now} denotes the current time, t_{joined} the time P joined the meeting, and n denotes the number of distinct leaders P encountered so far. Liveness holds if all those leaders have followed the protocol, while all other properties hold as long as the current leader is honest.

5 Meeting Stream Security

The notion of LL-CGKA formalizes the key agreement portion of Zoom's E2EE meeting protocol. While our formal analysis stops at the level of the key agreement, we now comment on how these guarantees extend to the full protocol.

The symmetric meeting key that participants agree upon is leveraged in a straightforward way to provide security guarantees for the whole meeting, by composing it with AEAD. Concretely, given the meeting key, Zoom clients derive a specific per-stream subkey by using HKDF and mixing in a specific stream identifier which depends on the stream type as well as the participant identifier. This subkey is used by each participant to encrypt their streams using AES-GCM. Incrementing nonces provide protection against replay and out of order delivery.

Confidentiality and authenticity. Informally, confidentiality of the meeting key (as formalized in the LL-CGKA abstraction) implies confidentiality of the streams, as distinguishing encrypted meeting streams from encryptions of random noise would require breaking the AEAD scheme. Similarly, AEAD provides integrity protection against external attackers who do not have access to the meeting key, guaranteeing that any received ciphertexts was produced by someone with knowledge of the meeting (sub)key. As noted in the whitepaper [11], it is possible for attendees with privileged network access to tamper with each other's streams.

Liveness. The liveness properties proven for the LL-CGKA directly guarantee that group operations in an E2EE meeting cannot be withheld, and extend analogously to the encrypted meeting streams, but with different parameters. Indeed, as of version 5.13 of the Zoom client, meeting participants stop decryption using old meeting keys shortly after a newer one is advertised from the key agreement, i.e., the LL-CGKA scheme (with a tolerance $\Delta_{\mathsf{stream}} = 10$ seconds to account for network latency). In addition, meeting leaders rotate these keys at least once every $t = 5$ minutes even when there is no change in the participant list. Assuming the above, the protocol guarantees that each packet sent by an honest participant and successfully decrypted was sent within $t + \Delta_{\mathsf{stream}} + \Delta$ of its decryption, where Δ is the liveness slack from the key agreement protocol. Alternatively, the protocol could include the heartbeat counter from the key agreement as associated data in the video encryption, yielding liveness $\Delta + \Delta_{\mathsf{stream}} + \Delta_{\mathsf{heartbeat}}$ without the need to frequently re-key.[12]

6 Conclusions

In this work, we provided the first formal security analysis of Zoom's E2EE meetings protocol, which is one of the most popular group video communication tools in the world. Our work lead to a deployed improvement of the Zoom E2EE meetings protocol, which strengthens its security properties. Of independent interest, our work is also the first that defines and studies *liveness* in the context of end-to-end encryption, which we hope should find other applications beyond Zoom meetings.

7 Acknowledgements

We thank Michael Maxim and the rest of the Zoom team for review and implementation of the end-to-end encrypted meeting protocol and liveness improvements detailed in this work.

References

1. Alwen, J., Blanchet, B., Hauck, E., Kiltz, E., Lipp, B., Riepel, D.: Analysing the HPKE standard. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021, Part I. LNCS, vol. 12696, pp. 87–116. Springer, Heidelberg (Oct 2021). `https://doi.org/10.1007/978-3-030-77870-5_4`
2. Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 129–158. Springer, Heidelberg (May 2019). `https://doi.org/10.1007/978-3-030-17653-2_5`
3. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Security analysis and improvements for the IETF MLS standard for group messaging. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part I. LNCS, vol. 12170, pp. 248–277. Springer, Heidelberg (Aug 2020). `https://doi.org/10.1007/978-3-030-56784-2_9`

[12] This is, however, non-trivial to achieve in a backwards compatible way.

4. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Modular design of secure group messaging protocols and the security of MLS. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021. pp. 1463–1483. ACM Press (Nov 2021). https://doi.org/10.1145/3460120.3484820
5. Alwen, J., Coretti, S., Jost, D., Mularczyk, M.: Continuous group key agreement with active security. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 261–290. Springer, Heidelberg (Nov 2020). https://doi.org/10.1007/978-3-030-64378-2_10
6. An, J.H.: Authenticated encryption in the public-key setting: Security notions and analyses. Cryptology ePrint Archive, Report 2001/079 (2001), https://eprint.iacr.org/2001/079
7. Apple: Facetime & privacy. https://www.apple.com/legal/privacy/data/en/face-time/
8. Barnes, R., Beurdouche, B., , Millican, J., Omara, E., Cohn-Gordon, K., Robert, R.: The messaging layer security (mls) protocol (draft-ietf-mls-protocol-latest). Tech. rep., IETF (Oct 2020), https://messaginglayersecurity.rocks/mls-protocol/draft-ietf-mls-protocol.html
9. Bellare, M., Goldwasser, S.: Verifiable partial key escrow. In: Graveman, R., Janson, P.A., Neuman, C., Gong, L. (eds.) ACM CCS 97. pp. 78–91. ACM Press (Apr 1997). https://doi.org/10.1145/266420.266439
10. Bienstock, A., Fairoze, J., Garg, S., Mukherjee, P., Raghuraman, S.: What is the exact security of the signal protocol? Preprint (2021), https://cs.nyu.edu/~afb383/publication/uc_signal/uc_signal.pdf
11. Blum, J., Booth, S., Chen, B., Gal, O., Krohn, M., Len, J., Lyons, K., Marcedone, A., Maxim, M., Mou, M.E., Namavari, A., O'Connor, J., Rien, S., Steele, M., Green, M., Kissner, L., Stamos, A.: Zoom cryptography whitepaper – v4.0. https://github.com/zoom/zoom-e2e-whitepaper/raw/master/archive/zoom_e2e_v4.pdf (2022)
12. Boneh, D., Naor, M.: Timed commitments. In: Bellare, M. (ed.) CRYPTO 2000. LNCS, vol. 1880, pp. 236–254. Springer, Heidelberg (Aug 2000). https://doi.org/10.1007/3-540-44598-6_15
13. Brendel, J., Cremers, C., Jackson, D., Zhao, M.: The provable security of ed25519: Theory and practice. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1659–1676 (2021). https://doi.org/10.1109/SP40001.2021.00042
14. Bresson, E., Chevassut, O., Pointcheval, D.: Dynamic group Diffie-Hellman key exchange under standard assumptions. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 321–336. Springer, Heidelberg (Apr / May 2002). https://doi.org/10.1007/3-540-46035-7_21
15. Canetti, R., Garay, J., Itkis, G., Micciancio, D., Naor, M., Pinkas, B.: Multicast security: a taxonomy and some efficient constructions. In: IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320). vol. 2, pp. 708–716 (1999)
16. Cisco: Zero-trust security for webex – white paper. https://www.cisco.com/c/en/us/solutions/collateral/collaboration/white-paper-c11-744553.html (2021)
17. Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. Journal of Cryptology **33**(4), 1914–1983 (Oct 2020). https://doi.org/10.1007/s00145-020-09360-1
18. Cohn-Gordon, K., Cremers, C.J.F., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017. pp. 451–466. IEEE (2017). https://doi.org/10.1109/EuroSP.2017.27, https://doi.org/10.1109/EuroSP.2017.27
19. Coron, J.S., Dodis, Y., Malinaud, C., Puniya, P.: Merkle-Damgård revisited: How to construct a hash function. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 430–448. Springer, Heidelberg (Aug 2005). https://doi.org/10.1007/11535218_26
20. Denis, F.: The sodium cryptography library. https://download.libsodium.org/doc/ (Jun 2013)
21. Dolev, D., Strong, H.R.: Polynomial algorithms for multiple processor agreement. In: 14th ACM STOC. pp. 401–407. ACM Press (May 1982). https://doi.org/10.1145/800070.802215
22. Dwork, C., Naor, M.: Pricing via processing or combatting junk mail. In: Brickell, E.F. (ed.) CRYPTO'92. LNCS, vol. 740, pp. 139–147. Springer, Heidelberg (Aug 1993). https://doi.org/10.1007/3-540-48071-4_10
23. Dwork, C., Naor, M., Sahai, A.: Concurrent zero-knowledge. In: 30th ACM STOC. pp. 409–418. ACM Press (May 1998). https://doi.org/10.1145/276698.276853
24. Feldman, P., Micali, S.: Optimal algorithms for byzantine agreement. In: 20th ACM STOC. pp. 148–161. ACM Press (May 1988). https://doi.org/10.1145/62212.62225
25. Fischlin, M., Günther, F.: Multi-stage key exchange and the case of Google's QUIC protocol. In: Ahn, G.J., Yung, M., Li, N. (eds.) ACM CCS 2014. pp. 1193–1204. ACM Press (Nov 2014). https://doi.org/10.1145/2660267.2660308
26. Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol with chains of variable difficulty. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part I. LNCS, vol. 10401, pp. 291–323. Springer, Heidelberg (Aug 2017). https://doi.org/10.1007/978-3-319-63688-7_10

27. Gruszczyk, J.: End-to-end encryption for one-to-one microsoft teams calls now generally available. Microsoft Teams Blog – December 14, 2021. https://techcommunity.microsoft.com/t5/microsoft-teams-blog/end-to-end-encryption-for-one-to-one-microsoft-teams-calls-now/ba-p/3037697 (12 2021)
28. Harder, E.J., Wallner, D.M.: Key Management for Multicast: Issues and Architectures. RFC 2627 (Jun 1999). https://doi.org/10.17487/RFC2627, https://www.rfc-editor.org/info/rfc2627
29. Isobe, T., Ito, R.: Security analysis of end-to-end encryption for zoom meetings. In: Baek, J., Ruj, S. (eds.) Information Security and Privacy. pp. 234–253. Springer International Publishing, Cham (2021)
30. Katz, J.: Efficient and non-malleable proofs of plaintext knowledge and applications. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 211–228. Springer, Heidelberg (May 2003). https://doi.org/10.1007/3-540-39200-9_13
31. Kim, Y., Perrig, A., Tsudik, G.: Tree-based group key agreement. Cryptology ePrint Archive, Report 2002/009 (2002), https://eprint.iacr.org/2002/009
32. Krawczyk, H.: Cryptographic extraction and key derivation: The HKDF scheme. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 631–648. Springer, Heidelberg (Aug 2010). https://doi.org/10.1007/978-3-642-14623-7_34
33. Krohn, M.: Zoom rolling out end-to-end encryption offering. Zoom Blog – October 14, 2020. https://blog.zoom.us/zoom-rolling-out-end-to-end-encryption-offering/ (10 2020)
34. Lowe, G.: A hierarchy of authentication specifications. In: Proceedings 10th Computer Security Foundations Workshop. pp. 31–43 (1997). https://doi.org/10.1109/CSFW.1997.596782
35. Marlinspike, M., Perrin, T.: The double ratchet algorithm (11 2016), https://whispersystems.org/docs/specifications/doubleratchet/doubleratchet.pdf
36. Panjwani, S.: Tackling adaptive corruptions in multicast encryption protocols. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 21–40. Springer, Heidelberg (Feb 2007). https://doi.org/10.1007/978-3-540-70936-7_2
37. Pass, R., Seeman, L., shelat, a.: Analysis of the blockchain protocol in asynchronous networks. In: Coron, J.S., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part II. LNCS, vol. 10211, pp. 643–673. Springer, Heidelberg (Apr / May 2017). https://doi.org/10.1007/978-3-319-56614-6_22
38. Perrig, A., Song, D., Canetti, R., Tygar, J.D., Briscoe, B.: Timed Efficient Stream Loss-Tolerant Authentication (TESLA): Multicast Source Authentication Transform Introduction. IETF RFC 4082 (Informational) (2005)
39. Pinto, A., Poettering, B., Schuldt, J.C.: Multi-recipient encryption, revisited. p. 229–238. ASIA CCS '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2590296.2590329, https://doi.org/10.1145/2590296.2590329
40. Poettering, B., Rösler, P., Schwenk, J., Stebila, D.: SoK: Game-based security models for group key exchange. In: Paterson, K.G. (ed.) CT-RSA 2021. LNCS, vol. 12704, pp. 148–176. Springer, Heidelberg (May 2021). https://doi.org/10.1007/978-3-030-75539-3_7
41. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto (1996)
42. Seggelmann, R., Tuexen, M., Williams, M.: Transport layer security (tls) and datagram transport layer security (dtls) heartbeat extension. IETF RFC 6520 (Standards Track) (2012)
43. Smart, N.P.: Efficient key encapsulation to multiple parties. In: Blundo, C., Cimato, S. (eds.) SCN 04. LNCS, vol. 3352, pp. 208–219. Springer, Heidelberg (Sep 2005). https://doi.org/10.1007/978-3-540-30598-9_15
44. WhatsApp: Whatsapp encryption overview (2017), retrieved 05/2020 from https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf
45. Wire Swiss GmbH: Wire security whitepaper. https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf (2021)
46. Yang, Z.: On constructing practical multi-recipient key-encapsulation with short ciphertext and public key. Sec. and Commun. Netw. **8**(18), 4191–4202 (dec 2015). https://doi.org/10.1002/sec.1334, https://doi.org/10.1002/sec.1334
47. Yuan, E.S.: Zoom acquires keybase and announces goal of developing the most broadly used enterprise end-to-end encryption offering. Zoom Blog – May 7, 2020. https://blog.zoom.us/zoom-acquires-keybase-and-announces-goal-of-developing-the-most-broadly-used-enterprise-end-to-end-encryption-offering/ (5 2020)

A Preliminaries

A.1 Notation

For $\mathbb{N} \coloneqq \{1, 2, \ldots\}$ and $x \in \mathbb{N}$, we write $[x] \coloneqq \{1, 2, \ldots, x\}$. We use the notation $\{x_i\}_{i \in [n]}$ to denote the set $\{x_1, x_2, \ldots, x_n\}$ and in slight abuse of notation x_j to denote the corresponding element if an ordering is clear from the context. We write $x \leftarrow a$ to assign the value a to the variable x, and for a set $\mathcal{B}$, $x \leftarrow\!\!\$\ \mathcal{B}$ for sampling an element in $\mathcal{B}$ uniformly at random. For a cyclic group $\mathbb{G}$, we use $\langle g \rangle$ to denote the subgroup generated by g and, thus, $\langle g \rangle = \mathbb{G}$ to denote that g is a generator. We use multiplicative group notation. The security parameter is denoted by κ.

A.2 Pseudocode and Games

We use pseudocode to describe both protocols and security games. Most algorithms we consider are stateful, i.e., they take a state as part of the input and produce an updated state as part of the output. In some cases, for simplicity, we omit this state from the pseudocode description.

Typically, those algorithms are fallible and in the case the given inputs are invalid for the current state, the algorithm can return the special error value $\bot$ instead. We may call such algorithms with the keyword **try** prepended, resulting in the error to propagate to the calling routine, by unwinding all state changes to the calling routine and then returning $\bot$ as well. We use the keyword **parse** to fallibly parse a message as a tuple of values, resulting in $\bot$ if the message was malformed. Furthermore, we use the keyword **req** followed by a boolean condition to specify preconditions, i.e., terminate the calling procedure with $\bot$ if the condition evaluates to `false`. In contrast, in games, the keyword **assert** followed by a boolean condition is used to denote winning conditions for the adversary with the special variable won of the game being set to `true` whenever the condition evaluates to `false`. The keyword **pub** followed by a variable denotes that the adversary has read-only access to that variable during the game.

Data structures. We make use of associative arrays. We denote by $A[i]$ the value at position i and by $A[i] \leftarrow x$ the respective assignment. Moreover, we use the special symbol $\bot$ as a shorthand for uninitialized values — i.e., the condition $A[i] = \bot$ evaluates to `true` iff position i is not set and $A[i] \leftarrow \bot$ clears said position. We use $A[\cdot] \leftarrow \bot$ to initialize an empty array and $B[\cdot] \leftarrow x$ to initialize one with the value x at every position. Additionally, we make use of FIFO queues. We write $\langle\rangle$ to denote the empty queue, $Q.\mathsf{enq}(x)$ for enqueueing x and $y \leftarrow Q.\mathsf{deq}()$ for dequeueing an item and storing it in y. We moreover use $y \leftarrow Q.\mathsf{peek}()$ to retrieve the first item without actually removing it from Q and $Q.\mathsf{reverse}()$ to reverse the order of the queue.

Games. We write our games using the following conventions. Most games have two special algorithms, $\mathsf{Initialize}$ and $\mathsf{Finalize}$, where the adversary $\mathcal{A}$ initiates the interaction calling the former, is then allowed to make an arbitrary number of queries to all other oracles before ending the interaction with a single invocation to $\mathsf{Finalize}$. The output bit of the oracle is then seen as the output of the interaction. Pure distinguishing games are written as two worlds representing the $b = 0$ and $b = 1$ experiments.

A.3 Cryptographic Primitives

Symmetric Encryption. A symmetric encryption scheme is a tuple of algorithms $\mathsf{SE} \coloneqq (\mathsf{SE.Enc}, \mathsf{SE.Dec})$. The encryption algorithm $\mathsf{SE.Enc}\colon \mathsf{SE}.\mathcal{K} \times \mathsf{SE}.\mathcal{M} \to \{0,1\}^*$ takes a key $\mathsf{k} \in \mathsf{SE}.\mathcal{K}$ and a message $\mathsf{m} \in \mathsf{SE}.\mathcal{M}$ to produce a ciphertext c. The deterministic decryption algorithm $\mathsf{SE.Dec}\colon \mathsf{SE}.\mathcal{K} \times \{0,1\}^* \to \mathsf{SE}.\mathcal{M} \cup \{\bot\}$, given the key and the ciphertext, outputs either a message $\mathsf{m} \in \mathsf{SE}.\mathcal{M}$ or $\bot$.

For correctness, we require that $\Pr[\mathsf{SE.Dec}(\mathsf{k}, \mathsf{SE.Enc}(\mathsf{k}, \mathsf{m})) = \mathsf{m}] = 1$, where the randomness is taken over the choice of $k \in \mathsf{SE}.\mathcal{K}$ and the encryption algorithm. For security, we either require the standard IND-CPA or IND-CCA2 notions, depending on the context.

Nonce-Based AEAD. A nonce-based authenticated encryption scheme with associated data is a tuple of deterministic algorithms ($\mathsf{AEAD.Enc}, \mathsf{AEAD.Dec}$). The algorithm $\mathsf{AEAD.Enc}\colon \mathsf{AEAD}.\mathcal{K} \times \mathsf{AEAD}.\mathcal{N} \times \mathsf{AEAD}.\mathcal{M} \times \mathsf{AEAD}.\mathcal{AD} \to \{0,1\}^*$ takes a key $\mathsf{k} \in \mathsf{AEAD}.\mathcal{K}$ and a nonce $\mathsf{nonce} \in \mathsf{AEAD}.\mathcal{N}$, a message $\mathsf{m} \in \mathsf{AEAD}.\mathcal{M}$, and associated data $\mathsf{ad} \in \mathsf{AEAD}.\mathcal{AD}$, to produce a ciphertext c. The decryption algorithm $\mathsf{AEAD.Dec}\colon \mathsf{AEAD}.\mathcal{K} \times \mathsf{AEAD}.\mathcal{N} \times \{0,1\}^* \times \mathsf{AEAD}.\mathcal{AD} \to \mathsf{AEAD}.\mathcal{M} \cup \{\bot\}$, given the key, the nonce, the ciphertext, and the associated data, outputs either a message $\mathsf{m} \in \mathsf{AEAD}.\mathcal{M}$ or $\bot$.

For correctness, it is required that

$$\mathsf{AEAD.Dec}(\mathsf{k}, \mathsf{nonce}, \mathsf{AEAD.Enc}(\mathsf{k}, \mathsf{nonce}, \mathsf{m}, \mathsf{ad}), \mathsf{ad}) = \mathsf{m}$$

for all $k \in \mathsf{AEAD}.\mathcal{K}$, $\mathsf{nonce} \in \mathsf{AEAD}.\mathcal{N}$, $\mathsf{m} \in \mathsf{AEAD}.\mathcal{M}$, and $\mathsf{ad} \in \mathsf{AEAD}.\mathcal{AD}$. Moreover, for simplicity we assume a fixed length message space, i.e., that $|m_i| = |m_j|$ for all $m_i, m_j \in \mathsf{AEAD}.\mathcal{M}$.

Security is then formalized using the distinguishing game between a real-world and ideal-world system in Fig. 8. The game simultaneously captures confidentiality (in the ideal-world a random message gets encrypted) and authenticity (the decryption of mauled/injected messages in the real-world must return $\bot$).

```
Game $\boxed{\mathsf{Real}^{\mathsf{IND\$}}_{\mathsf{AEAD}}}$ and $\mathsf{Ideal}^{\mathsf{IND\$}}_{\mathsf{AEAD}}$

Procedure: Initialize
  $\mathsf{k} \leftarrow\!\$\ \mathsf{AEAD}.\mathcal{K}$
  $C \leftarrow \varnothing$

Oracle: EncryptChall(m, ad)
  $\mathsf{nonce} \leftarrow\!\$\ \mathsf{AEAD}.\mathcal{N}$
  $\mathsf{m} \leftarrow\!\$\ \mathsf{AEAD}.\mathcal{M}$
  $\mathsf{c} \leftarrow \mathsf{AEAD.Enc}(\mathsf{k}, \mathsf{nonce}, \mathsf{m}, \mathsf{ad})$
  $C \leftarrow C \cup \{(\mathsf{c}, \mathsf{nonce}, \mathsf{ad})\}$
  return (c, nonce)

Oracle: EncryptTest(m, ad)
  $\mathsf{nonce} \leftarrow\!\$\ \mathsf{AEAD}.\mathcal{N}$
  $\mathsf{c} \leftarrow \mathsf{AEAD.Enc}(\mathsf{k}, \mathsf{nonce}, \mathsf{m}, \mathsf{ad})$
  $C \leftarrow C \cup \{(\mathsf{c}, \mathsf{nonce}, \mathsf{ad})\}$
  return (c, nonce)

Oracle: Decrypt(c, nonce, ad)
  $\mathsf{m} \leftarrow \bot$
  if $(\mathsf{c}, \mathsf{nonce}, \mathsf{ad}) \notin C$ then
    $\mathsf{m} \leftarrow \mathsf{AEAD.Dec}(\mathsf{k}, \mathsf{nonce}, \mathsf{c}, \mathsf{ad})$
  return m
```

Fig. 8: The nonce-based AEAD security experiments.

Public-Key Encryption. We use a public-key encryption scheme $\mathsf{PKE} := (\mathsf{PKE.kg}, \mathsf{PKE.enc}, \mathsf{PKE.dec})$, where $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{PKE.kg}(1^\kappa)$ denotes the key generation, $\mathsf{c} \leftarrow \mathsf{PKE.enc}(\mathsf{pk}, \mathsf{m})$ the encryption, and $\mathsf{m} \leftarrow \mathsf{PKE.dec}(\mathsf{sk}, \mathsf{c})$ the decryption, respectively. For security, we require the standard IND-CCA2 notion.

Digital Signatures. Many of our constructions use a digital signature scheme $\mathsf{Sig} := (\mathsf{Sig.KeyGen}, \mathsf{Sig.Sign}, \mathsf{Sig.Verify})$. Note that for convenience we require both signing and verification to split the input into the actual message m and an additional context string $\mathsf{context}$ (used for domain separation when the signing key can be used for multiple purposes). Hence, for $(\mathsf{isk}, \mathsf{ipk}) \leftarrow \mathsf{Sig.KeyGen}(1^\kappa)$ one signs a message under a context using $\mathsf{sig} \leftarrow \mathsf{Sig.Sign}(\mathsf{isk}, \mathsf{context}, \mathsf{m})$ and verifies the respective signature using $b \leftarrow \mathsf{Sig.Verify}(\mathsf{ipk}, \mathsf{context}, \mathsf{m}, \mathsf{sig})$. For security, we require EUF-CMA security such that $b = 1$ only if both the message m and the context match.

HKDF. We use a key derivation function $\mathsf{HKDF}\colon \mathbb{G} \times \{0,1\}^* \to \mathsf{AEAD}.\mathcal{K}$, which we model as a random oracle. The function takes as input a high entropy string (in our case, an element g in the Diffie Hellman group from which keys are sampled) and a domain separation string $\mathsf{context}$, and outputs a pseudorandom string of the appropriate length to be used as an AEAD key. To prove our schemes secure, we assume that this function behaves like a random oracle. In practice, Zoom's protocol uses the HMAC based Key Derivation Function defined in RFC5689 [32] (using 0^λ as the salt for the $\mathsf{HKDF.Expand}$ function).

Pseudo Random Generator (PRG). For the cmKEM construction we require a stateful PRG scheme $\mathsf{PRG} := (\mathsf{PRG.Init}, \mathsf{PRG.Eval})$. The algorithm $\mathsf{seed} \leftarrow \mathsf{PRG.Init}(1^\kappa)$ initializes a state seed, while $(\mathsf{seed}', k) \leftarrow \mathsf{PRG.Eval}(\mathsf{seed})$ evaluates the PRG outputting a string k as well as an updated state seed'.

For security, we require that the sequence of outputs $k_1, k_2, \ldots$ is indistinguishable from a sequence of independent and uniformly random values (of the appropriate length).

A.4 Gap Diffie Hellman

The Zoom scheme makes use a cyclic group $\mathbb{G} = \langle g \rangle$ with a fixed generator g (technically a family of groups indexed by the security parameter) for which the Gap Diffie-Hellman (Gap-DH) is assumed to be hard. Gap-DH states that the Computational Diffie-Hellman (CDH) problem remains hard even when given access to a decisional oracle taking $(X, Y, Z) \in \mathbb{G}^3$ and returning 1 iff $\mathrm{DH}(X, Y) = Z$.

We remark that the Gap-DH assumption (rather than, e.g., CDH) appears to be rather intrinsic to this kind of simple Diffie-Hellman based protocol, as exemplified by recent analyses of Signal [10] or the HPKE standard [1].

B Zoom's PKI

In analysis, we assume a simple (long-term) public-key infrastructure to make formal statements that assure security properties assuming user authenticity. The PKI provides each long-term identity id with their own private signing key isk, while allowing all other users to verify that the respective public verification key ipk belongs to id. Such a PKI is, however, not how — at the time of writing — Zoom actually verifies public keys, and we refer to the whitepaper [11] for further details on Zoom's ongoing efforts for improving user authentication. In the following, we briefly summarize those efforts at the time of writing.

Security codes. Currently, each user does have long-term keys but there is no actual PKI. Instead, security relies on the use of the so-called *meeting leader security code*, which is a digest of the long-term public key. The meeting leader is supposed to read out their security code at the beginning of the meeting (or whenever a new leader takes over) and all participants compare the code to what their client displays. Assuming an attacker cannot convincingly forge audio and video data, and that all participants personally know the meeting leader, this ensures that participants only stay in the meeting if they have the right key. Moreover, the leader is assured of the participants' identities by having them explicitly acknowledge the match of the security code. In a nutshell, this mechanism piggybacks on the meeting's E2E encryption to leverage the unidirectional security, established by the security code, to bidirectional one. A bit more concretely, in the meeting the leader securely distributes a shared symmetric key to all participants that is then used to encrypt the video stream. Now the leader security code ensures that a MITM cannot tamper with this key distribution process and, hence, the key material is known exactly to the set of recipients addressed by the leader. (At this point, the recipients may or may not be the intended ones.) If now each of them speaks up, and the leader knows them personally, the leader can verify that the set of participants matches their expectation.

Performed correctly, and under the right set of assumptions, an attacker controlling Zoom's infrastructure can, thus, not break long-term key authenticity.

Identity Provider Attestations. Zoom plans to introduce more sophisticated user identities which will contain additional data other than a changeable display name, such as email addresses and account identifiers. Organizations leveraging external Identity Providers (IDP) to manage authentication and access control for their members, will be able to delegate their IDP to attest those members' identities using an extension of OpenID Connect. At this point, our simplistic PKI will become more accurate, with the assumption of the PKI essentially corresponding to assuming the external IDPs to be honest.

Key Transparency. Further, Zoom envisions deploying a PKI based on a transparency tree. This complements the guarantees from external IDPs, especially for end-users not belonging to an organization providing an IDP. While this key transparency infrastructure will be hosted by Zoom, its append-only and public verifiability properties will make misbehavior detectable, resulting in stronger security.

C Details on cmKEM

C.1 The PKI

Let us briefly formalize the PKI assumed by both the protocol and the respective security game. See Appendix B for a discussion of how this PKI model applies to Zoom. The PKI provides the following interface:

- A user id can fetch their key pair. That is, when calling $\mathsf{PKI.get\text{-}sk}(\mathsf{id})$, the PKI looks up whether a triple $(\mathsf{id}, \mathsf{isk}, \mathsf{ipk})$ is recorded. Otherwise, it samples a new key pair using $(\mathsf{isk}, \mathsf{ipk}) \leftarrow \mathsf{Sig.KeyGen}(1^\kappa)$ and stores the respective triple. Finally, it returns $(\mathsf{isk}, \mathsf{ipk})$.
- Any user id can verify the public key of another user id'. That is, when id calls $\mathsf{PKI.verify\text{-}pk}(\mathsf{id}', \mathsf{ipk}')$, the PKI looks up whether a triple $(\mathsf{id}', \mathsf{isk}', \mathsf{ipk}')$ is recorded for some isk' and returns 1 iff so.

C.2 Correctness

Intuitively, correctness formalizes that all participants get the same sequence of keys. On one hand, this means ensuring that the invocations to the cmKEM algorithms succeed, i.e., do not abort when provided proper inputs, such as when the leader only adds existing parties who are not yet members of the group or when participants process honestly generated messages. On the other hand, it means ensuring that the keys output by all participants, as well as the leader, are indeed equal. (Note that the second property is formally implied by the consistency properties of the security game. For clarity, we nevertheless require it as part of correctness as well.)

Due to the rather complex interaction, correctness is formalized as a game depicted in Fig. 9. The game largely follows the structure of the security game (see Section 2.3 and Appendix C.3), with the main exception that participants immediately process their respective messages — as part of the $*\mathsf{verifyParticipants}$ helper function — rather than the adversary being able to schedule the delivery.

C.3 Security

In this section, we present the formal security definition of a cmKEM scheme. The security game is depicted in Fig. 10. We refer to Section 2.3 for a high-level discussion of the security game and in the following outline some of the more technical aspects.

Basic game state. The game keeps track of each user's current protocol state, leader, epoch, and period, using the arrays St, Leader, Epoch, and Period, respectively. Moreover, for each session, the game also keeps track of the keys and rosters. It is thereby assumed that each state is uniquely identified by the triple ($\mathsf{uid}_{\mathsf{lead}}$, e, p), i.e., the session's leader as well as the epoch and period within a session. As a result, the values are stored in $\mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$ and $\mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$, respectively.

The St and Leader are updated directly on the fly. The epoch and period are updated as part of the $*\mathsf{verifyProgress}$ helper method, which first checks that the updated values, as output by the protocol, match the expected ones. Similarly, Key is updated as part of the $*\mathsf{verifyConsistency}$ method that first checks key consistency. Group is updated via the $*\mathsf{setGroup}$ helper method. Finally, the game keeps track of corruptions and challenges as explained below.

Corruptions and member authentication. The game keeps track of corruptions using (1) the $\mathsf{CorrIds}$ set and (2) the $\mathsf{Corrupted}$ array. The former keeps track of all the corrupted long-term identities, while for the latter $\mathsf{Corrupted}[\mathsf{uid}]$ stores whether uid has been corrupted. It is modeled as an array (rather than set) for the following technicality: each uid starts out to be corrupted at the beginning of the game, then becomes "uncorrupted" once honestly generated, and may later become corrupted again. To this end, whenever the adversary chooses to corrupt id, then this also marks all associated ephemeral users that are *still active* as corrupted. (The game has a special $\mathsf{DeleteUser}$ oracle that allows a party to signify that it wants to terminate a ephemeral identity without having to be explicitly removed by another party.)

Game $\mathsf{Corr}^{\mathrm{cmKEM}}_{\Psi,\mathcal{A}}$

```
Main

Procedure: Initialize
  won ← false
  pub pub ← $\Psi$.InitSplitState()
  pub St[·], Sigs[·], Epoch[·], Period[·], Leader[·],
        Group[·,·,·], Key[·,·,·] ← $\bot$

Procedure: Finalize
  return won

Users and session management

Oracle: CreateUser(id, meetingId)
  (st, uid, sig) ← $\Psi$.CreateUser(id, meetingId)
  (id', ipk) ← $\Psi$.Identity(uid)
  assert id = id' $\wedge$ $\Psi$.Meeting(uid) = meetingId
  assert PKI.verify-pk(id, ipk)
  assert st $\neq \bot \wedge$ St[uid] = $\bot$
  St[uid] ← st
  Sigs[uid] ← sig
  return uid

Oracle: StartSession(uid_lead, $\{(\mathsf{uid}_i, \mathsf{ad}_i, \mathsf{sig}_i)\}_{i\in[n]}$, e')
  req St[uid_lead] $\neq \bot \wedge \neg$*isLeader(uid_lead)
  req e' = $\bot \vee$ Epoch[uid_lead] = $\bot \vee$ e' > Epoch[uid_lead]
  req $\forall i \in [n]$ : $\Psi$.Meeting(uid_i) = $\Psi$.Meeting(uid_lead)
        $\wedge$ uid_i $\neq$ uid_lead
  req $\forall i, j \in [n], i < j$ : uid_i $\neq$ uid_j
  (st', M) ← $\Psi$.StartSession(St[uid_lead], $\{(\mathsf{uid}_i, \mathsf{ad}_i, \mathsf{sig}_i)\}_{i\in[n]}$, e')
  if (st', M) $\neq \bot$ then
    St[uid_lead] ← st'
    Leader[uid_lead] ← uid_lead
    $G'$ ← $\{(\mathsf{uid}_i, \mathsf{ad}_i)\}_{i\in[n]} \cup \{(\mathsf{uid}_{\mathsf{lead}}, \bot)\}$
    if e' $\neq \bot$ then Epoch[uid_lead] ← e' $- 1$
    else Epoch[uid_lead] ← $-1$
    assert *verifyCorrectness(uid_lead, $G'$, M, true)
    return $M$
  else
    assert $\exists i \in [n]$ : St[uid_i] = $\bot \vee$ Sigs[uid_i] $\neq$ sig_i
    return $\bot$

Group and key management

Oracle: Add(uid_lead, $\{\mathsf{uid}_i, \mathsf{ad}_i, \mathsf{sig}_i)\}_{i\in[n]}$, newEpoch)
  req St[uid_lead] $\neq \bot \wedge$ *isLeader(uid_lead)
  (e, p) ← (Epoch[uid_lead], Period[uid_lead])
  req $\forall i \in [n]$ : (uid_i, $*$) $\notin$ Group[uid_lead, e, p] $\wedge$ uid_i $\neq$ uid_lead
        $\wedge$ $\Psi$.Meeting(uid_i) = $\Psi$.Meeting(uid_lead)
  req $\forall i, j \in [n], i < j$ : uid_i $\neq$ uid_j
  (st', M) ← $\Psi$.Add(St[uid_lead], $\{(\mathsf{uid}_i, \mathsf{ad}_i, \mathsf{sig}_i)\}_{i\in[n]}$, newEpoch)
  if (st', M) $\neq \bot$ then
    St[uid_lead] ← st'
    $G'$ ← Group[uid_lead, e, p] $\cup \{(\mathsf{uid}_i, \mathsf{ad}_i)\}_{i\in[n]}$
    assert *verifyCorrectness(uid_lead, $G'$, M, newEpoch)
    return M
  else
    assert $\exists i \in [n]$ : St[uid_i] = $\bot \vee$ Sigs[uid_i] $\neq$ sig_i
    return $\bot$

Oracle: Remove(uid_lead, $\{\mathsf{uid}_i\}_{i\in[n]}$)
  req St[uid_lead] $\neq \bot \wedge$ *isLeader(uid_lead)
  (e, p) ← (Epoch[uid_lead], Period[uid_lead])
  req uid_lead $\notin \{\mathsf{uid}_i\}_{i\in[n]} \wedge \{(\mathsf{uid}_i, *)\}_{i\in[n]} \subseteq$ Group[uid_lead, e, p]
  (St[uid_lead], M) ← $\Psi$.Remove(St[uid_lead], $\{\mathsf{uid}_i\}_{i\in[n]}$)
  assert (St[uid_lead], M) $\neq \bot$
  $G'$ ← Group[uid_lead, e, p] $\setminus \{(\mathsf{uid}_i, *)\}_{i\in[n]}$
  assert *verifyCorrectness(uid_lead, $G'$, M, true)
  return M
```

```
Helper Methods

Helper: *isLeader(uid)
  return Leader[uid] = uid

Helper: *verifyCorrectness(uid_lead, $G'$, M, newEpoch)
  (e, p) ← (Epoch[uid_lead], Period[uid_lead])
  $G$ ← Group[uid_lead, e, p]
  (e', p', k') ← (St[uid_lead].e, St[uid_lead].p, St[uid_lead].k')
  if $\neg$*verifyLeader(uid_lead, e, p, e', p', k', $G'$, newEpoch) then
    return false
  if $\neg$*verifyParticipants(uid_lead, e', p', k', $G$, $G'$, M) then
    return false
  return true

Helper: *verifyLeader(uid_lead, e, p, e', p', k', $G'$, newEpoch)
  (Epoch[uid_lead], Period[uid_lead]) ← (e', p')
  (Group[uid_lead, e', p'], Key[uid_lead, e', p']) ← ($G'$, k')
  if newEpoch then
    return $e' = e + 1 \wedge p' = 0$
  else
    return $e' = e \wedge p' = p + 1$

Helper: *verifyParticipants(uid_lead, e', p', k', $G$, $G'$, M)
  (pub, ms) ← Split(pub, M)
  if pub.e $\neq$ e' then return false
  for all uid : ms[uid] $\neq \bot$ do
    if (uid, $*$) $\notin G'$ then return false
  for all (uid, $*$) $\in G'$ do
    if ms[uid] = $\bot \wedge$ uid $\neq$ uid_lead then return false
  for all uid : uid $\neq$ uid_lead $\wedge$ St[uid] $\neq \bot \wedge$ (uid, $*$) $\in G'$ do
    processed ← false
    if (uid, $*$) $\in G$ then
      St[uid] ← $\Psi$.Process(St[uid], ms[uid])
      processed ← true
    else if e' > Epoch[uid] then
      let ad s.t. (uid, ad) $\in G'$
      St[uid] ← $\Psi$.JoinSession(St[uid], uid_lead, Sigs[uid_lead], ms[uid], ad)
      Leader[uid] ← uid_lead
      processed ← true
    if processed then
      if St[uid] = $\bot \vee$ e' $\neq$ St[uid].e $\vee$ p' $\neq$ St[uid].p $\vee$ k' $\neq$ St[uid].k then
        return false
      (Epoch[uid], Period[uid]) ← (e', p')
  return true
```

Fig. 9: The correctness game for a cmKEM scheme Ψ.

Game $\mathsf{Sec}^{\mathrm{cmKEM}}_{\Psi,\mathcal{A}}$

```
Main

Procedure: Initialize
  $b \leftarrow\$ \{0,1\}$
  $\mathsf{pub} \leftarrow \Psi.\mathsf{InitSplitState}()$
  $\mathsf{CorrIds}, \mathsf{Challs} \leftarrow \varnothing$
  $\mathsf{St}[\cdot], \mathsf{Epoch}[\cdot], \mathsf{Period}[\cdot], \mathsf{Leader}[\cdot],$
          $\mathsf{Group}[\cdot,\cdot,\cdot], \mathsf{Key}[\cdot,\cdot,\cdot], \mathsf{ChallKeys}[\cdot,\cdot,\cdot] \leftarrow \bot$
  $\mathsf{Corrupted}[\cdot] \leftarrow \mathtt{true}$
  return $\mathsf{pub}$

Procedure: Finalize$(b')$
  if $\neg{*}\mathsf{safe}()$ then return $\mathtt{false}$
  else if $\mathsf{won}$ then return $\mathtt{true}$
  else return $b' = b$

Session management

Oracle: $\mathsf{CreateUser}(\mathsf{id}, \mathsf{meetingId})$
  $(\mathsf{st}, \mathsf{uid}, \mathsf{sig}) \leftarrow \Psi.\mathsf{CreateUser}(\mathsf{id}, \mathsf{meetingId})$
  $(\mathsf{id}', \mathsf{ipk}) \leftarrow \Psi.\mathsf{Identity}(\mathsf{uid})$
  assert $\mathsf{id} = \mathsf{id}' \wedge \Psi.\mathsf{Meeting}(\mathsf{uid}) = \mathsf{meetingId}$
  assert $\mathsf{PKI.verify\text{-}pk}(\mathsf{id}, \mathsf{ipk})$
  assert $\mathsf{St}[\mathsf{uid}] = \bot$
  $\mathsf{St}[\mathsf{uid}] \leftarrow \mathsf{st}$
  $\mathsf{Corrupted}[\mathsf{uid}] \leftarrow \mathtt{false}$
  return $(\mathsf{uid}, \mathsf{sig})$

Oracle: $\mathsf{StartSession}(\mathsf{uid}_{\mathsf{lead}}, \{(\mathsf{uid}_i, \mathsf{ad}_i, \mathsf{sig}_i)\}_{i\in[n]}, \mathsf{e}')$
  req $\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}] \neq \bot$
  try $(\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}], \mathsf{M}) \leftarrow \Psi.\mathsf{StartSession}(\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}], \{(\mathsf{uid}_i, \mathsf{ad}_i, \mathsf{sig}_i)\}_{i\in[n]}, \mathsf{e}')$
  if $\mathsf{e}' \neq \bot$ then
    assert $\mathsf{e}' = \mathsf{St}[\mathsf{uid}_{\mathsf{lead}}].\mathsf{e}$
  $\mathsf{Leader}[\mathsf{uid}_{\mathsf{lead}}] \leftarrow \mathsf{uid}_{\mathsf{lead}}$
  for $i \in [n]$ do
    assert ${*}\mathsf{verifyCredentials}(\mathsf{uid}_n)$
  assert ${*}\mathsf{verifyProgress}(\mathsf{uid}_{\mathsf{lead}}, \text{‘}\mathsf{startedSession}\text{’})$
  ${*}\mathsf{setGroup}(\mathsf{uid}_{\mathsf{lead}}, \{(\mathsf{uid}_i, \mathsf{ad}_i)\}_{i\in[n]} \cup \{(\mathsf{uid}_{\mathsf{lead}}, \bot)\})$
  assert ${*}\mathsf{verifyConsistency}(\mathsf{uid}_{\mathsf{lead}}, *)$
  return $(M, {*}\mathsf{pubState}(\mathsf{uid}_{\mathsf{lead}}))$

Oracle: $\mathsf{JoinSession}(\mathsf{uid}, \mathsf{uid}_{\mathsf{lead}}, \mathsf{sig}_{\mathsf{lead}}, \mathsf{m}, \mathsf{ad})$
  req $\mathsf{St}[\mathsf{uid}] \neq \bot \wedge \mathsf{uid} \neq \mathsf{uid}_{\mathsf{lead}}$
  try $\mathsf{St}[\mathsf{uid}] \leftarrow \Psi.\mathsf{JoinSession}(\mathsf{St}[\mathsf{uid}], \mathsf{uid}_{\mathsf{lead}}, \mathsf{sig}_{\mathsf{lead}}, \mathsf{m}, \mathsf{ad})$
  $\mathsf{Leader}[\mathsf{uid}] \leftarrow \mathsf{uid}_{\mathsf{lead}}$
  assert ${*}\mathsf{verifyCredentials}(\mathsf{uid}_{\mathsf{lead}})$
  assert ${*}\mathsf{verifyProgress}(\mathsf{uid}, \text{‘}\mathsf{joinedSession}\text{’})$
  assert ${*}\mathsf{verifyConsistency}(\mathsf{uid}, \mathsf{ad})$
  return ${*}\mathsf{pubState}(\mathsf{uid})$

Oracle: $\mathsf{DeleteUser}(\mathsf{uid})$
  $\mathsf{St}[\mathsf{uid}] \leftarrow \bot$

Message processing (participants)

Oracle: $\mathsf{Process}(\mathsf{uid}, \mathsf{m})$
  req $\mathsf{St}[\mathsf{uid}] \neq \bot \wedge \neg{*}\mathsf{isLeader}(\mathsf{uid})$
  try $\mathsf{St}[\mathsf{uid}] \leftarrow \Psi.\mathsf{Process}(\mathsf{St}[\mathsf{uid}], \mathsf{m})$
  assert ${*}\mathsf{verifyProgress}(\mathsf{uid}, \text{‘}\mathsf{eitherChanged}\text{’})$
  assert ${*}\mathsf{verifyConsistency}(\mathsf{uid}, *)$
  return ${*}\mathsf{pubState}(\mathsf{uid})$
```

```
Group and key management (leader)

Oracle: $\mathsf{Add}(\mathsf{uid}_{\mathsf{lead}}, \{(\mathsf{uid}_i, \mathsf{ad}_i, \mathsf{sig}_i)\}_{i\in[n]}, \mathsf{newEpoch})$
  req $\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}] \neq \bot \wedge {*}\mathsf{isLeader}(\mathsf{uid}_{\mathsf{lead}})$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{Epoch}[\mathsf{uid}_{\mathsf{lead}}], \mathsf{Period}[\mathsf{uid}_{\mathsf{lead}}])$
  try $(\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}], \mathsf{M}) \leftarrow \Psi.\mathsf{Add}(\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}],$
            $\{(\mathsf{uid}_i, \mathsf{ad}_i, \mathsf{sig}_i)\}_{i\in[n]}, \mathsf{newEpoch})$
  if $\mathsf{newEpoch}$ then
    assert ${*}\mathsf{verifyProgress}(\mathsf{uid}_{\mathsf{lead}}, \text{‘}\mathsf{epochChanged}\text{’})$
  else
    assert ${*}\mathsf{verifyProgress}(\mathsf{uid}_{\mathsf{lead}}, \text{‘}\mathsf{periodChanged}\text{’})$
  for all $i \in [n]$ do
    assert $(\mathsf{uid}_i, *) \notin \mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$
              $\wedge\ \Psi.\mathsf{Meeting}(\mathsf{uid}_i) = \Psi.\mathsf{Meeting}(\mathsf{uid}_{\mathsf{lead}})$
              $\wedge\ {*}\mathsf{verifyCredentials}(\mathsf{uid}_i)$
  ${*}\mathsf{setGroup}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \cup \{(\mathsf{uid}_i, \mathsf{ad}_i)\}_{i\in[n]})$
  assert ${*}\mathsf{verifyConsistency}(\mathsf{uid}_{\mathsf{lead}}, *)$
  return $(\mathsf{M}, {*}\mathsf{pubState}(\mathsf{uid}_{\mathsf{lead}}))$

Oracle: $\mathsf{Remove}(\mathsf{uid}_{\mathsf{lead}}, \{\mathsf{uid}_i\}_{i\in[n]})$
  req $\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}] \neq \bot \wedge {*}\mathsf{isLeader}(\mathsf{uid}_{\mathsf{lead}})$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{Epoch}[\mathsf{uid}_{\mathsf{lead}}], \mathsf{Period}[\mathsf{uid}_{\mathsf{lead}}])$
  try $(\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}], \mathsf{M}) \leftarrow \Psi.\mathsf{Remove}(\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}], \{\mathsf{uid}_i\}_{i\in[n]})$
  assert ${*}\mathsf{verifyProgress}(\mathsf{uid}_{\mathsf{lead}}, \text{‘}\mathsf{epochChanged}\text{’})$
  assert $\mathsf{uid}_{\mathsf{lead}} \notin \{\mathsf{uid}_i\}_{i\in[n]} \wedge \{(\mathsf{uid}_i, *)\}_{i\in[n]} \subseteq \mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$
  ${*}\mathsf{setGroup}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \setminus \{(\mathsf{uid}_i, *)\}_{i\in[n]})$
  assert ${*}\mathsf{verifyConsistency}(\mathsf{uid}_{\mathsf{lead}}, *)$
  return $(\mathsf{M}, {*}\mathsf{pubState}(\mathsf{uid}_{\mathsf{lead}}))$

Challenges & corruptions

Oracle: $\mathsf{Test}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p})$
  req $\mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \neq \bot$
  if $\mathsf{ChallKeys}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] = \bot$ then
    $\mathsf{ChallKeys}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \leftarrow \mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$
  return $\mathsf{ChallKeys}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$

Oracle: $\mathsf{Challenge}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p})$
  req $\mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \neq \bot$
  $\mathsf{Challs} \leftarrow \mathsf{Challs} \cup \{(\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p})\}$
  if $\mathsf{ChallKeys}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] = \bot$ then
    if $b = 1$ then
      $\mathsf{ChallKeys}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \leftarrow \mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$
    else
      $\mathsf{ChallKeys}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \leftarrow\$ \mathcal{K}$
  return $\mathsf{ChallKeys}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$

Oracle: $\mathsf{Corrupt}(\mathsf{id})$
  $\mathsf{CorrIds} \leftarrow \mathsf{CorrIds} \cup \{\mathsf{id}\}$
  $\mathsf{L} \leftarrow \varnothing$
  for all $\mathsf{uid} : \mathsf{St}[\mathsf{uid}] \neq \bot$ do
    if $\mathsf{Identity}(\mathsf{uid}) = (\mathsf{id}, \cdot)$ then
      $\mathsf{Corrupted}[\mathsf{uid}] \leftarrow \mathtt{true}$
      $\mathsf{L} \leftarrow \mathsf{L} \cup \{\mathsf{St}[\mathsf{uid}]\}$
  $(\mathsf{isk}, \cdot) \leftarrow \mathsf{PKI.get\text{-}sk}(\mathsf{id})$
  return $(\mathsf{L}, \mathsf{isk})$

PKI Interaction

Oracle: $\mathsf{Verify\text{-}Pk}(\mathsf{id}, \mathsf{ipk})$
  return $\mathsf{PKI.verify\text{-}pk}(\mathsf{id}, \mathsf{ipk})$

Oracle: $\mathsf{Sign}(\mathsf{id}, \mathsf{context}, \mathsf{m})$
  req $\mathsf{context} \neq \text{‘}\mathsf{EncryptionKeyAnnouncement}\text{’}$
  $(\mathsf{isk}, \cdot) \leftarrow \mathsf{PKI.get\text{-}sk}(\mathsf{id})$
  return $\mathsf{Sig.Sign}(\mathsf{isk}, \mathsf{context}, \mathsf{m})$
```

Fig. 10: The game formalizing the security of a cmKEM scheme Ψ.

Game $\mathsf{Sec}^{\mathrm{cmKEM}}_{\Psi,\mathcal{A}}$ Helpers

```
Helper: $*\mathsf{pubState}(\mathsf{uid})$
  return $(\mathsf{Epoch}[\mathsf{uid}], \mathsf{Period}[\mathsf{uid}])$

Helper: $*\mathsf{isLeader}(\mathsf{uid})$
  return $\mathsf{Leader}[\mathsf{uid}] = \mathsf{uid}$

Helper: $*\mathsf{members}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p})$
  if $\mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \neq \bot$ then
      return $\mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$
  else if $\mathsf{p} > 0$ then
      return $*\mathsf{members}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p} - 1)$
  else return $\varnothing$

Helper: $*\mathsf{verifyCredentials}(\mathsf{uid})$
  $(\mathsf{id}, \cdot) \leftarrow \Psi.\mathsf{Identity}(\mathsf{uid})$
  return $\mathsf{id} \in \mathsf{CorrIds} \vee \neg\mathsf{Corrupted}[\mathsf{uid}]$

Helper: $*\mathsf{verifyProgress}(\mathsf{uid}, \mathsf{expected})$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{Epoch}[\mathsf{uid}], \mathsf{Period}[\mathsf{uid}])$
  $(\mathsf{e}', \mathsf{p}') \leftarrow (\mathsf{St}[\mathsf{uid}].\mathsf{e}, \mathsf{St}[\mathsf{uid}].\mathsf{p})$
  $(\mathsf{Epoch}[\mathsf{uid}], \mathsf{Period}[\mathsf{uid}]) \leftarrow (\mathsf{e}', \mathsf{p}')$
  if $\mathsf{expected} = \text{‘startedSession’}$ then
      return $(e = \bot \vee e' > e) \wedge p' = 0$
  else if $\mathsf{expected} = \text{‘joinedSession’}$ then
      return $(e = \bot \vee e' > e)$
  else if $\mathsf{expected} = \text{‘epochChanged’}$ then
      return $e' = e + 1 \wedge p' = 0$
  else if $\mathsf{expected} = \text{‘periodChanged’}$ then
      return $e' = e \wedge p' = p + 1$
  else if $\mathsf{expected} = \text{‘eitherChanged’}$ then
      return $[(e' = e + 1 \wedge p' = 0) \vee (e' = e \wedge p' = p + 1)]$
             $\wedge\ e' \leq \mathsf{Epoch}[\mathsf{Leader}[\mathsf{uid}]]$
  return true

Helper: $*\mathsf{verifyConsistency}(\mathsf{uid}, \mathsf{ad})$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{Epoch}[\mathsf{uid}], \mathsf{Period}[\mathsf{uid}])$
  $\mathsf{uid}_{\mathsf{lead}} \leftarrow \mathsf{Leader}[\mathsf{uid}]$
  // No assurance after (potential) active attack
  if $*\mathsf{triviallyInjectable}(\mathsf{uid})$ then
      return true
  // Group
  if $(\mathsf{uid}, \mathsf{ad}) \notin *\mathsf{members}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p})$ then
      return false
  // Keys
  if $\mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \notin \{\bot, \mathsf{St}[\mathsf{uid}].\mathsf{k}\} \vee \mathsf{St}[\mathsf{uid}].\mathsf{k} = \bot$ then
      return false
  $\mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \leftarrow \mathsf{St}[\mathsf{uid}].\mathsf{k}$
  return true

Helper: $*\mathsf{setGroup}(\mathsf{uid}_{\mathsf{lead}}, G')$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{Epoch}[\mathsf{uid}_{\mathsf{lead}}], \mathsf{Period}[\mathsf{uid}_{\mathsf{lead}}])$
  $\mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \leftarrow G'$

Helper: $*\mathsf{triviallyInjectable}(\mathsf{uid})$
  return $\mathsf{Corrupted}[\mathsf{uid}] \vee \mathsf{Corrupted}[\mathsf{Leader}[\mathsf{uid}]]$

Helper: $*\mathsf{safe}()$
  for $(\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}) \in \mathsf{Challs}$ do
      if $\mathsf{Corrupted}[\mathsf{uid}_{\mathsf{lead}}]$ then return false
      for $\mathsf{uid} \in *\mathsf{members}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p})$ do
          if $\mathsf{Corrupted}[\mathsf{uid}]$ then return false
  return true
```

Fig. 11: The helper functions for the cmKEM security game from Fig. 10.

This information is then, among others, used to formalize member authentication in the $*$verifyCredentials helper. In a nutshell, this helper ensures that the protocol only accepts a credential if it is valid. For instance in the StartSession oracle, if the protocol accepts (and **try** does not cause the oracle invocation to abort prematurely) then for each participant uid we must have $*$verifyCredentials(uid), or the adversary wins the game.

Note that the definition technically does not define the validity of a concrete credential sig, but rather defines the necessary conditions for such a valid credential to exist, which is either when the user is honest, or when the adversary knows the corresponding long-term key $\mathsf{id} \in \mathsf{CorrIds}$. (Encoding a concrete verification would not improve security, as in both cases the adversary can trivially input a valid credential and we would, hence, only rule out stupid attacks.)

Challenges. The game keeps track of all challenged keys using the Challs set. Moreover, to answer challenges consistently it uses the ChallKeys array, where once a key for a given state is output it is recorded. (That is, rather than ensuring that only either the Challenge or Test oracle can be invoked, we simply record the answer of the first such invocation.)

Safety predicate. The $*$safe helper method disallows trivial distinguishing strategies by determining whether the combinations of challenges and corruptions is "safe." More concretely it checks that if a user uid has been corrupted, then no state in which uid is a member must have been challenged, as otherwise the adversary could trivially distinguish using the leaked state.

Note that the $*$members helper function tackles a subtlety with respect to being a member and how ratcheting epochs and periods work. Assume that the leader $\mathsf{uid}_{\mathsf{lead}}$ removed a user uid while being in epoch e and period p, causing the leader to advance to the state $\mathsf{e} + 1$ and period 0. Zoom's scheme allows however for a network controlling adversary to instruct another member uid' to instead advance to $(\mathsf{e}, \mathsf{p}+1)$ first, for which $\mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}+1]$ has never been set. Since uid however could compute the respective key, uid needs to be considered a member of that given state.

Consistency properties. The helpers $*\mathsf{verifyProgress}$ and $*\mathsf{verifyConsistency}$ not only update the game's state but also ensure its consistency properties. First, $*\mathsf{verifyProgress}$ ensures that a given member ends up in the expected epoch and period. Note that the each call site in the game indicates whether the epoch or period should have been incremented using a special flag. Some special consideration has to be given to the invocation in the $\mathsf{JoinSession}$ oracle. When uid joins a session for the first time, we allow the party to end up in any arbitrary epoch and period (which $*\mathsf{verifyProgress}$ will allow, as so far $\mathsf{Epoch}[\mathsf{uid}]$ is not set), while when switching to another session, the party must end up in a strictly greater epoch.

Second, the $*\mathsf{verifyConsistency}(\mathsf{uid}, \mathsf{ad})$ helper first checks that only legitimate members end up in a given state. Note that, for simplicity, we often use the special wildcard symbol $*$ for the associated data, i.e., for a given uid and group G, the condition $(\mathsf{uid}, *) \in G$ is true if there exists ad such that $(\mathsf{uid}, \mathsf{ad}) \in G$. Similarly, we write $G \setminus \{(\mathsf{uid}_i, *)\}_{i\in[n]}$ to denote the set $\{(\mathsf{uid}, \mathsf{ad}) \in G : \mathsf{uid} \notin \{\mathsf{uid}_1, \ldots, \mathsf{uid}_n\}\}$. Note that, when a participant joins a group we do enforce an exact match of the ad components. Further, $*\mathsf{verifyConsistency}$ verifies that uid's key is the same as their leader's, i.e., that $\mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] = \mathsf{st.k}$ for st denoting uid's protocol state. Observe, however, that uid might run ahead of $\mathsf{uid}_{\mathsf{lead}}$ with respect to the period. To accommodate for this corner case, $*\mathsf{verifyConsistency}$ actually sets $\mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$ to uid's key in case it has not been assigned yet. This ensures that whenever any other member uid' — including $\mathsf{uid}_{\mathsf{lead}}$ — later moves to the same state, consistency is ensured.

PKI interaction. The cmKEM primitive makes use of a PKI for binding a party's the long-term signing key to their identity id. (For instance, id could be a user name, phone number or email address.) In reality, one has to assume that an adversary can verify any such signature, which is reflected in the game exposing access to $\mathsf{PKI.verify\text{-}pk}$. Furthermore, the signing key output by the PKI — in Zoom's protocol — is not exclusively used for the cmKEM protocol. We take this into account by exposing an additional signing oracle. Note that, however, that domain separation is ensured by preventing the adversary from using the cmKEM context identifier '$\mathsf{EncryptionKeyAnnouncement}$'.

Advantage. We say that a cmKEM scheme is secure if no PPT adversary has a better than negligible advantage in winning the game of Fig. 10. The advantage is defined as

$$\mathsf{Adv}^{\mathrm{cmKEM}}_{\Psi,\mathcal{A}} := 2\big(\Pr\big[\mathsf{Sec}^{\mathrm{LL\text{-}CGKA}}_{\Pi,\mathcal{A}} \Rightarrow 1\big] - \tfrac{1}{2}\big)$$

We intentionally do not define the advantage as a positive quantity by taking its absolute value, as the game returns `true` if the adversary wins by triggering an assertion or correctly guesses the bit b, and `false` if it performs any disallowed operations (such as challenging a key known to a corrupted party).

C.4 Zoom's cmKEM Scheme

Recall Zoom's cmKEM scheme from Section 2.4 and Fig. 1 in particular.

We first establish the following simple result.

Theorem 5. *Zoom's cmKEM scheme, which is presented in Fig. 1, is correct if the underlying nonce-based* AEAD *and signature schemes are correct.*

Proof. This follows directly from inspection of the protocol. Clearly, if $\mathbb{G} = \langle g \rangle$ is a Diffie-Hellman group, then `*encrypt-seed` derives the same symmetric key $k \leftarrow \mathsf{HKDF}(\mathsf{DL}_g(\mathsf{upk}_{\mathsf{uid}_{\mathsf{lead}}}, \mathsf{upk}_{\mathsf{uid}}), \text{'}\mathsf{KeyMeetingSeed}\text{'})$ for both the leader $\mathsf{uid}_{\mathsf{lead}}$ and a respective participant uid. Hence, assuming correctness of the AEAD scheme, uid will decrypt the seed that $\mathsf{uid}_{\mathsf{lead}}$ actually sent. Let's first consider the case of a new epoch. Here, $\mathsf{uid}_{\mathsf{lead}}$ samples a fresh seed', encrypts this to all participants (including potential new members) and then derives $(\mathsf{seed}, k) \leftarrow \mathsf{PRG.Eval}(\mathsf{seed}')$. The recipients, on the other hand, decrypt seed' and then also apply

$(\mathsf{seed}, k) \leftarrow \mathsf{PRG.Eval}(\mathsf{seed}')$, obviously resulting in the same key and seed. Second, consider the case of a period change. Here, $\mathsf{uid}_{\mathsf{lead}}$ encrypts the current seed' (before deriving seed and k) to all fresh parties, who then again derive the correct seed and k, while existing parties are simply told to apply $(\mathsf{seed}, k) \leftarrow \mathsf{PRG.Eval}(\mathsf{seed}')$ as well. □

Next, we consider the scheme's security.

Theorem 6 (Formal version of Theorem 1). *Zoom's cmKEM scheme presented in Fig. 1 is secure according to the game from Fig. 10, i.e.,*

$$\mathsf{Adv}^{cmKEM}_{\Psi,\mathcal{A}} = 2\big(\Pr\big[\mathsf{Sec}^{LL\text{-}CGKA}_{\Pi,\mathcal{A}} \Rightarrow 1\big] - \tfrac{1}{2}\big) \leq negl(\kappa),$$

under the Gap-DH assumption, when assuming that Hash *is collision resistant, the AEAD scheme is secure, the signature scheme is EUF-CMA secure, the PRG satisfies the standard indistinguishability from random notion, and modeling* HKDF *as a random oracle.*

Proof. We show this proof in three parts. First, we consider a sequence of hybrids leading to a game analogous to $\mathsf{Sec}^{\mathrm{cmKEM}}_{\Psi,\mathcal{A}}$ but with a partly idealized scheme cmKEM′ that uses independent shared symmetric key, rather than the actual keys derived from the Diffie-Hellman exchange, for the AEAD. Second, we consider a sequence of hybrids that gradually "disable" winning strategies for the adversary, e.g., where the game the adversary interacts with has certain winning conditions such as **assert** removed. We eventually end up with a game where the only winning strategy for the adversary is to guess the bit b, and show that the adversary's success probability remains unchanged from one such hybrid to the next. Finally, we show that in this "idealized" game guessing the bit b also cannot be done with more than negligible probability.

We first consider a hybrid execution $\mathcal{H}^{\mathsf{keys\text{-}1}}_{\Psi',\mathcal{A}}$ that behaves like $\mathsf{Sec}^{\mathrm{cmKEM}}_{\Psi,\mathcal{A}}$, but where the adversary wins whenever a collision in Hash (used by Ψ on the associated data) is detected. In particular the game monitors whether, during the execution of Ψ, two different ad and ad' are input such that $\mathsf{Hash}(\mathsf{ad}) = \mathsf{Hash}(\mathsf{ad}')$; if so, then the game immediately returns `true` (in addition to the regular ways of winning the game). Clearly, this hybrid is indistinguishable from the real game by collision resistance of Hash.

Then, we consider another hybrid $\mathcal{H}^{\mathsf{keys\text{-}2}}_{\Psi',\mathcal{A}}$ that behaves like $\mathcal{H}^{\mathsf{keys\text{-}1}}_{\Psi',\mathcal{A}}$, but uses fresh symmetric keys, independent of the public keys, for the AEAD encryption. More concretely, whenever $\mathcal{H}^{\mathsf{keys\text{-}2}}_{\Psi',\mathcal{A}}$ needs a symmetric key for $\mathsf{AEAD.Enc}$ for executing the scheme's `*encrypt-seed` helper — which can happen as part of a $\mathsf{StartSession}$, Add, or Remove oracle invocation — then $\mathcal{H}^{\mathsf{keys\text{-}2}}_{\Psi',\mathcal{A}}$ computes the scheme as follows:

- if the same key — i.e., for the same pair of parties uid and uid' — has been used before, then reuses that key;
- else if either the sender uid or the receiver uid' has been corrupted, or not honestly generated to begin with, then the game computes the proper symmetric key as in Ψ;
- else it samples a uniform random k of the appropriate length, and stores the tuple $(\mathsf{upk}, \mathsf{upk}', k)$ where upk and upk' are the public key stored in uid and uid', respectively.

If, later, a user with public key upk'' is corrupted, then the system takes all cached keys $(\mathsf{upk}, \mathsf{upk}', k)$ where either $\mathsf{upk}'' = \mathsf{upk}$ or $\mathsf{upk}'' = \mathsf{upk}'$, computes the respective Diffie-Hellman element (recall that $\mathcal{H}^{\mathsf{keys\text{-}2}}_{\Psi',\mathcal{A}}$ sampled the respective secret key usk''), and programs the random oracle at this position to return k.

Since in $\mathcal{H}^{\mathsf{keys\text{-}1}}_{\Psi',\mathcal{A}}$ the value k is the output of a random oracle, and thus uniformly randomly distributed, clearly $\mathcal{H}^{\mathsf{keys\text{-}2}}_{\Psi',\mathcal{A}}$ behaves exactly identically unless $\mathcal{A}$ queries the random oracle at one of the positions where it later is programmed. (Or two honest Diffie-Hellman key pairs collide, which only happens with negligible probability.) We now show that triggering such a bad event can be reduced to breaking the Gap-DH assumption.

Claim: Querying the ROM at a position that later needs programming, and thus distinguishing $\mathcal{H}^{\mathsf{keys\text{-}1}}_{\Psi',\mathcal{A}}$ and $\mathcal{H}^{\mathsf{keys\text{-}2}}_{\Psi',\mathcal{A}}$, implies breaking Gap-DH.

Proof: Assume N is an upper bound on the number of honest users created in the interaction. Then, the reduction first guesses which of the $N^2/2$ instances (i, j), the adversary will break. (If the reduction guesses wrong, and the adversary, e.g., corrupts one of those parties, the reduction forfeits. This incurs at most a quadratic loss in the reduction.)

The reduction then emulates $\mathcal{H}^{\mathsf{keys\text{-}1}}_{\Psi',\mathcal{A}}$ based on the Gap-DH instance as follows: Everything not directly related to the Diffie-Hellman keys is executed just as in $\mathcal{H}^{\mathsf{keys\text{-}1}}_{\Psi',\mathcal{A}}$. When tasked to honestly create a user, the reduction obtains the two group elements g^a and g^b from the Gap-DH instance and uses them as public keys for parties i and j. (For the other users, the reduction simply samples the secret keys themselves.) As a result, when required to compute a k involving either party, such as $\mathsf{HKDF}(g^{ax}, \text{‘}\mathsf{KeyMeetingSeed}\text{’})$ for a g^x input by the adversary, the reduction may no longer be able to do so. Instead, it proceeds as follows: it first checks that the adversary has not queried the respective group element at the random oracle HKDF — using the DDH oracle — and if such an element is found it uses the k selected by HKDF. Otherwise it just samples a fresh uniform k and for all subsequent random oracle queries checks whether they need to be programmed to this value accordingly. Overall, assuming that we guessed the instance correctly, this reduction thus behaves exactly as $\mathcal{H}^{\mathsf{keys\text{-}1}}_{\Psi',\mathcal{A}}$ with the adversary triggering the bad event implying a solution to the Gap-DH instance. □

This concludes the first part of the proof. So far, we have $\mathcal{H}^{\mathsf{keys\text{-}2}}_{\Psi',\mathcal{A}}$ that is a variant of $\mathsf{Sec}^{\text{cmKEM}}_{\Psi,\mathcal{A}}$ which executes an idealized protocol that uses uniform random symmetric encryption keys for the communication between uid and uid' rather than the ones obtained by applying Diffie-Hellman to the respective embedded public keys upk and upk', and we have

$$|\Pr[\mathcal{H}^{\mathsf{keys\text{-}2}}_{\Psi',\mathcal{A}} \Rightarrow 1] - \Pr[\mathsf{Sec}^{\text{cmKEM}}_{\Psi,\mathcal{A}} \Rightarrow 1]| \leq \text{negl}(\kappa) \qquad (1)$$

under the Gap-DH assumptions.

Next, we gradually modify the actual security game (rather than the scheme) to disable all its winning conditions except for guessing the bit b.

Claim: Consider the hybrid $\mathcal{H}^{\mathsf{checks\text{-}1}}_{\Psi',\mathcal{A}}$ that behaves like $\mathcal{H}^{\mathsf{keys\text{-}2}}_{\Psi',\mathcal{A}}$ except for the helper $*\mathsf{verifyCredentials}(\mathsf{uid})$ always returning `true`. (Analogously, that all statements **assert** $*\mathsf{verifyCredentials}(\mathsf{uid})$ are removed.) We claim that those experiments are computationally indistinguishable if the signing scheme is EUF-CMA secure. That is, $|\Pr[\mathcal{H}^{\mathsf{keys\text{-}2}}_{\Psi',\mathcal{A}} \Rightarrow 1] - \Pr[\mathcal{H}^{\mathsf{checks\text{-}1}}_{\Psi',\mathcal{A}} \Rightarrow 1]|$ is bounded by the probability of an a PPT $\mathcal{A}'$ breaking the EUF-CMA game, where $\mathcal{A}'$ has roughly the same running time as $\mathcal{A}$.

Proof: We establish this by observing that in order to have $*\mathsf{verifyCredentials}$ return false in the original game, id must not have been corrupted, i.e., $\mathsf{id} \notin \mathsf{CorrIds}$, and uid must have been recorded as corrupted, i.e., $\mathsf{Corrupted}[\mathsf{uid}] = \mathtt{true}$. The former condition implies that so far $\mathsf{Corrupt}(\mathsf{id})$ has not been invoked (and thus the signing key not leaked to the adversary) whereas the latter condition implies that either uid has not been honestly generated, or uid has been leaked by calling $\mathsf{Corrupt}(\mathsf{id})$. In short, $*\mathsf{verifyCredentials}(\mathsf{uid}) = \mathtt{false}$ implies that $\mathsf{Corrupt}(\mathsf{id})$ has not been invoked and uid has not been honestly generated. Consider now the $\mathsf{StartSession}$ oracle. There, $*\mathsf{verifyCredentials}$ is only invoked if the scheme did not reject the given inputs. In the scheme, sig_i is a signature of uid_i with associated data ‘$\mathsf{EncryptionKeyAnnouncement}$’, which then gets verified as part of the $\mathsf{StartSession}$ algorithm, for all $i \in [n]$. Hence, those checks passing for some uid_i without $\mathsf{Corrupt}(\mathsf{id})$ having been invoked or uid_i being the result of an honest $\mathsf{CreateUser}$ invocation clearly implies a signature forgery, as such signatures can also not be generated using the game's signing oracle. Analogous arguments can be made for the $\mathsf{JoinSession}$ and Add oracles. □

Claim: Consider a game $\mathcal{H}^{\mathsf{checks\text{-}2}}_{\Psi',\mathcal{A}}$ that behaves like $\mathcal{H}^{\mathsf{checks\text{-}1}}_{\Psi',\mathcal{A}}$ but has various **assert** statements removed. First, in the $\mathsf{CreateUser}$ oracle the following assertion are disabled: the ones about the long-term identity and public key matching the one returned by $\Psi.\mathsf{Identity}$, and the last one about $\mathsf{CreateUser}$ returning a valid state (not $\perp$) and uid being fresh, i.e., $\mathsf{St}[\mathsf{uid}] = \perp$. Second, throughout the game, all assertions with respect to the $\Psi.\mathsf{Meeting}$ algorithm and, finally, in the Add and Remove oracles the respective assertions about the group rosters.

We have that $|\Pr[\mathcal{H}^{\mathsf{checks\text{-}2}}_{\Psi',\mathcal{A}} \Rightarrow 1] - \Pr[\mathcal{H}^{\mathsf{checks\text{-}1}}_{\Psi',\mathcal{A}} \Rightarrow 1]|$ is negligible if the signing scheme is correct.

Proof: In the $\mathsf{CreateUser}$ oracle, the assertions about $\mathsf{Identity}$ immediately follow from inspection of the scheme, as does the one about the meeting id and the algorithm returning a valid state. The one about the credentials follows by correctness of the signing scheme, and the $\mathsf{CreateUser}$ algorithm returning a fresh and unpredictable uid follows from the scheme sampling a fresh and uniform random $\mathsf{upk} \in \mathbb{G}$.

For the remaining assertions related to $\mathsf{Meeting}$, observe that they are simply checked beforehand by the respective protocol algorithms that reject any input for which they would be violated. The same holds for the group roster assertions, essentially just reflecting the checks performed by the protocol. Hence, $\mathcal{H}^{\mathsf{checks\text{-}1}}_{\Psi',\mathcal{A}}$ and $\mathcal{H}^{\mathsf{checks\text{-}2}}_{\Psi',\mathcal{A}}$ behave the same assuming the correctness of the underlying signature scheme. □

Claim: Furthermore, consider $\mathcal{H}^{\mathsf{checks\text{-}3}}_{\Psi',\mathcal{A}}$ that replaces all the **assert** $*\mathsf{verifyProgress}$ statements by an invocation of $*\mathsf{verifyProgress}$ without the adversary winning the game if it returns false. We have that $|\Pr[\mathcal{H}^{\mathsf{checks\text{-}3}}_{\Psi',\mathcal{A}} \Rightarrow 1] - \Pr[\mathcal{H}^{\mathsf{checks\text{-}2}}_{\Psi',\mathcal{A}} \Rightarrow 1]|$ is negligible.
Proof: Again for the most part those checks are trivially satisfied by the way the protocol increments the epoch and period numbers, respectively. Note that if a user has not been in any session before, then $*\mathsf{verifyProgress}$ in $\mathsf{StartSession}$ or $\mathsf{JoinSession}$ always returns true. □

Claim: Finally, consider $\mathcal{H}^{\mathsf{checks\text{-}4}}_{\Psi',\mathcal{A}}$ that defuses the $*\mathsf{verifyConsistency}$ assertions, i.e., still invokes the helper methods but without the adversary winning based on the return value. We have that $|\Pr[\mathcal{H}^{\mathsf{checks\text{-}4}}_{\Psi',\mathcal{A}} \Rightarrow 1] - \Pr[\mathcal{H}^{\mathsf{checks\text{-}3}}_{\Psi',\mathcal{A}} \Rightarrow 1]|$ is negligible if the underlying AEAD scheme provides authenticity.
Proof: Observe that $*\mathsf{verifyConsistency}$ always returns `true` (and the assertion is not triggered) if either uid or their respective current leader $\mathsf{uid}_{\mathsf{lead}}$ have been corrupted. Hence, in the following we focus on the case where both are honest. First, consider the check of uid being a member of the group failing. That is $(\mathsf{uid}, \mathsf{ad}) \notin *\mathsf{members}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p})$. That in particular means that $(\mathsf{uid}, \mathsf{ad}) \notin \mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}']$ for all $\mathsf{p}' \leq \mathsf{p}$. If uid is the leader, then the game enforces that they are always part of the group. Otherwise, the only way for a participant uid to move to this epoch e is by receiving an encryption of a seed for some $\mathsf{p}' \leq \mathsf{p}$. If both uid and $\mathsf{uid}_{\mathsf{lead}}$ are honest, and given that no Hash collisions have occurred so far, this implies by the authenticity of the AEAD scheme that either of the two parties must have sent such a message. However, as a participant, uid would not accept such a message from themselves, even if they were a leader at some earlier point, due to enforcing that epochs increment monotonically upon a leader change. Furthermore, authenticity of the AEAD scheme ensures that the participant only accepts the message if the associated data matches. Hence, the only option is for $\mathsf{uid}_{\mathsf{lead}}$ to actually have sent such message with the matching associated data, in which case $(\mathsf{uid}, \mathsf{ad})$ must also be contained in $\mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}']$.

Next, consider the key consistency check to be failing. Observe that for $\mathsf{p} = 0$ all parties (except the leader) only transition upon receiving an encrypted seed, and an honest leader will send the same seed to all participants. Hence, analogous to the argument above, we can conclude that consistency of $\mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, 0]$ is implied by the authenticity of the AEAD scheme, for any $\mathsf{uid}_{\mathsf{lead}}$ and e. For $\mathsf{p} > 0$, observe that the protocol derives those keys deterministically from the previous period. Assume that for uid this check fails, after deterministically hashing forward, for some $\mathsf{p} > 0$ — with no previous check having failed at this point. Then, $*\mathsf{verifyConsistency}$ must have stored a different key k' when verifying another party uid'. Due to the determinism, that means that either those parties uid and uid' had incompatible keys in period $\mathsf{p} - 1$, in which case a previous consistency check would have failed, or uid' joined the meeting at this point without having been in $\mathsf{p} - 1$. In the latter case, uid', however, got this key encrypted from $\mathsf{uid}_{\mathsf{lead}}$, meaning that uid and $\mathsf{uid}_{\mathsf{lead}}$ would already have had an inconsistent state at $\mathsf{p} - 1$ (if uid had also just joined, they would have received the same seed from $\mathsf{uid}_{\mathsf{lead}}$ as uid'). Hence, in either case the first check to fail cannot be the key consistency check for a period $\mathsf{p} > 0$. □

In summary, we have $\mathcal{H}^{\mathsf{checks\text{-}4}}_{\Psi',\mathcal{A}}$ that is a variant of $\mathcal{H}^{\mathsf{keys\text{-}2}}_{\Psi',\mathcal{A}}$ where the winning condition has been modified such that only guessing b counts as winning, and combining the hybrid steps with Eq. (1), we have

$$|\Pr[\mathcal{H}^{\mathsf{checks\text{-}4}}_{\Psi',\mathcal{A}} \Rightarrow 1] - \Pr[\mathsf{Sec}^{\mathrm{cmKEM}}_{\Psi,\mathcal{A}} \Rightarrow 1]| \leq \mathrm{negl}(\kappa) \tag{2}$$

under the given assumptions. We, thus, conclude the proof by showing the following claim.

Claim: We have that guessing b in $\mathcal{H}^{\mathsf{checks\text{-}4}}_{\Psi',\mathcal{A}}$ is hard, i.e.,

$$2\big(\Pr\big[\mathcal{H}^{\mathsf{checks\text{-}4}}_{\Psi',\mathcal{A}} \Rightarrow 1\big] - \tfrac{1}{2}\big) \leq \mathrm{negl}(\kappa) \tag{3}$$

if the underlying AEAD and the PRG are secure, according to the respective standard notions.
Proof: First, consider an adversary $\mathcal{A}$ that never corrupts a party or injects their own key, i.e., in any interaction $\mathsf{Corrupted}[\mathsf{uid}] = \mathtt{false}$ whenever uid is used. Now consider a challenge for a state identified by the triple $\mathsf{uid}_{\mathsf{lead}}$, e, p. If $\mathsf{p} = 0$, then this means that $\mathsf{uid}_{\mathsf{lead}}$ encrypted a fresh PRG state seed to all participants, from which the key k is then derived using $(\mathsf{seed}', k) \leftarrow \mathsf{PRG.Eval}(\mathsf{seed})$. By the security of the AEAD scheme the adversary does not have any information on seed, and thus by the PRG security, we can conclude that k is indistinguishable from a uniform random one (as with $b = 1$) that is moreover independent of seed'. Now consider the period $\mathsf{p} = 1$, in which case the key is derived by simply iterating the PRG once more, i.e., $(\mathsf{seed}'', k') \leftarrow \mathsf{PRG.Eval}(\mathsf{seed}')$. If there are newly added parties in that period, then $\mathsf{uid}_{\mathsf{lead}}$ encrypts them seed'. Again, by AEAD security and security of the PRG, k' is indistinguishable from a uniform random key that is chosen independently of seed''. By iterating the same argument for periods $\mathsf{p} > 1$, we can conclude that all keys are indistinguishable from independent uniform random ones.

It remains to consider corruptions and convince ourselves that all possible resulting attacks are excluded as trivial by the $*\mathsf{safe}$ predicate. First, we consider an attacker that now can corrupt parties but will not inject any message to a party uid if either uid or their current leader $\mathsf{uid}_{\mathsf{lead}}$ have been corrupted (similar to an honest-but-curious adversary). To this end, assume that a party uid either starts out as honest and is then later corrupted as part of a $\mathsf{Corrupt}(\mathsf{id})$ call, or uid was not honestly generated in the first place. Assume that uid has been added to the session of a leader $\mathsf{uid}_{\mathsf{lead}}$ in (e, p) and (potentially) removed in $(\mathsf{e}', \mathsf{p}')$. According to $*\mathsf{safe}$ this means that challenging any state from (e, p) until $\mathsf{e}' + 1$ is prohibited. (Observe that $*\mathsf{members}$ still considers uid to be part of $(\mathsf{e}', \mathsf{p}' + 1)$ and so on.) Clearly, the leader does not send uid any information about epochs greater than e' or smaller than e. Hence, it suffices to consider periods before p in epoch e, where we can also observe that the seed uid obtains does not reveal information about the previous keys. The same argument extends to cases where either uid is part of multiple sessions (that in terms of keys are completely independent) or is later re-added to the same session. Finally, we observe that corrupting a session leader disables all challenges on that given session.

We conclude by considering fully active attacks: Whenever either uid or $\mathsf{uid}_{\mathsf{lead}}$ has been corrupted, then the adversary can make uid produce an inconsistent key — with respect to the key the honest leader would produce. We remark, however, that the adversary can only challenge honest keys, since by design $*\mathsf{verifyConsistency}$ returns early for uid, before the injected key could be stored in the game's state. This in particular implies that if the adversary would try to deliver a (mauled) ciphertext from another session to uid, either by authenticity of the AEAD scheme uid will reject (if uid and $\mathsf{uid}_{\mathsf{lead}}$ are honest) or the adversary is unable to challenge the resulting state, implying that he cannot obtain information from such an injection. In summary, our game simply disallows this kind of (trivial) active attacks. □

Combining Eqs. (2) and (3) directly yields that $\mathsf{Adv}^{\mathrm{cmKEM}}_{\Psi,\mathcal{A}}$ is negligible, concluding the proof. □

D Details on LL-CGKA

D.1 Security

In this section, we expand on Section 3.2 and discuss the formal security definition of a LL-CGKA scheme depicted in Figs. 12 and 13.

Game overview. The structure of the game is fairly similar to the one of the cmKEM primitive discussed in Section 2.3 and Appendix C.3. It notably differs in the handling of time, additional security guarantees and some other (minor) syntactic differences which we discuss below. (For instance, while the cmKEM primitive is a more technical abstraction centered around the notion of a session, the LL-CGKA primitive is tied more stringently to the notion of a meeting and, e.g., replaces $\mathsf{StartSession}$ with $\mathsf{CatchUp}$ followed by Lead.)

First, observe that the game maintains a *global clock* time that is advanced by the adversary. That is, for each fixed time, the adversary can specify as many operations (such as instructing the leader to add parties or letting a participant process a message) before incrementing time by 1. When creating a user uid, the adversary specifies that parties drift δ with respect to the global clock. This drift is then stored in $\mathsf{Offset}[\mathsf{uid}]$ and throughout the game used to convert global time to local time whenever any of the LL-CGKA algorithms is invoked; otherwise, all timestamps within the security game refer to global time.

Another important difference is that the LL-CGKA game keeps track of more *past state* in order to formalize stronger properties such as liveness. For instance, while the cmKEM game only kept each user uid's current epoch and period, the Epoch and Period now track uid's epoch and period over time. That is, $\mathsf{Epoch}[\mathsf{uid}, \mathsf{time}]$ stores the epoch in which uid was at that *global* time and analogously for $\mathsf{Period}[\mathsf{uid}, \mathsf{time}]$. Moreover, $\mathsf{Leader}[\mathsf{uid}, \mathsf{e}]$ keeps track of uid's leader for each epoch e. (Recall that each leader change implies an epoch change.)

Key consistency and confidentiality. The game defines the confidentiality and consistency of keys analogous to the cmKEM game. That is, each $\mathsf{Challenge}$ query is, depending on the bit b, either answered with the actual protocol key as stored in the Key array, or an independent uniform random one. Similarly, $*\mathsf{verifyConsistency}$ ensures that a participant has a consistent view on the key with their respective leader $\mathsf{uid}_{\mathsf{lead}}$ (and all other participants following the same leader).

The non-triviality condition, ruling out certain combinations of challenges and corruptions, is essentially also still the same: Due to the lack of either FS or PCS, a key is considered to be trivially computable by any member that is or has been part of the respective group, irrespective of when the corruption happens. We stress that, in this regard, the $\mathsf{Challenge}$ oracle only allows to challenge keys for which one member has actually been in that state; merely receiving (and setting aside for later) a key without transitioning to the respective epoch e' and period p' does not assign the key to $\mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}', \mathsf{p}']$.

No front running. In contrast to the cmKEM notion, the LL-CGKA game ensures that all participants only move to a state their leader has already been. The exception is the case where the leader's long-term identity but not their ephemeral identity has been compromised (this is due to in certain settings new states only being authenticated using long-term signing keys). To this end, the game sets keys (and groups) upon each action by the leader. Then, $*\mathsf{verifyConsistency}$ ensures that when a participant moves to a new epoch or period, either the leader's long-term identity has been corrupted or the relevant key has already been set.

Progress and credentials. The game ensures that either the epoch or period gets incremented with every change to the group. Since participants are expected to not run ahead of their leader, the difference between epochs and periods, however, becomes mostly irrelevant from a security perspective.[13] Hence, as a simplification, $*\mathsf{verifyProgress}$ simply checks that either of the two has been incremented. Note, however, that for a participant in LL-CGKA not every processed message necessarily triggers an advance to the next group state. As such, for $\mathsf{Process}$ the game only checks that the epoch-period pair does not decrease. Progress for participants is then guaranteed as part of liveness (see below).

As in the cmKEM game, $*\mathsf{verifyCredentials}$ ensures that the adversary cannot impersonate users without having compromised their long-term key material.

Group roster consistency. The LL-CGKA game formalizes that all parties have a consistent view on the group roster for each state. (Where a state is identified by the leader, epoch, and period respectively.) To this end, $*\mathsf{verifyConsistency}$ checks that the group stored as part of a participant uid's state, $\mathsf{St}[\mathsf{uid}].G$ matches the intended group by the leader. (Observe to this end, that in all operations changing the group roster, $*\mathsf{setGroup}$ stores the intended group of the leader $\mathsf{uid}_{\mathsf{lead}}$ in $\mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$ *before* $*\mathsf{verifyConsistency}$ is invoked for all parties including $\mathsf{uid}_{\mathsf{lead}}$.) Note that group roster consistency, in contrast to key consistency, only holds as long as the leader's long-term identity has not been compromised.

Moreover, as a sanity check, $*\mathsf{verifyConsistency}$ also ensures that if uid transitions to a state it considers themselves to be part of that state.

[13] The sole exception is the case where a leader's long-term identity but not ephemeral identity has been compromised.

Game $\mathsf{Sec}^{\text{LL-CGKA}}_{\Pi,\mathcal{A}}$

Main

```
Procedure: Initialize
  $b \leftarrow\$ \{0,1\}$
  $\mathsf{won} \leftarrow \texttt{false}$
  $\mathsf{time} \leftarrow 1$
  $\mathsf{CorrIds}, \mathsf{Challs} \leftarrow \varnothing$
  $\mathsf{St}[\cdot], \mathsf{Joined}[\cdot], \mathsf{Leader}[\cdot,\cdot], \mathsf{Group}[\cdot,\cdot,\cdot], \mathsf{Epoch}[\cdot,\cdot],$
      $\mathsf{Period}[\cdot,\cdot], \mathsf{Key}[\cdot,\cdot,\cdot], \mathsf{ChallKeys}[\cdot,\cdot,\cdot], \mathsf{Offset}[\cdot] \leftarrow \bot$
  $\mathsf{Corrupted}[\cdot] \leftarrow \texttt{true}$

Procedure: Finalize($b'$)
  if $\neg *\mathsf{safe}()$ then return $\texttt{false}$
  else if $\mathsf{won}$ then return $\texttt{true}$
  else return $b' = b$
```

PKI interaction

```
Oracle: Verify-Pk(id, ipk)
  return PKI.verify-pk(id, ipk)

Oracle: Sign(id, context, m)
  req context $\notin$ {'EncryptionKeyAnnouncement', 'LeaderParticipantList'}
  $(\mathsf{isk}, \cdot) \leftarrow$ PKI.get-sk(id)
  return Sig.Sign(isk, context, m)
```

Clock

```
Oracle: Tick
  $\mathsf{time} \leftarrow \mathsf{time} + 1$
  $\mathsf{M}[\cdot], P[\cdot] \leftarrow \bot$
  for all uid : $\mathsf{St}[\mathsf{uid}] \neq \bot$ do
    $(\mathsf{e},\mathsf{p}) \leftarrow (\mathsf{Epoch}[\mathsf{uid}, \mathsf{time}-1], \mathsf{Period}[\mathsf{uid}, \mathsf{time}-1])$
    $(\mathsf{Epoch}[\mathsf{uid}, \mathsf{time}], \mathsf{Period}[\mathsf{uid}, \mathsf{time}]) \leftarrow (\mathsf{e},\mathsf{p})$
    if $*\mathsf{isLeader}(\mathsf{uid}, \mathsf{time})$ then
      $(\mathsf{St}[\mathsf{uid}], \mathsf{M}[\mathsf{uid}]) \leftarrow$
          $\Pi.\mathsf{LeaderTick}(\mathsf{St}[\mathsf{uid}], \mathsf{time} + \mathsf{Offset}[\mathsf{uid}])$
      assert $*\mathsf{verifyProgress}(\mathsf{uid}, \mathsf{time}, \text{'weak'})$
      if $(\mathsf{Epoch}[\mathsf{uid}, \mathsf{time}], \mathsf{Period}[\mathsf{uid}, \mathsf{time}]) \neq (\mathsf{e},\mathsf{p})$ then
        $*\mathsf{setGroup}(\mathsf{uid}, \mathsf{time}, \mathsf{Group}[\mathsf{uid}, \mathsf{e}, \mathsf{p}])$
        $*\mathsf{setKey}(\mathsf{uid}, \mathsf{time})$
        assert $*\mathsf{verifyConsistency}(\mathsf{uid}, \mathsf{time})$
      $P[\mathsf{uid}] \leftarrow *\mathsf{pubState}(\mathsf{uid}, \mathsf{time})$
    else
      $(\mathsf{St}[\mathsf{uid}], \mathsf{alive}, \mathsf{M}[\mathsf{uid}]) \leftarrow \Pi.\mathsf{ParticipantTick}(\mathsf{St}[\mathsf{uid}], \mathsf{time} + \mathsf{Offset}[\mathsf{uid}])$
      if alive then assert $*\mathsf{verifyLiveness}(\mathsf{uid}, \mathsf{time}) \wedge \mathsf{St}[\mathsf{uid}] \neq \bot$
      else $\mathsf{St}[\mathsf{uid}] \leftarrow \bot$ // uid drops out
  return $(\mathsf{M}, P)$
```

User management

```
Oracle: CreateUser(id, meetingId, offset)
  $(\mathsf{ust}, \mathsf{uid}, \mathsf{sig}) \leftarrow \Pi.\mathsf{CreateUser}(\mathsf{time} + \mathsf{offset}, \mathsf{id}, \mathsf{meetingId})$
  $\mathsf{Offset}[\mathsf{uid}] \leftarrow \mathsf{offset}$
  assert $\Pi.\mathsf{Identity}(\mathsf{uid}) = \mathsf{id} \wedge \Pi.\mathsf{Meeting}(\mathsf{uid}) = \mathsf{meetingId}$
  assert $\mathsf{St}[\mathsf{uid}] = \bot$
  $\mathsf{St}[\mathsf{uid}] \leftarrow \mathsf{ust}$
  $\mathsf{Joined}[\mathsf{uid}] \leftarrow \mathsf{time}$
  $\mathsf{Corrupted}[\mathsf{uid}] \leftarrow \texttt{false}$
  return $(\mathsf{uid}, \mathsf{sig})$

Oracle: CatchUp(uid, grpPub)
  req $\mathsf{St}[\mathsf{uid}] \neq \bot$
  try $\mathsf{St}[\mathsf{uid}] \leftarrow \Pi.\mathsf{CatchUp}(\mathsf{St}[\mathsf{uid}], \mathsf{time} + \mathsf{Offset}[\mathsf{uid}], \mathsf{grpPub})$
  return $*\mathsf{pubState}(\mathsf{uid}, \mathsf{time})$
```

Group management

```
Oracle: Lead($\mathsf{uid}_{\mathsf{lead}}, \{\mathsf{uid}_i, \mathsf{sig}_i\}_{i \in [n]}$)
  req $\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}] \neq \bot$
  try $(\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}], \mathsf{M})$
        $\leftarrow \Pi.\mathsf{Lead}(\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}], \mathsf{time} + \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}], \{\mathsf{uid}_i, \mathsf{sig}_i\}_{i \in [n]})$
  for $i \in [n]$ do assert $*\mathsf{verifyCredentials}(\mathsf{uid}_n)$
  assert $*\mathsf{verifyProgress}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}, \text{'strict'})$
  $*\mathsf{setGroup}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}, \{\mathsf{uid}_{\mathsf{lead}}, \mathsf{uid}_1, \ldots, \mathsf{uid}_n\})$
  $*\mathsf{setKey}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time})$
  assert $*\mathsf{verifyConsistency}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time})$
  return $(\mathsf{M}, *\mathsf{pubState}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}))$
```

Group management (leader)

```
Oracle: Add($\mathsf{uid}_{\mathsf{lead}}, \{(\mathsf{uid}_i, \mathsf{sig}_i)\}_{i \in [n]}$)
  req $\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}] \neq \bot \wedge *\mathsf{isLeader}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time})$
  $(\mathsf{e},\mathsf{p}) \leftarrow (\mathsf{Epoch}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}], \mathsf{Period}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}])$
  $G \leftarrow \mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$
  try $(\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}], \mathsf{M}) \leftarrow \Pi.\mathsf{Add}(\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}], \mathsf{time} +$
  $\mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}], \{(\mathsf{uid}_i, \mathsf{sig}_i)\}_{i \in [n]})$
  for $i \in [n]$ do
    assert $\mathsf{uid}_i \notin G \wedge \mathsf{Meeting}(\mathsf{uid}_i) = \mathsf{Meeting}(\mathsf{uid}_{\mathsf{lead}})$
              $\wedge\ *\mathsf{verifyCredentials}(\mathsf{uid}_i)$
  assert $*\mathsf{verifyProgress}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}, \text{'strict'})$
  $*\mathsf{setGroup}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}, G \cup \{\mathsf{uid}_i\}_{i \in [n]})$
  $*\mathsf{setKey}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time})$
  assert $*\mathsf{verifyConsistency}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time})$
  return $(\mathsf{M}, *\mathsf{pubState}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}))$

Oracle: Remove($\mathsf{uid}_{\mathsf{lead}}, \{\mathsf{uid}_i\}_{i \in [n]}$)
  req $\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}] \neq \bot \wedge *\mathsf{isLeader}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time})$
  $(\mathsf{e},\mathsf{p}) \leftarrow (\mathsf{Epoch}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}], \mathsf{Period}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}])$
  $G \leftarrow \mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$
  try $(\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}], \mathsf{M}) \leftarrow \Pi.\mathsf{Remove}(\mathsf{St}[\mathsf{uid}_{\mathsf{lead}}], \mathsf{time} + \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}], \{\mathsf{uid}_i\}_{i \in [n]})$
  for $i \in [n]$ do
    assert $\mathsf{uid}_i \in G \wedge \mathsf{uid}_i \neq \mathsf{uid}_{\mathsf{lead}}$
  assert $*\mathsf{verifyProgress}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}, \text{'strict'})$
  $*\mathsf{setGroup}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}, G \setminus \{\mathsf{uid}_i\}_{i \in [n]})$
  $*\mathsf{setKey}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time})$
  assert $*\mathsf{verifyConsistency}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time})$
  return $(\mathsf{M}, *\mathsf{pubState}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}))$
```

Participants

```
Oracle: Follow($\mathsf{uid}, \mathsf{m}, \mathsf{uid}'_{\mathsf{lead}}, \mathsf{sig}'_{\mathsf{lead}}$)
  req $\mathsf{St}[\mathsf{uid}] \neq \bot$
  try $\mathsf{St}[\mathsf{uid}] \leftarrow \Pi.\mathsf{Follow}(\mathsf{St}[\mathsf{uid}], \mathsf{time} + \mathsf{Offset}[\mathsf{uid}], \mathsf{m}, \mathsf{uid}'_{\mathsf{lead}}, \mathsf{sig}'_{\mathsf{lead}})$
  assert $*\mathsf{verifyCredentials}(\mathsf{uid}'_{\mathsf{lead}})$
  assert $*\mathsf{verifyProgress}(\mathsf{uid}, \mathsf{time}, \text{'strict'})$
  $*\mathsf{setLeader}(\mathsf{uid}, \mathsf{time}, \mathsf{uid}'_{\mathsf{lead}})$
  assert $*\mathsf{verifyConsistency}(\mathsf{uid}, \mathsf{time})$
  return $*\mathsf{pubState}(\mathsf{uid}, \mathsf{time})$

Oracle: Process(uid, m)
  req $\mathsf{St}[\mathsf{uid}] \neq \bot \wedge \neg *\mathsf{isLeader}(\mathsf{uid}, \mathsf{time})$
  try $\mathsf{St}[\mathsf{uid}] \leftarrow \Pi.\mathsf{Process}(\mathsf{St}[\mathsf{uid}], \mathsf{time} + \mathsf{Offset}[\mathsf{uid}], \mathsf{m})$
  $\mathsf{uid}_{\mathsf{lead}} \leftarrow \mathsf{Leader}[\mathsf{uid}, \mathsf{Epoch}[\mathsf{uid}, \mathsf{time}]]$
  assert $*\mathsf{verifyProgress}(\mathsf{uid}, \mathsf{time}, \text{'weak'})$
  $*\mathsf{setLeader}(\mathsf{uid}, \mathsf{time}, \mathsf{uid}_{\mathsf{lead}})$
  assert $*\mathsf{verifyConsistency}(\mathsf{uid}, \mathsf{time})$
  return $*\mathsf{pubState}(\mathsf{uid}, \mathsf{time})$
```

Challenges & corruptions

```
Oracle: Test($\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}$)
  req $\mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \neq \bot$
  if $\mathsf{ChallKeys}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] = \bot$ then
    $\mathsf{ChallKeys}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \leftarrow \mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$
  return $\mathsf{ChallKeys}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$

Oracle: Challenge($\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}$)
  req $\mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \neq \bot$
  $\mathsf{Challs} \leftarrow \mathsf{Challs} \cup \{(\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p})\}$
  if $\mathsf{ChallKeys}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] = \bot$ then
    if $b = 1$ then
      $\mathsf{ChallKeys}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \leftarrow \mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$
    else
      $\mathsf{ChallKeys}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \leftarrow\$ \mathcal{K}$
  return $\mathsf{ChallKeys}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$

Oracle: Corrupt(id)
  $\mathsf{CorrIds} \leftarrow \mathsf{CorrIds} \cup \{\mathsf{id}\}$
  $\mathsf{L} \leftarrow \varnothing$
  for all uid : $\mathsf{St}[\mathsf{uid}] \neq \bot$ do
    if $\mathsf{Identity}(\mathsf{uid}) = \mathsf{id}$ then
      $\mathsf{Corrupted}[\mathsf{uid}] \leftarrow \texttt{true}$
      $\mathsf{L} \leftarrow \mathsf{L} \cup \{\mathsf{St}[\mathsf{uid}]\}$
  $(\mathsf{isk}, \cdot) \leftarrow$ PKI.get-sk(id)
  return (L, isk)
```

Fig. 12: The LL-CGKA security game. We define an LL-CGKA scheme to be secure if no PPT $\mathcal{A}$ can win this game with probability better than negligible above one half.

Game $\mathsf{Sec}^{\text{LL-CGKA}}_{\Pi,\mathcal{A}}$ Helpers

```
Helper: $*\mathsf{pubState}(\mathsf{uid}, \mathsf{time})$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{Epoch}[\mathsf{uid}, \mathsf{time}], \mathsf{Period}[\mathsf{uid}, \mathsf{time}])$
  $\mathsf{uid}_{\mathsf{lead}} \leftarrow \mathsf{Leader}[\mathsf{uid}, \mathsf{e}]$
  return $(\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}, \mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}])$

Helper: $*\mathsf{isLeader}(\mathsf{uid}, \mathsf{time})$
  $\mathsf{e} \leftarrow \mathsf{Epoch}[\mathsf{uid}, \mathsf{time}]$
  return $\mathsf{Leader}[\mathsf{uid}, \mathsf{e}] = \mathsf{uid}$

Helper: $*\mathsf{setGroup}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}, G')$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{Epoch}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}], \mathsf{Period}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}])$
  $\mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \leftarrow G'$
  $*\mathsf{setLeader}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}, \mathsf{uid}_{\mathsf{lead}})$

Helper: $*\mathsf{setLeader}(\mathsf{uid}, \mathsf{time}, \mathsf{uid}'_{\mathsf{lead}})$
  $\mathsf{e} \leftarrow \mathsf{Epoch}[\mathsf{uid}, \mathsf{time}]$
  $\mathsf{Leader}[\mathsf{uid}, \mathsf{e}] \leftarrow \mathsf{uid}'_{\mathsf{lead}}$

Helper: $*\mathsf{setKey}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{time})$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{Epoch}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}], \mathsf{Period}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}])$
  $\mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \leftarrow \mathsf{St}[\mathsf{uid}_{\mathsf{lead}}].\mathsf{k}[\mathsf{e}, \mathsf{p}]$

Helper: $*\mathsf{members}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p})$
  if $\mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \neq \perp$ then
    return $\mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$
  else if $\mathsf{p} > 0$ then
    return $*\mathsf{members}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p} - 1)$
  else return $\varnothing$

Helper: $*\mathsf{relevantPriorLeaders}(\mathsf{uid}, \mathsf{time})$
  // Finds all prior meeting leaders since the
  // last time the meeting “started over”,
  // i.e. the roster changed completely
  $L \leftarrow \varnothing$ // all leaders encountered
  $G \leftarrow \{\mathsf{uid}\}$ // all users encountered
  for $\mathsf{e}' \leftarrow \mathsf{Epoch}[\mathsf{uid}, \mathsf{time}], \dots, 0$ do
    $G' \leftarrow \varnothing$ // additional users
    for $\mathsf{uid}' \in G$ do
      $\mathsf{uid}'_{\mathsf{lead}} \leftarrow \mathsf{Leader}[\mathsf{uid}', \mathsf{e}']$
      if $\mathsf{uid}'_{\mathsf{lead}} \neq \perp$ then
        $L \leftarrow L \cup \{\mathsf{uid}'_{\mathsf{lead}}\}$
        for $\mathsf{uid}' : \mathsf{Meeting}(\mathsf{uid}') = \mathsf{Meeting}(\mathsf{uid}'')$
                    $\wedge\, \mathsf{uid}'_{\mathsf{lead}} = \mathsf{Leader}[\mathsf{uid}'', \mathsf{e}']$ do
          $G' \leftarrow G' \cup \{\mathsf{uid}'\}$
    $G \leftarrow G \cup G'$
  return $L$

Helper: $*\mathsf{verifyProgress}(\mathsf{uid}, \mathsf{time}, \mathsf{strictly})$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{Epoch}[\mathsf{uid}, \mathsf{time}], \mathsf{Period}[\mathsf{uid}, \mathsf{time}])$
  $(\mathsf{e}', \mathsf{p}') \leftarrow (\mathsf{St}[\mathsf{uid}].\mathsf{e}, \mathsf{St}[\mathsf{uid}].\mathsf{p})$
  $(\mathsf{Epoch}[\mathsf{uid}, \mathsf{time}], \mathsf{Period}[\mathsf{uid}, \mathsf{time}]) \leftarrow (\mathsf{e}', \mathsf{p}')$
  if $\mathsf{e} \neq \perp$ then
    if $\mathsf{strictly} = \text{‘strict’}$ then
      return $(\mathsf{e}' > \mathsf{e} \vee (\mathsf{e}' = \mathsf{e} \wedge \mathsf{p}' > \mathsf{p}))$
    else if $\mathsf{strictly} = \text{‘weak’}$ then
      return $(\mathsf{e}' > \mathsf{e} \vee (\mathsf{e}' = \mathsf{e} \wedge \mathsf{p}' \geq \mathsf{p}))$
  return true
```

```
Helper: $*\mathsf{verifyCredentials}(\mathsf{uid})$
  $(\mathsf{id}, \cdot) \leftarrow \mathsf{Identity}(\mathsf{uid})$
  return $\mathsf{id} \in \mathsf{CorrIds} \vee \neg\mathsf{Corrupted}[\mathsf{uid}]$

Helper: $*\mathsf{verifyConsistency}(\mathsf{uid}, \mathsf{time})$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{Epoch}[\mathsf{uid}, \mathsf{time}], \mathsf{Period}[\mathsf{uid}, \mathsf{time}])$
  $\mathsf{uid}_{\mathsf{lead}} \leftarrow \mathsf{Leader}[\mathsf{uid}, \mathsf{e}]$
  // No assurances during (potential) active attack
  if $*\mathsf{triviallyInjectable}(\mathsf{uid})$ then
    return true
  // Keys
  if $\mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \notin \{\perp, \mathsf{St}[\mathsf{uid}].\mathsf{k}\} \vee \mathsf{St}[\mathsf{uid}].\mathsf{k} = \perp$ then
    return false // basic consistency
  if $\mathsf{Identity}(\mathsf{uid}_{\mathsf{lead}}) \notin \mathsf{CorrIds} \wedge \mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] = \perp$ then
    return false // no front running
  $\mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}] \leftarrow \mathsf{St}[\mathsf{uid}].\mathsf{k}$
  // Group
  if $\mathsf{uid} \notin *\mathsf{members}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}) \vee \mathsf{uid} \notin \mathsf{St}[\mathsf{uid}].G$ then
    return false
  if $\mathsf{Identity}(\mathsf{uid}_{\mathsf{lead}}) \notin \mathsf{CorrIds} \wedge \mathsf{St}[\mathsf{uid}].G \neq \mathsf{Group}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$ then
    return false
  // History
  if $\mathsf{Identity}(\mathsf{uid}_{\mathsf{lead}}) \notin \mathsf{CorrIds}$ then
    for all $\mathsf{uid}' : \mathsf{Meeting}(\mathsf{uid}') = \mathsf{Meeting}(\mathsf{uid}) \;\wedge\; \mathsf{Leader}[\mathsf{uid}', \mathsf{e}] =$
    $\mathsf{Leader}[\mathsf{uid}, \mathsf{e}]$ do
      $\mathsf{e}_{\min} \leftarrow \min(i : \mathsf{Leader}[\mathsf{uid}, \mathsf{e}'] \neq \perp \;\wedge\; \mathsf{Leader}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}'] \neq \perp)$
      for $\mathsf{e}' \leftarrow \mathsf{e}_{\min}, \dots, \mathsf{e} - 1$ do
        if $\mathsf{Leader}[\mathsf{uid}, \mathsf{e}'] \neq \mathsf{Leader}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}']$ then
          return false
  return true

Helper: $*\mathsf{verifyLiveness}(\mathsf{uid}, \mathsf{time})$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{Epoch}[\mathsf{uid}, \mathsf{time}], \mathsf{Period}[\mathsf{uid}, \mathsf{time}])$
  $\mathsf{uid}_{\mathsf{lead}} \leftarrow \mathsf{Leader}[\mathsf{uid}, \mathsf{e}]$
  // No assurance after (potential) active attack
  for $\mathsf{uid}'_{\mathsf{lead}} \in *\mathsf{relevantPriorLeaders}(\mathsf{uid}, \mathsf{time})$ do
    if $\mathsf{Identity}(\mathsf{uid}'_{\mathsf{lead}}) \in \mathsf{CorrIds}$ then
      return true
  // Was leader recently here?
  $\Delta \leftarrow *\mathtt{liveness\text{-}slack}(\mathsf{uid}, \mathsf{time}, \mathsf{Leader}, \mathsf{Joined}, \mathsf{Offset})$
  for $\mathsf{time}' \leftarrow \mathsf{time} - \Delta, \dots, \mathsf{time}$ do
    if $(\mathsf{e} = \mathsf{Epoch}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}']$
              $\wedge\, \mathsf{p} \geq \mathsf{Period}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}'])$
          $\vee\, \mathsf{e} > \mathsf{Epoch}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}']$ then
      return true
  return false

Helper: $*\mathsf{triviallyInjectable}(\mathsf{uid}, \mathsf{time})$
  $\mathsf{e} \leftarrow \mathsf{Epoch}[\mathsf{uid}, \mathsf{time}]$
  return $\mathsf{Corrupted}[\mathsf{uid}] \vee \mathsf{Corrupted}[\mathsf{Leader}[\mathsf{uid}, \mathsf{e}]]$

Helper: $*\mathsf{safe}()$
  for $(\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}) \in \mathsf{Challs}$ do
    if $\mathsf{Corrupted}[\mathsf{uid}_{\mathsf{lead}}]$ then return false
    for $\mathsf{uid} \in *\mathsf{members}(\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p})$ do
      if $\mathsf{Corrupted}[\mathsf{uid}]$ then return false
  return true
```

Fig. 13: Additional functions used to define the the LL-CGKA security game from Fig. 12.

Meeting history consistency. As an additional property, the game ensures that if any two parties end up in the same state, they share a consistent view on the meeting's history for their common suffix, i.e., since the later one has been added. This, in particular, rules out attacks where a malicious Zoom server instructs different (disjoint) subsets of participants to first follow different leaders, splitting the meeting, and then later merging them under one common leader again.

Due to the key and group roster consistency, enforcing that participants have a consistent view on the history of leaders suffices to ensure an overall consistent view. This is formalized in $*\mathsf{verifyConsistency}$ by checking that, for each pair of parties that currently have the same leader, epoch, and period, they agree on the leader of each prior epoch since the last of them joined the meeting. (Observe that enforcing consistency between each uid and their current leader $\mathsf{uid}_{\mathsf{lead}}$ is insufficient, as $\mathsf{uid}_{\mathsf{lead}}$ might be much younger than some of the parties.)

Liveness. Finally, we consider our novel liveness property. Liveness is checked as part of the Tick oracle, ensuring that if a participant stays in the meeting after ParticipantTick, then their leader must have been in the same state recently, with the adversary winning otherwise. This, in turn, is formalized as part of ∗verifyLiveness by looking for a time within the given liveness slack for which the leader had the same state. Due to the game permitting multiple operations within one clock tick but only recording the latest state for each time, the notion formally cannot check that the leader has been in exactly that state but one that is at least as old. It is easy to see that this implies liveness as well.

Moreover, in the basic security definition, note we do not assure liveness in the presence of active attacks. This is formalized by ∗verifyLiveness checking that none of the leader's so far could have been (trivially) impersonated by an active attacker. To this end, the ∗relevantPriorLeaders helper computes all the past leaders of a given meeting, except for the case where a meeting at some point completely "started over" (i.e. the leader and set of participants changed completely from one epoch to the next), in which case past leaders no longer matter.

We remark that `*liveness-slack` is a parameter of the security notion, and it is expressed as a function rather than a single value so that the concrete guarantees can depend on the details of the protocol execution, for example how many different leaders a party has encountered. Indeed, Zoom's current scheme[1] and our improved proposal both achieve this notion with different liveness slack.

D.2 The protocol

In this section, we expand on Section 3.3 by discussing some of the more technical aspects. The protocol is parameterized in a signature scheme Sig, a hash function Hash, and a cmKEM scheme.

Client protocol. Recall the formal description of the client scheme presented in Fig. 3. For simplicity, the scheme is described implicitly maintaining state across invocations — a description of that state can be found in Table 1.

ust.e, ust.p, ust.k[·, ·], and ust.G	The exported fields as defined in the LL-CGKA syntax.
st	The state of a cmKEM protocol.
me	One's own ephemeral identity.
$\mathsf{uid}_{\mathsf{lead}}$	The identity of the leader.
isk	The user's (long-term) signing key (ipk denotes the corresponding verification key).
$\mathsf{e}_{\mathsf{next}}$ and $\mathsf{p}_{\mathsf{next}}$	The largest epoch and period, respectively, for which a key has been received (but not necessarily the corresponding LPL).
lplHash	The hash of the last LPL message sent or received.
hbHash	The hash of the last heart-beat message sent or received.
lastHb	An estimate on when the last known heartbeat has been sent by the leader. If none is known, this contains the time when CreateUser was executed.
Added and Removed	Parties added and removed, respectively, since the last time the LPL has been broadcast. (Maintained by the leader only.)
numLplLinks	The number of LPL messages since the latest coalesced one. (For the leader only.)

Table 1: The client's state of the LL-CGKA scheme.

cmKEM and LPL. Observe that the construction directly leverages a cmKEM scheme for the key exchange. To generate the LPL messages, a leader $\mathsf{uid}_{\mathsf{lead}}$ maintains the sets of parties added and removed since broadcasting the last LPL. For sending a differential LPL in the `*send-LPL` helper algorithm, $\mathsf{uid}_{\mathsf{lead}}$ sends out those sets (alongside the other information), while for a coalesced LPL $\mathsf{uid}_{\mathsf{lead}}$ sends out their current group roster G. Each LPL message further contains a monotonic counter v that is maintained across leader changes.

Heartbeats and liveness. In the following, we point out a number of subtleties with respect to the liveness component of the scheme. The lastHb variable denotes the (estimated) *send time* of the last known heartbeat. For a leader $\mathsf{uid}_{\mathsf{lead}}$, this simply corresponds to the actual sending time; for a participant uid, however, it corresponds to the last received heartbeat's indicated timestamp after accounting for any correctable clock drift (see the `*update-liveness` helper). As a result, uid will drop out in $\mathsf{ParticipantTick}$ whenever $\mathsf{time} > \mathsf{lastHb} + \Delta_{\mathsf{live}}$. We slightly abuse the lastHb variable, moreover, to ensure that joining the meeting takes no longer than Δ_{live}. To this end, when creating the ephemeral identity we simply initialize lastHb to the current time. While it does not correspond to anything heartbeat related at this point, the regular liveness mechanism then takes care that after Δ_{live} the party is deemed stale and no longer joins.

The protocol keeps track of an upper bound δ on the clock drift with the current leader. To this end, the protocol estimates the drift as the difference between the timestamp time' indicated in the heartbeat message, and the local time time it has been received at – that is, the protocol generally assumes no network delay. If the network does exhibit delays, then the drift estimates $\mathsf{time} - \mathsf{time}'$ exceeds the actual drift, and as such the party overestimates the recentness of the latest heartbeat, which results in the protocol prioritizing correctness over liveness. Note that for subsequent heartbeats from the same leader the drift estimate can only decrease. If the perceived drift becomes larger, the party can conclude that the later heartbeat must have been delayed in transit.

Server protocol. The sever protocol (implicitly) maintains the following state per meeting $\mathsf{meetingId}$, which is initialized by the Init algorithm:

- The server state of the cmKEM protocol $\mathsf{pub}_{\mathsf{cmKEM}}[\mathsf{meetingId}]$.
- The current group roster $G[\mathsf{meetingId}]$ consisting of the ephemeral identities.
- The list of LPL messages $\mathsf{lpls}[\mathsf{meetingId}]$ since and including the last coalesced one, represented as a FIFO queue.
- The last heartbeat $\mathsf{hb}[\mathsf{meetingId}]$.

We remark that the server does *not* have to worry about concurrent sessions within a given meeting, as those should never occur with an honest server that orchestrates the execution well. Hence, the server can keep just one state per meeting rather than per meeting and session.

The protocol is depicted in Fig. 14. Once more, we write the Split and $\mathsf{GroupState}$ algorithms to be implicitly stateful with Init simply setting up the state (rather than outputting it). The split algorithm considers two main cases. If the message M contains a cmKEM message M_K, then it splits this using the respective cmKEM algorithm. This produces a resulting cmKEM share for each current group member that the server then just forwards alongside any potential LPL or heartbeat messages. Additionally, it uses those shares to derive the current group roster. If M does not contain a cmKEM message, then the server just forwards the LPL message and heartbeat to the last known roster.

The $\mathsf{GroupState}$ algorithm simply outputs the latest cmKEM epoch, the queue of LPL messages, and the last heartbeat. The LPL queue gets updated as part of Split using the `*process-LPL` helper algorithm, and contains all LPL messages since the last coalesced one.

D.3 Proof of Theorem 2

In this section, we prove our main result. First, we translate the liveness slack of Theorem 2 into the respective helper function `*liveness-slack` of the formal security game as follows.

Protocol Zoom's Server LL-CGKA

```
Algorithm: Init()
  $\mathsf{pub}_{\mathsf{cmKEM}} \leftarrow \mathrm{cmKEM}.\mathsf{InitSplitState}()$
  $G[\cdot] \leftarrow \varnothing$
  $\mathsf{lpls}[\cdot] \leftarrow \langle\rangle$ // Empty FIFO-queue
  $\mathsf{hb}[\cdot] \leftarrow \bot$

Algorithm: GroupState(meetingId)
  $\mathsf{grpPub} \leftarrow (\mathsf{pub}_{\mathsf{cmKEM}}.\mathsf{E}[\mathsf{meetingId}], \mathsf{lpls}[\mathsf{meetingId}], \mathsf{hb}[\mathsf{meetingId}])$
  return grpPub

Helper: *process-LPL(meetingId, lpl)
  parse $(\mathsf{v}, \mathsf{coalesced}, \cdots) \leftarrow \mathsf{lpl}$
  if coalesced then
      $\mathsf{lpls}[\mathsf{meetingId}] \leftarrow \langle\rangle$ // Restart LPL queue
  lpls[meetingId].enq(lpl)

Algorithm: Split(M)
  parse $(\mathsf{uid}_{\mathsf{lead}}, \mathsf{M}_K, \mathsf{lpl}, \mathsf{hb}) \leftarrow \mathsf{M}$
  $\mathsf{meetingId} \leftarrow \mathrm{cmKEM}.\mathsf{Meeting}(\mathsf{uid}_{\mathsf{lead}})$
  $\mathsf{ms}[\cdot] \leftarrow \bot$
  if $\mathsf{M}_K \neq \bot$ then
      try $(\mathsf{pub}_{\mathsf{cmKEM}}, \mathsf{ms}_K) \leftarrow \mathrm{cmKEM}.\mathsf{Split}(\mathsf{pub}_{\mathsf{cmKEM}}, \mathsf{M}_K)$
      $G[\mathsf{meetingId}] \leftarrow \varnothing$
      for all $\mathsf{uid} : \mathsf{M}_K[\mathsf{uid}] \neq \bot$ do
          $G[\mathsf{meetingId}] \leftarrow G[\mathsf{meetingId}] \cup \{\mathsf{uid}\}$
          $\mathsf{ms}[\mathsf{uid}] \leftarrow (\mathsf{M}_K[\mathsf{uid}], \mathsf{lpl}, \mathsf{hb})$
  else
      for all $\mathsf{uid} : G[\mathsf{meetingId}]$ do
          $\mathsf{ms}[\mathsf{uid}] \leftarrow (\bot, \mathsf{lpl}, \mathsf{hb})$
  if $\mathsf{hb} \neq \bot$ then
      $\mathsf{hb}[\mathsf{meetingId}] \leftarrow \mathsf{hb}$
  if $\mathsf{lpl} \neq \bot$ then
      *process-LPL(lpl, meetingId)
  return ms
```

Fig. 14: The server protocol of the LL-CGKA scheme.

Definition 3. *Consider the following liveness slack algorithm*

$$\ast\textit{liveness-slack}(\mathsf{uid}, \mathsf{time}, \mathsf{Leader}, \mathsf{Joined}, \mathsf{Offset}) := \min\bigl(n \cdot \varDelta_{\mathsf{live}}, \mathsf{time} - \mathsf{Joined}[\mathsf{uid}]\bigr) + \varDelta_{\mathsf{live}},$$

where n is the number of distinct leaders encountered so far and $\varDelta_{\mathsf{live}}$ a fixed parameter.

Using this, we now set out to restate our main theorem.

Theorem 7 (Formal version of Theorem 2). *The LL-CGKA scheme from Figs. 3 and 14 is secure according to Figs. 12 and 13 with the liveness slack from Definition 3. That is, for any PPT adversary $\mathcal{A}$, we have*

$$\mathsf{Adv}_{\Pi,\mathcal{A}}^{LL\text{-}CGKA} := 2\bigl(\Pr\bigl[\mathsf{Sec}_{\Pi,\mathcal{A}}^{LL\text{-}CGKA} \Rightarrow 1\bigr] - \tfrac{1}{2}\bigr) \leq \mathit{negl}(\kappa),$$

for the security parameter κ, if the underlying cmKEM scheme is secure, the signature scheme is EUF-CMA secure, and the hash function is collision resistant.

For ease of presentation, we divide the proof into two parts: the first part considering all security properties except liveness, and a second part considering liveness only.

Lemma 3. *The advantage of any PPT adversary $\mathcal{A}$ in winning a modified version of $\mathsf{Sec}_{\Pi,\mathcal{A}}^{LL\text{-}CGKA}$ in which* $\ast$verifyLiveness *always returns true, is negligible in the security parameter κ if the underlying primitives are secure.*

Proof. In the following we sketch a simple reduction to the cmKEM game, while assuming that the signatures are unforgeable. In brief, the reduction emulates the LL-CGKA game toward the adversary $\mathcal{A}$ by internally using the cmKEM game such that $\mathcal{A}$ winning the former implies the reduction winning the latter. (Or $\mathcal{A}$ having forged a signature or found a hash collision.)

To this end, the reduction internally runs the LL-CGKA scheme Π as follows: for everything cmKEM related it queries the respective oracles of the cmKEM game, while it computes LPL and heartbeat messages itself. To compute the heartbeats' signatures, the reduction uses the signing oracle exposed by the cmKEM game with the context string 'LeaderParticipantList'.

First, we observe that the two games' states line up in that the state of the LL-CGKA game is just a more fine-grained version of cmKEM game's state. For instance, when the cmKEM game store's Epoch[uid], then the LL-CGKA game stores Epoch[uid, time] such that the two values coincide for the current time. (The analogous holds for Period, while Leader[uid, e] corresponds to Leader[uid] for uid's current epoch.) In particular, we can observe that the reduction can maintain all those arrays consistently with exception of the following:

- The state array $\mathsf{St}[\mathsf{uid}]$. Here, the reduction only maintains the LL-CGKA specific state it needs to execute the protocol. Note that the only time it needs the full state (including the cmKEM part) is upon a corruption, in which case it can simply issue the corruption to the underlying cmKEM game.
- The keys array $\mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$, which it does not maintain. For Challenge and Test queries it directly uses the underlying game, while the emulated $*$verifyConsistency checks can simply omit the key-consistency check (therefore potentially returning `true` instead of `false`). This is because the key-consistency checks of the LL-CGKA game and the underlying cmKEM game are essentially the same. (In particular, processing a heartbeat or LPL message cannot cause a key inconsistency in the Zoom protocol.) Hence, the very moment those checks were relevant to properly emulate the $*$verifyConsistency of the LL-CGKA game, the underlying cmKEM game is won. As a consequence, omitting those checks does not affect $\mathcal{A}$'s behavior until $\mathcal{A}$ won the cmKEM game — $\mathcal{A}$ has the same winning probability on the underlying cmKEM game in this reduction as they had in a reduction that perfectly emulated $*$verifyConsistency.

As a direct consequence of the matching state arrays, we observe that the reduction's use of the cmKEM game to run its scheme is not impeded by the latter game's preconditions — i.e., the preconditions of the two games align. For instance, the preconditions $\mathcal{A}$ must satisfy when calling the Add-oracle in the LL-CGKA game are the same as the ones of the cmKEM game.

We now discuss the remaining differences with respect of some of the game's assurances.

$*$verifyProgress: In Zoom's protocol the epoch and period is advanced upon receiving a heartbeat and LPL message. However, upon closer inspection, we observe that it is simply advanced to the one stored from the underlying cmKEM protocol. In order to thus violate the predicate, in reality $\mathsf{e}_{\mathsf{next}}$ and $\mathsf{p}_{\mathsf{next}}$ would need to violate the assurance. Furthermore, we can observe that the variant of the cmKEM game is strictly stronger (by enforcing whether the epoch or period progressed depending on the caller). This, implies that any $\mathcal{A}$ that violates the LL-CGKA game's condition (by violating it for $\mathsf{e}_{\mathsf{next}}$ and $\mathsf{p}_{\mathsf{next}}$) for sure also wins the cmKEM game.

$*$verifyConsistency: Next, we consider the $*$verifyConsistency helpers. By inspection we realize that this time some of LL-CGKA game's checks are stronger than the corresponding checks in the cmKEM game: (1) the additional group and key consistency checks and (2) the new history checks. Hence, we need to prove that violating any of those additional conditions violates the security of one of the other primitives.

To this end consider first the additional key consistency key ensuring that, unless the leader's long-term identity has been corrupted, $\mathsf{Key}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{e}, \mathsf{p}]$ has already been set when a party transitions to epoch e and period p. Here we observe that the scheme only transitions epoch upon receiving a LPL message for said epoch and period as well as a heartbeat message authenticating that LPL message. Since the latter contains a signature under the leader's long-term signing key, having a party accepting such a message without the leader being in that epoch and period consists a signature forgery. Similarly, consider the additional group consistency check ensuring that the participant agrees with the leader on the group roster. Again, this property is only required to hold as long as the leader's long-term identity has not been compromised. Since the group roster is communicated via the LPL messages which are authenticated via signatures as part of the heartbeat messages, violating this property would again require a signature forgery.

Finally, consider the history consistency check and assume that there are users uid and uid' (belonging to the same meeting) for which the assertion fails. Both users must have received at least one heartbeat from $\mathsf{uid}_{\mathsf{lead}}$ certifying the current epoch e — for now assume that there is one of the heartbeat of $\mathsf{uid}_{\mathsf{lead}}$ for e that both parties received. Then, since heartbeats form a hash chain that is checked by the parties, by collision resistance this implies that both users agree on the sequence of all heartbeats up to the moment the later user joined. Since each heartbeat, however, uniquely identifies an epoch and leader — and parties only accept epochs for which they know at least heartbeat — this implies the desired consistency. (In the special case where they parties have no shared heartbeat of their current epoch, one party must just have joined and in particular not been in any other epoch, rendering the check trivial.)

$*\mathsf{safe}$*:* Finally, consider the $*\mathsf{safe}$ predicate determining trivial wins. The predicate is identical to the one of the cmKEM game. Hence, any win deemed non-trivial for the LL-CGKA game is also deemed non-trivial for the cmKEM game. □

Liveness. We now show that an adversary $\mathcal{A}$ can also not break the liveness properties formalized as part of the LL-CGKA game with non-negligible probability. To this end, we consider a sequence of additional lemmas establishing intuitive invariants.

Lemma 4. *Consider an execution of the LL-CGKA protocol Π from Figs. 3, 4 and 14 within $\mathsf{Sec}^{LL\text{-}CGKA}_{\Pi,\mathcal{A}}$, with a PPT $\mathcal{A}$. Then, whenever $*\mathsf{verifyLiveness}$ is not trivially disabled and for each user uid, we have*

$$\delta_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] - (\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]) \leq \min(n \cdot \Delta_{\mathsf{live}}, \mathsf{time} - \mathsf{Joined}[\mathsf{uid}])$$

where n denotes the number of distinct leaders uid encountered so far and $\delta_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}]$ refers to uid's protocol state (i.e., their estimates on the drift to $\mathsf{uid}_{\mathsf{lead}}$) while $\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]$ refers to the game state (i.e., the actual drift).

Proof. Clearly, the left-hand side of the equation is only updated whenever uid receives a heartbeat. Moreover, between receiving heartbeats, the bound on the right-hand side monotonically increases. Thus, it suffices to consider the invariant right after processing an incoming heartbeat. In the following, assume that the user uid received the last heartbeat at local time $\mathsf{time}_{\mathsf{uid}} = \mathsf{time} + \mathsf{Offset}[\mathsf{uid}]$ and that this heartbeat contained a timestamp, i.e., the sending time, $\mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} = \mathsf{time}' + \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]$. Hence, after invoking `*update-drift` we have

$$\begin{aligned}
&\delta_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] - (\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]) \\
&\quad \leq \mathsf{time}_{\mathsf{uid}} - \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} - (\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]) \\
&\quad = (\mathsf{time}_{\mathsf{uid}} - \mathsf{Offset}[\mathsf{uid}]) - (\mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]) \\
&\quad = \mathsf{time} - \mathsf{time}'
\end{aligned}$$

where the inequality follows from the minima computed as part of `*update-drift`.

We now first prove the second part of the bound. To this end, observe that in order for uid to accept a heartbeat, this heartbeat must sign over an LPL link that includes — or that links to a prior LPL link that includes — uid's fresh ephemeral identity (see `*receive-LPL`). By causality we can, thus, conclude that the heartbeat has been sent after uid created their identity, i.e., $\mathsf{Joined}[\mathsf{uid}] \leq \mathsf{time}'$ which directly yields

$$\mathsf{time} - \mathsf{time}' \leq \mathsf{time} - \mathsf{Joined}[\mathsf{uid}].$$

Next, we prove the first part of the bound using induction over the invocations of `*update-drift`, i.e., we consider one particular invocation and assume that so far the invariant has been maintained. First, consider the case where uid receives their very first heartbeat. We know that due to the liveness mechanism making uid wait at most Δ_{live} to join, receiving this heartbeat must occur at some global time $\mathsf{time} \leq \mathsf{Joined}[\mathsf{uid}] + \Delta_{\mathsf{live}}$. Combining this with the prior established bound directly leads to the following inequality implying the claim:

$$\mathsf{time} - \mathsf{time}' \leq \mathsf{time} - \mathsf{Joined}[\mathsf{uid}] \leq \mathsf{Joined}[\mathsf{uid}] + \Delta_{\mathsf{live}} - \mathsf{Joined}[\mathsf{uid}] \leq \Delta_{\mathsf{live}}.$$

For subsequent heartbeats we distinguish two cases. If the incoming heartbeat is from the same leader as the current one, then we observe that by definition of the protocol $\delta_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}]$ only gets smaller (or stays) while the $n \cdot \Delta_{\mathsf{live}}$ term remain unchanged. Hence, the invariant is trivially preserved. Finally, consider the case that the heartbeat is from a new leader. Let $\mathsf{uid}''_{\mathsf{lead}}$ denote the previous leader and $\mathsf{time}''_{\mathsf{uid}''_{\mathsf{lead}}} = \mathsf{time}'' + \mathsf{Offset}[\mathsf{uid}''_{\mathsf{lead}}]$ the (local) time they sent their last heartbeat before $\mathsf{uid}_{\mathsf{lead}}$ took over. Since uid processed this new heartbeat and hasn't dropped out yet, we know that

$$\mathsf{time} + \mathsf{Offset}[\mathsf{uid}] \leq \mathsf{lastHb} + \Delta_{\mathsf{live}}$$

and using the definition of the protocol variable lastHb set in `*update-liveness` we get

$$\begin{aligned}
&\mathsf{lastHb} + \varDelta_{\mathsf{live}} \\
&\quad = \mathsf{time}''_{\mathsf{uid}''_{\mathsf{lead}}} + \delta_{\mathsf{uid}}[\mathsf{uid}''_{\mathsf{lead}}] + \varDelta_{\mathsf{live}} \\
&\quad = \mathsf{time}'' + (\mathsf{Offset}[\mathsf{uid}''_{\mathsf{lead}}] + \delta_{\mathsf{uid}}[\mathsf{uid}''_{\mathsf{lead}}]) + \varDelta_{\mathsf{live}} \\
&\quad \leq \mathsf{time}'' + (n-1) \cdot \varDelta_{\mathsf{live}} + \mathsf{Offset}[\mathsf{uid}] + \varDelta_{\mathsf{live}} \\
&\quad = \mathsf{time}'' + n \cdot \varDelta_{\mathsf{live}} + \mathsf{Offset}[\mathsf{uid}]
\end{aligned}$$

when using the induction hypothesis in the second last step (on the term in parentheses). Combining the two prior bounds we hence obtain

$$\mathsf{time} - \mathsf{time}' \leq \mathsf{time}'' + n \cdot \varDelta_{\mathsf{live}} - \mathsf{time}' \leq n \cdot \varDelta_{\mathsf{live}}$$

where in the last step we used that $\mathsf{time}'' \leq \mathsf{time}'$ by causality. □

It remains to show that this implies the desired liveness properties, as expressed in the following lemma.

Lemma 5. *In the following, consider the game that behaves like* $\mathsf{Sec}^{LL\text{-}CGKA}_{\Pi,\mathcal{A}}$*, with* `*liveness-slack` *as defined in Definition 3, but where the winning condition is modified to only account for* $\ast\mathsf{verifyLiveness}$*. Then the advantage of any PPT adversary* $\mathcal{A}$ *in winning that game is negligible, if the hash function is collision resistant and the signature scheme EUF-CMA secure.*

Proof. Let time denote the current global time, as defined in the game. Assume that $\mathsf{uid}_{\mathsf{lead}}$ sent (at local time $\mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}}$) the last heartbeat which uid received so far, and let lastHb be the variable in uid's state right after that. Inserting the definition of $\mathsf{lastHb} = \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} + \delta_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}]$ in the following expression yields

$$\begin{aligned}
&\mathsf{lastHb} - \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} - (\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]) \\
&\qquad = \delta_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] - (\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]) \\
&\qquad \leq \min(n \cdot \varDelta_{\mathsf{live}}, \mathsf{time} - \mathsf{Joined}[\mathsf{uid}]),
\end{aligned}$$

where the last step directly follows from Lemma 4. Rearranging the terms, we have

$$\mathsf{lastHb} - \mathsf{Offset}[\mathsf{uid}] \leq \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}] + \min\bigl(n \cdot \varDelta_{\mathsf{live}}, \mathsf{time} - \mathsf{Joined}[\mathsf{uid}]\bigr),$$

where the left hand side denotes the (global) time that uid thinks that this heartbeat certifies, while the right hand side denotes the actual sending time plus the error term.

The extra $\varDelta_{\mathsf{live}}$ term in Definition 3 accounts for uid waiting that long without receiving heartbeats before dropping out. Moreover, we stress that for each party uid, the game $\mathsf{Sec}^{\text{LL-CGKA}}_{\Pi,\mathcal{A}}$ only enforces liveness as long as all leaders uid encountered so far have been honest, satisfying the respective pre-condition of Lemma 4. □

Together, Lemmas 3 and 5 imply Theorem 7, concluding the proof.

D.4 Correctness

The security game described so far admits protocols in which parties reject all messages and immediately drop out of the meeting. To rule out such trivial protocols, and show that Zoom's protocol does not exhibit this behavior, in the following we discuss some basic correctness conditions.

The correctness game. We formalize correctness using the game depicted in Fig. 15. This game essentially models an honest execution of the LL-CGKA protocol and ensures that parties (1) agree on keys and participants, (2) don't get stuck, i.e., advance whenever they process a message, and importantly (3) do not drop out due to the liveness mechanism.

To model those guarantees, we consider an execution of the protocol that resembles a (simplified) deployment in a client-server setting.

Meeting flow. All communication is routed via the honest server. For instance, to have a new party to join a meeting the adversary can invokes the CreateUser oracle. This then generates a fresh uid and informs the server, which in turns then informs the current meeting leader to add uid to the meeting. This roughly corresponds to how Zoom deploys their protocol; we ignore access policies and advanced features such as locked meetings or the waiting room here. There are several ways in which a party can leave a meeting: A party can deliberately leave (using the Leave oracle) which again informs the server of that request. The same oracle, however, also supports a "silent" mode, modeling a party suddenly dropping off such as when losing internet connection. In the latter case the server is not informed. Rather the server may use the Expel oracle to instruct the leader to remove a party at any point in time, for instance when suspecting the party to be no longer actually online. The meeting host kicking out a party would also correspond to informing the server to expel the user — albeit this is not actually modeled in our correctness game. Similarly, for simplicity we only allow one participant at a time to be added or removed from the meeting, except at the beginning or during a leader change.

Note that for parties there is a joint ProcessOrFollow oracle. Simply put, the oracle will make the party Process a message from their current leader and switch leader using Follow when receiving a message from a different leader. Again, this roughly models, how one would actually deploy a LL-CGKA protocol in practice where the honest server is assumed to only forward messages from the current leader.

Honest server. Our correctness game keeps additional state for the server beyond the protocol's one: ServerLeader keeps track of the leader of each meeting meetingId while ServerGroup keeps track of the current group roster from the viewpoint of the server. Additionally, to deliver signatures sig and the public group information grpPub to the appropriate parties, the server uses ServerNewUser to keep track whether a given uid is fresh (i.e., has not received a LL-CGKA message yet) and ServerNewLeader whether a given leader has been freshly elected (i.e., has not sent a LL-CGKA message yet). Finally, the server keeps track of each party's most up-to-date signature using Sigs.

(Semi-)passive adversary. One of the main differences to the security game is that we assume a (mostly) passive adversary. In particular, we assume the honestly generated messages to be delivered unaltered and in order. (Zoom uses TLS between clients and the server for "control" messages, including all messages related to the key agreement protocol but not the audio and video streams.) In particular, in the correctness game the adversary cannot suppress messages (as the protocol depends on parties receiving all of them to ensure proper functionality). To this end, the game maintains a *message FIFO queue* UserMsgs[uid] for each party (modeling down-link communication between the server and the respective user) as well as a server queue ServerMsgs[meetingId] per meeting, modeling up-link communication. (For simplicity we assume that if different parties of the same meeting send something to the server, they get delivered perfectly in order.) In the correctness game, rather than providing all the arguments for the oracle calls, the adversary then merely instructs a party to process the next message in their queue. For ease of presentation, we however still maintain separate oracles and instead each oracle checks that it only is invoked if the *message type* of the message in front of the queue matches. For instance, the adversary may only invoke the Add oracle, to instruct the leader to add a party, whenever a message of the corresponding type is queued to be delivered to the leader next — the parties to be added as well as their matching signature are then part of the queued messages.

Note how the adversary has access to all the game and the honest parties' states, and can request any long term secret keys using Leak-PKI-Keys. However, we limit the ways in which the adversary can interact with the game: they cannot directly deliver arbitrary messages to parties other than the honest server. The adversary can create corrupted ephemeral identities using CreateMaliciousUser. They can also take over an existing participant's identity to send messages on their behalf using SendMaliciousSig, after which we consider that identity corrupted and stop enforcing their behavior or guaranteeing that they do not drop out. Note that the adversary cannot corrupt the meeting leader, or elect a corrupted party as the leader, and therefore we do not need to add parties to CorrIds. Moreover, the adversary is limited to send only messages of type 'sig' on behalf of malicious parties, as this is the only type of message the honest server would accept from a non-leader. Indeed, a malicious leader could trivially force parties to drop out by sending malformed messages.

While we believe that, for our scheme, correctness is restored whenever a participant starts following an honest leader again without having dropped out, for simplicity this is not reflected in the definition.

Game $\mathsf{Corr}^{\text{LL-CGKA}}_{\Pi,\mathcal{A}}$

Main

```
Procedure: Initialize
  won, lost ← false
  time ← 1
  Corrupted, CorrIds ← ∅
  pub pub ← Π.Init()
  pub St[·], Leader[·, ·], Group[·, ·, ·], Epoch[·, ·],
        Period[·, ·], Key[·, ·, ·], Sigs[·], Offset[·] ← ⊥
  pub ServerNewUser[·], ServerNewLeader[·] ← false
  pub ServerGroup[·] ← ∅
  pub ServerLeader[·], LeaderDrop[·] ← ⊥
  pub UserMsgs[·], ServerMsgs[·] ← ⟨⟩

Procedure: Finalize
  return won ∧ ¬lost
```

Clock

```
Oracle: Tick
  time ← time + 1
  for all meetingId : ServerMsgs[meetingId] ≠ ⟨⟩ do
    // Enforce network bound for server
    parse (time′, . . .) ← ServerMsgs[meetingId].peek()
    if time′ + $\Delta_{network}$ < time then
      lost ← true
  for all uid : St[uid] ≠ ⊥ ∧ uid ∉ Corrupted do
    meetingId ← Π.Meeting(uid)
    // Enforce network bound for users
    parse (time′, . . .) ← UserMsgs[uid].peek()
    if time′ + $\Delta_{network}$ < time then
      lost ← true
    // Enforce bound on leader selection
    if  LeaderDrop[meetingId] ≠ ⊥
      ∧ LeaderDrop[meetingId] + $\Delta_{election}$ < time  then
      lost ← true
    // Make parties tick
    (e, p) ← (Epoch[uid, time − 1], Period[uid, time − 1])
    (Epoch[uid, time], Period[uid, time]) ← (e, p)
    if ∗isLeader(uid, time) then
      (St[uid], M) ← Π.LeaderTick(St[uid], time + Offset[uid])
      assert St[uid] ≠ ⊥
      assert ∗verifyProgress(uid, time, ‘weak′)
      if (Epoch[uid, time], Period[uid, time]) ≠ (e, p) then
        ∗setGroup(uid, time, Group[uid, e, p])
        ∗setKey(uid, time)
        assert ∗verifyConsistency(uid, time)
      if M ≠ ⊥ then
        m ← (M, Epoch[uid, time], Period[uid, time])
        ServerMsgs[meetingId].enq((time, uid, ‘split′, m))
    else
      (St[uid], alive, sig) ← Π.ParticipantTick(St[uid], time + Offset[uid])
      if sig ≠ ⊥ then
        ServerMsgs[meetingId].enq((time, uid, ‘sig′, sig))
      if uid ∈ ServerGroup[meetingId] then
        assert alive ∧ St[uid] ≠ ⊥
        assert ∗verifyLiveness(uid, time)
```

User management

```
Oracle: CreateUser(id, meetingId, offset)
  (ust, uid, sig) ← Π.CreateUser(time + offset, id, meetingId)
  Offset[uid] ← offset
  assert Identity(uid) = id ∧ Meeting(uid) = meetingId
  assert St[uid] = ⊥ ∧ uid ∉ Corrupted
  St[uid] ← ust
  // Require adversary to select leader, if none
  if ServerLeader[meetingId] = ⊥ ∧ LeaderDrop[meetingId] = ⊥ then
    LeaderDrop[meetingId] ← time
  ServerMsgs[meetingId].enq((time, uid, ‘created′, sig)) // Inform server

Oracle: Leave(uid, silent)
  req St[uid] ≠ ⊥
  St[uid] ← ⊥
  meetingId ← Π.Meeting(uid)
  if ServerLeader[meetingId] = uid then
    LeaderDrop[meetingId] ← time
  if ¬silent then // inform server
    ServerMsgs[meetingId].enq((time, uid, ‘left′, ⊥))
```

Malicious participants

```
Oracle: CreateMaliciousUser(uid, sig)
  req St[uid] = ⊥ ∧ uid ∉ Corrupted
  Corrupted ← Corrupted ∪ {uid}
  ServerMsgs[meetingId].enq((time, uid, ‘created′, sig))

Oracle: SendMaliciousSig(uid, sig)
  meetingId ← Π.Meeting(uid)
  req Sigs[uid] ≠ ⊥ ∧ uid ≠ ServerLeader[meetingId]
  Corrupted ← Corrupted ∪ {uid}
  ServerMsgs[meetingId].enq((time, uid, ‘sig′, sig))

Oracle: Leak-PKI-Keys(id)
  (isk, ipk) ← PKI.get-sk(id)
  return (isk, ipk)
```

Group management

```
Oracle: Lead($uid_{lead}$)
  req St[$uid_{lead}$] ≠ ⊥
  parse (time′, ·, type, m′) ← UserMsgs[$uid_{lead}$].deq()
  req type = ‘lead′
  parse (m, grpPub) ← m′
  uids ← {uid | ∃sig : (uid, sig) ∈ m}
  e ← Epoch[$uid_{lead}$, time]
  if Leader[$uid_{lead}$, e] = ⊥ then
    St[$uid_{lead}$] ← Π.CatchUp(St[$uid_{lead}$], time + Offset[$uid_{lead}$], grpPub)
  (ust′, M) ← Π.Lead(St[$uid_{lead}$], time + Offset[$uid_{lead}$], m)
  if (ust′, M) ≠ ⊥ then
    St[$uid_{lead}$] ← ust′
    assert ∗verifyProgress($uid_{lead}$, time, ‘strict′)
    ∗setGroup($uid_{lead}$, time, uids ∪ {$uid_{lead}$})
    ∗setKey($uid_{lead}$, time)
    assert ∗verifyConsistency($uid_{lead}$, time)
    m ← (M, Epoch[$uid_{lead}$, time], Period[$uid_{lead}$, time])
    ServerMsgs[Meeting($uid_{lead}$)].enq((time, $uid_{lead}$, ‘split′, m))
    LeaderDrop[Meeting($uid_{lead}$)] ← ⊥
  else
    assert uids ∩ Corrupted ≠ ∅

Oracle: Add($uid_{lead}$)
  req St[$uid_{lead}$] ≠ ⊥ ∧ ∗isLeader($uid_{lead}$, time)
  parse (time′, ·, type, m) ← UserMsgs[$uid_{lead}$].deq()
  req type = ‘add′
  uids ← {uid | ∃sig : (uid, sig) ∈ m}
  (e, p) ← (Epoch[$uid_{lead}$, time], Period[$uid_{lead}$, time])
  $G$ ← Group[$uid_{lead}$, e, p]
  (ust′, M) ← Π.Add(St[$uid_{lead}$], time + Offset[$uid_{lead}$], m)
  if (ust′, M) ≠ ⊥ then
    St[$uid_{lead}$] ← ust′
    assert ∗verifyProgress($uid_{lead}$, time, ‘strict′)
    ∗setGroup($uid_{lead}$, time, $G$ ∪ uids)
    ∗setKey($uid_{lead}$, time)
    assert ∗verifyConsistency($uid_{lead}$, time)
    m ← (M, Epoch[$uid_{lead}$, time], Period[$uid_{lead}$, time])
    ServerMsgs[Meeting($uid_{lead}$)].enq((time, $uid_{lead}$, ‘split′, m))
  else
    assert uids ∩ Corrupted ≠ ∅

Oracle: Remove($uid_{lead}$)
  req St[$uid_{lead}$] ≠ ⊥ ∧ ∗isLeader($uid_{lead}$, time)
  parse (time′, ·, type, m) ← UserMsgs[$uid_{lead}$].deq()
  req type = ‘remove′
  (e, p) ← (Epoch[$uid_{lead}$, time], Period[$uid_{lead}$, time])
  $G$ ← Group[$uid_{lead}$, e, p]
  (St[$uid_{lead}$], M) ← Π.Remove(St[$uid_{lead}$], time + Offset[$uid_{lead}$], m)
  assert ∗verifyProgress($uid_{lead}$, time, ‘strict′)
  ∗setGroup($uid_{lead}$, time, $G$ \ m)
  ∗setKey($uid_{lead}$, time)
  assert ∗verifyConsistency($uid_{lead}$, time)
  m ← (M, Epoch[$uid_{lead}$, time], Period[$uid_{lead}$, time])
  ServerMsgs[Meeting($uid_{lead}$)].enq((time, $uid_{lead}$, ‘split′, m))
```

```
Participants

Oracle: ProcessOrFollow(uid)
  req $\mathsf{St}[\mathsf{uid}] \neq \bot \wedge \neg *\mathsf{isLeader}(\mathsf{uid}, \mathsf{time})$
  parse $(\mathsf{time}', \mathsf{uid}'_{\mathsf{lead}}, \mathsf{type}, \mathsf{m}') \leftarrow \mathsf{UserMsgs}[\mathsf{uid}].\mathsf{deq}()$
  req $\mathsf{type} = \text{`process'}$
  parse $(\mathsf{m}, \mathsf{sig}'_{\mathsf{lead}}, \mathsf{grpPub}, \mathsf{e}', \mathsf{p}') \leftarrow \mathsf{m}'$
  $\mathsf{e} \leftarrow \mathsf{Epoch}[\mathsf{uid}, \mathsf{time}]$
  if $\mathsf{uid}'_{\mathsf{lead}} = \mathsf{Leader}[\mathsf{uid}, \mathsf{e}]$ then
    $\mathsf{ust}' \leftarrow \Pi.\mathsf{Process}(\mathsf{St}[\mathsf{uid}], \mathsf{time} + \mathsf{Offset}[\mathsf{uid}], \mathsf{m})$
  else
    if $\mathsf{Leader}[\mathsf{uid}, \mathsf{e}] = \bot$ then
      $\mathsf{St}[\mathsf{uid}] \leftarrow \Pi.\mathsf{CatchUp}(\mathsf{St}[\mathsf{uid}], \mathsf{time} + \mathsf{Offset}[\mathsf{uid}], \mathsf{grpPub})$
    $\mathsf{ust}' \leftarrow \Pi.\mathsf{Follow}(\mathsf{St}[\mathsf{uid}], \mathsf{time} + \mathsf{Offset}[\mathsf{uid}], \mathsf{m}, \mathsf{uid}'_{\mathsf{lead}}, \mathsf{sig}'_{\mathsf{lead}})$
  assert $\mathsf{ust}' \neq \bot$ // no error
  $\mathsf{St}[\mathsf{uid}] \leftarrow \mathsf{ust}'$
  assert $*\mathsf{verifyProgress}(\mathsf{uid}, \mathsf{time}, \text{`weak'})$
  assert $(\mathsf{Epoch}[\mathsf{uid}, \mathsf{time}], \mathsf{Period}[\mathsf{uid}, \mathsf{time}]) = (\mathsf{e}', \mathsf{p}')$ // same as leader
  $*\mathsf{setLeader}(\mathsf{uid}, \mathsf{time}, \mathsf{uid}'_{\mathsf{lead}})$
  assert $*\mathsf{verifyConsistency}(\mathsf{uid}, \mathsf{time})$

Server

Oracle: Split(meetingId)
  parse $(\mathsf{time}', \mathsf{uid}_{\mathsf{lead}}, \mathsf{type}, \mathsf{m}) \leftarrow \mathsf{ServerMsgs}[\mathsf{meetingId}].\mathsf{deq}()$
  req $\mathsf{type} = \text{`split'}$
  if $\mathsf{uid}_{\mathsf{lead}} = \mathsf{ServerLeader}[\mathsf{meetingId}]$ then // Ignore old leaders
    parse $(\mathsf{M}, \mathsf{e}, \mathsf{p}) \leftarrow \mathsf{m}$
    $(\mathsf{pub}, \mathsf{ms}) \leftarrow \Pi.\mathsf{Split}(\mathsf{pub}, \mathsf{M})$
    for all $(\mathsf{uid}, \mathsf{m}) \in \mathsf{ms}$ do
      assert $\mathsf{uid} \in \mathsf{ServerGroup}[\mathsf{meetingId}]$
      if $\mathsf{ServerNewUser}[\mathsf{uid}] \vee \mathsf{ServerNewLeader}[\mathsf{meetingId}]$ then
        $\mathsf{sig}_{\mathsf{lead}} \leftarrow \mathsf{Sigs}[\mathsf{uid}_{\mathsf{lead}}]$
      else
        $\mathsf{sig}_{\mathsf{lead}} \leftarrow \bot$
      if $\mathsf{ServerNewUser}[\mathsf{uid}]$ then
        $\mathsf{grpPub} \leftarrow \Pi.\mathsf{GroupState}(\mathsf{pub}, \mathsf{meetingId})$
        $\mathsf{ServerNewUser}[\mathsf{uid}] \leftarrow \mathtt{false}$
      else
        $\mathsf{grpPub} \leftarrow \bot$
      $\mathsf{UserMsgs}[\mathsf{uid}].\mathsf{enq}((\mathsf{time}, \mathsf{uid}_{\mathsf{lead}}, \text{`process'}, (\mathsf{m}, \mathsf{sig}_{\mathsf{lead}}, \mathsf{grpPub}, \mathsf{e}, \mathsf{p})))$
    if $\mathsf{ServerNewLeader}[\mathsf{meetingId}]$ then
      $\mathsf{ServerNewLeader}[\mathsf{meetingId}] \leftarrow \mathtt{false}$

Oracle: InitiateAdd(meetingId)
  $\mathsf{uid}'_{\mathsf{lead}} \leftarrow \mathsf{ServerLeader}[\mathsf{meetingId}]$
  req $\mathsf{uid}'_{\mathsf{lead}} \neq \bot$
  parse $(\mathsf{time}', \mathsf{uid}', \mathsf{type}, \mathsf{sig}) \leftarrow \mathsf{ServerMsgs}[\mathsf{meetingId}].\mathsf{deq}()$
  req $\mathsf{type} = \text{`created'}$
  $\mathsf{Sigs}[\mathsf{uid}'] \leftarrow \mathsf{sig}$
  $\mathsf{ServerNewUser}[\mathsf{uid}'] \leftarrow \mathtt{true}$
  $\mathsf{ServerGroup}[\mathsf{meetingId}] \leftarrow \mathsf{ServerGroup}[\mathsf{meetingId}] \cup \{\mathsf{uid}\}$
  $\mathsf{m} \leftarrow \{\mathsf{uid}', \mathsf{Sigs}[\mathsf{uid}']\}$
  $\mathsf{UserMsgs}[\mathsf{uid}'_{\mathsf{lead}}].\mathsf{enq}((\mathsf{time}, \bot, \text{`add'}, \mathsf{m}))$

Oracle: UpdateSig(meetingId)
  parse $(\mathsf{time}', \mathsf{uid}', \mathsf{type}, \mathsf{sig}) \leftarrow \mathsf{ServerMsgs}[\mathsf{meetingId}].\mathsf{deq}()$
  req $\mathsf{type} = \text{`sig'}$
  $\mathsf{Sigs}[\mathsf{uid}'] \leftarrow \mathsf{sig}$
```

```
Oracle: InitiateRemove(meetingId)
  $\mathsf{uid}'_{\mathsf{lead}} \leftarrow \mathsf{ServerLeader}[\mathsf{meetingId}]$
  req $\mathsf{uid}'_{\mathsf{lead}} \neq \bot$
  parse $(\mathsf{time}', \mathsf{uid}', \mathsf{type}, \mathsf{M}) \leftarrow \mathsf{ServerMsgs}[\mathsf{meetingId}].\mathsf{deq}()$
  req $\mathsf{type} = \text{`left'}$
  $\mathsf{meetingId} \leftarrow \Pi.\mathsf{Meeting}(\mathsf{uid}')$
  $\mathsf{ServerGroup}[\mathsf{meetingId}] \leftarrow \mathsf{ServerGroup}[\mathsf{meetingId}] \setminus \{\mathsf{uid}\}$
  $\mathsf{m} \leftarrow \{\mathsf{uid}', \mathsf{Sigs}[\mathsf{uid}']\}$
  $\mathsf{UserMsgs}[\mathsf{uid}'_{\mathsf{lead}}].\mathsf{enq}((\mathsf{time}, \bot, \text{`remove'}, \mathsf{m}))$

Oracle: Expel(uid')
  // Like Leave but server initiated
  $\mathsf{meetingId} \leftarrow \Pi.\mathsf{Meeting}(\mathsf{uid}')$
  $\mathsf{ServerMsgs}[\mathsf{meetingId}].\mathsf{enq}((\mathsf{time}, \mathsf{uid}', \text{`left'}, \bot))$

Oracle: ElectLeader(uid'_lead)
  $\mathsf{meetingId} \leftarrow \Pi.\mathsf{Meeting}(\mathsf{uid}')$
  $\mathsf{uid}_{\mathsf{lead}} \leftarrow \mathsf{ServerLeader}[\mathsf{meetingId}]$
  req $\mathsf{uid}'_{\mathsf{lead}} \neq \mathsf{uid}_{\mathsf{lead}} \wedge \mathsf{St}[\mathsf{uid}'_{\mathsf{lead}}] \neq \bot \wedge \mathsf{uid}'_{\mathsf{lead}} \notin \mathsf{Corrupted}$
  // Process pending changes (w/o informing ceding leader)
  while $\mathsf{ServerMsgs}[\mathsf{meetingId}] \neq \langle\rangle$ do
    parse $(\mathsf{time}', \mathsf{uid}', \mathsf{type}, \mathsf{m}) \leftarrow \mathsf{ServerMsgs}[\mathsf{meetingId}].\mathsf{peek}()$
    if $\mathsf{type} = \text{`split'}$ then
      $\mathsf{Split}(\mathsf{meetingId})$
    else if $\mathsf{type} = \text{`sig'}$ then
      $\mathsf{UpdateSig}(\mathsf{meetingId})$
    else if $\mathsf{type} = \text{`created'}$ then
      $\mathsf{ServerMsgs}[\mathsf{meetingId}].\mathsf{deq}()$
      $\mathsf{Sigs}[\mathsf{uid}'] \leftarrow \mathsf{m}$
      $\mathsf{ServerNewUser}[\mathsf{uid}'] \leftarrow \mathtt{true}$
      $\mathsf{ServerGroup}[\mathsf{meetingId}] \leftarrow \mathsf{ServerGroup}[\mathsf{meetingId}] \cup \{\mathsf{uid}'\}$
    else if $\mathsf{type} = \text{`left'}$ then
      $\mathsf{ServerMsgs}[\mathsf{meetingId}].\mathsf{deq}()$
      $\mathsf{ServerGroup}[\mathsf{meetingId}] \leftarrow \mathsf{ServerGroup}[\mathsf{meetingId}] \setminus \{\mathsf{uid}'\}$
  // Detect race conditions
  if $\mathsf{UserMsgs}[\mathsf{uid}_{\mathsf{lead}}] \neq \langle\rangle$ then
    // Old leader has pending changes to roster
    // This will cause diverging, inconsistent, state
    // Thus, treat old leader as removed
    $\mathsf{ServerGroup}[\mathsf{meetingId}] \leftarrow \mathsf{ServerGroup}[\mathsf{meetingId}] \setminus \{\mathsf{uid}_{\mathsf{lead}}\}$
  // Inform new leader
  req $\mathsf{uid}'_{\mathsf{lead}} \in \mathsf{ServerGroup}[\mathsf{meetingId}]$
  $\mathsf{m} \leftarrow \{(\mathsf{uid}, \mathsf{Sigs}[\mathsf{uid}]) \mid \mathsf{uid} \in \mathsf{ServerGroup}[\mathsf{meetingId}] \wedge \mathsf{uid} \neq \mathsf{uid}'_{\mathsf{lead}}\}$
  if $\mathsf{ServerNewUser}[\mathsf{uid}'_{\mathsf{lead}}]$ then
    $\mathsf{grpPub} \leftarrow \Pi.\mathsf{GroupState}(\mathsf{pub}, \mathsf{meetingId})$
    $\mathsf{ServerNewUser}[\mathsf{uid}'_{\mathsf{lead}}] \leftarrow \mathtt{false}$
  else
    $\mathsf{grpPub} \leftarrow \bot$
  $\mathsf{UserMsgs}[\mathsf{uid}'_{\mathsf{lead}}].\mathsf{enq}((\mathsf{time}, \bot, \text{`lead'}, (\mathsf{m}, \mathsf{grpPub})))$
  $\mathsf{ServerLeader}[\mathsf{meetingId}] \leftarrow \mathsf{uid}'_{\mathsf{lead}}$
  $\mathsf{ServerNewLeader}[\mathsf{meetingId}] \leftarrow \mathtt{true}$
```

Fig. 15: The correctness game for the LL-CGKA notion. The helper algorithms are the same as in Fig. 13. We define an LL-CGKA scheme to be correct if no PPT $\mathcal{A}$ can win this game with better than negligible probability.

Bounded delay network. Given that liveness demands that parties do drop out if the adversary tries to withhold messages, we need to assume bounded network delay for the protocol to work. In the following, let $\Delta_{\mathsf{network}}$ denote a bound on the network delay. This is enforced in the game as follows: If at any point in time the any of the message queues contains a message that has been sitting there for too long (w.l.o.g., the first one to be delivered) then the adversary loses the game. That is, we only consider adversaries to be "valid" in the interaction that do not violate the network bound.

Leader election. Similarly, we also bound the time it takes the server to elect a new leader using $\Delta_{\mathsf{election}}$. Otherwise, if the server would not elect a leader for a long time, all participants would just drop out. Note that this is measured from the time the leader drops out in the Leave oracle to the time the next leader receives the instruction to take over. Further, this bound must also be adhered when the old leader *silently* drops out, for instance after suddenly losing internet connection. Hence, $\Delta_{\mathsf{election}}$ must be chosen such that the server actually can detect and correct such a situation — we discuss the the relationships between these various parameters below. Let us provide a few more comments on the leader election process. First, observe that from the viewpoint of the server there is always an active leader, stored as $\mathsf{ServerLeader}[\mathsf{meetingId}]$, except when a meeting is just about to start. The game, on the other hand, sets $\mathsf{LeaderDrop}[\mathsf{meetingId}]$ to the current timestamp as soon as the old leader drops out, even if the server is not notified. This field is then cleared out as part of $\mathsf{uid}_{\mathsf{lead}}$ whenever a new leader successfully takes over, and the oracle Tick enforces that this process takes no longer than $\Delta_{\mathsf{election}}$.

Moreover, observe that the ElectLeader oracle either allows to just switch the leader or to additionally process an addition or removal message first. The latter mode differs from separately processing said message in not informing the old leader. In the case of an addition, this can be used to make a new participant immediately the new leader (e.g., to support the join-before-host feature of Zoom meetings) whereas the case of a removal for instance models the old leader being the party to be removed. Finally, notice that race conditions may occur when switching over to a new leader. For instance, if the server suspects the old leader to have lost connection it must elect a new one, even if messages from the old leader might later still arrive. In such a case the old leader proceeded to a state which is now incompatible with the one distributed by the new leader, which continued the meeting at the latest state the server was still aware of. As a result, the server treats the old leader as having dropped out if such a race condition occurs.[14]

Correctness assurances. The game checks various correctness properties. In particular, the game enforces the following properties not covered by the security notion:

- *No dropping out:* The game enforces that parties do not drop out unless they intentionally leave or are kicked out. This is formalized by checking the respective return value of ParticipantTick whenever a party is still supposed to be in the group.
- *No errors:* While the LL-CGKA syntax defines most operations as fallible, the correctness notion ensures than in case of an honest execution algorithms do not reject. More concretely, whenever the leader is instructed to perform an operation, such as adding or removing a party, then they must succeed and produce a LL-CGKA message. Two exceptions apply: Upon being instructed to take over as a leader with a given group the leader is allowed to reject if the group contains corrupted parties. Similarly, upon being instructed to add additional parties, the leader may reject if any of those parties are corrupted. Participants, on the other hand, must never reject a message generated by the honest leader even if other malicious participants are involved — either the leader already rejected or the meeting must progress as expected.
- *Correct state:* When a participant obtains a LL-CGKA message, processing said message must result in the participant transitioning to the same epoch and period as the leader. If the leader after sending a LL-CGKA message is in epoch e and period p, all participants upon processing their respective message share must transitioning to the same epoch and period. To this end, the correctness game annotates LL-CGKA messages with their respective epoch and period and then enforces the property as part of ProcessOrFollow. Note that annotating LL-CGKA messages is done for bookkeeping by the game only, e and p are not actually sent (separately) in a protocol execution.

In addition, the correctness game also enforces various properties that directly mirror the ones from the security game:

- *Consistency:* Analogous to the security game $*$verifyConsistency checks that all parties in the same state, i.e. parties with the same leader, epoch and period, agree on the key, the group roster as well as the meeting's history. In combination with ensuring that parties do end up in the expected epoch and period, this ensures that parties do agree on all relevant state.

[14] In the game, this is done by directly inspecting the network queues. In a real implementation this would of course need to be implemented by keeping track of group-change requests sent to a leader without receiving a corresponding LL-CGKA message.

- *Progress:* The game assures in $*\mathsf{verifyProgress}$ that upon each action the leader moves to the next state, i.e., epoch or period.
- *Liveness:* The correctness game also checks liveness, i.e., the fact that parties at all times are in a state for which their leader has been in recently.

Theorem 8. *The LL-CGKA scheme from Figs. 3 and 14 satisfies correctness under the following constraints:*

(1) $\Delta_{\mathsf{live}} \geq 4 \cdot \Delta_{\mathsf{network}} + \max(\Delta_{\mathsf{LPL}}, \Delta_{\mathsf{election}})$*;*
(2) $\Delta_{\mathsf{live}} \geq \Delta_{\mathsf{heartbeat}} + 2 \cdot \Delta_{\mathsf{network}}$*;*
(3) $\Delta_{\mathsf{live}} \geq \Delta_{\mathsf{heartbeat}} + \Delta_{\mathsf{election}} + 2 \cdot \Delta_{\mathsf{network}}$*;*
(4) $\Delta_{\mathsf{LPL}} \leq \Delta_{\mathsf{heartbeat}}$*.*

In particular, when modeling a server who considers a leader to have dropped out and elects a new one immediately after $\mathsf{M} \geq 1$ *missed heartbeats, then we can set*

$$\Delta_{\mathsf{election}} = 2 \cdot \Delta_{\mathsf{network}} + \mathsf{M} \cdot \Delta_{\mathsf{heartbeat}}$$

(where one $\Delta_{\mathsf{network}}$ *is the time it takes for the server to get heartbeats and the other the time it takes to inform the new leader) and obtain correctness if the parameters satisfy*

$$\Delta_{\mathsf{live}} \geq 6 \cdot \Delta_{\mathsf{network}} + (\mathsf{M} + 1) \cdot \Delta_{\mathsf{heartbeat}}.$$

Proof (Sketch). We note that the consistency properties follow by inspection of the scheme and correctness of the underlying primitives. In the following, we give a brief argument on the no-dropout property.

First, we observe that condition (1) implies that parties will join a meeting successfully as long as the meeting's current leader remains active long enough to process their request. It takes up to $2 \cdot \Delta_{\mathsf{network}}$ for the server to receive the join request from the new participant uid and deliver it to the current leader, which will immediately generate the respective cmKEM message for the new party but might take up to Δ_{LPL} for the next refresh of the LPL and heartbeat (in case the last LPL message has just been sent before receiving the request). It then takes up to another $2 \cdot \Delta_{\mathsf{network}}$ to deliver the message from the leader to the joining participant, at which point the participant will delay dropping out by an additional Δ_{live}. If the leader drops out before processing the request, we know that at most $\Delta_{\mathsf{election}}$ later a new leader will have been elected (and informed). Since the leader drops out no later than $2 \cdot \Delta_{\mathsf{network}}$ after the new participant created their uid, in order for uid to not be admitted, we know that no later than $2 \cdot \Delta_{\mathsf{network}} + \Delta_{\mathsf{election}}$ after uid got created a new leader takes over and immediately generates cmKEM message, LPL message, and heartbeat for the current group that does include uid. This is then received by uid after at most an additional $2 \cdot \Delta_{\mathsf{network}}$. Hence, (1) ensures that uid successfully joins even in that case. Finally, if uid is elected to be the new leader, this happens no later than $2 \cdot \Delta_{\mathsf{network}} + \Delta_{\mathsf{election}}$ after uid got created.

Next, consider a party who has accepted at least a heartbeat in the meeting. We observe that if a leader sends a heartbeat message at *global* time t that is still accepted by a party, then this party will not drop out before global time $t + \Delta_{\mathsf{live}}$, as $\delta[\mathsf{uid}_{\mathsf{lead}}]$ is an upper bound on the drift between a party and their leader, which makes lastHb an upper bound on the (local) time when the heartbeat was sent.

Now assume that the party uid still accepted a heartbeat sent at global time t. If the adversary does not switch leaders, the same leader will send the following heartbeat at time $t' \leq t + \Delta_{\mathsf{heartbeat}}$ (where the smaller-equal follows from $\Delta_{\mathsf{LPL}} \leq \Delta_{\mathsf{heartbeat}}$). This new heartbeat will be delivered no later than global time $t' + 2 \cdot \Delta_{\mathsf{network}} \leq t + \Delta_{\mathsf{heartbeat}} + 2 \cdot \Delta_{\mathsf{network}} \leq t + \Delta_{\mathsf{live}}$ by condition (2), and therefore it will be accepted as well.

Finally, consider a leader switch. Assume that the old leader sent their last heartbeat at global time t, before leaving at time $t + \Delta_{\mathsf{heartbeat}}$, i.e., right before being supposed to generate the next heartbeat. By assumption, at global time $t' \leq t + \Delta_{\mathsf{heartbeat}} + \Delta_{\mathsf{election}}$ the new leader will have been notified, which immediately generates a cmKEM message as well as LPL and heartbeat messages. By the time $t + \Delta_{\mathsf{heartbeat}} + \Delta_{\mathsf{election}} + 2\Delta_{\mathsf{network}}$ participants will have received those messages and, thus, condition (3) ensures that they still did not drop out and will prolong liveness until at least global time $t' + \Delta_{\mathsf{live}}$. □

E Details on Improved Liveness

E.1 Additional Interaction: Proof of Theorem 3

Recall that our proposal improves the liveness properties twofold. First, the liveness slack no longer degrades in the number of leader changes. Second, liveness now holds even if a *past leader* has been corrupted or malicious as long as the current leader is honest, which is formalized by altering the $*\mathsf{verifyLiveness}$ helper in the LL-CGKA security game, as detailed in Fig. 16.

```
Game Sec$^{\text{LL-CGKA}}_{\Pi,\mathcal{A}}$ Helpers

Helper: $*\mathsf{verifyLiveness}(\mathsf{uid}, \mathsf{time})$
  $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{Epoch}[\mathsf{uid}, \mathsf{time}], \mathsf{Period}[\mathsf{uid}, \mathsf{time}])$
  $\mathsf{uid}_{\mathsf{lead}} \leftarrow \mathsf{Leader}[\mathsf{uid}, \mathsf{e}]$
  // No assurance with malicious leader
  if $\mathsf{Identity}(\mathsf{uid}_{\mathsf{lead}}) \in \mathsf{CorrIds}$ then
      return true
  // Was leader recently here?
  $\Delta \leftarrow *\texttt{liveness-slack}(\mathsf{uid}, \mathsf{time}, \mathsf{Leader}, \mathsf{Joined}, \mathsf{Offset})$
  for $\mathsf{time}' \leftarrow \mathsf{time} - \Delta, \ldots, \mathsf{time}$ do
      if $(\mathsf{e} = \mathsf{Epoch}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}']$
                  $\wedge\ \mathsf{p} \geq \mathsf{Period}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}'])$
              $\vee\ \mathsf{e} > \mathsf{Epoch}[\mathsf{uid}_{\mathsf{lead}}, \mathsf{time}']$ then
          return true
  return false
```

Fig. 16: The strengthened liveness properties compared to Fig. 12.

Theorem 9 (Theorem 3 restated). *The modified LL-CGKA scheme from Fig. 5 is secure according to Fig. 12 with the modified* $*\mathsf{verifyLiveness}$ *depicted in Fig. 16 and the following liveness slack*

$$*\textit{liveness-slack}(\mathsf{uid}, \mathsf{time}, \mathsf{Leader}, \mathsf{Joined}, \mathsf{Offset}) := \min\bigl(2 \cdot \Delta_{\mathsf{nonce}} + \Delta_{\mathsf{live}}, \mathsf{time} - \mathsf{Joined}[\mathsf{uid}]\bigr) + \Delta_{\mathsf{live}},$$

if the underlying cmKEM scheme is secure, the signature scheme is EUF-CMA secure, and the hash function is collision resistant. Additionally, it is secure with the original (weaker) $*\mathsf{verifyLiveness}$ *from Fig. 12 with*

$$*\textit{liveness-slack}(\mathsf{uid}, \mathsf{time}, \mathsf{Leader}, \mathsf{Joined}, \mathsf{Offset}) := \min\bigl(2 \cdot \Delta_{\mathsf{nonce}} + \Delta_{\mathsf{live}}, \mathsf{time} - \mathsf{Joined}[\mathsf{uid}], n \cdot \Delta_{\mathsf{live}}\bigr) + \Delta_{\mathsf{live}},$$

which is at most the slack of the current protocol.

It is easy to see that the scheme only modifies the liveness properties, i.e., all other properties such as key confidentiality, key consistency, and authenticity directly translate over from Theorem 7. The existing liveness term

$$*\texttt{liveness-slack}(\mathsf{uid}, \mathsf{time}, \mathsf{Leader}, \mathsf{Joined}, \mathsf{Offset}) := \min\bigl(\mathsf{time} - \mathsf{Joined}[\mathsf{uid}], n \cdot \Delta_{\mathsf{live}}\bigr) + \Delta_{\mathsf{live}},$$

in case all past leaders have been honest, also carry over from the proof of Lemma 4. Moreover, the modified $*\mathsf{verifyLiveness}$ depicted in Fig. 16 is strictly stronger — i.e., ensures liveness in a broader set of circumstances — implying that the we can focus on that version of $*\mathsf{verifyLiveness}$ for the additional Δ_{nonce} term. We, thus, observe that it suffices to strengthen Lemma 4 as follows.

Lemma 6 (Adaptation of Lemma 4). *Consider an execution of the modified LL-CGKA scheme from Fig. 5 within* $\mathsf{Sec}_{\Pi,\mathcal{A}}^{LL\text{-}CGKA}$ *with the modified* $*\mathsf{verifyLiveness}$ *Fig. 16. Then, whenever* $*\mathsf{verifyLiveness}$ *is not trivially disabled and for each user* uid*, we have*

$$\begin{aligned}\delta_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] - (\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]) \\ \leq \min\bigl(2 \cdot \varDelta_{\mathsf{nonce}} + \varDelta_{\mathsf{live}}, \mathsf{time} - \mathsf{Joined}[\mathsf{uid}]\bigr)\end{aligned}$$

where $\delta_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}]$ *refers to* uid*'s protocol state (i.e., their estimates on the drift to* $\mathsf{uid}_{\mathsf{lead}}$*) while* $\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]$ *refers to the game state (i.e., the actual drift).*

Proof. Recall from the proof of Lemma 4 that it suffices to consider the moments right after a party processed a heartbeat message. Moreover, recall that if the user uid received the last heartbeat at local time $\mathsf{time}_{\mathsf{uid}} = \mathsf{time} + \mathsf{Offset}[\mathsf{uid}]$ and this heartbeat contained a timestamp, i.e., the sending time, $\mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} = \mathsf{time}' + \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]$, then after invoking `*update-drift` we have

$$\delta_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] - (\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]) \leq \mathsf{time} - \mathsf{time}'.$$

First, we observe that the proof of the $\mathsf{time} - \mathsf{Joined}[\mathsf{uid}]$ — i.e., the fact that a party is never off by more than the duration they spent in a meeting — follows analogous to the current protocol. Indeed, we observe that the respective proof didn't rely on leaders being honest at all (and would even hold against a malicious current leader).

We now prove the term $2 \cdot \varDelta_{\mathsf{nonce}} + \varDelta_{\mathsf{live}}$ on the bound. First, consider the case where uid receives a first heartbeat from a new leader. Since the party will drop out unless they get this heartbeat within $\varDelta_{\mathsf{live}}$, and this heartbeat must have been generated after the party generated their identity, we can conclude (analogously as in Lemma 4) that the term can be bounded by $\varDelta_{\mathsf{live}}$. We henceforth only consider the case of a leader switch.

The protocol enforces that upon a leader switch the first heartbeat of the new leader (and the LPL message) is delivered alongside the first cmKEM message. For the nonce included in the cmKEM message being at most that old and unpredictable, we have that the cmKEM message can have been at most $2\varDelta_{\mathsf{nonce}}$ old whenever uid at this point in time. Formally, this holds due to the authenticity of the associated data ad, which is set to the nonce, being enforced as part of the $*\mathsf{verifyConsistency}$ condition in the cmKEM security game from Fig. 10. (Crucially, this property holds irrespective whether uid previously interacted with a malicious leader or not.) Since the protocol verifies that the heartbeat message certifies the cmKEM message's epoch and period, and the new leader is assumed to be honest, we can conclude that this heartbeat is indeed the one the new leader generated at the same time as the cmKEM message. Therefore, we conclude that the drift estimate when processing the first heartbeat processed from an honest leader (except when joining the meeting) is at most $2 \cdot \varDelta_{\mathsf{nonce}}$ off.

To keep the equations simpler, we bound the error after processing the first heartbeat from a new leader with the sum of the two terms, $2 \cdot \varDelta_{\mathsf{nonce}} + \varDelta_{\mathsf{live}}$, instead of their maximum.

For subsequent heartbeats of the same leader, the same argument as in the proof Lemma 4 applies to establish that liveness only improves. □

E.2 Additional Interaction: Correctness

Our modified protocol introduces nonces sent by participants that a new leader has to acknowledge to establish liveness upon a leader change. More concretely, the protocol accepts any of the two most recent nonces. It remains to show that this does not adversely affect correctness as long as the nonce generation interval $\varDelta_{\mathsf{nonce}}$ is chosen to be not too small, i.e., as long as the two most recent nonces stay relevant enough for the information to disseminate.

Theorem 10. *The modified LL-CGKA scheme from Fig. 5, satisfies correctness if* $\varDelta_{\mathsf{nonce}} \geq 4 \cdot \varDelta_{\mathsf{network}}$ *and* $2 \cdot \varDelta_{\mathsf{nonce}} \geq 4 \cdot \varDelta_{\mathsf{network}} + \varDelta_{\mathsf{LPL}}$ *in addition to the constraints of the base protocol. That is, it satisfies correctness according to the game from Fig. 15 if:*

(1) $\mathit{\Delta}_{\mathsf{live}} \geq 4 \cdot \mathit{\Delta}_{\mathsf{network}} + \max(\mathit{\Delta}_{\mathsf{LPL}}, \mathit{\Delta}_{\mathsf{election}})$*;*
(2) $\mathit{\Delta}_{\mathsf{live}} \geq \mathit{\Delta}_{\mathsf{heartbeat}} + 2 \cdot \mathit{\Delta}_{\mathsf{network}}$*;*
(3) $\mathit{\Delta}_{\mathsf{live}} \geq \mathit{\Delta}_{\mathsf{heartbeat}} + \mathit{\Delta}_{\mathsf{election}} + 2 \cdot \mathit{\Delta}_{\mathsf{network}}$*;*
(4) $\mathit{\Delta}_{\mathsf{LPL}} \leq \mathit{\Delta}_{\mathsf{heartbeat}}$*;*
(5) $\mathit{\Delta}_{\mathsf{nonce}} \geq 4 \cdot \mathit{\Delta}_{\mathsf{network}}$*;*

In particular, when choosing $\mathit{\Delta}_{\mathsf{nonce}} \geq \mathit{\Delta}_{\mathsf{LPL}}$ *then constraint* (5) *implies constraint* (6).

Proof (Sketch). We observe that the modified protocol only introduces the additional nonce freshness checks. Hence, it suffices to argue that the above conditions are sufficient for those checks not to fail. First, consider the case of a leader change. We observe that at the time $\mathsf{ElectLeader}(\mathsf{uid}'_{\mathsf{lead}})$ the nonce the server knows for a particular participant uid is at most $\mathit{\Delta}_{\mathsf{nonce}} + \mathit{\Delta}_{\mathsf{network}}$ old, which models the case of the server just missing the next newer nonce that is still in transit. From the time $\mathsf{ElectLeader}(\mathsf{uid}'_{\mathsf{lead}})$ is called, it then takes at most $3 \cdot \mathit{\Delta}_{\mathsf{network}}$ for uid to receive the first message from $\mathsf{uid}'_{\mathsf{lead}}$, with messages sent from the server to $\mathsf{uid}'_{\mathsf{lead}}$, back to the server, and finally to uid. Hence, at the time uid receives the first message from $\mathsf{uid}'_{\mathsf{lead}}$ the included nonce is at most $\mathit{\Delta}_{\mathsf{nonce}} + 4 \cdot \mathit{\Delta}_{\mathsf{network}}$ old. Since uid accepts the two most recent nonces — and hence as long as it is less than $2 \cdot \mathit{\Delta}_{\mathsf{nonce}}$ old — condition (5) implies that uid does not reject the nonce.

Second, consider the case of uid freshly joining the meeting. In that case, uid distributes their nonce as part of their initial message to the server which is then handed to the current leader $\mathsf{uid}_{\mathsf{lead}}$ at most $2 \cdot \mathit{\Delta}_{\mathsf{network}}$ after the nonce has been created. The leader will immediately sent the cmKEM message back, acknowledging the nonce within time $4 \cdot \mathit{\Delta}_{\mathsf{network}} + \mathit{\Delta}_{\mathsf{LPL}}$. (Note that while it may take up to $\mathit{\Delta}_{\mathsf{LPL}}$ longer for the LPL and heartbeat to be received, the freshness of the nonce is evaluated at the time the cmKEM message arrives.) Therefore, uid and accepts the nonce as long as $2 \cdot \mathit{\Delta}_{\mathsf{nonce}} \geq 4 \cdot \mathit{\Delta}_{\mathsf{network}}$, which is implied by (5). The case of a leader change during uid joining is covered by the above argument of a regular leader change. □

E.3 Leveraging Synchronicity: Proof of Theorem 4

Theorem 11 (Theorem 4 restated). *The modified LL-CGKA scheme from Fig. 7 is secure according to Fig. 12 with the following improved liveness slack*

$$\begin{aligned}&{*}\mathtt{liveness\text{-}slack}(\mathsf{uid}, \mathsf{time}, \mathsf{Leader}, \mathsf{Joined}, \mathsf{Offset})\\ &\qquad := \min\big(|\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]|, n \cdot \mathit{\Delta}_{\mathsf{live}}, \mathsf{time} - \mathsf{Joined}[\mathsf{uid}]\big) + \mathit{\Delta}_{\mathsf{live}},\end{aligned}$$

if the underlying cmKEM scheme is secure, the signature scheme is EUF-CMA secure, and the hash function is collision resistant.

Again, it is easy to see that the scheme only modifies the liveness properties, i.e., all other properties such as key confidentiality, key consistency, and authenticity directly translate over from Theorem 7. We, thus, only consider liveness.

We start by establishing the lemma, stating that the protocol maintains proper bounds on the clock drift. In the remainder of this section, we use the following notational convention: For protocol variables we use subscripts to denote the respective party. For example, we use $\mathsf{lastHb}_{\mathsf{uid}}$ to refer to the variable lastHb as maintained by the protocol of party uid. For time related variables we moreover disambiguate local versus global clocks using those subscripts: Global times (as used by the security game) are denoted without subscript — e.g., time refers to the current global time of the security game — whereas $\mathsf{time}_{\mathsf{uid}} = \mathsf{time} + \mathsf{Offset}[\mathsf{uid}]$ refers to the respective local time as used in the protocol by party uid. (Note that uid's protocol maintains all variables with respect to their local time, i.e., any time related variable with subscript uid can always be understood with respect to uid's local clock. Variables like $\mathsf{elapsed}$, which are computed by a client but represent an amount of time and not a specific instant relative to their clock, can be added to both local and global times interchangeably.)

Lemma 7. *Consider an execution of the LL-CGKA protocol from Fig. 7 within* $\mathsf{Sec}_{\Pi,\mathcal{A}}^{LL\text{-}CGKA}$*, with a PPT* $\mathcal{A}$*. Then, for each user* uid *that so far only encountered honest leaders, we have*

$$\delta_{\mathsf{uid}}^{\mathsf{min}}[\mathsf{uid}_{\mathsf{lead}}] \leq \mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}] \leq \delta_{\mathsf{uid}}^{\mathsf{max}}[\mathsf{uid}_{\mathsf{lead}}],$$

where the first and last terms refer to uid*'s protocol state (i.e., their estimates on the offset to* $\mathsf{uid}_{\mathsf{lead}}$*) while the middle term refers to the game state (i.e., the actual offset).*

Proof. We prove this statement using induction over all invocations of `*update-drift`. To start, observe that the protocol initializes $\delta_{\mathsf{uid}}^{\mathsf{min}}[\mathsf{uid}_{\mathsf{lead}}]$ and $\delta_{\mathsf{uid}}^{\mathsf{max}}[\mathsf{uid}_{\mathsf{lead}}]$ to $-\infty$ and $+\infty$, respectively. Hence, the invariant holds initially, and we only need to show that it is preserved by the `*update-drift` procedure from Fig. 7.

Now, consider the case that uid at global time time receives a heartbeat from an honest leader $\mathsf{uid}_{\mathsf{lead}}$ with timestamp $\mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}}$. Assuming unforgeability of signatures, $\mathsf{uid}_{\mathsf{lead}}$ actually sent that heartbeat at global time $\mathsf{time}' = \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} + \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}] \leq \mathsf{time}$. If $\delta_{\mathsf{uid}}^{\mathsf{max}}[\mathsf{uid}_{\mathsf{lead}}]$ gets updated in that execution of `*update-drift`, at local time $\mathsf{time}_{\mathsf{uid}} = \mathsf{time} + \mathsf{Offset}[\mathsf{uid}]$, it gets set to

$$\begin{aligned}\delta_{\mathsf{uid}}^{\mathsf{max}}[\mathsf{uid}_{\mathsf{lead}}] \leftarrow \mathsf{time}_{\mathsf{uid}} - \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} &= (\mathsf{time} + \mathsf{Offset}[\mathsf{uid}]) - (\mathsf{time}' + \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]) \\ &= \mathsf{time} - \mathsf{time}' + \mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}] \\ &\geq \mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}],\end{aligned}$$

implying the second part of the inequality being preserved.

Analogously, if $\delta_{\mathsf{uid}}^{\mathsf{min}}[\mathsf{uid}_{\mathsf{lead}}]$ is updated in `*update-drift`, it is set to $\mathsf{earliest}_{\mathsf{uid}} - \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}}$. Therefore, to show that the lower bound is preserved throughout the execution, it is enough to establish that the following invariant is preserved:

$$\mathsf{earliest}_{\mathsf{uid}} - \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} \leq \mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}],$$

where $\mathsf{earliest}_{\mathsf{uid}}$ denotes the value uid computes in `*update-drift`. If this is uid's first heartbeat, i.e., $\mathsf{lastHb}_{\mathsf{uid}}^{\mathsf{min}} = \bot$, then we have that $\mathsf{earliest}_{\mathsf{uid}} = \mathsf{lastHb}_{\mathsf{uid}} = \mathsf{Joined}[\mathsf{uid}] + \mathsf{Offset}[\mathsf{uid}]$, i.e., the (local) time at which the ephemeral identity has been created. Since the heartbeat signs over a LPL message that must contain this ephemeral identity we know that $\mathsf{Joined}[\mathsf{uid}] \leq \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]$, implying the claim. Otherwise, if $\mathsf{lastHb}_{\mathsf{uid}}^{\mathsf{min}} \neq \bot$, consider first the case where the previous heartbeat has been sent by the same (honest) leader $\mathsf{uid}_{\mathsf{lead}}$. Then we have that this previous heartbeat contained the timestamp $\mathsf{time}''_{\mathsf{uid}_{\mathsf{lead}}} = \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} - \mathsf{elapsed}'$ (where $\mathsf{elapsed}'$ denotes the value contained in the current heartbeat), and upon receiving it uid set $\mathsf{lastHb}_{\mathsf{uid}}^{\mathsf{min}} = \mathsf{time}''_{\mathsf{uid}_{\mathsf{lead}}} + \delta_{\mathsf{uid}}^{\mathsf{min}}[\mathsf{uid}_{\mathsf{lead}}]$. Therefore, when computing `*update-drift` for the current heartbeat we have that

$$\begin{aligned}\mathsf{earliest}_{\mathsf{uid}} &= \mathsf{lastHb}_{\mathsf{uid}}^{\mathsf{min}} + \mathsf{elapsed}' \\ &= \mathsf{time}''_{\mathsf{uid}_{\mathsf{lead}}} + \delta_{\mathsf{uid}}^{\mathsf{min}}[\mathsf{uid}_{\mathsf{lead}}] + \mathsf{elapsed}' \\ &= \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} + \delta_{\mathsf{uid}}^{\mathsf{min}}[\mathsf{uid}_{\mathsf{lead}}] \\ &\leq \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} + \mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}],\end{aligned}$$

where in the last step we used our assumption that in all prior `*update-drift` invocations the invariant from the lemma statement has been preserved.

Finally, consider the case of the heartbeat being from a new leader, with $\mathsf{uid}''_{\mathsf{lead}}$ being the prior one. We now adapt the above argument. To this end, consider the last heartbeat of said leader sent at local time $\mathsf{time}''_{\mathsf{uid}''_{\mathsf{lead}}}$, before $\mathsf{uid}_{\mathsf{lead}}$ taking over. The new leader $\mathsf{uid}_{\mathsf{lead}}$ will set $\mathsf{elapsed}' = \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} - \mathsf{lastHb}_{\mathsf{uid}_{\mathsf{lead}}}^{\mathsf{max}}$. If $\mathsf{uid}_{\mathsf{lead}}$ has been part of the meeting before, then $\mathsf{lastHb}_{\mathsf{uid}_{\mathsf{lead}}}^{\mathsf{max}}$ has been set by $\mathsf{uid}_{\mathsf{lead}}$ when receiving the heartbeat with timestamp $\mathsf{time}''_{\mathsf{uid}''_{\mathsf{lead}}}$, for which we have

$$\mathsf{lastHb}_{\mathsf{uid}_{\mathsf{lead}}}^{\mathsf{max}} = \mathsf{time}''_{\mathsf{uid}''_{\mathsf{lead}}} + \delta_{\mathsf{uid}_{\mathsf{lead}}}^{\mathsf{max}}[\mathsf{uid}''_{\mathsf{lead}}] \geq \mathsf{time}''_{\mathsf{uid}''_{\mathsf{lead}}} + \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}] - \mathsf{Offset}[\mathsf{uid}''_{\mathsf{lead}}],$$

where we used the first inequality of the lemma. If $\mathsf{uid}_{\mathsf{lead}}$ has not been part of the meeting before, $\mathsf{lastHb}_{\mathsf{uid}_{\mathsf{lead}}}^{\mathsf{max}}$ has been set by $\mathsf{uid}_{\mathsf{lead}}$ to the current local time during $\mathsf{CatchUp}$. For uid to accept the new leader's heartbeat, that leader must have gotten the correct heartbeat as part of $\mathsf{CatchUp}$, which includes the old leader's signature (even if $\mathsf{uid}_{\mathsf{lead}}$ does not verify that signature). Hence, we can conclude that $\mathsf{CatchUp}$ happened after the previous leader's last heartbeat and hence the same lower bound on $\mathsf{lastHb}^{\mathsf{max}}_{\mathsf{uid}_{\mathsf{lead}}}$ applies. We can derive

$$\mathsf{elapsed}' \leq (\mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]) - (\mathsf{time}''_{\mathsf{uid}''_{\mathsf{lead}}} - \mathsf{Offset}[\mathsf{uid}''_{\mathsf{lead}}]),$$

i.e., that $\mathsf{elapsed}'$ is a lower bound on the actually elapsed time between the two heartbeats, and thus

$$\begin{aligned} \mathsf{earliest}_{\mathsf{uid}} &= \mathsf{lastHb}^{\mathsf{min}}_{\mathsf{uid}} + \mathsf{elapsed}' \\ &= \mathsf{time}''_{\mathsf{uid}''_{\mathsf{lead}}} + \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}''_{\mathsf{lead}}] + \mathsf{elapsed}' \\ &\leq \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} + \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}''_{\mathsf{lead}}] + \mathsf{Offset}[\mathsf{uid}''_{\mathsf{lead}}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}] \\ &\leq \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}} + \mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}], \end{aligned}$$

where in the last step we used our assumption of $\delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}''_{\mathsf{lead}}] \leq \mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}''_{\mathsf{lead}}]$. □

Next, we show that the following bound on the bounds difference. (Unsurprisingly, this resulting difference resembles the liveness assurance of Zoom's original protocol.)

Lemma 8. *For any stage of an execution of the LL-CGKA protocol Π from Fig. 3 within* $\mathsf{Sec}^{LL\text{-}CGKA}_{\Pi,\mathcal{A}}$, *we have that*

$$\delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] - \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] \leq \min(n \cdot \Delta_{\mathsf{live}}, \mathsf{time} - \mathsf{Joined}[\mathsf{uid}])$$

whenever $*\mathsf{verifyLiveness}$ *is not trivially disabled and where n is defined as in Theorem 4.*

Proof. We again prove this by induction over the invocations of `*update-drift`, i.e., we consider one particular invocation and assume that so far the invariant has been maintained.

First, consider the case where uid receives their very first heartbeat, with timestamp $\mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}}$. Due to the liveness mechanism making uid wait at most Δ_{live} to join, we know that this must occur at some global time $\mathsf{time} \leq \mathsf{Joined}[\mathsf{uid}] + \Delta_{\mathsf{live}}$, and therefore in this case proving the bound for the second term in the minimum implies it holds for the first. Moreover, we know that $\mathsf{earliest}_{\mathsf{uid}} = \mathsf{Joined}[\mathsf{uid}] + \mathsf{Offset}[\mathsf{uid}]$. Observe that $\delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}]$ gets set to $\mathsf{time} + \mathsf{Offset}[\mathsf{uid}] - \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}}$ and $\delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}]$ to $\mathsf{earliest}_{\mathsf{uid}} - \mathsf{time}'_{\mathsf{uid}_{\mathsf{lead}}}$, implying the claim.

Second, if the heartbeat is from a leader from which uid already received a previous one, then the invariant is trivially preserved as $\delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}]$ only decreases, while $\delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}]$ and the terms on the right side only increase.

Third, consider a heartbeat from a new leader $\mathsf{uid}_{\mathsf{lead}}$, when the previous leader[15] $\mathsf{uid}''_{\mathsf{lead}}$ sent the last heartbeat at time $\mathsf{time}''_{\mathsf{uid}''_{\mathsf{lead}}}$. Then, `*update-drift` sets the bounds such that

$$\delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] - \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] = \mathsf{time} + \mathsf{Offset}[\mathsf{uid}] - \mathsf{earliest}_{\mathsf{uid}} \leq \mathsf{time} + \mathsf{Offset}[\mathsf{uid}] - \mathsf{time}''_{\mathsf{uid}''_{\mathsf{lead}}} - \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}''_{\mathsf{lead}}], \quad (4)$$

by using that $\mathsf{elapsed}' \geq 0$ (which can be deduced by inspection) to bound $\mathsf{earliest}_{\mathsf{uid}}$. Moreover, since uid is still processing this heartbeat and hasn't dropped out already, we know that

$$\mathsf{lastHb}_{\mathsf{uid}} + \Delta_{\mathsf{live}} \geq \mathsf{time} + \mathsf{Offset}[\mathsf{uid}]$$

and we know (the proof is the same as the one that $\delta^* \leq \delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}]$ in Lemma 9) that

$$\mathsf{lastHb}_{\mathsf{uid}} - \mathsf{time}''_{\mathsf{uid}''_{\mathsf{lead}}} \leq \delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}''_{\mathsf{lead}}].$$

Hence, we can conclude by combining the three above inequalities that

$$\delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] - \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] \leq \delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}''_{\mathsf{lead}}] - \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}''_{\mathsf{lead}}] + \Delta_{\mathsf{live}},$$

[15] Here we assume $\mathsf{uid}''_{\mathsf{lead}} \neq \mathsf{uid}$, but the case where the participant itself was the previous leader can be treated analogously and leads to stronger guarantees as it resets liveness.

and so if $\mathsf{uid}''_{\mathsf{lead}}$ was the $(n-1)$-th leader by our assumption we obtain the respective bound for the first term of the minimum. For the second term, consider that

$$\mathsf{earliest}_{\mathsf{uid}} \geq \mathsf{Joined}[\mathsf{uid}] + \mathsf{Offset}[\mathsf{uid}]. \tag{5}$$

We have seen above that this holds when processing the very first heartbeat, and the inequality can be extended to further heartbeats by showing that $\mathsf{earliest}_{\mathsf{uid}}$ only increases over time. In particular when processing each heartbeat

$$\mathsf{earliest}_{\mathsf{uid}} \leftarrow \mathsf{elapsed}' + \mathsf{lastHb}^{\mathsf{min}} = \mathsf{elapsed}' + \mathsf{time}' + \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}''_{\mathsf{lead}}] \geq \mathsf{elapsed}' + \mathsf{time}' + \mathsf{earliest}'_{\mathsf{uid}} - \mathsf{time}' \geq \mathsf{earliest}'_{\mathsf{uid}}$$

where $\mathsf{earliest}'_{\mathsf{uid}}$ was the value of the variable before the last heartbeat was processed. Combining Eqs. (4) and (5) gives the desired bound. □

We now use the above two lemma to bound the error on the estimated timestamps, i.e., the maximal error a participant uid makes when estimating the time a certain heartbeat has been sent.

Lemma 9. *Assume an honest leader* $\mathsf{uid}_{\mathsf{lead}}$ *sends a heartbeat at global time* time'*, i.e., with timestamp* $\mathsf{time}' + \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]$*, which gets successfully processed by a participant* uid *at global time* time*. After executing* `*update-liveness`*, we have*

$$\left|\mathsf{time}' - (\mathsf{lastHb}_{\mathsf{uid}} - \mathsf{Offset}[\mathsf{uid}])\right| \leq \min\!\big(|\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]|, n \cdot \mathit{\Delta}_{\mathsf{live}}, \mathsf{time} - \mathsf{Joined}[\mathsf{uid}]\big),$$

i.e., the difference between the actual sending time time' *and the estimated sending time* $\mathsf{lastHb}_{\mathsf{uid}} - \mathsf{Offset}[\mathsf{uid}]$ *by* uid *(converted into global time) is bounded by the minimum over the given three terms.*

Proof. We first show the inequality holds for the second and third terms of the minimum using Lemma 8. Let

$$\delta^* := \mathsf{lastHb}_{\mathsf{uid}} - (\mathsf{time}' + \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}])$$

denote the correction factor applied by uid in `*update-liveness`. Substituting this into the left-hand side yields

$$\begin{aligned} &\left|\mathsf{time}' - (\mathsf{lastHb}_{\mathsf{uid}} - \mathsf{Offset}[\mathsf{uid}])\right| \\ &\quad = \left|\mathsf{time}' - (\delta^* + \mathsf{time}' + \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}] - \mathsf{Offset}[\mathsf{uid}])\right| \\ &\quad = |\delta^* + \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}] - \mathsf{Offset}[\mathsf{uid}]| \\ &\quad = |\delta^* - (\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}])|. \end{aligned} \tag{6}$$

By inspection of `*update-liveness` we observe that δ^* can be written as

$$\delta^* = \begin{cases} \max(0, \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}]), & \text{if } \delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] \geq 0 \\ \delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}], & \text{if } \delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] < 0 \end{cases} \tag{7}$$

and applying Lemma 7 lets us deduce that $\delta^* \leq \delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}]$ in either case. Therefore, we obtain

$$\begin{aligned} &\delta^* - (\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]) \\ &\quad \leq \delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] - (\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]) \\ &\quad \leq \delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] - \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] \\ &\quad \leq \min\!\big(n \cdot \mathit{\Delta}_{\mathsf{live}}, \mathsf{time} - \mathsf{Joined}[\mathsf{uid}]\big), \end{aligned}$$

where we used Lemma 7 in the second step and Lemma 8 in the last step. Analogously we can write δ^* as

$$\delta^* = \begin{cases} \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}], & \text{if } \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] \geq 0 \\ \min(0, \delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}]), & \text{if } \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] < 0 \end{cases} \tag{8}$$

yielding $\delta^* \geq \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}]$. As a result, we furthermore obtain

$$\begin{aligned}\delta^* &- (\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]) \\ &\geq \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] - (\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]) \\ &\geq \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] - \delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] \\ &\geq -\min\bigl(n \cdot \mathit{\Delta}_{\mathsf{live}}, \mathsf{time} - \mathsf{Joined}[\mathsf{uid}]\bigr),\end{aligned}$$

concluding the proof of the second and third term.

Finally, consider the first term of the bound. First, we observe that in case $\delta^* = 0$, the bound is directly implied by Eq. (6).

If $\delta^* < 0$ we have

$$0 > \delta^* \overset{(7)}{=} \delta^{\mathsf{max}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] \overset{\text{Lem. 7}}{\geq} \mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]$$

and therefore subtracting $(\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}])$ we obtain

$$-(\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]) > \delta^* - (\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]) \geq 0$$

which immediately gives the bound by taking the absolute value.

If $\delta^* > 0$ we have

$$0 < \delta^* \overset{(8)}{=} \delta^{\mathsf{min}}_{\mathsf{uid}}[\mathsf{uid}_{\mathsf{lead}}] \overset{\text{Lem. 7}}{\leq} \mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]$$

from which we can analogously obtain the same bound. □

It remains to show that this implies the desired liveness properties, as expressed in the following lemma. (Recall that all other properties have already been proven.)

Lemma 10. *In the following, consider the game that behaves like* $\mathsf{Sec}^{\mathit{LL\text{-}CGKA}}_{\Pi,\mathcal{A}}$*, with* `*liveness-slack` *as defined in Theorem 4, but where the winning condition is modified to only account for* $*\mathsf{verifyLiveness}$*. Then the advantage of any PPT adversary* $\mathcal{A}$ *in winning that game is negligible.*

Proof. Assume that $\mathsf{uid}_{\mathsf{lead}}$ sent the last heartbeat uid successfully processed at global time time'. Then, it suffices to show

$$\mathsf{lastHb}_{\mathsf{uid}} - \mathsf{Offset}[\mathsf{uid}] \leq \mathsf{time}' + \min\bigl(|\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]|, n \cdot \mathit{\Delta}_{\mathsf{live}}, \mathsf{time} - \mathsf{Joined}[\mathsf{uid}]\bigr),$$

where the left hand side denotes the (global) time that uid thinks that this heartbeat certifies, while the right hand side denotes the actual sending time plus the error term. The slack then simply results from uid waiting for $\mathit{\Delta}_{\mathsf{live}}$ until dropping out. The result then follows directly from Lemma 9. □

Together, Lemmas 3 and 10 imply Theorem 4, concluding the proof.

E.4 Leveraging Synchronicity: Correctness

We now formalize correctness of the enhanced scheme. As discussed in Section 4, the scheme's improved liveness assurances come at the cost of degraded correctness. In the following, we show that correctness still holds as long as clocks are synchronized, or the drift is small — we conjecture correctness to hold in additional settings such as small network delay and carefully managed leader changes, but do not make any corresponding formal statement. More formally, we show the following modified correctness theorem where conditions (2) and (3) account for the need of synchronized clocks.

Theorem 12. *The modified LL-CGKA scheme from Fig. 7 satisfies correctness under the following constraints:*

(1) $\mathit{\Delta}_{\mathsf{live}} \geq 4 \cdot \mathit{\Delta}_{\mathsf{network}} + \max(\mathit{\Delta}_{\mathsf{LPL}}, \mathit{\Delta}_{\mathsf{election}})$*;*

(2) $\mathit{\Delta}_{\mathsf{live}} \geq \mathit{\Delta}_{\mathsf{heartbeat}} + 2 \cdot \mathit{\Delta}_{\mathsf{network}} + |\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]|$*;*
(3) $\mathit{\Delta}_{\mathsf{live}} \geq \mathit{\Delta}_{\mathsf{heartbeat}} + \mathit{\Delta}_{\mathsf{election}} + 2 \cdot \mathit{\Delta}_{\mathsf{network}} + |\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]|$*;*
(4) $\mathit{\Delta}_{\mathsf{LPL}} \leq \mathit{\Delta}_{\mathsf{heartbeat}}$*.*

The first and last inequality only depend on protocol parameters, while the middle ones bound the clock offset between any meeting participant and any of their meeting leaders over an execution of the protocol. This can be formalized as a restriction on the adversary's behavior in the correctness experiment.

Proof (Sketch). We remark that the only change with respect to the original protocol is the more stringent liveness check. Hence, we focus exclusively on liveness in the following.

First, we observe that with respect to initially joining the meeting the enhanced protocol remains unchanged to the base protocol: The first heartbeat needs to arrive within $\mathit{\Delta}_{\mathsf{live}}$ from the time the participant's ephemeral identity uid has been created. As a result, condition (1) still ensures that parties will join a meeting successfully.

Next, consider a party who has accepted at least a heartbeat in the meeting. We observe that if a leader $\mathsf{uid}_{\mathsf{lead}}$ sends a heartbeat message at *global* time time' that is still accepted by a party uid at time time, then uid delay dropping out until global time

$$\mathsf{lastHb}_{\mathsf{uid}} - \mathsf{Offset}[\mathsf{uid}] + \mathit{\Delta}_{\mathsf{live}},$$

for $\mathsf{lastHb}_{\mathsf{uid}}$ as computed during `*update-liveness`. Using Lemma 9 we can thus conclude that uid will not drop out before

$$\begin{aligned}&\mathsf{lastHb}_{\mathsf{uid}} - \mathsf{Offset}[\mathsf{uid}] + \mathit{\Delta}_{\mathsf{live}}\\ &\quad \geq \mathsf{time}' - \min\bigl(|\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]|, n \cdot \mathit{\Delta}_{\mathsf{live}}, \mathsf{time} - \mathsf{Joined}[\mathsf{uid}]\bigr) + \mathit{\Delta}_{\mathsf{live}}.\end{aligned}$$

If the adversary does not switch leaders, the same leader will send the following heartbeat at global time $\mathsf{time}'' \leq \mathsf{time}' + \mathit{\Delta}_{\mathsf{heartbeat}}$ (where the smaller-equal follows from $\mathit{\Delta}_{\mathsf{LPL}} \leq \mathit{\Delta}_{\mathsf{heartbeat}}$). This new heartbeat will be delivered no later than global time

$$\begin{aligned}&\mathsf{time}'' + 2 \cdot \mathit{\Delta}_{\mathsf{network}}\\ &\quad \leq \mathsf{time}' + \mathit{\Delta}_{\mathsf{heartbeat}} + 2 \cdot \mathit{\Delta}_{\mathsf{network}}\\ &\quad \leq \mathsf{time}' + \mathit{\Delta}_{\mathsf{live}} - \min\bigl(|\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]|, n \cdot \mathit{\Delta}_{\mathsf{live}}, \mathsf{time} - \mathsf{Joined}[\mathsf{uid}]\bigr),\end{aligned}$$

by condition (2), and therefore it will be accepted as well.

Finally, consider a leader switch. Assume that the old leader sent their last heartbeat at global time time', before leaving at time $\mathsf{time}' + \mathit{\Delta}_{\mathsf{heartbeat}}$, i.e., right before being supposed to generate the next heartbeat. By assumption, at global time $\mathsf{time}'' \leq \mathsf{time}' + \mathit{\Delta}_{\mathsf{heartbeat}} + \mathit{\Delta}_{\mathsf{election}}$ the new leader will have been notified, which immediately generates a cmKEM message as well as LPL and heartbeat messages. Let time denote the time at which the participant uid obtains those messages, where clearly $\mathsf{time} \leq \mathsf{time}' + \mathit{\Delta}_{\mathsf{heartbeat}} + \mathit{\Delta}_{\mathsf{election}} + 2\mathit{\Delta}_{\mathsf{network}}$. Using condition (3) and Lemma 9 we can conclude

$$\begin{aligned}&\mathsf{time}' + \mathit{\Delta}_{\mathsf{heartbeat}} + \mathit{\Delta}_{\mathsf{election}} + 2\mathit{\Delta}_{\mathsf{network}}\\ &\quad \leq \mathsf{time}' - \min\bigl(|\mathsf{Offset}[\mathsf{uid}] - \mathsf{Offset}[\mathsf{uid}_{\mathsf{lead}}]|, n \cdot \mathit{\Delta}_{\mathsf{live}}, \mathsf{time} - \mathsf{Joined}[\mathsf{uid}]\bigr) + \mathit{\Delta}_{\mathsf{live}}\\ &\quad \leq \mathsf{lastHb}_{\mathsf{uid}} - \mathsf{Offset}[\mathsf{uid}] + \mathit{\Delta}_{\mathsf{live}},\end{aligned}$$

where the last term denotes the time uid would drop out of the meeting. Hence, uid will not have dropped out and still accept the message from the new leader. □