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 have since been deployed in production.

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.

⋆ Research conducted while contracting for Zoom

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 [46] and rolled out basic E2EE support to the public in October 2020 [32]. 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 [34] 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)

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

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.
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 deployed one of them in production (in version 5.13 of the Zoom meetings 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, however, 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 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 the first option, which is deployed in version 5.13 of the Zoom meetings 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 [7,16,11,43,44] offering E2EE group calls, and others offering this feature only for two-party calls [27,34]. While some of the solutions do offer intuitive security descriptions in the form of a whitepaper, such as Wire [44], Cisco [16], and WhatsApp [43], 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 [33] (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 [39] — 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., [37,23,30]) use timing assumptions to improve efficiency (or overcome impossibility results) for problems which do not inherently require timing assumptions. Other works (e.g., [22,40,9,12]) 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., [26,36]). 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 [41].

Related Notions The cmKEM notion is an extension of multi-recipient Key Encapsulation (mKEM) [42,38,45] 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., [35], and line of work on Logical Key Hierarchies, e.g., [15].

Another related notion to both cmKEM and LL-CGKAis 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.

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

[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.

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 assume that the leader is told whom to add or remove, ignoring policy aspects.

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 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* ust *is assumed to expose the current key* $\mathsf{ust.k}$*, epoch* $\mathsf{ust.e}$*, and period* $\mathsf{ust.p}$.

User management:

– $(\mathsf{ust}, \mathsf{uid}, \mathsf{sig}) \leftarrow \mathsf{CreateUser}(\mathsf{id}, \mathsf{meetingId})$ *creates an ephemeral user belonging to* id *and the meeting* $\mathsf{meetingId}$*. It outputs the initial state* ust*, the user's identity* uid*, and credentials* 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.*

[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 those concepts can be decoupled.

Session management:

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

Group and management (leader):

- $(\mathsf{ust}', \mathsf{M}) \leftarrow \mathsf{Add}\big(\mathsf{ust}, \{(\mathsf{uid}_i, \mathsf{ad}_i, \mathsf{sig}_i)\}_{i \in [n]}, \mathsf{newEpoch}\big)$ *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.*
- $(\mathsf{ust}', \mathsf{M}) \leftarrow \mathsf{Remove}(\mathsf{ust}, \{\mathsf{uid}_i\}_{i \in [n]})$ *removes the users* uid_1 *to* uid_n *from the group.*

Message processing (non-leaders):

- $\mathsf{ust}' \leftarrow \mathsf{Process}(\mathsf{ust}, \mathsf{m})$ *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.*

2.2 PKI

The ephemeral user id's uid are bound to the long-term identity id via the credentials. 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, whose analysis is left for future work. We refer to the full version of this work 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 the full version of this work.

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. (The protocol might of course reject such 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. (This would have to be done by instructing another party to assume the role of the leader, excluding 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 it is not subject to the same restrictions on compatible corruptions described below.

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 sessions' 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 other, due to those properties differing. Participants must only move to an epoch once their leader 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 of the 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 — i.e., does not leverage any efficiency gains from sending the same message to multiple recipients — to communicate fresh keys to the participants. It is based around Diffie-Hellman key

[5] I.e., similar to 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.

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, and 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 $\mathsf{usk} \coloneqq \mathsf{DLog}_g(\mathsf{upk})$ 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 $\mathsf{newEpoch} = \texttt{true}$, the leader communicates the seed to all participants. More efficiently, however, when adding participants with $\mathsf{newEpoch} = \texttt{false}$, 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 scheme first derives for each recipient a shared symmetric key from a Diffie-Hellman element of its own secret key usk and the recipient's public key upk'. The scheme uses HKDF for this derivation, which for the purpose of the security analysis we model as an 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 nonce-based AEAD, for a random nonce that is transmitted as part of the resulting ciphertext. The associated data contains the meeting and sender identifiers, and a fixed context string.

Server protocol. The 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. For simplicity, we model that the message $\mathsf{M} = (G, \mathsf{C})$ sent to the server includes the current set of recipients. More concretely, each user uid' for which C contains a share obtains $\mathsf{m} = (\text{`}\mathsf{epoch}', \mathsf{C}[\mathsf{uid}'])$, while for other users the server delivers $\mathsf{m} = \text{`}\mathsf{period}'$.

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 the full version of this work. In short, based on the security of Gap Diffie Hellman, we can first switch to an 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

Protocol cmKEM Client Protocol

User management

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

Algorithm: Meeting(uid)
  parse (meetingId, id, ipk, upk) ← uid
  return meetingId

Algorithm: Identity(uid)
  parse (meetingId, id, ipk, upk) ← uid
  return (id, ipk)
```

Session management

```
Algorithm: StartSession($\{(\mathsf{uid}_i, \mathsf{ad}_i, \mathsf{sig}_i)\}_{i \in [n]}$)
  $G$ ← $\{\mathsf{uid}_1, \ldots, \mathsf{uid}_n\}$
  req me ≠ ⊥ ∧ me ∉ $G$
  AD[·] ← ⊥
  for $i \in [n]$ do
    req *verify-user($\mathsf{uid}_i$, $\mathsf{sig}_i$)
    AD[$\mathsf{uid}_i$] ← $\mathsf{ad}_i$
  $\mathsf{uid}_{\mathsf{lead}}$ ← me
  if e ≠ ⊥ then e ← e + 1
  else e ← 1
  p ← 0 ; seed ← PRG.Init($1^\kappa$)
  C ← *encrypt-seed($G$, AD)
  M ← ($G$, C)
  (seed, k) ← PRG.Eval(seed)
  return M

Algorithm: JoinSession($\mathsf{uid}'_{\mathsf{lead}}$, $\mathsf{sig}'_{\mathsf{lead}}$, m, ad)
  req me ≠ ⊥ ∧ $\mathsf{uid}'_{\mathsf{lead}}$ ≠ me ∧ $\mathsf{uid}'_{\mathsf{lead}}$ ≠ $\mathsf{uid}_{\mathsf{lead}}$
  req *verify-user($\mathsf{uid}'_{\mathsf{lead}}$, $\mathsf{sig}'_{\mathsf{lead}}$)
  $\mathsf{uid}_{\mathsf{lead}}$ ← $\mathsf{uid}'_{\mathsf{lead}}$
  parse ('epoch', c) ← m
  (e', p, seed') ← *decrypt-seed(c, $\mathsf{uid}'_{\mathsf{lead}}$, ad)
  if e ≠ ⊥ then
    req (e', p') = (e + 1, 0)
    e ← e'
  (seed, k) ← PRG.Eval(seed')
```

Group and key management (leader)

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

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

Message processing (participants)

```
Algorithm: Process(m)
  req me ≠ ⊥ ∧ $\mathsf{uid}_{\mathsf{lead}}$ ≠ ⊥ ∧ $\mathsf{uid}_{\mathsf{lead}}$ ≠ me
  if m = ('epoch', c) then
    (e', p', seed') ← *decrypt-seed(c, $\mathsf{uid}_{\mathsf{lead}}$, ⊥)
    req (e', p') = (e + 1, 0)
    (e, p) ← (e', p')
    (seed, k) ← PRG.Eval(seed')
  else if m = 'period' then
    p ← p + 1
    (seed, k) ← PRG.Eval(seed)
```

```
Helper: *encrypt-seed($G'$, AD)
  C[·] ← ⊥
  for uid' ∈ $G'$ do
    parse (·, ·, ·, upk') ← uid'
    nonce ←$ AEAD.$\mathcal{N}$
    if $K$[uid'] = ⊥ then
      $K$[uid'] ← HKDF($\mathsf{upk}'^{\mathsf{usk}}$, 'KeyMeetingSeed')
    ad' ← (meetingId, me, AD[uid'])
    ad'' ← Hash('EncryptionKeyMeetingSeed'
                  ∥ Hash(ad'))
    c' ← AEAD.Enc($K$[uid'], nonce, (e, p, seed), ad'')
    C[uid'] ← (c', nonce)
  return C
```

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

Helper: *verify-user(uid', sig')
  parse (meetingId', id', ipk', upk') ← uid'
  return meetingId' = meetingId
   ∧ Sig.Verify(ipk', 'EncryptionKeyAnnouncement', uid', sig')
   ∧ PKI.verify-pk(id', ipk')
```

Fig. 1: The client protocol of Zoom's cmKEM scheme. The protocol implicitly maintains a state ust, which exposes the key ust.k, epoch ust.e, and period ust.p.

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].

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*, which all 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, limiting any practical clock drift significantly and, thus, justifying the former 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*

[7] Technically, Curve25519 breaks the abstraction of cyclic groups we, for simplicity, use for the presentation of 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.

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 credentials* 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*.*

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*.*
- $(\mathsf{alive}, \mathsf{sig}') \leftarrow \mathsf{ParticipantTick}(\mathsf{ust}, \mathsf{time})$ *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 credentials (for the server) with* $\mathsf{sig}' = \perp$ *denoting no update.*

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

Server's algorithms:

– $\mathsf{pub} \leftarrow \mathsf{Init}()$ *generates an initial server state.*
– $(\mathsf{pub}', \{(\mathsf{uid}_i, \mathsf{m}_i)\}_{i\in[n]}) \leftarrow \mathsf{Split}(\mathsf{pub}, \mathsf{M})$ *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 the full version of this work.

Meeting Flow Let us briefly discuss how Zoom uses the above defined LL-CGKA abstraction to orchestrate a meeting. To start a meeting, Zoom instructs the initial host to invoke the Lead algorithm. For a participant to join the meeting, the server first hands them the most recent public group state (using $\mathsf{GroupState}$) that the participant processes using $\mathsf{CatchUp}$. Afterwards, the participant is instructed of the leader (using Follow) where alongside it is given their respective message share from the message the leader generated in the respective Add invocation. Observe that at this point the participant might not have a usable symmetric key yet. Instead, it might take up to the next message generated by $\mathsf{LeaderTick}$ for the participant to fully join the meeting.

To switch leaders, the new one is instructed to invoke Lead and all other participants are instructed to invoke Follow. Note that it is not required for the new leader to have joined the meeting beforehand — the $\mathsf{CatchUp}$ 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 the full version of this work 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. Obviously 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 $\mathsf{LeaderTick}$ or $\mathsf{ParticipantTick}$ 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 — then 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 to the one of a cmKEM scheme. That is, upon a challenge, the security game outputs, depending on a bit b, either the real or an independent uniform random key. 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. A malicious server can 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, the 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 broadcasting the group membership to the participants and regularly broadcast so-called heartbeat messages for liveness. A formal description is presented in Fig. 2. Additional details can be found in the full version of this work.

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 for has been certified by an LPL message. (Re-keying is nevertheless done eagerly, potentially leading to unused keys.)

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

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

[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.

Protocol Zoom's Client LL-CGKA

User management

```
Algorithm: CreateUser(time, id, meetingId)
    $(\mathsf{st}, \mathsf{me}, \mathsf{sig}) \leftarrow \mathrm{cmKEM}.\mathsf{CreateUser}(\mathsf{id}, \mathsf{meetingId})$
    $(\mathsf{isk}, \cdot) \leftarrow \mathsf{PKI.get\text{-}sk}(\mathsf{id})$
    $\mathsf{lastHb} \leftarrow \mathsf{time}$
    $\mathsf{uid}_{\mathsf{lead}} \leftarrow \perp$
    $G \leftarrow \varnothing$
    $\mathsf{e}, \mathsf{p}, \mathsf{e}_{\mathsf{next}}, \mathsf{p}_{\mathsf{next}}, \mathsf{v}, \mathsf{t} \leftarrow 0$
    $\mathsf{k}[\cdot, \cdot] \leftarrow \perp$
    $\delta[\cdot] \leftarrow \infty$
    $\mathsf{lplHash}, \mathsf{hbHash} \leftarrow \perp$
    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(ust, time, grpPub)
    req $\mathsf{uid}_{\mathsf{lead}} = \perp$
    parse $(\mathsf{lpls}', \mathsf{hb}') \leftarrow \mathsf{grpPub}$
    // 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)
    $\mathsf{hbHash} \leftarrow \mathsf{Hash}(\mathsf{hb}')$
```

Participants' algorithms

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

Algorithm: Process(time, $\mathsf{m}'$)
    req $\mathsf{uid}_{\mathsf{lead}} \neq \perp \land \mathsf{uid}_{\mathsf{lead}} \neq \mathsf{me}$
    parse $(\mathsf{m}'_K, \mathsf{lpl}', \mathsf{hb}') \leftarrow \mathsf{m}'$
    if $\mathsf{m}'_K \neq \perp$ 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 $\mathsf{lpl}' \neq \perp$ then
        try *receive-LPL($\mathsf{lpl}'$)
    if $\mathsf{hb}' \neq \perp$ then
        try *receive-heartbeat($\mathsf{hb}'$)
```

Leader's algorithms

```
Algorithm: Lead(time, $\{(\mathsf{uid}_i, \mathsf{sig}_i)\}_{i\in[n]}$)
    $\mathsf{uid}_{\mathsf{lead}} \leftarrow \mathsf{me}$
    try $(\mathsf{st}, \mathsf{M}_K)$
            $\leftarrow \mathrm{cmKEM}.\mathsf{StartSession}(\mathsf{st}, \{(\mathsf{uid}_i, \perp, \mathsf{sig}_i)\}_{i\in[n]})$
    $G \leftarrow \{\mathsf{uid}_1, \ldots, \mathsf{uid}_n\} \cup \{\mathsf{me}\}$
    $\mathsf{Added}, \mathsf{Removed} \leftarrow \varnothing$
    $\mathsf{numLplLinks} \leftarrow \mathsf{max\text{-}links}$
    $\mathsf{lpl} \leftarrow$ *send-LPL()
    $\mathsf{hb} \leftarrow$ *send-heartbeat()
    $\mathsf{M} \leftarrow (\mathsf{me}, \mathsf{M}_K, \mathsf{lpl}, \mathsf{hb})$
    return 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 $\mathsf{uid}_i \notin G$
    $G \leftarrow G \cup \{\mathsf{uid}_1, \ldots, \mathsf{uid}_n\}$
    $\mathsf{Added} \leftarrow \mathsf{Added} \cup \{\mathsf{uid}_1, \ldots, \mathsf{uid}_n\}$
    if $\mathsf{p} \geq \mathsf{p}_{\mathsf{MAX}}$ then
        try $(\mathsf{st}, \mathsf{M}_K) \leftarrow$
                $\mathrm{cmKEM}.\mathsf{Add}(\mathsf{st}, \{(\mathsf{uid}_i, \perp, \mathsf{sig}_i)\}_{i\in[n]}, \mathtt{true})$
        $(\mathsf{e}, \mathsf{p}) \leftarrow (\mathsf{st.e}, \mathsf{st.p})$
    else
        try $(\mathsf{st}, \mathsf{M}_K) \leftarrow$
                $\mathrm{cmKEM}.\mathsf{Add}(\mathsf{st}, \{(\mathsf{uid}_i, \perp, \mathsf{sig}_i)\}_{i\in[n]}, \mathtt{false})$
    return $(\mathsf{me}, \mathsf{M}_K, \perp, \perp)$

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

Time driven

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

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

Fig. 2: The client part of Zoom's overall LL-CGKA scheme. The description implicitly keeps the state ust.

Protocol Zoom's Client LL-CGKA Helpers

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

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

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

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

Helper: *update-drift(time')
  $\delta[uid_{lead}] \leftarrow \min(\delta[uid_{lead}], time - time')$

Helper: *update-liveness(time')
  $lastHb \leftarrow time' + \delta[uid_{lead}]$
```

Fig. 3: Helper algorithms for the client part of Zoom's overall LL-CGKA scheme.

as well, for this to go unnoticed. As a countermeasure, participants drop out from the meeting if they do not receive a heartbeat message for too long.

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 estimate on the clock drift (between the participant and their respective leader) and Δ_{live} denotes a protocol parameter. In a best effort to prevent this from happening, the server will elect a new leader whenever the current one struggles to upload heartbeats.

The protocol estimates 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 matches the expected one.

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 the full version of this work for details.

Security Security is summarized in our main result below, with a more detailed proof given in the full version of this work.

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 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, $\mathsf{p_{MAX}} = 0$, 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 but 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 shared obtained by split are handed to the parties to execute Follow without any delay. (To 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 at time $t_2 \geq t + \Delta_{\mathsf{live}}$ (which is delivered to all parties as previously described) and immediately afterwards the

[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.

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 the 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 a 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_A, thinking that the clock of P_A 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 of each participant. This proposal has been implemented in the Zoom meeting client since version 5.13.

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
  $uid_{lead}$ ← ⊥
  $G$ ← ∅
  e, p, $e_{next}$, $p_{next}$, v, t ← 0
  k[·, ·] ← ⊥
  δ[·] ← ∞
  lplHash, hbHash ← ⊥
  return (me, sig)
```

Participants' algorithms

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

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

Leader's algorithms

```
Algorithm: Lead(time, $\{(uid_i, sig_i)\}_{i \in [n]}$)
  $uid_{lead}$ ← me
  for $i \in [n]$ do
      parse ($sig'_i$, $nonce_i$) ← $sig_i$
  try (st, $M_K$) ← cmKEM.StartSession(st,
          $\{(uid_i, \boxed{nonce_i, sig'_i})\}_{i \in [n]}$)
  $G$ ← {$uid_1, \ldots, uid_n$} ∪ {me}
  Added, Removed ← ∅
  numLplLinks ← max-links
  lpl ← *send-LPL()
  hb ← *send-heartbeat()
  M ← (me, $M_K$, lpl, hb)
  return M

Algorithm: Add(time, $\{(uid_i, sig_i)\}_{i \in [n]}$)
  req $uid_{lead}$ = me
  for $i \in [n]$ do
      req $uid_i \notin G$
      parse ($sig'_i$, $nonce_i$) ← $sig_i$
  $G$ ← $G$ ∪ {$uid_1, \ldots, uid_n$}
  Added ← Added ∪ {$uid_1, \ldots, uid_n$}
  if p ≥ $p_{MAX}$ then
      try (st, $M_K$) ← cmKEM.Add(st,
              $\{(uid_i, \boxed{nonce_i, sig'_i})\}_{i \in [n]}$, true)
      (e, p) ← (st.e, st.p)
  else
      try (st, $M_K$) ← cmKEM.Add(st,
              $\{(uid_i, \boxed{nonce_i, sig'_i})\}_{i \in [n]}$, false)
  return (me, $M_K$, ⊥, ⊥)
```

Fig. 4: The proposed changes with respect to Zoom's scheme from Fig. 2.

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) at regular intervals. These nonces are seen as part of a party's credential and hence ParticipantTick outputs new credentials whenever the nonce is update. (In practice one would of course only upload the nonce, not the entire credentials, each time.) For simplicity, we choose the same parameter Δ_{live} as for the overall liveness slack.

Whenever a new leader is elected, they get each participants latest nonce from the server. We encode this as part of the credentials sig for the Lead algorithm, which can now be thought as some 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 stores their second latest nonce nonce' and try with that one if $\mathsf{JoinSession}$ initially fails. See Fig. 4 for a formal description of the changes with respect to Zoom's current scheme from Fig. 2.

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 the full version of this work.

Theorem 3. *The modified LL-CGKA scheme from Fig. 4 is secure with the liveness slack of P being at most*

$$\min\big\{\min(3,n)\cdot\Delta_{\mathsf{live}}, t_{\mathsf{now}} - t_{\mathsf{joined}}\big\} + \Delta_{\mathsf{live}},$$

where t_{now} denotes the current time, t_{joined} the time P joined the meeting, and n denotes the number of distinct leaders P encountered so far. 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.

4.3 Leveraging Clock Synchronicity

In this section, we explore an alternative approach towards mitigating the degrading liveness properties. Concretely, we propose to leverage pre-existing clock synchronicity to achieve better liveness properties 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 (some 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. 5: 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 setting and instruct the client to re-synchronize their clock (either displaying a warning or do 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.

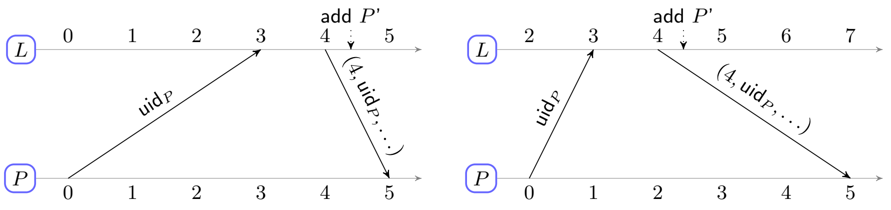

Fig. 5: The leader L's clock running ahead (right) negatively affects liveness as the addition of P' can be withhold longer from P.

The protocol We now discuss our proposed mechanism. In a nutshell, our proposed improvement works by each party P not maintaining a single (best-effort) estimate $\delta_P[L]$ to their current leader L, but strict lower and upper bounds $\delta_P^{\mathsf{min}}[L] \leq \mathsf{offset}_{L\to P} \leq \delta_P^{\mathsf{max}}[L]$ on their respective drift $\mathsf{offset}_{L\to P}$ gradually improved over the course of the protocol execution. Analogous to the current estimate, those bounds are derived from simple causality observations and in turn used to adjust the timestamp indicated as part of the heartbeat messages. See Fig. 6 for a formal description of the proposed modifications with respect to Zoom's current scheme.

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

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

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

In our protocol, P will have a meaningful such lower bound t_{then} 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

Protocol Client LL-CGKA (Improvement 2)

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

Algorithm: $\mathsf{CatchUp}(\mathsf{ust}, \mathsf{time}, \mathsf{grpPub})$
    req $\mathsf{uid}_{\mathsf{lead}} = \bot$
    parse $(\mathsf{lpls}', \mathsf{hb}') \leftarrow \mathsf{grpPub}$
    // 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 $\mathsf{hb}$

Helper: *receive-heartbeat(hb)
    parse $(\mathsf{t}', \mathsf{time}', \boxed{\mathsf{elapsed}'}, \mathsf{sig}'_{\mathsf{hb}}) \leftarrow \mathsf{hb}$
    req $\mathsf{t} = \bot \lor \mathsf{t}' = \mathsf{t} + 1$
    $\mathsf{t} \leftarrow \mathsf{t}'$
    $(\cdot, \mathsf{ipk}') \leftarrow \mathrm{cmKEM}.\mathsf{Identity}(\mathsf{uid}_{\mathsf{lead}})$
    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$(\mathsf{time}', \mathsf{elapsed}')$
    // Upon receiving hb with timestamp time'
    if $\mathsf{lastHb}^{\mathsf{min}} \neq \bot$ then
        $\mathsf{time}_{\mathsf{then}} \leftarrow \mathsf{lastHb}^{\mathsf{min}} + \mathsf{elapsed}'$
    else
        $\mathsf{time}_{\mathsf{then}} \leftarrow \mathsf{lastHb}$ // time of creating uid
    $\delta^{\mathsf{max}}[\mathsf{uid}_{\mathsf{lead}}] \leftarrow \min(\delta^{\mathsf{max}}[\mathsf{uid}_{\mathsf{lead}}], \mathsf{time} - \mathsf{time}')$
    $\delta^{\mathsf{min}}[\mathsf{uid}_{\mathsf{lead}}] \leftarrow \max(\delta^{\mathsf{min}}[\mathsf{uid}_{\mathsf{lead}}], \mathsf{time}_{\mathsf{then}} - \mathsf{time}')$

Helper: *update-liveness$(\mathsf{time}')$
    // 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. 6: The proposed changes with respect to Zoom's scheme from Fig. 2.

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 deducing a lower bound on when the last heartbeat of the old leader was sent, and the new leader L' indicating as part of the heartbeat a lower bound on the *elapsed duration* $\mathsf{elapsed}$ 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 deduce a lower bound on $\mathsf{elapsed}$ 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 assumes that this has been sent at local time $\mathsf{time}'_P := \mathsf{time}'_L + \delta_P[L]$ and correspondingly delays dropping out until $\mathsf{time}'_P + \Delta_{\mathsf{live}}$. Unfortunately, after a number of leader changes the uncertainty on $\delta_P[L]$ (and thus in our improved protocol the difference between $\delta_P^{\mathsf{min}}[L]$ and $\mathsf{offset}_{L \to P} \leq \delta_P^{\mathsf{max}}[L]$) can become quite large. This potentially means that this "converted" timestamp might be actually less accurate than the sent one, leading to the degradation in provable liveness observed for Zoom's protocol.

We, thus, want to be careful not to destroy the assurances in case the clocks are well synchronized. Hence, the protocol conservatively adjusts the received timestamp if and only if the leader's clock is surely behind or ahead, respectively:

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

Security We now state the respective security statement. A more formal version thereof and a proof is given in the full version of this work.

Theorem 4. *The modified LL-CGKA scheme from Fig. 6 is secure with the following improved liveness slack*

$$\min\left\{|\mathsf{offset}_{L \to P}|,\ n \cdot \Delta_{\mathsf{live}},\ t_{\mathsf{now}} - t_{\mathsf{joined}}\right\} + \Delta_{\mathsf{live}},$$

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 pointed out 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 meetings 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.

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,

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

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
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. 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)
33. 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`
34. Marlinspike, M., Perrin, T.: The double ratchet algorithm (11 2016), `https://whispersystems.org/docs/specifications/doubleratchet/doubleratchet.pdf`
35. 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`
36. 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`
37. 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)
38. 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`
39. 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`
40. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto (1996)
41. Seggelmann, R., Tuexen, M., Williams, M.: Transport layer security (tls) and datagram transport layer security (dtls) heartbeat extension. IETF RFC 6520 (Standards Track) (2012)
42. 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`
43. WhatsApp: Whatsapp encryption overview (2017), retrieved 05/2020 from `https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf`
44. Wire Swiss GmbH: Wire security whitepaper. `https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf` (2021)
45. 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`
46. 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)