Zoom Cryptography Whitepaper*

Josh Blum[1], Simon Booth[1], Brian Chen[1], Oded Gal[1], Maxwell Krohn[1], Julia Len[1], Karan Lyons[1], Antonio Marcedone[1], Mike Maxim[1], Merry Ember Mou[1], Armin Namavari[1], Jack O'Connor[1], Surya Rien[1], Miles Steele[1], Matthew Green[2], Lea Kissner, and Alex Stamos[3]

[1]Zoom Video Communications
[2]Johns Hopkins University
[3]Stanford University

August 21, 2023
Version 4.2

*Zoom is grateful for the collaboration and expertise of our co-authors, who appear here if they contributed to any version of this whitepaper.

Contents

1 Introduction

Millions of people use Zoom to learn, connect with friends and family, collaborate with colleagues, and deliver vital services. Zoom users deserve excellent security, and Zoom is working to provide these protections in a transparent and peer-reviewed process. This document describes the cryptographic protocols powering several Zoom products and features, both upcoming and already released.

We are actively engaging in a process of consultation with multiple stakeholders, including clients, cryptography experts, and civil society. As we receive feedback, we will update this document to reflect changes in roadmap and cryptographic design.

1.1 Outline

In Section 2, we start by providing some context on the Zoom platform and discussing the goals and limitations of our approach.

Many of our most secure offerings, including end-to-end encryption (E2EE), require Zoom client devices to have cryptographic keys whose secret components are not available to Zoom servers. Section 3 describes these keys, how they are managed by Zoom clients, and how we use *sigchains* to bind keys from a user's devices to that user's account and identifiers.

In Section 4, we propose a key transparency architecture that will force Zoom servers to be consistent about each user's identity, empowering Zoom client devices to monitor their own identities and detect any attempts at impersonation. In Section 5, we describe how users can leverage external identity providers to certify their own keys, allowing communication partners to independently verify them without relying on Zoom. Both of these mechanisms reduce the need for fingerprints and security codes to achieve authentication.

The following sections introduce the cryptographic protocols powering Zoom Mail Service (Section 6), Meetings (Section 7) and Phone (Section 8).

1.1.1 E2E Encryption for Zoom Meetings

Before version[1] 4.0, this document was titled "E2E Encryption for Zoom Meetings," as it focused primarily on the Meetings product, for which it proposed an incrementally deployable four-phase roadmap to strong end-to-end security and identity. Since then, we have deployed the first phase of the roadmap and expanded the use of end-to-end encryption and other advanced cryptography to more Zoom products, such as Phone and Email. To better describe these new developments, we moved away from a chronological presentation in favor of an evergreen document that describes the cryptographic design behind each product.

[1] Previous versions available at `https://github.com/zoom/zoom-e2e-whitepaper/`

2 Background and Security Goals

Zoom offers a comprehensive communications platform consisting of a variety of products, including Zoom Team Chat, Zoom Phone, Zoom Mail Service, Zoom Whiteboard, and Zoom Meetings.

For users of these products, Zoom provides software for desktop and mobile operating systems and embeds software in Zoom Room devices. In this document, when we refer to "Zoom clients" or simply "clients," we include all these various forms of packaging. Crucially, these are systems to which we can deploy cryptographic software. In contrast, some Zoom products can also be accessed through other systems which are unable to support custom cryptographic protocols, or cannot offer the same level of security when doing so. For example, users can join a meeting by dialing in from a landline phone (with no cryptography support), or from their web browsers (which can perform cryptography, but make it easy for the server to surreptitiously provide tampered cryptographic code).

Our goal is to provide the best security protections across all of these devices, mindful of the constraints that each environment poses, and without compromising the easy and seamless experience that our customers expect. When analyzing the security of our products, we consider a wide range of potential adversaries, namely:

Outsiders: Individuals who are not authorized to access specific information or data streams (such as everyone who is not mentioned as a sender or recipient in a given email thread, or everyone except the participants of a specific meeting), and do not have access to non-public information related to it (e.g., user passwords, meeting passwords, IDs, SSO systems). These attackers may monitor, intercept, and modify network traffic, but do not have privileged access to Zoom's infrastructure.

Participants: Zoom users who are authorized to participate in a specific communication or have been granted access to some information: for example, meeting participants who can access a meeting, because they know the meeting's ID and password or exercise other qualifying credentials.

Insiders: Those who develop and maintain Zoom's server infrastructure and its cloud providers, as well as attackers who have gained (even partial) access to this infrastructure.

Against these adversaries, colluding or working independently, we seek the following security goals:

Confidentiality: Only authorized participants should have access to the contents of end-to-end encrypted communications. If the Zoom product supports removing users from encrypted communication channels, then those users should no longer have access to those communications after they are removed.

Integrity: Those who are not authorized participants should have no ability to corrupt those end-to-end encrypted communications.

Abuse Prevention: When authorized participants engage in abusive behavior, there is an effective mechanism to report them to Zoom's safety team, to help prevent further abuse.

Broadly speaking, end-to-end encrypted products aim to achieve confidentiality and integrity goals even against insiders: we aim to detect, and if possible, prevent any violations of these properties. While prevention is preferable, we stress that even detection can be a powerful deterrent: it allows users to monitor their communications and quickly move to limit any damage, while demonstrating our commitment to be transparent about breaches.

All Zoom products also have many server-driven security and access control mechanisms, including transport-layer encryption, meeting passwords, and granular user and account level permissions. While these are not meant to protect against insiders with privileged access to our infrastructure (and are not the focus of this document), they are highly effective against outsiders.

2.1 Limitations

Our encryption protocols offer strong guarantees for our customers. However, as with any security system, there are limitations to our approach. There are certain classes of attacks and threats that we deem out of scope, including:

Metadata and traffic analysis: Insiders and outsiders may learn information about the timing, duration, and bandwidth usage patterns of end-to-end encrypted communications, as well as the list of participants and IP addresses.

Software flaws: Zoom's client code or the third-party libraries it links against can have bugs, or worse, intentional backdoors. Zoom's binary build procedures might become compromised. In these cases, there are no good cryptographic guarantees we can make. Zoom relies on extensive analyses by independent third party auditors to reassure customers in this domain.

Third party compromise: We leverage third-party Single Sign-Ons (SSOs) and Identity Providers (IDPs) to independently vouch for the identity of Zoom's users. Doing so moves the trust away from Zoom and to identity providers that many of Zoom's enterprise users already trust for sensitive identity operations. Where we do rely on SSOs and IDPs, the additional security guarantees they offer might be lost due to attacks on their infrastructure.

Denial of service: The encryption architecture does not offer any guarantees about availability of the service. Insiders can always refuse service to a specific user, including when they try to perform security critical operations such as revoking a compromised key to prevent further data from being encrypted and thus exposed.

In addition, we note that many Zoom products have rich feature sets and work on multiple platforms. Some of these features and use cases are incompatible with our security guarantees. For example, Zoom Mail Service can be used to send and receive emails from other

email services that do not support end-to-end encryption (see Section 6.3). Or, dial-in phones can be used to join Zoom Meetings, but do not support end-to-end encryption. E2E security of the type described in this document is not possible in settings where Zoom products interface with these existing standards. Moreover, users can access some Zoom products through their web browser, and without installing Zoom's client. Supporting E2EE for web users poses certain challenges: secure, long-term storage for cryptographic private keys might be unavailable; and worse, malicious web servers could serve backdoored source code to web clients with little chance of detection. We intend to participate in the web standards development process to facilitate the creation of browsers upon which we could offer dependable E2E security.

We will further comment on the specific properties and limitations for each of the products and features described in this whitepaper in their respective sections.

3 User Identity and Key Management

As part of an overarching end-to-end encryption solution, we introduce the concept of *cryptographic identity* to Zoom. This allows Zoom users to ensure they are communicating only with the intended parties across many Zoom products, and helps prevent impersonation and "meddler-in-the-middle attacks" (MitM)[2]. End-to-end encryption is only as secure as the ends: if Alice thinks she is talking to her coworkers, but instead her competitors are participating in the meeting (perhaps with their video off) or there is a MitM, encryption will not be sufficient to protect her. Zoom aims to give users helpful and trustworthy information to evaluate the identity of users they interact with.

This section describes how we define and represent a user's identity to the people they interact with on Zoom, as well as how we enforce that these identities are consistent over time and cannot be tampered with.

3.1 Non-Cryptographic Identity at Zoom

Zoom organizes its users into accounts. Accounts can be held by individual people, businesses or institutions, and they consist of one or more users: if Example Corporation uses Zoom, then each Example Corporation employee would be a Zoom user belonging to the Example Corporation account. Each user can have more than one device (e.g., a computer or a phone) on which they can use Zoom products.

Each account is part of a cloud infrastructure that hosts the data relating to the account and its users, such as email addresses and login information. Some Zoom users are in the Zoom commercial cloud; there is also a Zoom for Government cloud for U.S. government employees and contractors, as well as separate white-labeled private Zoom instances, each with their own cloud. Zoom products generally support cross-cloud communication, though there may be some limitations.

[2] "Meddler-in-the-middle attack" is also known as "man-in-the-middle attack."

Zoom users authenticate to Zoom in a variety of ways. Users can log in using their email address and a password, or via an OAuth or SAML-based flow with an external Identity Provider (IDP) that has been set up for their account. In all of these cases, an email address is used as a unique user identifier. If the account settings allow it, users can change their email address or authentication method.

Some Zoom products, such as meetings, do not require individuals to sign in as a Zoom user in order to participate, unless configured otherwise. Users can join a meeting by clicking a link or by entering the meeting ID and password in the app.

The identity information displayed for a given user depends on the Zoom product, product-specific settings, account settings, and whether the viewer is in the same account or a different account. Identity information may include name, job title, company, phone number, and email address. Users may be able to modify their identity information, though account administrators can restrict their users to approved names. This identity information, and mechanisms that control changes to it, are controlled by the Zoom servers and cannot be independently verified by clients.

Zoom products may provide mechanisms to enforce access control: for example, meetings support meeting passwords, the waiting room feature, and the ability to restrict the meeting to users in the host's account or users whose emails have a specific domain name. These features are enforced by the Zoom servers, so they can be circumvented if the server is compromised. They also do not prevent one member of an account from impersonating another member of the same account, and they may not give users enough information to make access control decisions themselves: for example, whether to admit an attendee from a meeting's waiting room.

3.2 Cryptographic User Identity

The cryptographic identity of a Zoom user consists of two components. The first component is a set of human-readable identifiers unique to each user and the account they belong to. This allows users to be identified by displaying their email addresses and information about their Zoom accounts to other users. Second, each user's identity includes the set of devices (and their cryptographic keys) controlled by that user. We describe a device model that lets us reason about how a user's devices and keys change over time, and helps us formalize the concept of trust between devices. The device model allows a user's devices to communicate amongst themselves in addition to securely communicating with other users. When a device, logged in for a given user, is first used for a feature requiring cryptographic identity, it automatically generates encryption and signing key pairs, a process called *provisioning*. These device key pairs will then be used to negotiate additional encryption key pairs shared between the users' devices.

The components of a user's identity can change over time, and it is important to keep track of these changes so that they are auditable. For this purpose, we introduce a data structure called a signed hashchain, or *sigchain*.

Finally, the device model is reflected in the user interface. Users are notified when new

devices are added, and are able to revoke devices that are lost, stolen, or no longer used.

3.3 Displaying Identity

Note: Displaying cloud identifiers is not currently available. We plan to release it in future updates.

This section describes how we could display the identity of a Zoom user to others in various products. Because cryptographic keys are not easy to read, compare, or keep track of, we only show human-readable identifiers in the user interface. A user's set of identifiers consists of three components:

1. A Cloud Identifier, which represents the cloud infrastructure a user's information is hosted on (omitted if the user is on the Zoom commercial cloud)
2. An Account Domain Name (ADN), which identifies the account that the user is part of, where applicable. Before the ZTT (Section 4) is deployed, the ADN will only be displayed for users whose identities are vouched for by a trusted third-party IDP (Section 5).
3. An email address, which can be used to distinguish individual users within the account

Here are a few examples of how a user's identifiers can be displayed to another user:

John Smith `example.com` (`jsmith@example.net`)	The display name, "John Smith," is freely chosen and not authenticated. `example.com` is the ADN. Note that the email domain, `example.net`, can differ from the ADN.
Lucy Lee `example.org` (`lucy.lee@example.org`) **GOV**	Since the `example.org` company works with the US government, their identities and keys are hosted on the separate Zoom for Government Cloud, and this is noted in the UI.
Anna Smith `example.com`	Anna might decide not to disclose her email address but still be identified as a member of the `example.com` account. In this case, although the user's email address would not be revealed, their devices' long-term cryptographic public keys could be leveraged by a determined attacker to ascertain when they have interacted with the same device multiple times, even when the display name is altered.

Richard Roe	Users can use some Zoom products as guests and display no identifying information to other users, other than the freely-chosen display name. Since they generate fresh long-term keys each time, the aforementioned tracing attack is not possible.
Mike Doe (`mike.doe@example.com`)	Mike is associated with an email address but not an ADN.

3.3.1 Identifying Accounts

Accounts on Zoom can be optionally identified using a domain name, which we call the Account Domain Name (ADN). Domain names make good identifiers because they are unique (while for example two companies with the same name might exist in two countries), and many users are already familiar with them.

To set the ADN, an account administrator may choose one of the account's Associated Domains. Associated Domains is a Zoom feature where an account admin can prove ownership of domain names (e.g. by adding a specific DNS record, adding a header tag to the home page, or hosting a file at a specific location on the domain) for the purposes of account and user management. Domain names consist of alphanumeric characters and hyphens.

Only a single account at a time can use a domain name as their ADN. Account administrators are allowed to change their ADN if needed. In the future, Zoom may provide a new dedicated subdomain as an option for accounts to use as the ADN.

3.3.2 Identifying Users

Zoom users are also identified by email address. Emails (unlike names) are unique, and they sometimes represent people better than the legal name held in a company's HR system. Most users already have email addresses they are associated with, such as the one they use to log in. Zoom Mail Service (see Section 6) also issues users a dedicated email address upon signup. For Zoom Mail Service users, the email address used as identifier will be their Zoom Mail Service address. We may support associating multiple email addresses per user in the future.

In some cases, users will be able to change the email associated with themselves. When a user Alice changes their email from `support@example.com` to `alice@example.com`, for example, other users that interacted with Alice are not notified of this change. However, if a new user Bob takes over Alice's old email `support@example.com` and associates it with his existing user, then people who interacted with `support@example.com` when it was associated with Alice will get a prompt explaining that `support@example.com` is now associated with a new user. These notifications are supported by the Contact Sync feature (planned for a future release); see Section 3.9.

3.4 Multi-Device Support

Zoom users often have several devices that they use Zoom on: their work computer, personal computer, mobile phone, and so on. Each of these devices stores long-term signing and encryption keys (called *device keys*), which it uses to secure communications and data across a variety of Zoom products. To obtain a rigorous concept of identity, we formalize the set of a user's device keys as well as the ways this set can change over time, a crucial step towards achieving the goal of linking a user's identifiers with their device keys.

Users have three main operations to change their set of valid devices:

1. DeviceAdd, which adds a new device to the set. The new device generates a new long-term signing key, which is used directly in some Zoom products (such as E2EE meetings) and also to sign statements about the set, and an encryption key, which can be used to communicate with other devices.
2. DeviceRevoke, which revokes the validity of a previously-added device. Revoked devices are still recorded as part of the set for auditing purposes, but new signatures from their keys are no longer considered valid, and the encryption keys they have access to are rotated as soon as possible.
3. BatchApprove, which indicates that an existing device considers all devices added to the set until this BatchApprove event legitimate and trustworthy. This operation is signed by the approving device's key.

Ensuring that each user has a single set of devices that is consistent over time serves several purposes. For the user themselves, it ensures that all of their devices know about each other, so they can be notified when a new device gets added and quickly react if their user account has been compromised.

A user's set of devices is also of interest to other users they interact with, because not all devices associated with a user may be trusted equally. If Bob is in a meeting with Alice's work computer today, he can trust the connection (i.e., that there is no MitM) by either checking the security code or by noticing that Alice's public key is the same as was used in all past interactions Bob had with Alice. This way, Bob only has to trust that there was no MitM the first time the connection was established, and from that assumption deduce that all communications where Alice has the same public key are also secure. This assumption is commonly referred to as Trust-On-First-Use (TOFU). If tomorrow, Bob meets with Alice's new mobile phone, Bob might not trust its key as much as her work computer's: a malicious server could have added it, or a hacker could have stolen Alice's Zoom password. It'd be unfortunate if Alice and Bob had to recheck their security code. By performing a BatchApprove operation from her work computer, Alice can indicate that all of her other devices are trusted, so Bob (who trusts the public key on Alice's work laptop) can use the signed BatchApprove statement to extend his trust to Alice's new mobile phone's key.

BatchApprove operations induce a trust graph over a user's set of devices, where each device represents a node and each BatchApprove adds an edge from the device performing the approval to all non-revoked devices introduced after that device. We call each connected

component in this graph an *approval class*, and we assume devices in the same approval class trust each other. When a Zoom user provisions a new device, they'll have access to their complete device list and so will be able to revoke any that are unrecognized, lost, or stolen. Because of this, we assume that later devices implicitly trust the validity of earlier ones.

Consider the following scenario:

1. Bob provisions Device **a**
2. Bob provisions Device **b**
3. Bob provisions Device **c**

Bob's graph is disconnected: he has 3 separate devices and 3 different approval classes. **c** does implicitly trust **b**, but we don't consider them part of the same approval class as the trust is not mutual.

4. Bob logs onto **b** and performs a BatchApprove

This operation partially connects Bob's trust graph. **b** trusts **c** (the device provisioned after **b**). Now, there are only two approval classes: one with **a** and one with **b** and **c**.

5. Bob provisions Device **d**
6. Bob logs onto **c** and performs a BatchApprove

Now, **c** trusts **d**. Because **b** already trusted **c**, we know that **b** trusts **d** as well, even though it never made this claim explicitly.

7. A malicious server provisions Device **e** in order to impersonate Bob
8. Bob logs onto **a**, revokes **e** since it's unrecognized, and performs a BatchApprove

Having all of Bob's devices know about each other allows Bob to take action if his account is compromised. After he revokes **e**, all of Bob's devices, as well as the users he communicates with on Zoom, know not to trust statements signed by **e**'s device key. Figure 1 summarizes Bob's trust graph after these steps.

3.4.1 Per-User Keys

One application of the trust graph is to facilitate *per-user keys* (PUKs): a set of keys shared between all of a user's devices, rotated on device addition or revocation.

In each DeviceAdd or DeviceRevoke operation, devices generate a new PUK seed. The PUK seed is encrypted for each unrevoked device's device key: devices implicitly trust all

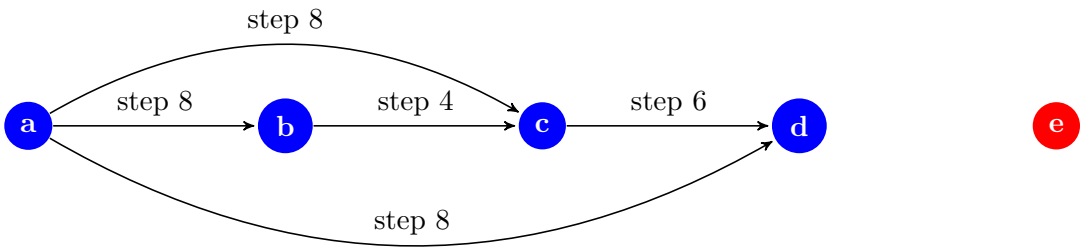

Figure 1: Device approval graph, where nodes are devices and edges are approvals.

older devices (otherwise, they would revoke them). Each seed is associated with a number called a *PUK generation*, which starts at 1 and increments every time the seed rotates. All devices within an approval class share the same set of known PUK seeds.

From each PUK seed, devices generate subseeds, symmetric subkeys, and asymmetric subkeys for various applications such as email and voicemail.

Symmetric PUKs facilitate syncing encrypted data between a user's devices. Devices use the latest per-user key to encrypt all content, but previous per-user keys are still useful for decrypting older data. In the above example, if **f** is provisioned, it doesn't yet have access to older PUKs; only the one it just created. But if an older device performs a BatchApprove that includes **f**, it will also encrypt all the PUK seeds it knows about for the devices it's approving, which means **f** can now decrypt data encrypted with keys created before **f** was introduced.

Asymmetric encryption PUKs can be used to encrypt data for other users. If Alice encrypts a piece of content for Bob using his most recent asymmetric PUK, Bob can read the content on all devices added before the content was encrypted, as well as all the devices added afterwards, as long as those devices have been approved by an earlier one. The public keys corresponding to the asymmetric keys derived from the PUK seed are published in the user's sigchain (Section 3.5.3).

To generate a set of PUKs, devices:

1. Generate a new 32-byte secret seed
2. Use HKDF on the seed to generate different 32-byte keys:
   (a) private X25519, context `"Zoombase-2-ClientOnly-KDF-PerUserX25519"`
   (b) email seed, context `"Zoombase-2-ClientOnly-KDF-PerUserEmailSeed"`
      i. private X25519 for email, context `"Zoombase-2-ClientOnly-KDF-PerUserEmailX25519"`
   (c) voicemail seed, context `"Zoombase-2-ClientOnly-KDF-PerUserVoicemailSeed"`
      i. private X25519 for voicemail, context `"Zoombase-2-ClientOnly-KDF-PerUserVoicemailX25519"`
   (d) symmetric, context `"Zoombase-2-ClientOnly-KDF-PerUserSymmetricKey"`

See Appendix B for a deeper analysis of multi-device configurations and the guarantees we can achieve given these rules. In summary, a newly-added device will immediately be able to send and receive new encrypted content that is accessible from all devices. Approving a device will give that device access to past encrypted content. If a device is revoked, that device is unable to decrypt any content encrypted after the revocation.

Note that if a device is self-revoked, the necessary key rotation might not happen immediately if there are no other suitable devices online to perform it. Before the key rotation is completed, we rely on the server to enforce that the revoked device does not access further user data, including ciphertexts sent both before and after the revocation, by no longer accepting the per-device authentication tokens and signatures using the keys the device owns. This is not a cryptographic guarantee and relies on the server's honesty, and also does not cover any content already in the device's memory before revocation.

3.4.2 Backup Keys

In addition to their physical devices, a user can add so-called "virtual devices" to their sigchain. Virtual devices also have signing and encryption key pairs associated with them, but instead of strictly corresponding to the physical device that generated them, these keys may be exported or communicated to other parties and are used to provide additional functionality. Virtual devices are treated like physical ones: they can be added and removed from the user's device list, other devices encrypt PUKs for them, and the corresponding private keys can be used to approve other devices or rotate PUKs.

A backup key is a string of letters and digits which the user can write down on paper. By entering the string on a new device, the user can decrypt and recover their encrypted data if all other existing physical devices become unavailable.

Backup keys can be generated by one of the user's existing devices. Note that the backup key can have at most the PUK access of the generating device. Backup keys are added to a user's device list using a DeviceAddAndApprove operation (which combines DeviceAdd and BatchApprove, as described above).

The device generates a high entropy string, which is displayed as a sequence of letters and digits, such as

`ZR30 4D11 5HJM RJG2 6H75 78DH B0VS 4KSF`.

The first four characters are used as a key identifier and are not considered private. Backup keys have at least 128 bits of private entropy, as well as built-in error correction to tolerate small copy-pasting mistakes. The backup key string is used to derive a seed using sodium's [9] `argon2id`, which is then used to derive the device's signing and encryption keys.

Assume a user provisions a new device and does not have access to any of their previous physical devices (either temporarily or permanently). The user can enter a backup key on this new device to recover all the data available to the backup key: the new device will use the backup key to re-derive the signing and encryption key corresponding to the virtual backup device and then perform a BatchApprove operation. In particular, the backup encryption key can be used to encrypt all the PUKs that the backup device had access to

for the current device's key.

3.4.3 Escrow Keys

Escrow (see Section 3.8) allows accounts to designate some of their members as *Escrow Administrators (EAs)*, who can access other account members' (*escrowees'*) encrypted data in order to support features like legal discovery, retention, and accidental loss prevention[3]. To support these use cases, an account can enable escrow, which prompts each account member with an unskippable notification to add a virtual device to their device list, the *escrow device*, whose secret keys are encrypted for the EAs.

The escrowee (or potentially one of their account's EAs) can rotate their escrow device's key (and therefore concurrently rotate their PUK) whenever one of the EAs' devices with access to the escrowee's keys is revoked (see Section 3.8.2).

3.4.4 Lockdown Mode

Note: As of version 5.15.10, lockdown mode is only available (and required) for EAs (see Section 3.8.1).

As detailed in Section 3.4, a standard user can add new devices at any time, and start using them immediately (except for accessing previously encrypted data) without approval from previous devices. This allows newly added unapproved devices to immediately start using E2EE Zoom products to send encrypted data on behalf of the user, while not granting it access to previously encrypted data until a previous device comes online to approve it. While this device model may be appropriate for most users, some users may benefit from additional security guarantees, namely that new devices cannot be used without the consent of an existing device. This prevents a compromised server or an attacker who learns the user's password from adding a device and impersonating the user without prior user approval[4](though imposing additional usability burden on the user).

Lockdown mode can be enabled or disabled for the user from a user's device in the oldest non-empty approval class. When in lockdown mode, new devices can be added, but are not permitted to rotate the PUK or approve new devices on behalf of the user until they are themselves approved by a device in the oldest class (and thus join that class). We refer to these new devices as *unconfirmed*, and older devices in the oldest class as *confirmed*.

If a confirmed device revokes an unconfirmed device, no key rotations are needed (as the revoked device did not have access to any PUKs). If a confirmed device approves an unconfirmed device, the existing PUK is encrypted for the newly confirmed device, again without rotations. The PUK is only rotated if a confirmed device revokes another

[3]Some companies are required by law to keep records of all their communications, and make them available to law enforcement when requested. In addition, employees' devices might get lost, stolen or otherwise unavailable.

[4]Note that compromising the server would allow an attacker to temporarily reassign a user's email identifier to a user whose keys are controlled by the attacker, thus circumventing the need for prior approval for impersonation even if lockdown mode is enabled. This attack would still be detectable after the fact, and currently is not applicable: only EA sigchains can enable lockdown mode, and they do not have email identifiers.

confirmed device, including itself (although in the self-revoke case, the rotation will be performed by another confirmed device once it comes online). Unconfirmed devices are allowed to perform revocations of any device, but other clients will not accept any PUK rotations performed by unconfirmed devices while the user is in lockdown mode.

Lockdown mode reduces the number of necessary key rotations, in addition to preventing a compromised server from impersonating a user. On the other hand, the user will need to have access to at least one confirmed device in order to add and use new devices. If the user loses or revokes all of their devices, then data encrypted for their user will unrecoverable, and their account will be essentially unusable.

The "locked out" user would have to create a new user account (note that Zoom allows email addresses to be re-assigned). To reduce the likelihood of this "locked out " scenario, the Zoom server requires users to create a backup key (see Section 3.4.2) before entering lockdown mode. The server enforces that users in lockdown mode have at least one unrevoked device. One exception to this is that if an account disables escrow, the server will revoke all devices belonging to the account's EA.

3.5 Consistent Identities With Sigchains

Both accounts and users have states that change over time. An account can change its identity provider or its ADN, and a user can change their email address or add and remove devices.

We need to keep track of states that change over time in a way that is auditable. To do so, we describe the sequence of changes in a data structure called a signed hashchain, or *sigchain*.

Once a client learns of a sigchain, the only changes to this chain that will be considered valid are extensions of the sequence. Since changes cannot be "forgotten," the Zoom server cannot rewrite history.

Still, this model doesn't force the Zoom server to be consistent across different devices it talks to. We will add a transparency layer called the Zoom Transparency Tree (Section 4) to ensure that the Zoom server must present the same information about sigchains to all users.

3.5.1 Sigchains

A sigchain is a sequence of statements (called *links*), where each link includes a collision-resistant hash of the previous link. These links can be thought of as state transitions that modify an object (the sigchain state). For a user sigchain, the sigchain state contains the list of active devices, the list of revoked devices, the trust graph, and the list of email addresses and accounts historically associated with the user.

In order to accept a transition as valid, clients check that it satisfies several conditions, including that:

1. The link is of a known type.
2. The link has the correct fields for that type.
3. The transition is admissible given the current state.
4. The link correctly includes the hash of the previous link.
5. Some links require cryptographic signatures by the devices authorizing the transition to be considered valid. In these cases, the signatures are encoded as part of the links to compute link hashes.

Examples of admissibility rules for a user sigchain include that a device can only be revoked if it was active in the previous state, and that signatures over revocation links must be by a device that was active in the previous state.

Since each of the links in a sigchain contains a hash of the previous link, the hash of the last link is a compact commitment to the entire sigchain state. Each sigchain link also contains an incrementing sequence number. We refer to an object consisting of the sigchain type, the last link's sequence number, and the last link's hash as the *sigchain tail*:

```
{
  "sigchainType": "User",
  "lastSequenceNumber": 15,
  "lastLinkHash": "484ad7..."
}
```

When clients want to update their view of a sigchain from the server, they can just query for the new links and ensure that the first new link contains a previous hash matching the cached tail.

Note that the examples of objects in this document are encoded in JSON and simplified for ease of exposition. The actual implementation may use different application encodings and data structures.

Different applications require different levels of access to sigchains. For example, although a user should be able to fully audit the history of past email addresses stored in their sigchain, meeting participants might only need to see the most recent one to display it in the UI. For this reason, rather than being directly encoded, sensitive information on a sigchain link is stored as a commitment: rather than `alice@example.com`, a sigchain link could contain

$$\mathsf{COMMIT}(\texttt{alice@example.com}) = \mathsf{HMAC}(\mathsf{randomKey}, \texttt{alice@example.com}).$$

The server gives all users the entire sigchain link so that they can check that its signatures and hashes are valid, but only Alice's devices will receive the plaintext email addresses and 32-byte random keys corresponding to previous email addresses. COMMIT can also be used to selectively delete parts of links such as device names: a server can throw away the random HMAC key as well as the plaintext data, and the signature over the link will still verify.

3.5.2 Overview of Sigchain Types

Zoom devices, users, and accounts are each internally identified by unique immutable identifiers called deviceID, userID, and accountID respectively. The deviceID is generated randomly for each new device, while the userID and accountID are assigned by the server.

Each device, user, and account is also associated with more user-friendly (but mutable) identifiers: respectively, device names, email addresses, and ADNs.

Representing these different components and their relationships requires different types of sigchains:

1. User sigchains store, for each userID, information related to that user's identity, such as the user's email address, accountID, and the set of their devices and their trust relationships.
2. Email sigchains store, for each email address, the associated userID.
3. Account sigchains store, for each accountID, both the ADN and identity provider associated with the account.
4. ADN sigchains keep track of, for each domain name, the accountID to which the domain is associated.
5. Membership sigchains keep track of, for each accountID, the userIDs within the account.

Note that some of the information stored on these sigchains is redundant: for example, a mapping between an email and the corresponding userID is recorded both in a user sigchain and in an email sigchain. This is necessary so that the server cannot claim that two separate userIDs are associated with the same email address at the same time. Accordingly, some operations will cause multiple chains to be updated at the same time.

As detailed earlier, every Zoom user is part of an account. As potential optimizations, if this account only has a single user and doesn't have an ADN, then that user's sigchain might not mention their accountID, and there would be no corresponding account or membership sigchains until the account either gets another user or an ADN.

3.5.3 User Sigchains

User sigchains record changes to a user's identity. There are several types of user sigchain links, each representing a different way to change a user's identity.

EmailChange. As mentioned in Section 3.3, users can set and change the email addresses that will be displayed for or used to communicate with other users. Two users can switch email addresses, but the server will prove that two users do not have the same email address at the same time. An EmailChange link will have the following fields:

```
{
  "sigchainType": "User",
  "linkType": "EmailChange",
  "sequenceNumber": 10,
  "prev": "484ad7...",
  "cloudName": "commercial",

  "userID": "ebc0d2...",

  "emailChange": COMMIT({
    "email": "alice@example.com",
    "emailChainSequenceNumber": 5
  })
}
```

The first six fields are common to every sigchain link. sequenceNumber is an incrementing counter that starts from 1 for the first link, prev is the (canonical) hash of the previous link in the chain (in this case, the one with sequence number 9), and cloudName specifies which cloud the sigchain belongs to.

Every user sigchain link also specifies the userID.

Here, the email field specifies the new email address to be associated with this user, which supersedes any previous email. Every time an EmailChange link associates this user with a new email, the email sigchain for that email address is also extended with a corresponding UserIDChange link referring to this user's userID, and the sequence number of that link is reported in this link as emailChainSequenceNumber.

Because the Zoom website can be used to change one's email address, EmailChange links do not have any signatures (i.e., they can be inserted into sigchains by the Zoom server).

AccountChange. Users can also transfer between accounts, similarly to how they can switch emails.

```
{
  "sigchainType": "User",
  "linkType": "AccountChange",
  ...
  "accountChange": COMMIT({
    "accountID": "c2d8aa...",
    "membershipChainSequenceNumber": 12,
    "additionIndex": 5
  })
}
```

Since multiple users can be added to a membership sigchain in a single link, additionIndex specifies the corresponding position in that link. If a user is removed from an account, the accountChange section specifies accountID `null` and the other two fields are omitted.

Since Zoom servers are allowed to move users between accounts, AccountChange links do not have any signatures.

DeviceAdd. A device addition link specifies the long-term public device keys for the new device and a human-readable name:

```
{
  "sigchainType": "User",
  "linkType": "DeviceAdd",
  ...
  "deviceID": "ebc0d2...",
  "deviceName": COMMIT({
    "name": "Alice's Work Smartphone",
    "version": 1
   }),
  "deviceType": "PHONE",
  "deviceEd25519PublicKey": "ce8564...",
  "deviceX25519PublicKey": "ad7913...",

  "userEmailX25519PublicKey": "f40b66...",
  "userVoicemailX25519PublicKey": "d600fc...",
  "userX25519PublicKey": "c2cce1...",

  "revokeDeviceIDs": [
     "ac98ad...",
     ...
  ]
}
```

This link specifies a device identifier (unique among the devices associated with this user), a signing public key, an encryption public key, as well as a device name and type. The device can be a virtual device, such as a backup key (see Section 3.4.2). Note that the device name is hidden by a commitment, because users other than the owner of the chain do not need to see it. In order to support reusing names, the device name includes an incrementing version component, which will be visible in the user interface. Device names allow the user to have a human-readable, unambiguous way to distinguish their devices.

The link also specifies the new per-user key public keys. The encryptions of the new PUK seeds for older devices are not represented within the sigchain link, but are sent separately to the server, which propagates them to the older devices.

After the user adds a new device, they can immediately use it to revoke any existing undesired devices, which justifies our claim that later devices should trust the validity of earlier (unrevoked) ones.

To ensure that users know the corresponding device private key, each DeviceAdd link requires a signature from that key, which can be checked against deviceEd25519PublicKey in the link.

UserRoot. This link should always be the first link in a user's sigchain, and can only appear as such. It is equivalent to a DeviceAdd and specifies all the same kinds of information about the user's first device, but can also include emailChange and accountChange fields, which convey similar information as the corresponding sigchain links. Including these fields here allows us to reduce the overall number of sigchain links.

DeviceRename. Users can change their device names if desired. This change is signed with the device's public key.

DeviceRevoke. When a device is stolen, lost, or no longer used for Zoom, the user should revoke it. If one of the user's valid devices performs the revocation, it will also rotate the PUKs and sign this link to guarantee integrity of the new key. If instead the revocation is done through the Zoom website, the PUK cannot be rotated and no signature is made. After a device signing key is revoked, it can no longer be used to sign sigchain links.

DeviceKeyRotate. If a user suspects their device was temporarily compromised, or if they have institutional key rotation policies, they might want to rotate their device keys and PUKs. This operation keeps the same deviceID but chooses new signing and encryption keys, as well as a new set of PUKs, and uploads encryptions of each PUK secret known to the device for the new encryption key so that the old encryption key can be safely discarded. This link is signed by the device's previous public key in addition to the new key. As with DeviceRevoke links, the old signing device key cannot sign further sigchain links.

BatchApprove. A BatchApprove link lets a device indicate that it trusts the validity of all devices created after that device until the point in the sigchain where the BatchApprove link appears. To reduce the number of sigchain links, the user can also specify a list of devices to revoke within this link. The link can specify a new set of PUKs, and this is required if any devices are being revoked. This link is signed by the approving device, and can be used to construct a trust graph as described in Section 3.4.

DeviceAddAndApprove. This link is equivalent to a fused DeviceAdd and BatchApprove, but allows us to reduce the overall number of sigchain links and also guarantee that intermediate states of some operations can't appear on the sigchain. For example, this link is used to add and approve escrow keys (see Section 3.4.3). It is posted by an existing device, the approver, but also specifies a device identifier, name, and keys for a new device. Uniqueness of the new device's identifier, name, and keys are enforced just like in DeviceAdd. This link implies that the approver trusts all devices created up to and including the device being added by this link. Signatures from both the approver and the new device are required. When this link is used to add an escrow device, the link also records the identifier for the EA sigchain (see Section 3.8.1), its tail, and its PUK generation.

PerUserKeyRotate. If a device notices that a device revocation (either a self revocation, or one from the website) has occurred after the most recent PUK was generated, the device will perform another PUK rotation using a PerUserKeyRotate link. This guarantees that even if the revoked device was compromised, this newest PUK is still confidential. For personal storage, staleness is not an issue, as any device trying to encrypt data will first rotate the per-user keys. But if other users are encrypting data for a compromised PUK and the server cooperates, data could be readable by a revoked device. This link is signed by the device that is rotating the PUK.

LockdownMode. A device can enable or disable lockdown mode (see Section 3.4.4) using this link. A user can enable lockdown mode if all of the user's unrevoked devices are a single approval class. When in lockdown mode, only approved devices are allowed to rotate the PUK or approve new devices.

3.5.4 Email Sigchains

Users can change their email over time. It is important that at any specific point in time, each email corresponds to a unique user, so that Zoom users can be unequivocally identified. Also, users should be able to audit whether at any point their email has been associated with another user. We record the mapping between an email and the corresponding userIDs in an email sigchain. These sigchains only have one kind of link, UserIDChange.

```
{
  "sigchainType": "Email",
  "linkType": "UserIDChange",
  ...
  "email": COMMIT("alice@example.com"),

  "userIDChange": COMMIT({
    "userID": "ebc03d...",
    "userChainSequenceNumber": 9
   })
}
```

The userChainSequenceNumber refers to the position in the user sigchain of the corresponding EmailChange link (or the initial UserRoot link).

Since users can change their email on the Zoom website, this link does not require any signatures.

3.5.5 Account Sigchains

Account sigchains consists of two kinds of links, which record the identity provider and ADN that each account is using. IDPUpdate links contain the domain name of the IDP (say `examplecorp.generic-idp.com`). IDPs should not use the same domain for multiple accounts to prevent equivocation of their attestations.

```
{
  "sigchainType": "Account",
  "linkType": "IDPUpdate",
  ...
  "accountID": "abef02...",

  "idpDomain": "examplecorp.generic-idp.com"
}
```

ADNChange links associate an ADN with the account. Only the latest ADNChange link is considered valid, and each one corresponds to an AccountIDChange link in the appropriate ADN sigchain. If an account is no longer using an ADN, the adnChange section specifies ADN `null` and the other field is omitted.

```
{
  "sigchainType": "Account",
  "linkType": "ADNChange",
  ...
  "accountID": "abef02...",

  "adnChange": COMMIT({
    "adn": "example.org",
    "adnChainSequenceNumber": 9
  })
}
```

EscrowAdmin links enable escrow for the account by including the Escrow Admin sigchain's userID and its current latest sequence number (see Section 3.8.1). To disable escrow, the link should set the userID to null with the other field omitted.

```
{
  "sigchainType": "Account",
  "linkType": "EscrowAdmin",
  ...
  "accountID": "abef02...",

  "escrowAdmin": COMMIT({
    "userID": "ebc03d...",
    "escrowAdminChainSequenceNumber": 9
  })
}
```

Links in account sigchains do not require any signatures.

3.5.6 ADN Sigchains

ADN sigchains track which accountID a specific Account Domain Name is associated with. There is only one type of link, AccountIDChange, which corresponds to and points to an ADNChange link in the account sigchain for the appropriate accountID:

```
{
  "sigchainType": "ADN",
  "linkType": "AccountIDChange",
  ...
  "adn": "example.org",

  "accountIDChange": COMMIT({
    "accountID": "873c34...",
    "accountChainSequenceNumber": 29
  })
}
```

Links in ADN sigchains do not require any signatures.

3.5.7 Membership Sigchains

Membership sigchains record changes to the set of users which are part of each account. They have a single type of link, ChangeMembers:

```
{
  "sigchainType": "Membership",
  "linkType": "ChangeMembers",
  ...
  "accountID": "19aebb...",

  "added": [
    COMMIT({
      "userID": "db2f1c...",
      "userChainSequenceNumber": 9,
    }),
    ...
  ],
  "removed": [
    COMMIT({
      "userID": "9ae3d2...",
      "userChainSequenceNumber": 4
    }),
    ...
  ]
}
```

Each link can add or remove multiple users. Each user is hidden behind a commitment so that it is possible to prove that an individual user is part of an account without also leaking the userIDs of the other members that are being added as part of the same link. Hiding the userIDs of the users being removed from the account would potentially make it harder to prove that a user is indeed still a member of a specific account without opening all the commitments. We will solve this problem in the future by leveraging the transparency layer, but until then, clients will not rely on these sigchains, although the server will start keeping track of them.

Links in membership sigchains do not require any signatures.

3.6 Sigchain Fingerprints

A sigchain fingerprint summarizes a user's sigchains. It is constructed as a hash of the user's user sigchain and account sigchain tails (note that this includes the user's present and past email addresses, as well as the current ADN due to the reference in the account sigchain). Sigchain fingerprints are useful in two contexts: to verify new devices before approving, and to verify the keys used to communicate with other users.

During the device addition and approval process, a malicious insider has the opportunity to conduct a MitM attack. After the new device is added, the insider creates a copy of the user's sigchain but replaces the keys in the final DeviceAdd link with keys controlled by the attacker, using the same device name. The forged copy is served to older devices while the honest sigchain is served to the new device. When the older device sees the approval interface, only the device name is shown, and so the user may be tricked into accepting the server's device unintentionally (which would imply sharing all previously encrypted data with the attacker). However, in this case the two devices would necessarily display different fingerprints (as they bind to sigchains containing different keys, and the hash is collision resistant). Users concerned about MitM attacks can check, before approving new devices, that the fingerprint shown on the approving device matches the one from the device(s) they intend to approve. This ensures that an attacker cannot interfere with the process and gain access to historical encrypted data.

Sigchain fingerprints are also useful when communicating with other users. For example, when a user emails another user for the first time, a malicious server could potentially swap out the intended user's sigchain for an entirely fabricated sigchain with devices controlled by the server. In these contexts, the user interface will allow users to verify the other user's sigchain fingerprint before sending the encrypted message. This process requires the two users to securely communicate out-of-band (for example, in-person). The sender can check that the fingerprint they see in their client matches the recipient's fingerprint to ensure the message can be decrypted only by the intended recipient. Similarly, the recipient can do the same for the sender to verify that the message came from the expected sender and was not altered.

In the user interface, fingerprints are represented in hexadecimal. Users can always view their own fingerprint from the Zoom client, and Zoom products will show the fingerprints of other users in specific scenarios where it is useful.

3.7 Client Key Management

3.7.1 Storing Secret Keys on Device

Clients persist device key pairs indefinitely until a DeviceRevoke or DeviceKeyRotate occurs. Device keys are never transmitted to any other device or the server. They may sometimes be lost after a disk corruption or operating system reinstall. In this case, the user must go through the provisioning process once again as a new device would.

Long-term device keys are stored in the local operating system's keychain (where available),

but with some added protection for two Zoom users using the same OS account. Before storing keys in the local OS-provided keychain, we encrypt them using a key-wrapping key stored by the server and specific to each user and each device. When a user revokes their device, they will delete the keychain entry and the server will delete the corresponding key. This also guarantees that keys cannot be recovered from a backup of a device that has since been revoked. If two users are using the same computer, the key-wrapping key prevents one user from being able to access the other's keys.

One exception are keys corresponding to devices on the EA sigchain (see Section 3.8.1). These keys are also stored in the OS keychain, but they are encrypted using a key-wrapping key which is unique to each EA sigchain and each device. The server will only provide this key-wrapping key to users with EA permissions (see Section 3.8.3). This allows two EAs in the same account sharing the same device to use the same EA device key, which therefore only needs to be provisioned (and approved) once.

We use the committing AEAD scheme $\mathsf{CtE1}$ [12] to prevent the server from supplying malicious key-wrapping keys. On provisioning, after generating the device encryption and signing key pairs, the client will encrypt each secret key k as follows:

1. Generate a 32-byte random string KWK and request the server to store it persistently associated with the user and device.
2. Define $\mathsf{Context} \leftarrow$ `"Zoombase-1-ClientOnly-KDF-SecretStore"`.
3. Compute $C \leftarrow \mathsf{CtE1\text{-}Enc}(\mathsf{K}{=}\mathsf{KWK}, \mathsf{H}{=}\mathsf{Context}, \mathsf{M}{=}k)$, where H is the associated data parameter for the underlying AEAD, and store it in the system keychain.

For $\mathsf{CtE1}$, we use $\mathsf{HMAC}_{\mathsf{SHA256}}$ as the commitment function and `libsodium` [9]'s `crypto_aead_chacha20poly1305_ietf` as the AEAD.

We emphasize that this feature will not protect against an insider who also has access to a user's device (for example, by colluding with another user using the same device). It will also not prevent different users of the same device from installing malware to steal the other user's keys.

3.7.2 Device Management Interface

We offer a dedicated UI to manage devices that are part of the user's sigchain, available in the Zoom client's settings. Upon visiting this device list, clients ask the Zoom server for the latest sigchain tail and process any new links in order to make sure that the view is up-to-date. The device list contains all active devices (which can be used to participate in E2EE communications) and revoked devices (which can no longer be used), indicating their device name and type based on the sigchain;[5] it also has the user's own fingerprint and the user's current and past email addresses (if any). Users can revoke devices from this view. If a device realizes that it is revoked (by processing updates to its own sigchain,

[5]The list may also include some information about devices that is not directly stored on the sigchain and that we rely on the server to report honestly. For example, it may include the earliest time of any E2E-encrypted communication decryptable by each device, determined as the time at which the oldest device in its approval class was added.

for example as part of a periodic refresh or because of a server notification), it will delete all private ephemeral and long-term keys as well as sensitive data, and then log itself out.

When the user first uses a feature that requires sigchains from one of their devices, that device generates a new set of device keys and adds them to the first link in the user's sigchain. From then on, the user's other devices (existing and future) will also generate their own keys and extend the chain; each time, the user is prompted to review the device list and revoke any devices that are unrecognized, lost, stolen, or no longer used.

In addition, after provisioning each new device, the user gets notifications on their old devices asking them to approve or revoke any new untrusted devices. This list might include devices that are already revoked but are still new from the perspective of the old device. Users also get notifications regarding changes made to their email address.

Once the Zoom Transparency Tree (Section 4) is deployed, Zoom servers and insiders will not have the ability to, e.g., suppress notifications in order to hide a malicious device addition or email change.

3.8 Account Escrow

Some of the data that organizations store on the Zoom platform is encrypted with keys that are only known to the devices of the users authorized to access it; the data is not available to Zoom servers. For example, emails between active users of the Zoom Mail Service (Section 6) and some phone calls between Zoom Phone clients (Section 8.1) are in most cases encrypted using the sender and recipient's keys only. However, administrators of these organizations may also wish to have access to this data, for example, to protect against accidental data loss (if the account member loses their devices) or to support legal retention and discovery. We refer to solutions that give such organizations cryptographic access to their members' encrypted data as *escrow*, and in this section we present Zoom's solution to this problem.

When an account enables escrow, all the account members (*escrowees*) receive an unskippable notification informing them that escrow is being enabled. Upon acknowledgement, each member adds a "virtual" escrow device (Section3.4.3) to their own user sigchain. This escrow device's public keys corresponds to secret keys known to a set of devices which are controlled by *Escrow Administrators (EAs)*, account members designated with escrow permissions. This enables all account members to automatically share their PUKs (and by extension their encrypted data) with these admins.

3.8.1 Escrow Administrators and the EA Sigchain

To enable escrow for their account, an account administrator assigns EA permissions (Section 3.8.3) to one or more users to designate them as EAs. EAs will then be able to provision one or more of their devices with fresh *EA device keys*, which ultimately will be used to decrypt the encrypted content escrowed from all the account's users.

EA device keys are independent of the device keys listed on the EA's own user sigchain (and used to encrypt their own data); instead, the EA enabled devices and their public

keys are added to a special user sigchain, which we call the *EA sigchain*. EA sigchains follow all the rules of regular user sigchains (Section 3.5.3), but they have no associated email, and devices listed in this chain can belong to different EAs (instead of a single user).

In addition, the server adds an EscrowAdmin link to the account sigchain referring to the EA sigchain's userID, so that account members can be on the same page on whether escrow is enabled or not. Each EA sigchain is associated with at most one account, and vice versa.

When provisioning an EA's device to have access to all the account's data, the device generates fresh EA device keys and adds a new link to the EA sigchain. These users' devices therefore have two sets of device keys: one on the user's own sigchain, and one on the EA sigchain. Once it is approved by an existing device on the EA sigchain, the device will receive all the PUKs associated with this EA sigchain, allowing it access to all the escrowed data. Provisioned EA devices will display to their corresponding EA users the list of active EA devices (which EAs can use to monitor which devices have access to the account's data) and the EA sigchain fingerprint. Monitoring EA devices and comparing fingerprints prevents MitM attacks. The fingerprint can be compared before approving a new EA device, and shared with all the account members so that they can confirm it before enabling escrow. After enabling escrow, account members can continue to monitor the EA fingerprint to ensure that their escrow devices are rotated for every EA PUK rotation.

Given that the Zoom server is responsible for enforcing which users are EAs, and therefore can add devices to the EA sigchain, the first device will enable lockdown mode on the EA sigchain (see Section 3.4.4), and members' clients will enforce that this is the case. This prevents an attacker compromising the Zoom server from generating new PUKs on the EA sigchain and therefore gaining access to account members' data. Lockdown mode also avoids having to rotate the PUK whenever a new device is added to the EA sigchain, which has a significant performance impact as it would trigger a PUK rotation in all the account member's chains as well (Section 3.4.1).

When an account administrator disables escrow, the Zoom server revokes all escrow admin management permissions, adds a new link to the account sigchain with an empty escrow admin userID, and revokes all device keys on the EA's chain. As soon as clients are aware of this, either through a server notification or from their own monitoring, they revoke the virtual escrow devices from their own chains (and rotate their PUKs). EAs' devices will also revoke the EA device's secret keys from durable storage. If the user is moved between different accounts, and the new account has escrow enabled, the user will be notified of the new escrow being setup and given the opportunity to check the new EA sigchain's fingerprint. If the former account has escrow enabled, those EAs will only have access to the user's ciphertexts before the account change, while the user's devices will share all known PUKs with the new EAs. As detailed in the Security Limitations section, we rely on the server to enforce proper access control for cyphertexts when the encryption keys cannot be promptly rotated due to the user's devices being offline, which can happen in both the above cases.

It is important that the EAs maintain access to at least one device in the oldest non-empty approval class at all times, as otherwise escrowed data could be irreparably lost. For this reason, administrators are required to generate a backup key on the EA sigchain when

enabling escrow. If all the devices on the EA's oldest class are unavailable, the only option to recover is to turn escrow off and back on, which would trigger notifications to all account members to acknowledge escrow again.

When escrow is enabled, or re-enabled when old escrow devices are unavailable, the new EAs will have access to a specific user's data only after that user comes online to encrypt their keys for the EAs. Therefore, there may be unrecoverable gaps in the content that can be recovered for the escrowed users. To avoid this, we recommend account administrators to enable escrow before encryption features that depend on the key management infrastructure we describe (such as Zoom Mail Service and Advanced Encryption for Voicemail), to avoid turning escrow off and on, and to generate multiple EA backup keys and keep them in safe locations.

3.8.2 Users' Escrow Device Management

When an account enables escrow and sets up an EA sigchain, all devices belonging to the account's members will receive an unskippable notification that escrow is being enabled for their account, and that the user's data will be available to the account's EAs. Clients will validate that escrow is turned on by checking that the EA sigchain's userID is in the account sigchain, and only accept EA sigchains with lockdown mode enabled.

The notification shows the EA sigchain's fingerprint, which account admins should share with their users through a secure out-of-band channel. Users should compare the fingerprint received from the out-of-band channel with the one displayed by their clients. If the fingerprints differ, users should reach out to their account admins to notify them of the potential MitM attack. Users will not be able to use features leveraging their device keys until escrow is enabled.

Upon acknowledgement of the notification, clients generate a fresh set of device keypairs corresponding to their new virtual escrow device, encrypt the secret keys for the EA sigchain's latest PUK, delete the secret keys, and add the corresponding public keys with a DeviceAddAndApprove link on their own sigchain. The server will prevent users from revoking this virtual device (unless the administrators disable escrow).

If the EA sigchain's PUK rotates, the escrow device keys of all the users in the account (and their PUKs) have to rotate as well, as all those keys are known to a revoked device. A DeviceKeyRotate link that rotates an escrow device's key can either be signed by any of the users' own devices or by one of the EA's device keys. At the moment, these rotations are performed by each user's own devices when they come online, but in the future we can have the EA's devices also perform them to ensure that the escrow device key rotation is completed in a timely manner, especially for users who might not come online frequently. Escrowee clients do not display UI notifications for these escrow device key rotations, because the system enforces that the only devices that can rotate the escrow device keys are those on the user sigchain, and those trusted by the device on the EA sigchain that enabled escrow (and that the user has acknowledged they trust). This is achieved by having clients enforce that the EA sigchain is in lockdown mode, and including the EA sigchain tail in the DeviceAddAndApprove introducing the escrow device. If later the client detects that the EA sigchain is no longer in lockdown mode, or that the device which posted the

latest PUK on this chain is not trusted by the one which originally enabled lockdown mode (for example, this could happen when lockdown mode is disabled and re-enabled by a different device on that chain), it either throws an error or asks the user to acknowledge escrow again (with an updated fingerprint).

The Zoom server immediately notifies escrowee clients to handle escrow admin's PUK rotations; in the future, users will monitor the Zoom Transparency Tree for the EA and account sigchains, which would prevent the server from withholding such updates. When learning that the EA or account sigchain have been updated, for example before performing an escrow device key rotation as above, clients enforce that escrow is still enabled and that the userID of the EA sigchain mentioned has not changed (by looking at the account sigchain), and that the EA sigchain has lockdown mode enabled. If clients detect that escrow has been turned off, or that the EA userID has changed, they will revoke their escrow devices from their own chains, rotate their PUKs, and potentially notify the user and re-enable escrow as above using the new userID.

3.8.3 EA Permissions

While all the EAs' devices in the oldest class have access to the same PUK keys and can perform the same actions from a cryptographic perspective, the Zoom servers allow assigning a more granular set of permissions to each Escrow Administrator, which would apply to all their devices on the EA sigchain. These additional restrictions align with typical business requirements our customers might have, as detailed below, but could be circumvented if the Zoom server were to be compromised.

Users with the `escrow-manager` permission can enable escrow, create the first escrow device, grant escrow permissions to other users, and approve new devices on the EA sigchain.

To support user recovery (see Section 3.8.4) and legal discovery (see Section 3.8.5), we introduce two more server-enforced permissions.

Users with the `escrow-write` permission (used for recovery) and the `escrow-read` permission (used for discovery) both can add devices to the EA sigchain, but cannot approve the device themselves (only users with the `escrow-manager` permission can approve new devices for the EA). Therefore, to receive access to the EA PUKs, their newly added devices need to be approved by a user with the `escrow-manager` permission. In addition, users with the `escrow-write` permission can use their EA device keys to write approval links on other account member's user sigchains (therefore helping them recover their data), while users with the `escrow-read` permission can request from the server ciphertexts encrypted for any of the account member in order to decrypt them.

3.8.4 User Recovery

In the case that a user in an account with escrow enabled loses access to their devices (including any backup keys), they can ask one of their escrow account administrator for help recovering their data on a newly logged-in device.

An EA with the `escrow-write` permission can use one of their authorized devices on the EA sigchain to approve the user's new device and give them access to the necessary encryption

keys. As in a regular device approval (Section 3.7.2), the EA is first presented with an approval review screen which includes all of the escrowee's devices (and the escrowee's fingerprint, which can be checked out-of-band with them). If the admin accepts, the admin's device signs a BatchApprove sigchain link on the escrowee's chain, and encrypts all of the user's previous PUKs for the user's new device. The EA signs the approval link with both the virtual escrow device key (from the user's sigchain) and the escrow admin's own device key (from the EA sigchain).

3.8.5 Legal Discovery

Some situations, such as legal discovery, will require the account admin to obtain data from its account members.

If escrow is enabled, an EA with the `escrow-read` permission can request from the server escrowees' data ciphertexts (such as emails encrypted with escrowee PUKs) and ciphertexts containing the escrowee PUKs (encrypted for the virtual escrow device, the secret key of which is known to the EA devices), and therefore decrypt the necessary data without cooperation from other account members.

3.9 Highlighting Untrusted Devices with Contact Sync

Note: Contact Sync is not currently available. We plan to release it in a future update. The description here is specific to E2EE Meetings (Section 7), but similar guarantees can be extended to other products.

Even with a strong concept of identity, impersonation attacks are still possible in meetings. If Bob's coworker, Alice, has the email `alice@company.org`, Bob might not realize if he joins a meeting with an impersonator using `al1ce@company.org`, especially if the impersonator has their video turned off. Or if a hacker stole Alice's username and password, they could provision a new device and pretend to be Alice. We'd like to have a TOFU-style UI feature to let Bob know if it's the first time he's seeing a device in a meeting—but we also don't want to bother Bob for his first meeting with Alice on every new device that Alice makes, which could lead to alert fatigue. Specifically, we want to provide warnings when Bob is in a meeting with a device that is not approved by a device that Bob has seen before. We'd also like Bob to share his meeting history between his devices securely so they all have the latest information.

Each device maintains, for every other user that it has been in a meeting with, a record containing that user's user sigchain tail, the time of their first meeting together, and the total number of E2EE meetings they were in together. These records are updated after each meeting as appropriate, though they are only updated when either

1. The meeting has less than 25 participants and the device has been in the meeting with the participant for over 10 minutes
2. The participant has been speaking for over 30 seconds

In order for the meeting history to be shared across devices, clients periodically send encrypted meeting records to the server. Records are individually encrypted with the latest PUK, signed with the device key, and tagged with $t = \mathsf{HMAC}(k, \mathsf{userID})$, where k is a key derived from the PUK. The server stores a mapping between $(\mathsf{deviceID}, t)$ and the encrypted record, which the clients can update as necessary. When the client learns of a new PUK, records and tags are updated to use the new PUK lazily.

The records are encrypted to minimize the privacy loss in case they are leaked (Zoom servers already learn who is participating in which meeting). Because the tag is generated deterministically, these records would reveal if two of Alice's devices had meetings with the same user (but not which user), given that both devices used the same per-user key. However, we find this tradeoff acceptable as it allows for a more efficient synchronization. Further, the server has the ability to rollback a device's record at any time, but doing so could only cause additional warnings.

When joining a meeting, clients generate tags for each other participant using each known PUK and request the server to send any corresponding records. Any record signed by a revoked device is not considered. Given this data, we can provide warnings for each device in a meeting:

1. If the device has been seen before, or if the device is trusted by a device that has been seen before as indicated by the trust graph, display the number of meetings with this user in the last month.
2. Otherwise, if the device is untrusted but the user has been seen before, display "This is the first time you are talking to this device of this person."
3. Otherwise, display "This is the first time you are talking to this person."

Note that devices may not have access to all the per-user keys used to encrypt records, which could result in extraneous warnings. For example, imagine Alice provisions Zoom on her phone and has a meeting with Bob. Then, Alice provisions Zoom on her laptop and has another meeting with Bob without approving the laptop from the phone. In this case, Alice's laptop does not have the PUK used by Alice's phone to encrypt the record, so there will be another warning for Bob in the second meeting.

Though Contact Sync warnings improve Zoom users' security, alert fatigue can make the warnings less useful. If a user's contacts regularly fail to approve new devices with earlier ones, or if the server does not properly sync records of prior communication, users can grow accustomed to the warnings and possibly ignore them in the event of a real attack. Even without an attack, excessive warnings make for a suboptimal and tiresome user experience.

3.10 Compromise Prevention for Device Provisioning

Note: The features in this section are not currently available. We plan to release them in future updates.

With the ZTT (see Section 4), we have a mechanism for users to detect sigchain corruptions

after-the-fact. For instance, if an insider adds a new device onto Alice's sigchain to impersonate her in a meeting while she is offline, Alice will detect the attack after she comes back online, since the attacker had to commit the malicious change to Alice's sigchain to the ZTT. Ideally, Alice could prevent unauthorized device additions from happening in the first place. Note that users whose identities are vouched for by a trusted third-party IDP (Section 5) already have good protections as long as their IDP is not compromised. We want to offer a strong alternative to all users that does not rely on external parties.

Recall that users currently login to Zoom via SSO, OAuth, or simple username and password. Their devices' keys are authenticated by multiple signatures: a self signature, an optional approval signature by an older device (Section 3.4), a signature by Zoom (as part of the inclusion in the ZTT, once it is deployed), and an optional attestation from the IDP.

Further validations of the device keys are possible. For example, users can sign new devices with existing devices at the time they are first added (as opposed to in a later approval), potentially through a QR-code-based flow similar to the one Keybase has now. One device shows a QR code of a random session key, and the other scans the QR code; this establishes an end-to-end secure tunnel between the two devices that a compromised server cannot interfere with. Over this tunnel, devices can exchange public keys and cross-sign each other.

Alternatively, organizations can set up policies that require an IT administrator to sign off on device additions. The new user device can display the hash of its public key in QR code form. The IT administrator can scan it, signing the new device with their administrator's signing key.

Devices without the required signatures might be prevented from signing on behalf of the user or rotating the user's PUKs, thus denying them access to any content encrypted for the user. They might also trigger extra UI warnings to other participants when joining meetings, or even be excluded from meetings marked "high security."

3.11 Security Properties

The identity and key management system described in this section is leveraged by multiple Zoom products, and it provides some lower level security properties that these products build upon.

First, we note that the secret keys corresponding to any device public keys included in any sigchain are known only to the device that generated them (unless an attacker has somehow gained access to the device's memory or storage): secret keys never leave the device,[6] and are only used to perform encryption and signing. A consequence is that ciphertexts/signatures for/by each of the device keys can only be decrypted/created by the device that generated these keys. Virtual device keys are an exception. For example, backup keys are shown once to the user as text strings that can be written down, and therefore our guarantees depend on these keys being kept securely by the user, and not

[6] Except possibly when the user voluntarily sends crash reports to Zoom. We try to minimize this risk, but cannot exclude it.

being available to an attacker. A second exception is escrow keys, where the escrow device secret key is possibly generated by one of the user's own devices and encrypted for the account's escrow admin's PUK: this gives the EA's devices the same access to the user's keys as the user's own devices.

Similarly, consider any valid user sigchain. The secret keys corresponding to any per-user public key that appears in that sigchain are only known to any devices that were added but not revoked before the per-user key was added to that sigchain, plus any devices that were added afterwards and approved by one of those devices (as recorded in each device's own view of the user's sigchain). If escrow is enabled or the user creates backup keys, some EA devices and those with access to the backup keys might also learn these PUKs.

Also, note that the Zoom server cannot force any devices to forget identity updates like device revocations: when receiving sigchains, devices only accept new sigchain links that extend the ones they are already aware of.

3.11.1 MitM Between a User's Devices

These properties allow us to obtain strong guarantees about the confidentiality of the data that is encrypted for a user's device keys and PUKs. For example, consider any ciphertext (such as an email draft) encrypted for a given per-user key whose secret key is only known to a set of uncompromised devices controlled by the corresponding user (and the users who have permissions to manage the escrow admin virtual user, if escrow is enabled). In order to obtain the plaintext, an adversary (even an insider) would have to either break the encryption scheme, compromise one of the user's (or escrow admin's) current devices or backup keys, or trick the user into using one of their devices to approve a maliciously-controlled device (so that the existing device encrypts the PUK secrets for the malicious one). If escrow is enabled, an adversary could also compromise one of the escrow admin's device keys, or a malicious insider could trick the escrow admin user into approving a maliciously-controlled device.

A cautious user would only approve an additional device on their sigchain in a situation where they expect such a request: for example, after they sign in on Zoom for the first time on a new device, or in the case of enabling escrow, after they receive a trustworthy notice from their account administrator that escrow is being enabled. An insider could take this as an opportunity to attempt a MitM attack, by lying about the new device's public key to the old device. However, outsiders cannot perform this attack: even if the attacker had access to the user's credentials, the attacker would have to add an extra additional device and cannot prevent the legitimate one from appearing in the approval modal. Comparing the sigchain fingerprints as known to the approving device and the new device before confirming approval (and comparing the EA's fingerprint before adding or approving the virtual escrow device) also prevents insiders from performing this attack. Because the sigchain is an append-only data structure, comparing fingerprints after approval would still allow the user to detect the attack, providing evidence of server compromise/misbehavior and allowing the user to mitigate its effects.

In the future, the Zoom Transparency Tree will offer (assuming trusted auditors) an automated and transparent way to ensure that devices have a consistent view of all sigchains

and public keys such that checking fingerprints before device approvals will no longer be critical.

3.11.2 MitM Between Different Users

Another class of attacks involves an MitM between the sender and recipient of an encrypted communication. An insider might lie to the sender about the recipient's sigchain and PUKs or add a malicious device to the recipient's sigchain, causing the sender to encrypt for a key controlled by the attacker. These attacks can be prevented or detected by checking fingerprints, and can in many cases be detected after-the-fact with minimal user burden by leveraging the ZTT.

Consider an adversary compromising one of the recipient's devices or backup keys, thus obtaining their secrets. With access to the device, an attacker could obtain all the recipient's data, both locally cached and stored by the server (at least until their session expires). If the user discovers the compromise, they could revoke this device and rotate their keys, preventing further data leakage. However, if the attacker also compromises (even at a later point) the Zoom server infrastructure, they might attempt a more subtle attack: when a sender requests this recipient's sigchain, the server could provide a legitimate but stale version of this chain, i.e., one that does not include the device revocation and the subsequent rotation. This attack can be prevented by comparing fingerprints. However, detecting this compromise using fingerprints after the fact is not that straightforward: the sender and recipient would have to ensure that they agree on their views of the recipient's sigchain at the time the *message was encrypted and sent*, rather than at the time of the fingerprint comparison. Assuming all clients have synchronized clocks, one way to achieve this would be to have each sigchain link include a timestamp. Each client would remember the last time they updated their view of any sigchain and refuse to accept new links for that sigchain that have an earlier timestamp. In order to be tolerant to time misconfigurations, we currently do not enforce these properties, but we are considering them for future updates.

Eventually, the ZTT might also ensure, under similar time synchronization assumptions, that everyone's view of all sigchains is not only consistent, but also relatively up-to-date, i.e. that any attempt to withhold updates beyond some reasonable tolerance bound is detected. This will ensure that the server is not able to trick senders into encrypting for out-of-date keys unnoticed.

3.11.3 Integrity

Ensuring that all devices have a consistent view of a user's sigchain and that no extraneous devices have been added to it (by comparing fingerprints or relying on the ZTT and monitoring one's own sigchain) also helps with integrity guarantees. Users can ensure communications haven't been tampered with by checking that they are signed by devices belonging to the claimed author. However, as above, in some cases the evidence of compromise might be less conclusive: if the sender signs a message using a device's signing key, and later revokes that device (for example, because the device was lost or compromised), the recipient has no way to tell if the message was signed before or after the revocation/compromise. The user interface may communicate this potential risk to the

recipient so that they can confirm the integrity of any sensitive communications with the sender out-of-band, or ask them to resend the message with an up-to-date key.

3.11.4 Security Limitations

While our concept of identity provides strong security guarantees, there are still attacks it won't be able to prevent.

An attacker who is able to register a new device in the system on behalf of a non-consenting user, whether by stealing the user's credentials or compelling the server, will be able to read data that is E2E-encrypted to the targeted user after the compromise (but not before, as explained above), as it will be able to add a new compromised PUK to the user's chain, which might be used by other users to encrypt messages intended for the recipient before the recipient has had a chance to come online to review and revoke the new device. This attack can be prevented if the sending user compares fingerprints out-of-band with the potentially compromised recipient(s) before encrypting for the new PUKs, or detected by comparing fingerprints (possibly implicitly through the ZTT) after the fact.

When one of a user's devices is revoked, the per-user public encryption keys for that user become stale and should be rotated as soon as possible. When another of the user's devices comes back online, they will pick new encryption key pairs, publish the public keys in a new sigchain link, and encrypt the secret portions for all the user's devices that are still active. If one of the user's devices is revoking another one, key rotation can happen immediately. However, in case of self-revocation or revocation from the web or by an account administrator, if none of the other user's devices are online, this rotation might be delayed. A delay can also happen if escrow is disabled, or the user is moved between accounts (with different escrow administrators) while being offline. Any data encrypted to this user during this period would be using an encryption key known to a revoked device (or to old escrow administrators' devices), and we rely on the server to enforce that these devices can no longer access user data. An attacker who both compromised a revoked (or escrow administrator's) device *and* had read access to the server might thus be able to decrypt this data. Although the attack window seems limited and hard to exploit in practice, this is a limitation of the current design. Similarly, when a key is revoked and rotated out, we do not re-encrypt old ciphertexts for the new key, and rely on the server for access control to the ciphertexts.

We stress that, while escrow enables necessary enterprise features such as account recovery and data retention, EAs' devices have keys that allow them to decrypt all data in their account: as such, this privilege should be restricted to a few security conscious users to minimize the potential for compromise.

3.11.5 Privacy Limitations

From a privacy perspective, in addition to the identifiers displayed in the user interface, our solution provides some limited extra information about a user to other users they interact with: the sigchains reveal the history of the user's devices, including when they were added and revoked (but not their names, which are protected behind a commitment), as well as which device was used to sign any specific E2E-encrypted communication. Similarly, the

number of times that a user changes their email address or account is visible (but not the previous emails or account IDs). Moreover, since the server might report the timestamps of sigchain statements (and, once deployed, the ZTT will also necessarily reveal the same information), time correlations between different statements might be exploited to infer, for example, that two users swapped their email addresses, or that two users are in the same account. While this information is not displayed in the user interface, the client needs this data to perform the sigchain validation and therefore a motivated attacker might extract such information. We believe that this is acceptable; it is similar to the security code change warnings in applications like Signal and WhatsApp.

Note that sigchains are not publicly available and are subject to server-side access control: they are provided as needed to users. For example, if Bob is a Zoom Mail Service user and knows Alice's email address, they can ask the server for the sigchain associated with that address in order to encrypt an email. We currently rate limit requests for users' sigchains and other personal data as a partial mitigation for this leakage.

In the future, we are considering a different mechanism that allows the server to offer "randomized" versions of a user's encryption keys, in a way that trades the opportunity to check a user's fingerprints (while still being able to detect impersonation after the fact) for increased privacy.

4 Transparency Tree

Note: The Zoom Transparency Tree is not currently available. We plan to release it in a future update.

In this section, we describe a mechanism that expands the authentication guarantees from Sections 3 and 5 to ensure that all Zoom users have a consistent view of each others' devices and keys.

Imagine an insider, Mallory, who wants to eavesdrop on a meeting between honest users Alice and Bob, who have never interacted on Zoom before and haven't checked the meeting leader security code. To succeed in this attack, Mallory could instruct the Zoom server to lie to Alice about Bob's keys and to Bob about Alice's keys, replacing them with keys she controls. If Bob's client is the only one to see the fake key for Alice, and similarly Alice's is the only client who gets the fake key for Bob, then such an attack would be hard to detect after the fact.

Some possible countermeasures for such attacks require trusted external entities or manual validation steps (such as checking the security codes described in Section 7.7) that potentially have to be performed out-of-band. Instead, we will be able to detect equivocation by Zoom servers and identity providers while minimizing active checking by the user.

To do so, we will ensure that Zoom servers provide the same mapping between user accounts and public keys to all clients, sign such a mapping, and are held accountable for these signed statements. This way, in order to compromise a single meeting, Zoom would have to lie

not only to Alice about Bob's keys (and vice versa), but also to every other Zoom user about those keys, including lying to Bob about his own keys. Bob's client can thus easily review the list of his devices and discover any suspicious activity. External auditors will routinely verify that the server's mapping is consistent over time.

Thus, key fingerprint comparisons and other related warnings can be demoted in the user experience, to be replaced with targeted security alerts (which we expect never to be triggered). Key security becomes virtually invisible to the user.

4.1 Zoom Transparency Tree

The idea that there should be a single and consistent mapping between an identity and its public keys has already been explored successfully to solve similar issues. Most notably, Certificate Transparency [14] limits the damage that a compromised certificate authority can do by signing fake TLS certificates. It does so by requiring that all signed X.509 certificates must be submitted to a publicly auditable log before being accepted by browsers. Industry projects such as Key Transparency [1] and Keybase [2] (which is now part of Zoom), and academic works such as SEEMless [7] and CONIKS [15] have explored applying a similar approach to individual users' identities for messaging applications, with Keybase being the only instance in production use today, as far as we know. However, all the existing solutions in this space that we are aware of do not currently match Zoom's security and privacy requirements while offering usability features like multi-device support.

We build on prior work to design a new mechanism tailored to Zoom's use cases: the Zoom Transparency Tree (ZTT). The ZTT is backed by a Rotatable Zero-Knowledge Set [8], which is similar to the Merkle tree as used by Keybase, but with privacy-preserving path-lookup features such as in CONIKS. This data structure offers a key-value store interface where key-value pairs, once inserted, cannot be removed or altered. The state of the structure can be summarized by a small commitment, and lookup queries can be accompanied by a short proof that they are consistent with the commitment. Whenever a client is given a signed sigchain statement (as introduced in Section 3.5) about another user's identity or their keys, this statement is accompanied by an inclusion proof in the ZTT.

4.2 Integration Details

4.2.1 ZTT Auditing

The design of the ZTT requires auditing to verify the structure of the tree. Zoom will partner with independent external auditors which (in a privacy-preserving way) ensure that the append-only property of the ZTT is respected.

Clients query the auditors to ensure that their view of the ZTT's commitment has been audited and is consistent with everyone else's. If the client can reach the auditor and detects a fork in the ZTT, they can send the auditor the forked and signed commitments in addition to the warning, so that the auditor can disclose the inconsistency. If Zoom clients cannot reach any of the auditor servers, they will signal a degraded encryption

level.

We will publish code so that interested parties can also audit the ZTT.

Additionally, organizations using Zoom will be able to review updates to the ZTT and track their employees' device changes.

4.2.2 Provisioning

When provisioning a new device, the client ensures that the sigchain statement is included in the ZTT by first sending it to the Zoom servers and then querying the ZTT to check that the sigchain update has been included.

4.2.3 Self-Audit and Refresh

Periodically, the user's client should ask the server for an updated ZTT commitment, ensure that this commitment is consistent with past data, possibly verify it with external auditors, and review the user's sigchain for any new statements. If new keys are added to the sigchain, the client should ask the user to review the changes. If the user notices an unexpected change, they may be prompted to change their password or talk to their IT department.

4.2.4 Validating User Identity

Because we only trust keys stored within the ZTT, users can verify that each others' public keys are included in the ZTT, before proceeding with using the keys.

4.2.5 Contact List Updates

The contact lists that users accumulate, described in Section 3.9, can also be stored in the ZTT. The immutability guarantees that the ZTT provides means that Alice can note an update to Bob's identity in the client installed on her phone, and Zoom is obliged to relay that update to her desktop client.

4.3 Security Properties

Though the ZTT allows an equivocating server to be detected, we rely on the user to validate device additions. Users might be offline or might be ignoring notifications and therefore compromises might not be detected, or only detected after an attack.

The ZTT requires external auditors to provide security guarantees. If the auditors are not honest, or have poor uptime, this can limit the ability to detect server misconduct. We can mitigate this risk by relying on multiple auditors or implement partial auditing by clients.

5 Identity Provider Attestations

Note: As of version 5.13.10, we only support Identity Provider attestations that are issued by Okta and are displayed in the context of E2EE meetings. This feature is called "Okta Authentication for End-to-End Encryption".

Accounts that have an ADN and a compatible identity provider[7] (IDP) are able to have the IDP vouch for their users' identities in a way that other Zoom users can independently verify. This mechanism restricts the ability, even for Zoom insiders, to impersonate account members. Many organizations already trust an IDP for authentication purposes, so this feature does not increase the attack surface or require additional trust in the IDP.

In order for clients to be able to verify identity attestations by an external IDP, we need two components:

1. A way for clients to determine the IDP associated with a Zoom account (that cannot be tampered with by the Zoom servers)
2. A mechanism for IDPs to issue—and for clients to verify—a signed attestation that binds a user's email address to the cryptographic key(s) they use to communicate (see Section 3)

5.1 Associating Accounts with Identity Providers

In order to associate an account to its IDP, the account's ADN hosts a DNS TXT record, per cloud, pointing to the corresponding IDP domain. Specifying the cloud identifier in the record supports ADNs configuring different IDPs for different clouds. Note that since accounts are expected to change their IDP rarely, clients can cache this mapping aggressively.

For example, if an account hosted on the Zoom commercial cloud is using `example.org` as their ADN and `generic-idp.com` as their IDP, then the DNS entry for `example.org` should include a TXT record with a value like:

```
v=zoomadn us.zoom.idp.commercial=examplecorp.generic-idp.com
```

As part of the process of validating IDP attestations (see Section 5.4), clients request the IDP domain value from the Zoom server, and compare it with the value returned in the account's DNS TXT record. In addition, once the ZTT is deployed, clients will also check that this information matches what is in the ZTT for auditing purposes. If the values do not match, then clients won't complete verification of or display any identifiers for the account's users.

[7] Compatible identity providers need to support a custom extension of the OpenID Connect (OIDC) protocol which we describe below.

5.2 IDP Attestations

IDP attestations are generated and verified according to the OpenID Connect (OIDC) protocol. OIDC is an extension of the widely used OAuth 2.0 authentication protocol, an industry standard that many IDPs and Single Sign-On providers already support. OIDC provides a standardized format, the ID token, to express claims about identities and their attributes. It also specifies how users can request attestations for their own identity and verify ones obtained from other users.

We customize the protocol by:

1. Introducing an additional attribute `"zoom-identity-snapshot"` to the ID token in order to encode the state of a user's identity on Zoom. The IDP keeps track of the latest value of this attribute for every account user.
2. Specifying how this attribute can only be updated by the authorized user, and not by any other user or entity, including Zoom servers.
3. Specifying how this customized ID token for a specific user identity can be validated by other users.

Our modified OIDC ID token (which we will also refer to as an *IDP attestation*) is a signed JSON Web Token (JWT) data structure which contains claims about a user's Zoom identity. The payload might look like the following:

```
{
  "iss": "https://examplecorp.generic-idp.com", // issuer

  "email": "alice@example.com",

  "zoom-identity-snapshot": "409788...",

  "exp": 1311281970 // expiration time
  "iat": 1311280970, // issue time

  [...]
}
```

The token contains an `issuer` field which identifies the OIDC issuer of the token (the IDP). In order for Zoom clients to accept an identity claim, the `issuer` field must match the IDP domain associated with the account. `iat` and `exp` specify the validity of the token. The `zoom-identity-snapshot` field encodes the state of a user's identity, such as the user's sigchain-backed identity; it is treated by the IDP as an opaque string, and the IDP does not need to check its validity (the identity snapshot is further described in Section 5.5).

The fetched attestation may be shared with and validated by anyone, so it doesn't include the `aud` field. Unlike standard OIDC ID tokens, which are "bearer tokens" in that they can be presented by a user to prove their identity to the party indicated in the token's

`aud` field, our attestations are meant to be exchanged with potentially many other clients to prove a binding of a particular client's identity to their keys. Because compliant OIDC implementations enforce that the `aud` field contains their own identifier, they will not accept our attestations (which lack this field) as a standard ID token for authentication purposes.

The `email` value must match the email address that the user logged into Zoom with (and that the server gives to the verifying client) in order for validation to succeed. Currently, the `email` field is required; in the future, we might make this field optional to allow for proving that a user is a member of a specific account without revealing their full identity.

5.3 Updating Snapshots

In order for a user to receive a valid OAuth access token to read and update their `zoom-identity-snapshot` attribute, they must successfully complete the OAuth 2.0 Authorization Code Flow with PKCE [16] with their IDP for the same email address they use to login to Zoom. This is implemented as a second, separate flow from the Zoom user login flow. To ensure that only an authorized user on a Zoom native client is able to update the identity snapshot stored by the IDP, our protocol requires IDPs to:

1. Introduce new OAuth scopes `idpSnapshot.manage` (required to update one's snapshot) and `idpSnapshot.read` (required to read one's snapshot).
2. Issue access tokens with the `idpSnapshot.manage` scope only for requests that use PKCE, and where the redirect URI is one of the fixed custom URIs intended to refer to the native Zoom desktop and mobile clients (such as `zoommtg://zoom.us/oauth2`).
3. Issue access tokens with the `idpSnapshot.read` scope, regardless of the specific OIDC flow. This scope supports future use cases where the Zoom server can fetch valid user attestations without modifying the attested keys.

With the custom URI redirect, we trust the operating system and browser to redirect to the native Zoom app and not to a website in a browser: such a website might be serving malicious JavaScript from a compromised web server that could hijack the authorization flow. PKCE is an OAuth 2.0 extension that prevents other apps installed on the user's device from intercepting the authorization code. The Zoom app will not share the resulting "write" access token with anyone else, including the Zoom server, but read-only access to snapshots can be extended to all access tokens, including those issued to browser sessions.

After the client has obtained an access token which includes the `idpSnapshot.manage` scope, they can use it for a dedicated IDP API endpoint to request an attestation with an arbitrary `zoom-identity-snapshot` value.

We realize that the protections given to write ID tokens depend on the security of the underlying platform including the user's browser, their OS and their hardware, but we intend these protections to be best effort measures.

5.4 Validating IDP Attestations

IDP attestations can be validated like standard OIDC ID tokens in the Authorization Code Flow [3] with a few modifications. Users must:

1. Verify that the IDP domain in the `iss` field of the JWT matches the IDP in the ADN's DNS TXT record and the IDP returned by the Zoom server. This ensures that the IDP is authorized to sign on behalf of the account ADN (as specified in Section 5.1).
2. Use OpenID Connect Discovery to ensure that the key used to sign the JWT is valid. To do this, make a request to `https://examplecorp.generic-idp.com/.well-known/openid-configuration`) for the JWKS, the keys used to sign the OIDC ID token.
3. Validate the JWT, including checking its signature and expiration date.
4. Validate that the `zoom-identity-snapshot` value and email address match what is provided by the Zoom server.

We take several measures to improve the security and privacy of the attestation validation process.

To ensure confidentiality and authenticity of DNS queries in step 1, clients obtain the DNS TXT record by making a DNS over HTTPS (DoH) request to the DNS resolver 1.1.1.1 (operated by Cloudflare), using Cloudflare's JSON-formatted DoH API endpoint.

We have deployed Zoom-hosted proxy servers, described below, to hide client IPs from external servers in steps 1 and 2. However, a client's proxy and network configurations may prevent the client from successfully connecting to our Zoom-hosted proxy. If the Zoom client detects a proxy, it will attempt sending requests both via the detected proxy (which reveals its proxy's IP to the target domain), and via the Zoom proxy. To reduce latency, the client will send the requests simultaneously and use the response that returns first. If neither request succeeds, then attestation validation will fail.

In more detail, the client uses the Zoom-hosted proxy servers in step 1 when sending a DoH request to a supported DNS resolver, and in step 2 when requesting JWKS belonging to other users' IDPs. The client first establishes an HTTPS connection to the proxy service, over which it sends a `HTTP CONNECT` request to establish a tunneled connection to the target domain. The proxy server authenticates the request, and also confirms that the target domain is an expected domain (e.g. that it is a subdomain of a known IDP domain). The client now establishes a HTTPS connection with the target server through the tunnel. This allows the client to use TLS to fetch the requisite data, but not reveal the client IP address to network eavesdroppers or the target domain. In order to mitigate abuse, the proxy checks that the target domain (in the TLS ClientHello) and the target port (in the HTTP CONNECT) are expected, but is not otherwise involved in the standard TLS handshake and certificate validation process for the target domain.

5.5 Zoom Identity Snapshots

The `zoom-identity-snapshot` field of an IDP attestation binds the email and ADN to the set of cryptographic keys that the client will use in their interactions and, in some cases, to their whole cryptographic identity by including the user and account sigchain tails.

For example, in the context of E2EE meetings (Section 7.10), the snapshots currently include only the identity verification key IVK used by the device posting the attestation to join that specific meeting. In the future, once we start leveraging sigchains for identity in E2EE meetings, we will also include the sigchain tails in the snapshot.

There are several advantages of binding to the full cryptographic user identity: devices will be able to share a single attestation, which would need to be updated only when it expires (a configurable interval) or when there are changes to the user's identity, thus simplifying interactions with identity providers and improving the user experience. Note that these attestations of sigchain tails are not considered valid if the sigchain's user has not reviewed all changes to their identity (e.g. reviewed newly added devices). In addition, because these attestations would cover the full history of a user's cryptographic identity, any past misbehavior by the server is more likely to be detected.

5.6 Security Properties

When clients verify an IDP attestation, they obtain the following informal guarantee: the user who claims to control the cryptographic keys included in the attestation has successfully authenticated to the given IDP (w.r.t. to the email address in the attestation), and the owner of the attestation's ADN domain has delegated the IDP to make these statements on the account's behalf.

This property relies on several assumptions. Beyond the security of the cryptographic primitives, the verifier relies on the web PKI to obtain the ADN's DNS TXT entry from the DNS resolver Cloudflare over DoH, as well as the IDP's public keys used to verify the attestations. A fraudulent attestation may also be produced upon compromise of the attested user's device or of the user's authentication credentials (including any two-factor authentication, if enforced by the IDP), or by compromising the IDP's server infrastructure.

Clients verify Zoom's suggestion for an ADN's preferred IDP through a DNS query. While relying on TLS (similarly to how JWKs are fetched) or DNSSEC would prevent some attacks from compromised DNS providers or network attackers, using a DNS TXT record eases the setup for our customers. We rely on Cloudflare as a trusted DNS resolver to confirm that the IDP indicated by Zoom is indeed the one the ADN owner intends to use. The DNS resolver can convince clients that a certain account is not using an IDP by e.g., returning a DNS NXDOMAIN message, or a wrong IDP value: in this case, because an honest Zoom and the ADN have returned different IDP domain values, the client will not consider the attestation valid and will not display the related information in the user interface. We stress that control of the Zoom server infrastructure alone is not sufficient to convince clients of a wrong IDP value. In the future, with the transparency layer offered by the ZTT (Section 4), any server tampering of an account's ADN or IDP will be detected.

We plan to offer users other options for DNS resolvers in addition to 1.1.1.1 in the future.

From a privacy perspective, users who opt into sharing attestations share a binding of their email address and the long-term device key(s) used by their devices, and thus reveal their user, account and device identifiers to their communication partners.

We attempt to minimize the information disclosed to third-party servers: by using the Zoom-hosted HTTPS CONNECT proxies, the verifying client can avoid leaking its client IP to supported DoH resolvers and other users' IDPs (for example, if a user who does not have an IDP is verifying another user's attestation from Okta, their requests through the proxy would hide the client IP from Cloudflare and Okta). As previously described, if the client has a proxy configured, its proxy's IP address may be revealed to the target domain.

The concrete properties users obtain when leveraging IDP attestations depend on the specific application. For example, in an E2EE meeting (Section 7.10), users are able to confirm that their meeting partners belong to a specific account (identified by a domain they know) even if the users have never previously met. Meeting hosts can leverage attestations to prevent MitM attacks without manually checking security codes. We discuss how attestations are leveraged in E2EE Zoom Meetings in Section 7.10, and plan to extend support to other products in the future.

6 Encryption for Zoom Mail Service

Zoom Mail Service is a Zoom-hosted product that allows users to send and receive emails. In this section, we describe the cryptographic design behind Zoom Mail Service, and how it supports end-to-end encryption where possible. In particular, given the lack of a modern and widely adopted end-to-end encrypted email standard,[8] and the desire to leverage the strong identity properties and the multi-device key management architecture described in the rest of this document, we only offer end-to-end encryption for emails sent between two Zoom Mail Service users (who have provisioned their clients to generate the required keys). Emails from external providers are encrypted at rest with client-controlled per-user keys as soon as possible after they are received, and the server deletes any unencrypted copy of emails to external providers after they are successfully processed.

Zoom Mail Client also supports external email providers, but here we only analyze the security of the product when leveraging our own Zoom Mail Service.

Note: Implementing end-to-end encryption in Zoom Mail Service is an ongoing process, and not all planned features and functionality in our design will be available as of version 5.12.8. This section describes the overall plan for the cryptographic design of Zoom Mail Service and identifies limitations that remain as of the current release.

[8] Though some open protocols exist, such as PGP, they have limited adoption and ease of use, a fragmented ecosystem, and frequent security vulnerabilities.

6.1 Encrypted Email Protocol

In this section, we describe the protocol that clients and the Zoom Mail Service server can use to encrypt and decrypt emails. The encryption format supports optional signatures over the emails: clients will sign all the emails they send to other Zoom Mail Service users using their device keys, while emails encrypted by the server (for example, those from external email providers) are not signed. For cryptographic purposes, an email consists of a sequence of opaque byte strings, which we call **pieces**; these may include the email body itself (which contains a MIME tree structure and email headers such as lists of "To" and "CC" email addresses) and attachments (we encrypt and treat pieces separately to allow, e.g., downloading attachments separately on demand). In addition, an email has a (possibly empty) BCC header string which lists the email addresses of any BCC recipients. The BCC header string is visible to the sender but not to recipients, and is used by the server for routing purposes.

To encrypt an email for one or more Zoom Mail Service users, the sender or Zoom server uses the following procedure:

1. Encrypt each piece with a fresh uniformly random 32-byte key (different for each piece) using Cake-AES.[9] Collect each key and the SHA-256 hash of each ciphertext[10] into a list called the **manifest**. Then, encrypt the manifest itself using Cake-AES with another fresh uniformly random 32-byte key, called the **shared symmetric key**.
2. Compute the $\mathsf{HMAC}_{\mathsf{SHA256}}$ of the BCC header string with a random 32-byte commitment key to produce the BCC commitment.
3. Generate a random Curve25519 ephemeral sender private and public key.
4. For each recipient (including the sender themselves):
   (a) Resolve the recipient's email address (using sigchains) into a recipient user ID and their email PUK (a public per-user Curve25519 key advertised in the recipient's sigchain, derived as described in Section 3.4.1). Compute a Diffie-Hellman shared secret between this key and the ephemeral sender private key.
   (b) Using SHA-256, hash the following information into the **recipient-associated digest**: all user IDs and public keys used by the sender and recipient, the hash of the manifest ciphertext, the BCC commitment, and other context such as the version number. The BCC commitment is only used by the sender to verify and display the BCC header string in their own outbox.
   (c) If the sender is a Zoom Mail Service user, then using Ed25519, sign this hash with the sender's private device signing key. (If the Zoom server is encrypting an email, omit this signature.)

[9] Cake-AES is a custom encryption format built on top of AES-GCM. Its ciphertexts commit to both the key and plaintext used to produce them, and it supports random-access decryption and large ciphertext sizes. See Appendix C for details.

[10] As an optimization, when forwarding or replying to an email containing an attachment, clients reuse the same ciphertext, key and hash related to the attachment in the new email to avoid having to download, re-encrypt and re-upload the attachment. The additional leakage is acceptable, given that the server could likely figure that the attachment is the same based on context and attachment size.

(d) Using $\mathsf{HMAC}_{\mathsf{SHA256}}$, derive a key from the recipient-associated digest and the Diffie-Hellman shared secret.

(e) Using XChaCha20-Poly1305 with this key, encrypt the shared symmetric key and, if it exists, the signature over the recipient-associated digest to produce a **recipient box ciphertext**.

5. Sign the list of recipient-associated digests (including BCC recipients) with the sender's private device signing key to produce the recipient list signature. The recipient list signature is only used by the sender's client to verify the email's recipients in their own outbox, and is not sent to recipients.

6. Send all ciphertexts, the BCC commitment and key, the recipient list signature, and the ephemeral sender public key to the server, along with the BCC header string and the other headers needed to route the email.

There are several use cases where the sender needs to give the server access to the email contents, which we detail below. To handle these cases, the sender follows the above procedure except that they create an additional recipient box using a fresh random public key as the recipient key and reveal the corresponding secret key to the server. This preserves the authenticity guarantees of E2E encryption for the other recipients, and also allows the sender to upload the encrypted email body and any attachments only once. In addition, the sender sets a flag in the email manifest indicating that the email is not end-to-end encrypted, which clients use to display the encryption type in the UI.[11]

When the server receives an email from a sender, it checks that the sender is logged in to an active device. It then puts a copy of the email in each recipient's inbox and a copy in the sender's "sent" folder. Each recipient's copy only contains their own recipient box ciphertext and, except for the sender's copy, doesn't contain the BCC header string, the BCC commitment key, or the recipient list signature. Although the sender's copy omits other recipients' box ciphertexts, it must retain the user ID and PUK generation of all other recipients to allow the recipient list signature to be verified. If there are any external recipients (such that the email is no longer E2EE), the server is able to decrypt the email and forward it to the external recipients.

Upon receipt of an email, the recipient follows these steps. Note that this includes senders viewing emails from their "sent" folder.

1. If a signature is present, load the signer's user sigchain to check that the sender signing key is valid. If the key is not the signing key of some device on the sigchain, decryption returns an error. Clients allow messages signed by revoked devices because the email could have been legitimately sent from the device before it was revoked, and rely on Zoom servers to prevent revoked devices from sending further messages. In future releases, messages signed by revoked devices will be highlighted in the user interface, so that users can be especially cautious. (Note that we allow unsigned messages even if the sender's email address is known to be using Zoom Mail Service: for example,

[11] We trust the sender to report this flag honestly. A malicious sender could modify their client to tamper with this flag, but doing so would only harm their own emails' privacy.

the sender might have sent this specific email from a different email service before migrating their address to Zoom Mail Service. Such emails are indicated as not E2E-encrypted.)

2. If the email being viewed was sent by the current user, assert that the BCC header string, the BCC commitment key, and the recipient list signature are all present. Verify the commitment, then recompute all recipient-associated digests and verify the signature. The BCC header and recipient list are now trusted and can be displayed in the UI.

3. Independently compute the recipient-associated digest and the Diffie-Hellman share between the recipient's key and the sender's ephemeral public key. Using that hash and share, derive the key to decrypt the recipient box ciphertext and get the signature over the recipient-associated digest and shared symmetric key. Verify the signature.

4. Using the shared symmetric key, decrypt the manifest ciphertext to get a list of piece keys and piece ciphertext hashes.

5. On demand, fetch each piece ciphertext, verify its hash, and decrypt it with its piece key.

Recipients do not receive or verify public keys for other recipients, but do see their email addresses in the "To" and "CC" headers.

6.2 Emails to Users without Devices

Some users may have a Zoom Mail Service account, but not have logged in on any devices or generated any keys yet. We say that these users are "pre-provision." For example, the IT team of an organization might create user accounts for every new hire before their start date. We wish to allow pre-provision users to receive emails, but these emails cannot be end-to-end encrypted because the users don't yet have keys. Instead, when emailing a pre-provision user, the sender flags the email as not E2EE and shares a decryption key with the server, as described earlier. The server stores these emails for the user until they create their first device, at which point the server decrypts and re-encrypts those emails for the user's first email PUK. Since the server performs the final encryption, these emails will not be signed by the original sender.

Pre-provision users are different from users who do have a sigchain, but whose devices are all revoked. The Zoom client doesn't allow sending emails to such users.

6.3 Emails to and from External Users

Zoom Mail Service users may send or receive emails from external users who aren't using the same platform. Such emails cannot be E2E-encrypted, as this encryption is not compatible with external mail providers; the server must see the email contents while receiving from or sending to external email providers using standard protocols. We nevertheless wish to provide the strongest feasible security guarantees.

When Zoom Mail Service servers receive emails from external users, they follow essentially the same procedure as a Zoom Mail Service sender to encrypt the email for each recipient's most recent PUK, except that they omit the sender public signing key and corresponding signatures, as previously described. After encryption, any plaintext copies of the incoming email are deleted in order to prevent later memory compromise from violating email confidentiality.

When sending an email that includes external recipients, clients flag the email as not E2EE and share a decryption key with the server, as described earlier. The server uses this key to decrypt the email and relay it to the external recipient using standard email protocols.

We also offer password-protected (sometimes called "expiring" or "access restricted") emails as a more secure option when emailing external recipients. The client freshly samples a high-entropy key (which we call a password), then creates an additional recipient box by encrypting the shared symmetric key for that password. Both the ciphertext and the password are sent to the server.

In lieu of the plaintext email contents, Zoom Mail Service sends a link to a Zoom-hosted web page to the recipient. The fragment[12] component of the link contains both the password and the recipient-associated digest (Section 6.1). After sending the email to the recipient, Zoom Mail Service erases the password from its memory and storage.

When the recipient's browser visits the link, the JavaScript on the web page validates the provided ciphertext against the recipient-associated digest, then decrypts the email client-side using the password. Neither the password nor the plaintext email contents are sent back to the Zoom servers.

Additionally, senders set an expiration time after which password-protected emails are automatically deleted by the Zoom server.

Password-protected emails offer several security benefits. The recipient's (and any intermediate) email servers never process email contents directly, limiting exposure, and all requests to visit the link can be logged by Zoom. If the Zoom server's storage is compromised after it sends such an email and deletes the password, the email ciphertext remains undecryptable. This does not rule out an active attack where a compromised server waits for the recipient to visit the link and serves them malicious JavaScript to obtain the password. However, this attack could only happen before the email expires and the ciphertext itself is deleted. We consider this an acceptable tradeoff given the convenience it provides to the recipient, who is not required to install additional software.

6.4 Mailing Lists

Zoom Mail Service supports mailing lists but, at the moment, membership is entirely managed by the server, and we do not support end-to-end encrypted emails to these lists. When a client sends an email to a mailing list, the server obtains the plaintext and re-

[12] The fragment of a link is the part after the `#`, which is not sent to the server when the user visits the link.

encrypts it for each member of the mailing list individually (as for emails from external senders). The re-encryption is the same as for emails that came from an external sender. We will continue to improve this design in the future.

6.5 Calendar Email Integration

Zoom Mail supports integration with other Zoom products. For example, Zoom Calendar Service may notify users over email when they are invited to an event. These emails will only be encrypted at rest, like emails from external services, rather than E2E-encrypted; they will be clearly indicated as such in the UI.

6.6 Encrypting Non-Email Data

Zoom Mail Service users can also have sensitive information in their user accounts that don't directly correspond to any email they sent or received. These include custom labels, folder names, and email signatures. The Zoom Mail Client encrypts all of this information for the user's own most recent email PUK, but the server still learns metadata such as the fact that two emails are in the same folder.

Email drafts are encrypted and decrypted in essentially the same way as they would be when they are sent/received, except that the sender does not create recipient box ciphertexts for recipients (other than themselves) until the email is actually sent.

6.7 Security Properties

Zoom Mail Service shares the goals, threat model, and limitations of Section 2 with the other products outlined in this document. In this case, we stress that Zoom servers get access to email metadata such as message size, send time, recipient list, number of attachments, attachment sizes, information on threading (e.g., whether an email is a response to a previous email), and in some cases (e.g., when replying or forwarding), whether two emails share the same attachment. Moreover, in future releases user reporting of potentially inappropriate content may share decrypted email content with Zoom servers, though this functionality is not available as of version 5.12.8.

The email encryption format is key-committing (each ciphertext, even if maliciously generated, can only be decrypted to a single plaintext) and guarantees both confidentiality and integrity, assuming the key material is kept private and correctly authenticated. Guarantees related to keys are inherited from the underlying key management architecture, as detailed in Section 3.11 (with the caveats below). For example, an attacker with read-only access to the Zoom server infrastructure and without access to any of the recipient's devices cannot violate the confidentiality of E2EE email contents, or of non-E2EE emails whose processing has been completed (i.e. that have been sent out[13] or encrypted for the recipient's keys) before the compromise began.

[13] Here we are only considering attacks to the Zoom infrastructure, excluding attacks to the external recipient's email account or on the mail delivery infrastructure beyond Zoom servers.

The user interface around fingerprints is still under active development (as of version 5.12.8, the product is still in beta). The current release only offers partial detection and prevention mechanisms against active attacks, which we are working to strengthen in the very near future. There is currently no way to mark that a fingerprint has been verified out-of-band so that the user gets notified when the corresponding sigchain changes. The guarantee that we offer is the following. Assume a user looks at a recipient's fingerprint (by hovering on their name/email address) while in the compose window, and then hits "Send" **without closing the compose window between the two operations**. If so, the client enforces that the email will be encrypted using the latest email PUK that is part of the sigchain corresponding to the recipient fingerprint that the user saw. If the sigchain is updated before the user hits the "Send" button, the send operation will fail so that the user has the opportunity to check the updated fingerprint. We stress that this mechanism does not work if the user closes the window between seeing the fingerprint and sending the email, which may occur when continuing composing from a draft, or across different emails sent to the same recipient. We deem this an acceptable temporary solution until we are able to offer comprehensive support for remembering "trusted" fingerprints and notifications of fingerprint changes. We will continue to develop a more robust and feature-rich experience around fingerprints and change notifications, as well as more automated solutions such as those involving the ZTT to reduce the burden on the user.

Lastly, while it is possible for the sender to check a recipient's fingerprint before sending, there is currently no intuitive way for recipients to check the fingerprint of the sender of a given email to verify the claimed authorship. Support for this use case is also upcoming.

Beyond key management, Zoom Mail Service does not make any guarantees regarding the following security properties:

- Deniability (as ciphertexts are signed with the sender's own device keys).
- Authentication of email threading, ordering, or the relationship between forwarded messages (i.e. unrelated emails could be displayed as if belonging to the same thread or out of order, some emails might be hidden, and any quoted text in a reply or forwarded email can be altered by the responder/forwarder).
- Authentication that an email has a particular label or is in a folder (e.g., whether an email has been "read").
- Rollback attacks on label/folder names or other encrypted non-email data (the server can always "undo" a rename by re-supplying the old signed ciphertext instead of the updated one).
- Forward secrecy (users cannot delete key material related to, e.g., a single specific email).
- Post-compromise security. We offer partial guarantees: encryption keys are rotated as soon as possible after a device is explicitly revoked, and ciphertexts encrypted after the rotation cannot be decrypted using the old keys.

We will prioritize improving on these issues depending on customer feedback.

6.7.1 Spam Detection and Content Monitoring

Confidentiality and integrity are not the only aspect of email security; some encrypted emails could contain spam, viruses, phishing messages, or other undesirable content. By design, Zoom servers cannot scan E2E-encrypted emails for such threats. Since sending E2EE emails requires a paid subscription, we expect abuse to be very limited. Moreover, even if the contents are inaccessible, user reporting and metadata-based techniques, such as rate limiting by account or IP address, can block or detect some abusive behavior. In addition, emails from external email addresses will be subject to server-side content analysis, including virus scanning and spam filtering using standard tools, as well as SPF, DMARC, DKIM and STS enforcement.

7 Encryption for Zoom Meetings

This section describes the cryptographic design that powers Zoom Meetings. We offer two types of encryption for Meetings: enhanced encryption encrypts the meeting stream with a key which is available to the Zoom server infrastructure, allowing for features such as cloud recording and live captioning, as well as the opportunity to dial into a meeting from a mobile or landline phone. End-to-end encryption, on the other hand, ensures that only the clients of the participants who are in the meeting have access to the meeting key. This offers stronger security and privacy, but lacks all the features that require the server to be able to decrypt meeting contents.

7.1 Zoom Meetings

A Zoom meeting is initiated by a designated individual, who we will refer to as the host. The host has the ability to configure meetings, notify participants, select meeting passwords, and control meeting functions while a meeting is in progress. The host's configured policies (e.g., whether meeting participants may share their screens by default) are applied to the meeting. The host need not be present for the entire duration of a meeting: if "Join Before Host" is enabled, individuals can begin a meeting before the host joins. Similarly, a host can appoint one or more additional individuals as co-hosts and can leave the meeting under the control of a replacement host.

Each Zoom meeting involves up to 1,000 participants. Meetings are identified externally by a short meeting identifier (the meeting ID), known to the Zoom infrastructure, which each participant must possess as a precondition for joining a meeting.

During meeting setup/scheduling, users can select enhanced or end-to-end encryption. This option cannot be changed once the meeting starts, and all participants receive a clear indication of the type of encryption during the meeting.

Zoom meetings also feature several access control mechanisms, among which are:

- A shared meeting password, which can be selected by the host at the time the meeting is configured.
- A "Waiting Room" feature, in which the host (and replacement host) has the ability to manually approve entry of participants throughout the course of a live meeting. Participants are identified by a name of their choosing.
- A mechanism by which meeting participants must register prior to the meeting.
- A setting to limit attendees of a meeting to those who are signed-in and authenticated members of certain domains.

We stress that these mechanisms are powered by the server, and might be circumvented by insiders.

7.2 Enhanced Encryption

When a Zoom client gains entry to a Zoom meeting with enhanced encryption, it gets a 256-bit per-meeting key (MK) generated by the Zoom server, which retains the key to distribute it to participants as they join. During meetings, each participant might produce different data streams such as audio, video and screen sharing. Since Zoom client version 5.0, each stream consists of UDP packets that are encrypted (in both enhanced and end-to-end encrypted meetings) using AES-GCM with a unique per-stream key, derived from the meeting key and a (non-secret) stream identifier using HMAC.

Those packets are relayed and multiplexed via one or more Multimedia Routers (MMR) in Zoom's infrastructure. The MMR servers do not decrypt these packets to route them, and use the per-meeting key only to provide the special features (such as PSTN dial-in and Cloud Recording) as detailed below. There is also no mechanism to re-key a meeting.

If a PSTN or SIP client is authorized to join, the MMR provides the per-meeting encryption key to specialized connector servers in Zoom's infrastructure. These servers act as a proxy: they decrypt and composite the meeting content streams in the same manner as a Zoom client and then re-encode the content in a manner appropriate for the connecting client. Zoom's optional Cloud Recording feature works similarly, recording the decrypted streams and hosting the resulting file in Zoom's cloud for the user to access. In the current design, Zoom's infrastructure brokers access to the meeting key.

This design provides confidentiality and authenticity for all Zoom data streams by providing encryption between Zoom client endpoints and the servers. However, it does not provide end-to-end key management. In meetings using enhanced encryption, a passive adversary who has access to the memory of the relevant Zoom servers may be able to breach confidentiality, by observing the shared meeting key, deriving session keys, and decrypting all meeting data. Zoom meetings with enhanced encryption, as well as virtually every other cloud product, rely on securing the server infrastructure in order to achieve overall security; end-to-end encryption, using keys at the endpoints only, allows us to reduce reliance on the security of the Zoom infrastructure.

7.3 End-to-End Encryption

Our end-to-end encryption design eliminates Zoom servers' role in generating (and controlling access to) the meeting key, shifting this responsibility to the participants' Zoom clients. No secret key material or unencrypted meeting contents will be provided to Zoom infrastructure servers, except in specific cases where this sharing is explicitly authorized by meeting participants (e.g., to support abuse reporting).

In end-to-end encrypted meetings:

- "Join Before Host" and Cloud Recording cannot be enabled.
- All participating clients must run the official Zoom client software; it is not possible to join using dial-ins, web browsers, or legacy Zoom-enabled devices.
- Participants can see a "meeting leader security code" that they can use to verify that no one's connection to the meeting is being tampered with. The host can read this code out loud, and all participants can check that their clients display the same code.

7.3.1 Security Goals

Zoom Meetings shares the goals, threat model, and limitations of Section 2 with the other products outlined in this document. Specific to Zoom Meetings, we additionally deem **in-meeting impersonation attacks** to be out of scope: a malicious but otherwise authorized meeting participant colluding with a malicious server can masquerade as another authorized meeting participant.

There are also several legacy standards and platforms that E2E encryption for Zoom Meetings is not compatible with. For example, dial-in phones or SIP/H.323 devices can be used to join Zoom Meetings, but these devices cannot be modified to support end-to-end encryption and require meeting content to be decrypted and re-encoded in an "end" in Zoom's data center. The E2E security guarantees described in this section do not apply to meetings that support such features.

7.4 System Components

Our end-to-end encryption protocol assumes the following components:

Identity management system. The system depends on the existence of a Zoom ID management system that will be responsible for distributing cryptographic public keys generated by individual clients. This server will bind keys to Zoom user accounts where possible, and will also support clients who do not have explicit Zoom identities.

Signaling channel. The system will make use of a signaling channel to distribute cryptographic messages between participants in a meeting. Currently, meeting participants route control messages on TLS-tunnels over TCP, through the MMRs. TLS is terminated at Zoom's servers. This channel is suitable for our needs.

Bulletin board. Participants in the channel can post cryptographic messages to a meeting-specific "bulletin board," where all other participants can see them. This

abstraction can be implemented over the signaling channel. The server controls the bulletin board, as it controls the signaling channel itself, and therefore can tamper with it.

Meeting leader. The protocol requires that, at all times, one of the participants plays the role of the meeting "leader." This client will have the responsibility of generating and updating the shared meeting key, and to distribute this key and other meeting metadata to the other participants. Zoom servers will select the leader, and replace them with a new one if the current one leaves. Note that the meeting leader role is different from the meeting host, who can authorize and kick out participants and has administrative control over meeting functionality. Before version 5.11.3, clients enforce that the leader is always the host, and this will still be the case in most meetings; however, newer clients allow the server to assign these roles independently to support functionality like E2EE Breakout Rooms, which do not have a dedicated host.

7.5 Cryptographic Algorithms

All meeting data sent over UDP gets encrypted with AES in GCM mode [11]. Key derivation uses the HKDF algorithm [13]. For public key encryption and signing, we rely on Diffie-Hellman over Curve25519 [4] and EdDSA over Ed25519 [5]. We use the interface and implementation of the NaCl [6]-inspired `libsodium` library [9], as detailed below.

7.5.1 Signing

For signing, we use `libsodium`'s EdDSA implementation directly:

- Sign.KeyGen generates a key pair (vk, sk) (via `crypto_sign_keypair`).
- Sign.Sign takes as input a context string Context and a message M and outputs a "detached" signature Sig over SHA256(Context)||SHA256(M) (via `crypto_sign_detached`).
- Sign.Verify takes as input a detached signature Sig, a context string Context, and a message M; it outputs true on verification success and false on failure (via `crypto_sign_verify_detached`).

7.5.2 Authenticated Public-Key Encryption

Authenticated public-key encryption also uses `libsodium`. Note that in encryption and decryption, we derive shared keys and use them to encrypt/decrypt the message as separate steps, firstly so we can cache the derived keys, and secondly because `libsodium` does not expose a function that directly supports using associated data in public-key encryption, only in symmetric encryption.

Box.KeyGen
Input: None
Output: an encryption key pair $(pk_{\mathsf{Box}}, sk_{\mathsf{Box}})$

To generate a key pair:

1. Return $(pk_{\mathsf{Box}}, sk_{\mathsf{Box}})$ as generated by `crypto_box_keypair`.

Box.Enc
Input: Sender's secret key $sk^{\mathsf{S}}_{\mathsf{Box}}$ and receiver's public key $pk^{\mathsf{R}}_{\mathsf{Box}}$, a context string $\mathsf{Context}_{\mathsf{KDF}}$, a second context string $\mathsf{Context}_{\mathsf{cipher}}$, metadata Meta, and a message M.
Output: a ciphertext C

To encrypt:

1. Generate a 192-bit random string $\mathsf{RandomNonce}$.
2. Compute $K' \leftarrow \texttt{crypto\_box\_beforenm}(pk^{\mathsf{R}}_{\mathsf{Box}}, sk^{\mathsf{S}}_{\mathsf{Box}})$, which is the DH key-exchange of the public key $pk^{\mathsf{R}}_{\mathsf{Box}}$ and the private key $sk^{\mathsf{S}}_{\mathsf{Box}}$.
3. Compute $K \leftarrow \mathsf{HKDF}(K', \mathsf{Context}_{\mathsf{KDF}})$, using an empty HKDF salt parameter. (K may be cached for this key pair and context.)
4. Compute $D \leftarrow \mathsf{SHA256}(\mathsf{Context}_{\mathsf{cipher}})||\mathsf{SHA256}(\mathsf{Meta})$.
5. Compute $C' \leftarrow \texttt{crypto\_aead\_xchacha20poly1305\_ietf\_encrypt}(M, D, \mathsf{RandomNonce}, K)$, which computes XChaCha20-Poly1305 over the plaintext M with the symmetric key K, the associated data D, and the nonce $\mathsf{RandomNonce}$.
6. Output $C \leftarrow (C', \mathsf{RandomNonce})$.

Box.Dec
Input: Receiver's secret key $sk^{\mathsf{R}}_{\mathsf{Box}}$ and sender's public key $pk^{\mathsf{S}}_{\mathsf{Box}}$, a context string $\mathsf{Context}_{\mathsf{KDF}}$, a second context string $\mathsf{Context}_{\mathsf{cipher}}$, metadata Meta, and a ciphertext C.
Output: a message M, or error

To decrypt:

1. Parse C as $(C', \mathsf{RandomNonce})$.
2. Compute $K' \leftarrow \texttt{crypto\_box\_beforenm}(pk^{\mathsf{S}}_{\mathsf{Box}}, sk^{\mathsf{R}}_{\mathsf{Box}})$.
3. Compute $K \leftarrow \mathsf{HKDF}(K', \mathsf{Context}_{\mathsf{KDF}})$, using an empty HKDF salt parameter. (K may be cached for this key pair and context.)
4. Compute $D \leftarrow \mathsf{SHA256}(\mathsf{Context}_{\mathsf{cipher}})||\mathsf{SHA256}(\mathsf{Meta})$.
5. Compute $M \leftarrow \texttt{crypto\_aead\_xchacha20poly1305\_ietf\_decrypt}(C', D, \mathsf{RandomNonce}, K)$. If decryption fails, output error. Otherwise output M.

7.6 Join/Leave Protocol flow

Each client needs a device signing key pair to join E2EE meetings: we denote the public verification key as IVK, and the secret signing key as ISK.

Once sigchain-backed identity for meetings (Section 7.11) is available, devices will directly use their signing key pairs as advertised in the user's sigchain. Until then, each device generates a dedicated IVK/ISK pair using $\mathsf{Sign.KeyGen}$ on their first login. In all cases, these signing keys are securely stored as described in Section 3.7.1.

If the user is joining a meeting as a guest (without logging in), this key pair is freshly generated for every meeting and never recorded in the sigchain. This prevents other participants from tracing them across meetings by noticing when a long-term key is reused.

We assume each meeting is identified by its unique meetingID, as in the current system. Each meeting gets its own "bulletin board" that's accessible to everyone who has server-gated access to the meeting. The server clears it when the meeting ends. Note that meetings can be ended then later restarted, and a meeting ID can refer to a standing or repeating meeting.

From a cryptographic perspective, the server is free to tamper with all values posted on the bulletin board. In Section 7.12, we describe further that a malicious server that sends stale messages from a previous meeting incarnation can at best deny service, which it can do regardless.

Figure 2 describes the basic flow of a leader admitting a participant into the meeting.

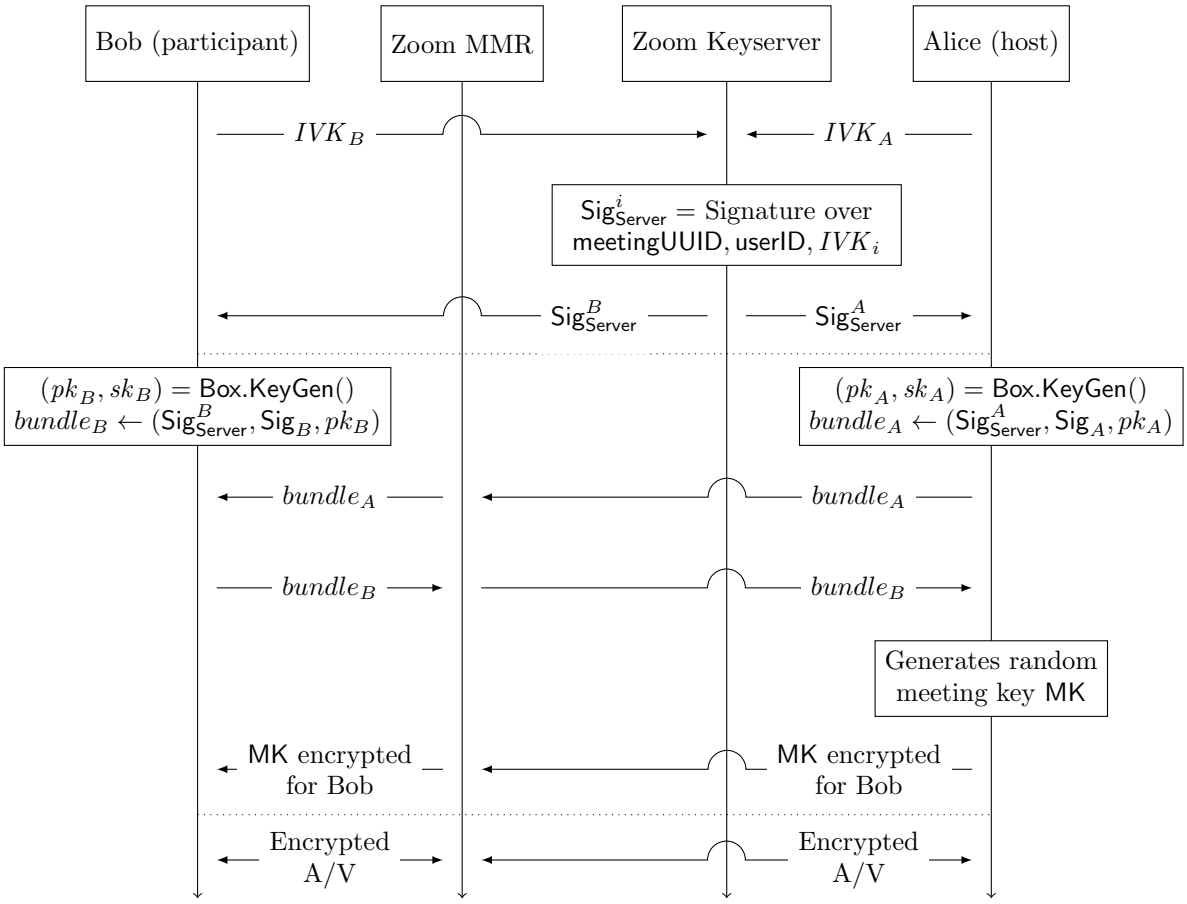

Figure 2: Protocol flow diagram for a leader accepting a participant into the meeting

7.6.1 Server Key Certificate Chains

When a client i joins a meeting, the Zoom server signs a statement $\mathsf{Sig}^i_{\mathsf{Server}}$ indicating that the client's userID, hardwareID, and IVK are authorized. userID and hardwareID are non-cryptographic identifiers used by the Zoom server to distinguish between users and devices.

We use certificate pinning to strengthen the security of the server signature. Zoom clients will ship with a DigiCert root certificate and they only trust certificates authorized for a specific Zoom domain via a certificate chain originating from the pinned DigiCert root. Hardware Security Modules (HSMs) are used to manage keys for an internal intermediate CA, which will in turn attest to the servers' signing keys. Server keys are valid for a week and are rotated daily. In order to detect certificate revocation in the event of CA or server compromise, clients require stapled OCSP responses on the intermediate certificates they receive.

These signatures help protect against MitMs injecting users into the meeting. This feature was released in version 5.7.0 (see Appendix A).

7.6.2 Participant Key Generation

When any participant i joins the meeting, whether before or after it starts, and whether the leader or not, it performs the following operations:

1. Generates new public-key *ephemeral* encryption key pair: $(pk_i, sk_i) = \mathsf{Box.KeyGen}()$.
2. Queries the Zoom infrastructure for the server-generated meetingUUID for this instance of this meeting; this is server-generated per-meeting-instance randomness that the individual participants cannot control.
3. Computes $\mathsf{Binding}_i \leftarrow (\mathsf{meetingID} \| \mathsf{meetingUUID} \| i \| \mathsf{hardwareID} \| IVK_i \| pk_i)$.
4. Defines $\mathsf{Context} \leftarrow$ `"Zoombase-1-ClientOnly-Sig-EncryptionKeyAnnouncement"`.
5. Computes $\mathsf{Sig}_i \leftarrow \mathsf{Sign.Sign}(ISK_i, \mathsf{Context}, \mathsf{Binding}_i)$
6. Stores sk_i for the duration of the meeting.
7. Posts Sig_i and pk_i to the bulletin board, so that all participants can see it.

7.6.3 Leader Join

When the leader joins the meeting meetingID, they:

1. Fetches meetingUUID from the Zoom infrastructure.
2. Generates a symmetric 32-byte seed MK using a secure random number generator.
3. Gets the full list of participants I from the MMR.
4. For each participant $i \in I$, it runs the "Participant Join (Leader)" subroutine for i.

Each MK has an associated sequence number mkSeqNum, starting at 1 and incrementing whenever the key changes as described in Section 7.6.6.

7.6.4 Participant Join (Leader)

Given a leader ℓ and a participant i joining meeting meetingID on hardwareID, the leader:

1. Fetches IVK_i from the key server.
2. Fetches Sig_i and pk_i from the meeting's "bulletin board."

3. Computes $\mathsf{Binding}_i \leftarrow (\mathsf{meetingID} \| \mathsf{meetingUUID} \| i \| \mathsf{hardwareID} \| IVK_i \| pk_i)$.
4. Defines $\mathsf{Context}_{\mathsf{sign}} \leftarrow$ `"Zoombase-1-ClientOnly-Sig-EncryptionKeyAnnouncement"`.
5. Verifies the signature: $\mathsf{Sign.Verify}(IVK_i, \mathsf{Sig}_i, \mathsf{Context}_{\mathsf{sign}}, \mathsf{Binding}_i)$.
6. If verification fails, it aborts.
7. Computes $\mathsf{Meta} \leftarrow (\mathsf{meetingID} \| \mathsf{meetingUUID} \| \ell \| i)$.
8. Defines $\mathsf{Context}_{\mathsf{KDF}} \leftarrow$ `"Zoombase-1-ClientOnly-KDF-KeyMeetingSeed"`.
9. Defines $\mathsf{Context}_{\mathsf{cipher}} \leftarrow$ `"Zoombase-1-ClientOnly-Sig-EncryptionKeyMeetingSeed"`.
10. Computes $C \leftarrow \mathsf{Box.Enc}(sk_\ell, pk_i, \mathsf{Context}_{\mathsf{KDF}}, \mathsf{Context}_{\mathsf{cipher}}, \mathsf{Meta}, (\mathsf{MK}, \mathsf{mkSeqNum}))$.
11. Posts (i, C) to the "bulletin board."

7.6.5 Participant Join (Non-Leader)

When participant i joins meeting meetingID, it performs the reverse of the above procedure:

1. Fetches IVK_ℓ from the Key server for the leader ℓ.
2. Fetches Sig_ℓ and pk_ℓ from the meeting's "bulletin board."
3. Fetches (i, C_i) from the "bulletin board."
4. Fetches the meetingUUID from the server.
5. Computes $\mathsf{Binding}_\ell \leftarrow (\mathsf{meetingID} \| \mathsf{meetingUUID} \| \ell \| \mathsf{hardwareID} \| IVK_\ell \| pk_\ell)$.
6. Defines $\mathsf{Context}_{\mathsf{sign}} \leftarrow$ `"Zoombase-1-ClientOnly-Sig-EncryptionKeyAnnouncement"`.
7. Verifies the signature: $\mathsf{Sign.Verify}(IVK_\ell, \mathsf{Sig}_\ell, \mathsf{Context}_{\mathsf{sign}}, \mathsf{Binding}_\ell)$.
8. If verification fails, it aborts.
9. Computes $\mathsf{Meta} \leftarrow (\mathsf{meetingID} \| \mathsf{meetingUUID} \| \ell \| i)$.
10. Defines $\mathsf{Context}_{\mathsf{KDF}} \leftarrow$ `"Zoombase-1-ClientOnly-KDF-KeyMeetingSeed"`.
11. Defines $\mathsf{Context}_{\mathsf{cipher}} \leftarrow$ `"Zoombase-1-ClientOnly-Sig-EncryptionKeyMeetingSeed"`.
12. Decrypts $(\mathsf{MK}, \mathsf{mkSeqNum}) \leftarrow \mathsf{Box.Dec}(sk_i, pk_\ell, \mathsf{Context}_{\mathsf{KDF}}, \mathsf{Context}_{\mathsf{cipher}}, \mathsf{Meta}, C)$.

Now all participants have access to the shared meeting key MK, and can encrypt and decrypt meeting streams accordingly. Participants use an additional HKDF step to derive different subkeys for different message types (e.g. chat, video), mixing in a distinct flag for the message type and also the meetingID, the meetingUUID, and the ID of the message sender.

7.6.6 Key Rotation

At any point later in the meeting, the leader can generate a new 32-byte value MK'. The leader performs steps 10-11 of "Participant Join (Leader)" for all participants, with the updated MK' value. All participants see the rekey signal on their signaling channel, and perform step 12 of "Participant Join (Non-Leader)." Each participant ensures mkSeqNum in the ciphertext sent by the leader is greater than the previously known mkSeqNum; otherwise the key rotation is ignored.

Participants do not immediately encrypt using the new meeting key; they wait about 2 seconds to ensure all participants smoothly transition over. Additionally, as specified in Section 7.6.7, they wait for a signature from the leader certifying the list of users the key has been shared with. This ensures that users always know for whom they are encrypting.

All encrypted UDP packets are prefaced with the 4-byte $\mathsf{mkSeqNum}$, so participants know which version to use for decryption.

The leader should trigger a rekey whenever a participant enters or leaves the meeting. However, if multiple users join in short succession, the leader may choose to wait for a short amount of time and add all the newly-joined users at once. On the other hand, rekeys when users leave meetings might be delayed for up to 10 seconds, which ensures that leaving users can at most only decrypt meeting content sent shortly thereafter (or up to a couple of minutes if the server is suppressing messages to prevent the leader from rotating the key). Until a user begins to encrypt using a new key unknown to the leaving participant, the leaving participant will continue to be displayed as participating in the user interface (detailed in Section 7.6.7).

As a final security measure, leaders rotate the meeting key every five minutes even if there have not been any participant changes, which provides certain *liveness* properties detailed in Section 7.6.8.

We stress that each MK is independently generated, so knowing the previous MK provides no information about the subsequent MK'.

Note: Before Zoom client version 5.12, newly joined users might have received keys up to 15 seconds old. Additionally, participants did not wait for the corresponding leader-signed participant list to start encrypting with a new meeting key.

7.6.7 Leader Participant List

A meeting leader maintains a "leader participant list" (LPL) tabulating all the users in the meeting. For each user currently in the meeting, the LPL keeps track of a hash over their $\mathsf{Binding}_i$, which includes their IVK_i, pk_i, userID, and $\mathsf{hardwareID}$, as well as their display name. For users who have left the meeting, the LPL tracks only their userID, $\mathsf{hardwareID}$, IVK_i, and display name.

The LPL is used to drive the participant list in the user interface, which records both users currently in the meeting and those who have left.

Currently, the participant list in the user interface only identifies users through their self-selected display name and profile picture. Changes to participants' display names are relayed by the server to the leader who includes them in the LPL: a compromised server can change the display names of any meeting participant. As such, display names should not be relied on to establish the participants' identities. Attestations from identity providers (Section 7.10) can also strengthen in-meeting identity by displaying additional identifiers that cannot be tampered with. Later, we will introduce a strong notion of cryptographic identity in meetings (Section 3) to further address these limitations.

The LPL is represented as a sequence of operations such as adding a user to the meeting, or noting when a user has left. Every time there is such an operation, the leader increments a counter v representing the total number of operations and signs over a data structure (called a *link*) containing the counter, the hash of the previous link, and the current operation. If there are more than 20 links in the chain, the leader can coalesce all the previous links

into a smaller number of links. The old links are then deleted in order to save space.

Leaders post a signature over the latest link to the bulletin board whenever membership changes, and broadcast it over the signaling channel at designated "heartbeat intervals":

$$H_i = \mathsf{SHA256}(\mathsf{Binding}_\ell \| \mathsf{SHA256}(LPL_v) \| v \| t \| \mathsf{mkSeqNum} \| H_{i-1} \| \mathsf{timestamp}_l)$$
$$S_i = \mathsf{Sign.Sign}\,(ISK_\ell, \mathsf{Context}, H_i)$$

where Context is `"Zoombase1-ClientOnly-Sig-LeaderParticipantList"`, t increments on every send, v increments whenever the LPL changes, mkSeqNum increments on every MK rotation, H_{i-1} is the previous heartbeat's hash, and $\mathsf{timestamp}_l$ is a monotonically increasing timestamp as recorded by the leader's local clock.

By replaying the sequence of operations in the bulletin board, the other participants can reconstruct the current list of participants, so they know who to rekey for if the leader drops out and they become the new leader. Evil servers might try to withhold updates the leader makes here, to hide when bad actors are kicked out. As such, the leader also sends a low bandwidth "heartbeat" over the signaling channel. Heartbeats should go out at least every 10 seconds. All participants observe and verify these heartbeats, and if they fail to receive ten heartbeats in a row, they should drop out of the meeting. This mechanism prevents the server from withholding updates to the LPL for an extended period of time.

Heartbeats certify both the mkSeqNum and the list of participants for whom mkSeqNum is accessible. Because users wait for the heartbeat certifying a certain meeting key before encrypting with it, the participant list in the user interface always reflects the list of participants users are encrypting for.

When a leader does drop out of a meeting, the Zoom server picks a new leader arbitrarily and sends a signal to participants indicating that the leader has changed. The new leader then coalesces the chain as described above, and other participants verify that the new leader is present in the new LPL.

For users in the meeting, clients remember the mapping between (userID, hardwareID) and hash over the corresponding $\mathsf{Binding}_i$, and ensure that the hash remains stable across new links and leader changes. If a user leaves the meeting, it is enforced that any user who rejoins with the same userID and hardwareID must have the same IVK_i, but not necessarily the same pk_i. These guarantees persist only over the course of a single meeting.

One unavoidable attack the Zoom server can perform is *partitioning* the meeting: for example, split the participants of an ongoing meeting in two groups (each with its own leader), and tell each group that the other half dropped out. Partitioning attacks cannot be avoided while tolerating participants abruptly dropping out of meetings. However, since each heartbeat includes the previous heartbeat's hash in the signature material, if two participants accept the same heartbeat (i.e., they are in the same partition) they must also agree on the history of the meeting. In other words, partitions cannot be reconciled as their heartbeat chains have diverged.

Note: Before Zoom client version 5.13, the heartbeat signature did not include the previous heartbeat's hash in the signature material. As such, different meeting participants might have disagreed on the past history of the meeting, in the case of meeting partitions

due to bad network conditions or server compromise.

7.6.8 Liveness

An important security property in video meetings (and other synchronous communications) is *liveness*: attackers should not be able to significantly delay data streams or meeting management actions (such as adding or removing users).

To strengthen liveness, each meeting participant generates and uploads 192-bit unpredictable nonces N at regular intervals[14]. When the meeting leader rekeys the meeting, each participant's ciphertext includes their most recent nonce as associated data. Participants only accept ciphertexts whose associated data includes one of their two most recent nonces (to tolerate race conditions). This ensures that received keys have been sent recently, i.e., *after* the recipient generated the relevant nonce. This mechanism guarantees the recency of *meeting keys*. Because leaders rotate the meeting key at least every five minutes (more frequently if the set of participants changes), and users stop using old meeting keys for decryption 10 seconds after receiving a newer key, meeting streams are also guaranteed to be relatively recent.

Note: Before version 5.13 of the Zoom meetings client, the liveness bounds worsened with the number of leader changes. Despite that, attacking liveness by forcing a high number of leader changes would likely make participants suspicious: the UX notifies users of each leader change, which is not frequent in regular meetings.

7.6.9 Locked Meetings

Hosts and co-hosts have the ability to lock and unlock the meeting. While a meeting is locked, no new users will be admitted into the meeting. In non-E2EE meetings, the server performs all access control, but locked E2EE meetings will offer stronger guarantees.

When the leader presses the "Lock Meeting" or "Unlock Meeting" buttons in the user interface, their client adds a corresponding link to the *LPL*. Other participants' clients show that the meeting is locked only when it is set in *LPL*; the server does not have the ability to influence this part of the user interface. When a locked meeting becomes unlocked (as indicated by the *LPL*), all participants' user interfaces display a prominent warning indicating the change.

Note that due to the tolerances in propagating the *LPL*, the server might prevent some participants from learning that the host has locked a meeting by withholding the relevant link and selecting a new host within 100 seconds. However, because each heartbeat includes the hash of the previous one, this would result in a meeting partition between the participants who received the lock link and those who didn't.

While the meeting is locked, the leader's client will refuse to send the meeting key to any new participants who request to join. However, if a user leaves and then rejoins with the same *IVK* (as is recorded in the *LPL*), the leader allows them to rejoin. This lets

[14] To limit bandwidth, the frequency degrades quadratically with the number of participants in the meeting. For meetings of at most 10 participants, the interval is about 3 minutes.

participants who drop out due to network issues automatically reconnect even while the meeting is locked.

If the leader changes while the *LPL* indicates that the meeting is locked, participants ensure that the new leader was in the previous *LPL*. If not, they drop out of the meeting. The new leader can then copy over the locked bit and the list of participants from the old *LPL* into the new link.

Co-hosts are also able to lock and unlock the meeting. They do this by sending a signed message to the leader via the bulletin board. The leader ensures that the co-host is really a member of the current *LPL* before processing the change, and trusts the server to identify whether each participant is actually a co-host. Since the server also has the ability to select a new meeting host among existing participants of a locked meeting, this change does not significantly degrade the security of the meeting. In order to prevent replays, co-hosts sign over the latest *LPL* link hash, and leaders ensure that this matches their view of the *LPL* before accepting. If the *LPL* changes before the co-host's message was received and processed, the co-host tries again.

From a security perspective, once a participant learns that the meeting is locked and checks that all current participants are trustworthy (e.g., via a meeting leader security code check), they can be sure that no unintended parties have or will have access to the meeting until an "unlocked meeting" warning is displayed.

These additional guarantees for locked E2EE meetings were added in Zoom client version 5.6.0 (see Appendix A).

7.6.10 Meeting Teardown

At the end of the meeting, or when leaving a meeting early, all participants should discard all meeting keys, all keys derived from those meeting keys, and the ephemeral DH private keys sk_i they generated when they joined.

The intent here is to provide forward secrecy. That is, if an adversary can record all encrypted messages relayed between Zoom clients during the meeting, and can later recover all keys stored on a user's device after the meeting ends, they still cannot recover the meeting data.

7.7 Meeting Leader Security Code

If all participants can verify the authenticity of the leader's public key (IVK_ℓ), they are safe from MitM attacks. The Zoom client exposes the following "meeting leader security code" in the security tab:

$$\mathsf{Digits}(\mathsf{SHA256}(\mathsf{SHA256}(\texttt{"Zoombase-1-ClientOnly-MAC-SecurityCode"})||\mathsf{SHA256}(IVK_\ell)))$$

Digits extracts a string of 39 decimal digits from a SHA-256 hash, representing just over

129 bits of information. This representation is more human-readable and more internationalizable than the full hexadecimal hash. Crucially, every Zoom client in the meeting independently computes these codes from the IVK_ℓ used in the handshake protocol. The length of the code is long enough to protect against second pre-image attacks. The leader reads out the meeting leader security code, after which everyone in the meeting in turn does the same thing. If the code does not match, the participant should speak up in the meeting, and the leader should rotate the meeting key by kicking them out; they may be allowed to rejoin and try again. By having the leader go first, participants verify that they all agree both on which of them is the leader, and on their IVK_ℓ. Both properties are necessary to detect MitM attacks.

If deep fake technology[15] is a concern, or the participants do not know each other in advance, this verification can also happen over a different out-of-band secure channel.

Non-leader participants see a notification prompting them to re-perform the security code checks whenever the meeting leader changes. These additional checks prevent a compromised Zoom server from changing the meeting leader over the course of a meeting without being detected.

We considered other approaches to the meeting leader security code, such as mixing more of the handshake data into the displayed code. While more mixins would be more robust to attacks that try to confuse participants by mixing members from different meetings, we see a UX advantage of "one leader, one code."

7.8 E2E Encryption for Breakout Rooms

The Breakout Rooms feature allows splitting a Zoom meeting into multiple sub-meetings. The host can assign participants to rooms, or allow them to choose the room that they want to join, and can broadcast chat messages to all breakout rooms at once. While main meeting participants (including the host) do not have access to meeting content from breakout rooms that they are not a participant of, the host does receive some metadata about the "activity status" of participants in each breakout room to help them monitor engagement, including for example whether participants have their video on, are using reactions, or sharing their screen.

In E2EE meetings, we implement Breakout Rooms by having each room function as its own independent meeting, with its own leader, participant list, and sequence of meeting keys. As users leave the main meeting and join a breakout room, both meetings rotate their keys. In addition, after they join a breakout room, participants are also re-added by the leader to the main meeting so that they can keep decrypting messages broadcast by the main meeting host. Since they continue to have access to the meeting key, breakout room participants are still part of the main meeting LPL, but the participants panel of the meeting UX indicates that they are "In a breakout room." If the main meeting is locked, participants are still able to go in and out of it to join breakout rooms, as the leader of the

[15] We use "deep fakes" to refer to manipulated and/or fabricated audio/video data that uses synthetic media techniques to replace the likeness of one person with another. Using a "deep fake" (especially in real-time) of a meeting participant could potentially deceive others about the identity of that participant.

main meeting will let them back in given that they keep the same *IVK* (as explained in Section 7.6.9).

Users in a breakout room can check the security codes with the breakout room leader to ensure that there are no MitM attacks on the breakout room itself. Since there is no cryptographically enforced relationship between the main meeting and breakout rooms, such as the fact that the assignments of people to rooms reflect the intentions of the meeting host, breakout room participants should also make sure that other participants in their breakout room are expected to be there before discussing sensitive matters (as in regular meetings). However, breakout room participants cannot see the main meeting's leader security code, and are not notified if the leader of the main meeting changes while they are in a breakout room. Therefore, an active insider could add a breakout room participant to a different main meeting when they try to rejoin it after entering a breakout room. Then, the attacker could broadcast messages to that participant which would appear to come from the original host of the main meeting (this would work even if the main meeting were locked). As such, users worried about active attacks from insiders should be suspicious of broadcast messages received while in a breakout room.

An attacker might also try to join an unlocked main meeting and immediately go to a breakout room to avoid being identified by having to turn their camera on or being asked to check security codes. The attacker could use the same display name and picture of another meeting participant, either one who recently left the meeting or one that the attacker has kicked out themselves (by controlling the network or the Zoom infrastructure). In this case, the same participant name would appear twice as "left" and "In a breakout room" in the participant list, which may be hard for other participants or the leader to spot. While analogous attacks are possible even when breakout rooms are not in use, they might be harder to recognize and act on in this scenario: for example, participants in a breakout room do not get a video tile in the gallery view. We recommend that all meeting participants carefully monitor the participant list, and that hosts lock their meeting whenever possible.

Finally, when participants leave a breakout room to rejoin the main meeting, an insider might trick them into joining a different main meeting than the original one (even if the latter were locked). As such, when switching between the main meeting and breakout rooms, all participants should recheck leader security codes.

In future updates, we plan to further strengthen the guarantees offered by this feature, by not making participants leave and then rejoin the main meeting when entering or exiting breakout rooms, and having breakout room participants enforce that other participants of their breakout room are also listed in the main meeting's participant list. This will better mitigate the risks above, and require less trust in the Zoom server when using E2EE Breakout Rooms.

7.9 Abuse Management and Reporting

If a user experiences abusive behavior and wishes to report it to Zoom's Trust and Safety team, they simply upload the unencrypted data normally collected in an abuse report (e.g., a description of the abuse and some portion of the meeting content) to Zoom for review.

This protocol is imperfect, since it potentially could allow a bad meeting participant to "frame" an honest meeting participant for abuse that didn't happen. For the same reason, it allows an actual abuser to disavow uploaded evidence of their abuse. We think for now, the framing behavior is rare and only possible with access to good "deep fake" technology.

Future refinements are possible. Participants could sign their outgoing video streams, and other participants will only allow meetings to proceed if all streams are appropriately signed. This change would defeat the two attacks above, but with major drawbacks:

Performance: Signing and verifying individual UDP video streams is expensive in terms of bandwidth and computation. More research is required to make this change practical.

Repudiation: Honest participants might not want an indelible record that they said something. They might understandably want to treat meetings as ephemeral in accordance with their standard data retention practices.

Given these challenges, we will revisit our decisions at a later date as we gain more operational experience with the current proposal.

7.10 IDP Attestations for E2EE Meetings

If a user's account is configured to support IDP attestations (Section 5), they can use an attestation in E2EE meetings to prove to other meeting participants that they are part of a specific organization (identified by its ADN) in which they hold a specific email address (as vouched for by the organization's designated IDP, and which they logged into Zoom with). Attestations by a trusted IDP reduce the need to explicitly check security codes.

Upon joining a meeting, each device obtains an IDP attestation, which authenticates the signing key IVK used to join the meeting as described in Section 5.3. Currently, the `zoom-identity-snapshot` field of the attestation includes the IVK; once we start leveraging sigchains in meetings (see Section 7.11 for details), we plan to extend the snapshot with the tails of the given user's sigchains.

The user signs the IDP attestation and their ADN with their IVK, and temporarily stores the attestation and signature on the Zoom servers. To share this information only with the intended meeting participants, the user posts to the meeting bulletin board an "identity sharing token" (along with the signed $\mathsf{Binding}_i$ described in Section 7.6.2 "Participant Key Generation"), which other participants can present to the server in exchange for the attestation. Identity sharing tokens are computed as an HMAC of some metadata about the attestation (such as whether it contains the user's email address), using a random 32-byte symmetric key generated by the Zoom servers for each user's device in each meeting. Identity sharing tokens have a lifetime of 24 hours, and can be used to fetch any matching attestation for that user's device. An IDP attestation can be used across multiple meetings but has a relatively short lifetime, which is set by the IDP, defaulting to 48 hours.

When displaying another participant's identity in the user interface during a meeting, clients that successfully fetch an identity attestation will verify the signature of ADN

and attestation by *ISK*, verify that this ADN (in addition to the email address in the attestation) matches what is provided by the Zoom server, and verify the attestation as detailed in Section 5.4; in particular, the *IVK* in the $\mathsf{Binding}$ must be consistent with the contents of attestation's `zoom-identity-snapshot` field. If the checks are successful, the interface displays to the user the email address from the attestation and details about the authenticating ADN and IDP. The verifying user should review the displayed email address and ADN and confirm that they are the expected identifiers for the user they are meeting with.

Meeting participants' clients may perform these checks asynchronously during a meeting, but in the future we may offer the option to configure a meeting such that the host must complete this verification before admitting a joining participant. Requiring this verification before key exchange can help increase the security of meetings that are, for example, restricted to users in specified accounts.

7.11 E2EE Meetings with Cryptographic Identity

Note: This feature is not currently available. We plan to release it in a future update.

In Section 3 we describe how we leverage sigchains to build a strong multi-device notion of cryptographic user identity. We plan to leverage this notion in the context of E2EE meetings in order to strengthen the guarantees provided by security codes and IDP attestations in detecting and preventing MitM attacks.

The server signature $\mathsf{Sig}^i_{\mathsf{Server}}$ as described in Section 7.6.1 will also include the sigchain tails for the corresponding user, email, account, and ADN sigchains. These tails will also be included in the signed $\mathsf{Binding}_i$ generated in the "Participant Key Generation" procedure of Section 7.6.2, and in the `zoom-identity-snapshot` field of the IDP attestation (if one is used in the meeting).

In a meeting, Alice's client verifies Bob's sigchains (and IDP attestation, if present) before Alice's client displays identifiers for Bob in the UI. To do so, Alice's client fetches Bob's user sigchain (which includes IVK_i), email sigchain, account sigchain, ADN sigchain. Alice's client verifies the server signature $\mathsf{Sig}^i_{\mathsf{Server}}$, checks that the tails of the received sigchains match those in $\mathsf{Sig}^i_{\mathsf{Server}}$ and in the $\mathsf{Binding}$, checks that Bob's latest sigchain is consistent with any previous retrievals of Bob's sigchain, and verifies the IDP attestation if one is present (including checking that the sigchain tails included in the attestation match). As for IDP attestations, these checks might be performed asynchronously, or mandated before the host performs the key exchange to limit who can access a meeting.

To minimize the need to request new attestations, users accept attestations that do not necessarily cover the latest sigchain tails as long as the new links added since the IDP's snapshot do not revoke the device currently being used in the meeting and do not change the user's email or account identifiers.

The Zoom server will provide access control to ensure that sigchains are visible to other meeting participants only for a short duration after a meeting begins. If Alice has never been in a meeting with Charlie, Charlie will have no information regarding Alice's sigchain's

contents, length, or update frequency.

7.12 Security Properties for E2EE Meetings

The Identity Management System, Bulletin Board and Signaling Channel as enumerated in Section 7.4 are deployed by Zoom, and protect against outsiders using TLS. Attackers classified as insiders by our threat model could monitor or meddle with these components. An insider monitoring such components (a passive attack) would expose meeting metadata, which is stated as a limitation of our designs in Section 2.1, but would not otherwise compromise the confidentiality of the meeting.

We prevent outsiders from joining meetings through passwords, waiting rooms, and the other non-cryptographic server-enforced access control features described in Section 7.1. TLS and encryption of the meeting streams protect the confidentiality and integrity of the meeting against outsiders who might control the participants' network.

Within the same meeting, the encrypted streams sent by each participant are protected against replay attacks by using encryption nonces as counters. Even across different meetings, the streams cannot be replayed thanks to the fact that participants delete all the ephemeral keys once a meeting is over (which also guarantees forward secrecy).

We aim to minimize the damage that an active insider can perform without being detected. First, an insider can force participants to drop out of a meeting, as well as partition them into separate meetings, each with its own leader, and add extra participants circumventing all the above server-enforced features. Our protocol ensures that the meeting leader will be able to detect any participant obtaining access to meeting encryption keys by monitoring the participant list for unexpected entries (for example, by recognizing the participants' faces in their video streams). If the host observes the same user leaving the meeting four or more times, the participant list will reflect the number of observed leaves. Liveness properties ensure that actions like rekeying the meeting or adding/removing participants cannot be arbitrarily delayed, and furthermore, that audio/video streams displayed are relatively recent.

Detecting unauthorized participants can be challenging in large meetings or in certain views, such as when the leader is sharing their screen. In the future, a stronger notion of identity will be leveraged to highlight potential eavesdroppers in the UI, such as those outside of the host's organization or guest users.

We stress that, even when using IDP attestations, seeing a specific user identity in the participant list of a meeting does not imply that the corresponding user has chosen to participate in the meeting or is still actively participating, but only that that user could potentially have access to the encrypted meeting contents. A malicious insider could either trick the leader into including a user in the participant list (when the user is not actually present in the meeting), or hide the fact that a participant has left a meeting (so that other users are convinced they are still participating). All such participants would still trigger Contact Sync warnings as detailed in Section 3.9 (once that feature is deployed), and clients will remember their identities for future meetings. We believe that preventing

these issues would add too much complexity and overhead to the protocol.

An active attacker can also try to perform a MitM attack against the meeting. This class of attacks is mitigated by the meeting leader security code, which should be re-checked every time a participant joins or rejoins the meeting (including when moving in and out of breakout rooms), or whenever the meeting leader security code changes as indicated by a UI notification.

When leveraged in E2EE meetings (under the assumptions in Section 5.6) an IDP attestation confirms that a meeting participant's long-term device key belongs to a member of a specific account (identified by its ADN), which has vouched for that participant's identity (e.g. email address) through a third party identity provider. IDP attestations are a mitigation against MitM attacks and serve as an alternative to checking security codes.

We also stress that, as discussed in Section 7.8, when in an E2EE meeting using Breakout Rooms, participants should be skeptical of broadcast messages from the meeting host and should carefully monitor the participant list of the main meeting.

We have published [10] a formal security analysis of our core E2EE meetings protocol, which precisely characterizes the liveness, confidentiality, and security properties it achieves.

7.12.1 Areas to Improve

While E2E meetings offer strong security properties to Zoom users, there are still opportunities for improvement, both in the key agreement layer and in the identity layer.

Meddler-in-the-Middle. The meeting leader security code and IDP attestations are countermeasures for the classic MitM attack, wherein Bob isn't actually connecting to Zoom; he's connecting to Eve who is proxying his communications. Both solutions have limitations: they can be defeated by deep fake technology, introduce some UX friction, or require additional trust assumptions. Sigchain-backed identities (Section 7.11) and the ZTT (Section 4) will further improve the security properties and UX.

Anonymous Eavesdropper. An adversary, in conjunction with a malicious Zoom server, types in a name of their choosing, turns off video, mutes their microphone and just observes. Checking security codes and cryptographic identity can help address this problem partially.

Impersonation Attacks Within the Meeting. Even if Alice and Bob are both authorized to be in the meeting, if Alice has the help of a malicious server, she can inject audio/video for Bob. Charlie would have no way of knowing that Bob's stream was being faked.

8 Encryption for Zoom Phone

Zoom Phone is a cloud phone system, in which Zoom Phone users are assigned phone numbers and can call other Zoom Phone users, as well as other telephone landline and mobile numbers. As with Zoom Meetings using enhanced encryption, all Zoom Phone calls are encrypted in transit between client endpoints and the server, but not in an end-to-end manner: a passive adversary with Zoom server access may be able to decrypt the data streams. To increase the guarantees for Zoom Phone users and reduce reliance on the server, we offer the ability to upgrade an ongoing Zoom Phone call to use end-to-end encryption.

8.1 E2EE Zoom Phone Calls

The design of E2EE for Zoom Phone has very similar goals, threat models, and limitations as in Zoom Meetings. We wish to prevent even insiders with access to Zoom servers from compromising the confidentiality and integrity of E2E-encrypted Zoom Phone calls. Like Zoom Meetings, Zoom Phone has many features that are not compatible with strong cryptographic guarantees, such as dialing in from PSTN phones. Such features will be disabled when E2E is enabled.

At the moment, we only support E2EE for calls between up to two Zoom Phone clients in the same account, which greatly simplifies the protocol.

8.1.1 Join/Leave Protocol

E2EE Zoom Phone calls leverage the key management system from Sections 3.4 and 3.7, and the same cryptographic primitives as Zoom Meetings (described in Section 7.5). Zoom Phone calls are identified with CallSessionID instead of meetingID and meetingUUID.

To join an E2E call, participants ask for signed statements from the Zoom server over their ID and long-term key, just as in Zoom meetings (described in Section 7.6.1). They also generate per-call ephemeral DH keys (pk_i, sk_i), sign them with their device keys (similar to Section 7.6.2), and send the key and signature to the other participant.

The call is encrypted with a shared meeting key obtained from the DH key exchange of the clients' ephemeral keys (using the participants' UIDs and the CallSessionID as context). Since the set of participants is fixed, the key does not rotate and does not have a sequence number, and there is also no Leader Participant List, heartbeats, or any analogue of the "locked meeting" feature. Participants still fetch each other's long-term public keys from the key server. Instead of the bulletin board, the server simply offers an interface for the two callers to message each other. At the moment, the only way to have an E2EE Zoom Phone call is to upgrade an in-progress call to use E2EE. When upgrading to E2EE, each client first sends its own signed ephemeral key, and when it receives the other party's key material, it responds with an explicit acknowledgement message. Once it receives the other party's acknowledgement, the client can start encrypting its own call stream.

8.1.2 Phone Security Code

To defend against MitM attacks, Zoom Phone provides a "phone security code" that has a similar format to the meeting leader security code (Section 7.7), but that is derived from the ephemeral public keys of both parties. Since the set of participants is fixed, there is no concern about this code changing too frequently. The security code is computed as

$$\mathsf{Digits}(\mathsf{SHA256}(\mathsf{Context}||pk_{\mathsf{Caller}}||pk_{\mathsf{Callee}}||\mathsf{CallSessionID})),$$

where Context is the string `"Zoombase-2-ClientOnly-KDF-PhoneSecurityCode"`. The user who initiated the call is designated the Caller, and the other user is the Callee.

8.2 Advanced Encryption for Voicemail

Zoom Phone users may receive voicemail sent from either mobile and landline phones using PSTN, or other Zoom Phone clients. By default, voicemail messages are received and recorded by Zoom servers, who encrypt them at rest with keys controlled by the Zoom infrastructure.

We also offer Advanced Encryption for Voicemail (AEV) as a more secure alternative: Zoom servers still receive and record the voicemail themselves, but they encrypt it for keys known only to the intended recipient's devices. We accomplish this using a voicemail-specific PUK subkey, as described in Section 3.4.1. Accordingly, AEV-encrypted voicemails can only be accessed by the devices belonging to the intended recipient with access to the PUK, i.e. devices created before the voicemail was generated, or (transitively) approved by such a device. We do not support retrieving these voicemail messages from PSTN, the Zoom website, or generic desk phone/SIP devices.

Voicemails encrypted with AEV use a simplified version of the email encryption scheme (Section 6.1). Voicemail ciphertexts include the audio recording as the only encrypted piece, and there is always only a single (non-BCC) recipient. Email addresses are omitted as they are not relevant in the voicemail context, and there are no signatures as, even when sent among Zoom Phone users, it is the Zoom servers that are performing the recording and encryption.

8.2.1 Security Properties

The security properties of AEV largely follow from those of the key management layer that it is built upon, as detailed in Section 3.11. In particular, a malicious insider cannot decrypt any AEV-encrypted voicemail messages recorded before its compromise began (unless it also corrupts a device to obtain the necessary decryption keys). Key rotations happen as soon as possible when the set of devices changes and ensure that revoked devices cannot decrypt future voicemails, while we rely on the server to withhold past ciphertexts.

We stress that, even if Zoom Phone servers delete plaintext voicemail messages as soon as possible after encryption (and cannot perform decryption afterwards), they have temporary access to the voicemails during recording, and therefore AEV has inherently weaker guarantees than end-to-end encryption. In addition to the confidentiality limitations, voicemail

recipients cannot independently verify, and therefore rely on the Zoom servers for, the integrity of the voicemail recording (i.e. that it wasn't tampered with) and of any metadata such as the recording time, claimed author, intended recipient and their respective phone numbers. This is unavoidable at least in the PSTN case, as this technology does not support encryption natively. We are considering offering E2EE voicemail between Zoom Phone users in the future. Moreover, we note that the above metadata is not encrypted with per-user keys, and remains available to the server to provide the service even after the audio recording is not.

Since Zoom servers are performing the encryption and keeping track of everyone's keys, fingerprints are not displayed when sending or listening to an AEV voicemail message (but the user interface distinguishes between messages using standard and AEV encryption). Fingerprint verification is still useful during device approvals to prevent an attacker (including a recently compromised server) from gaining access to previously encrypted voicemails.

9 Acknowledgements

We thank Deirdre Connolly, Adriaan De Vos, Yevgeniy Dodis, Takanori Isobe, Ryoma Ito, Daniel Jost, Nadim Kobeissi, Chelsea Komlo, Anna Kornfeld Simpson, Paul Miller, Tim Ruffing, Nitesh Saxena, Soatok, Ryan Thomas, all GitHub issue authors, and the many internal reviewers and colleagues for helpful conversations and feedback on previous versions of this document.

References

[1] Key transparency. Available at `https://github.com/google/keytransparency/`.

[2] Keybase. Available at `https://keybase.io`.

[3] OpenID connect core 1.0 authorization code flow. `https://openid.net/specs/openid-connect-core-1_0.html#CodeFlowAuth`.

[4] Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography - PKC 2006 (9th International Conference on Practice and Theory in Public-Key Cryptography, New York NY, USA, April 24-26, 2006, Proceedings)*, Lecture Notes in Computer Science, pages 207–228, Germany, 2006. Springer.

[5] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering 2*, 2012.

[6] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. NaCl: Networking and cryptography library. Available at `https://nacl.cr.yp.to/`.

[7] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. SEEMless: Secure end-to-end encrypted messaging with less trust. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1639–1656, 2019.

[8] Brian Chen, Yevgeniy Dodis, Esha Ghosh, Eli Goldin, Balachandar Kesavan, Antonio Marcedone, and Merry Ember Mou. Rotatable zero knowledge sets: Post compromise secure auditable dictionaries with application to key transparency. In *Advances in Cryptology–ASIACRYPT 2022*, pages 547–580. Springer, 2023.

[9] Frank Denis. The sodium cryptography library. Available at `https://download.libsodium.org/doc/`, Jun 2013.

[10] Yevgeniy Dodis, Daniel Jost, Balachandar Kesavan, and Antonio Marcedone. End-to-end encrypted zoom meetings: Proving security and strengthening liveness. In *Advances in Cryptology–EUROCRYPT 2023*. Springer, 2023.

[11] Morris J. Dworkin. SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, 2007.

[12] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In *Advances in Cryptology–CRYPTO 2017*, pages 66–97. Springer, 2017.

[13] Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010.

[14] Adam Langley, Emilia Kasper, and Ben Laurie. Certificate transparency. *Internet Engineering Task Force (IETF)*, 2013.

[15] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing key transparency to end users. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 383–398, 2015.

[16] N. Sakimura, J. Bradley, and N. Agarwal. Proof Key for Code Exchange by OAuth Public Clients. RFC 7636, 2015.

A Release Schedule

This appendix details which Zoom client versions first support the features described in this document.

Version 5.4.0 E2EE Meetings (Section 7), except for Locked Meetings (Section 7.6.9), Server Key Certificate Chains (Section 7.6.1), Breakout Rooms (Section 7.8), and support for sigchain-backed identity (Section 7.11) and IDP attestations (Section 7.10).

Version 5.6.0 Locked Meetings (Section 7.6.9).

Version 5.7.0 Server Key Certificate Chains (Section 7.6.1).

Version 5.11.3 Breakout Rooms (Section 7.8). E2EE Zoom Phone (Section 8.1).

Version 5.12 Improvements to E2EE Meeting Key Rotation (Section 7.6.6) and Leader Participant List (Section 7.6.7).

Version 5.12.6 Cryptographic User Identity (Section 3), except for cloud identifier and ADN support (Section 3.3, Section 3.3.1), Contact Sync (Section 3.9), and Compromise Prevention for Device Provisioning (Section 3.10). Advanced Encryption for Voicemail (Section 8.2).

Version 5.12.8 Encryption for Zoom Mail Service (Section 6).

Version 5.13 E2EE Meeting Liveness (Section 7.6.8).

Version 5.13.10 Okta Authentication for End-to-End Encryption (E2EE), which supports Okta IDP attestations for E2EE Meetings (Sections 5, 7.10).

Version 5.15.10 Account Escrow (Section 3.8).

B Understanding Multiple Devices

As detailed in Section 3.4, device graphs can become complicated; we introduce a formal model to reason about them and what guarantees are available.

Definition B.1. A **device family state** is a tuple:

$$\mathcal{F}_t = \langle \mathbb{D}_t, d_t, \mathbb{R}_t, \mathbb{A}_t, \mathbb{P}_t, \mathbb{B}_t \rangle$$

Where:

- $t \in \mathbb{N}_0$ is a non-negative integer which is used to order states in a sequence, which we refer to as "time."
- $\mathbb{D}_t$ is a finite set whose elements represent the devices at time t.
- $d_t : \mathbb{D}_t \to \mathbb{N}_0$ is a function that maps devices to the time t that they were provisioned.
- $\mathbb{R}_t \subseteq \mathbb{D}_t$ is the set of revoked devices at time t. $\mathbf{x} \in \mathbb{R}_t$ means that $\mathbf{x}$ is revoked at time t or before.
- $\mathbb{A}_t \subseteq \mathbb{D}_t \times \mathbb{D}_t$ is the set of device approvals at time t. $\langle \mathbf{x}, \mathbf{y} \rangle \in \mathbb{A}_t$ means that $\mathbf{x}$ approved $\mathbf{y}$ at time t or before.
- $\mathbb{P}_t$ is the set of per-user-keys (PUKs).
- $\mathbb{B}_t \subseteq \mathbb{P}_t \times \mathbb{D}_t \times \mathbb{D}_t$ is the set of PUK boxes, where one device boxes the private keys of a PUK for itself or another device. That is, $\langle k, \mathbf{x}, \mathbf{y} \rangle \in \mathbb{B}_t$ means that device $\mathbf{x}$ boxed k for device $\mathbf{y}$ at time t or before.

It's also worth defining, for convenience, a function that maps a device $\mathbf{x}$ to all the PUKs that it knows. Recall that $\mathbf{x}$ knows a PUK k if another device $\mathbf{y}$ boxed k for it; or symbolically, that $\langle k, \mathbf{x}, \mathbf{y} \rangle \in \mathbb{B}_t$. Note that in some cases, $\mathbf{x} = \mathbf{y}$.

Definition B.2. Let $p_t : \mathbb{D}_t \to \mathbb{P}_t$ be defined as:

$$p_t(\mathbf{x}) = \{k \mid \exists \mathbf{y} \in \mathbb{D}_t \text{ such that } \langle k, \mathbf{y}, \mathbf{x} \rangle \in \mathbb{B}_t\}$$

We now define how a device family transitions from one configuration to another over time.

Definition B.3. A **device family** consists of a sequence $\mathcal{F} = (\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_n)$. Each of the $\mathcal{F}_i$ is a family state as defined above. $\mathcal{F}_0$ is the empty family state (all the components of the tuple are empty sets or functions with empty domain). Moreover, we require that the difference between any two consecutive states can be seen as the result of one of the following transitions below (which leave the components of the tuple which are not explicitly updated as unchanged):

1. DeviceAdd. Device $\mathbf{x}$ is provisioned at time t:
   - Preconditions:
     (a) $\mathbf{x} \notin \mathbb{D}_{t-1}$
   - Effects:
     (a) $\mathbb{D}_t \leftarrow \mathbb{D}_{t-1} \cup \{\mathbf{x}\}$
     (b) $d_t \leftarrow d_{t-1} \cup \{\langle \mathbf{x}, t \rangle\}$

   (c) $\mathbb{P}_t \leftarrow \mathbb{P}_{t-1} \cup \{k\}$ for some new key k
   (d) $\mathbb{B}_t \leftarrow \mathbb{B}_{t-1} \cup \{\langle k, \mathbf{x}, \mathbf{y}\rangle \mid \mathbf{y} \in \mathbb{D}_t \backslash \mathbb{R}_t\}$

2. DeviceRevoke. Device $\mathbf{x}$ is revoking devices $\mathbb{S}$ at time t:

   - Preconditions:
     (a) $\mathbf{x} \in \mathbb{D}_{t-1}$
     (b) $\mathbf{x} \notin \mathbb{R}_{t-1}$
     (c) $\mathbb{S} \subseteq \mathbb{D}_{t-1}$
     (d) $\mathbb{S} \cap \mathbb{R}_{t-1} = \varnothing$
   - Effects:
     (a) $\mathbb{R}_t \leftarrow \mathbb{R}_{t-1} \cup \mathbb{S}$
     (b) $\mathbb{P}_t \leftarrow \mathbb{P}_{t-1} \cup \{k\}$ for some new PUK k
     (c) $\mathbb{B}_t \leftarrow \mathbb{B}_{t-1} \cup \{\langle k, \mathbf{x}, \mathbf{y}\rangle \mid \mathbf{y} \in \mathbb{D}_t \backslash \mathbb{R}_t\}$

   Note that our model also allows the server to initiate a DeviceRevoke, in which case the PUK will not be rotated. We don't model this transition here for simplicity.

3. PerUserKeyRotate. Device $\mathbf{x}$ rotates PerUserKey at time t (usually after another device self-revoked):

   - Preconditions:
     (a) $\mathbf{x} \in \mathbb{D}_{t-1}$
     (b) $\mathbf{x} \notin \mathbb{R}_{t-1}$
   - Effects:
     (a) $\mathbb{P}_t \leftarrow \mathbb{P}_{t-1} \cup \{k\}$ for some new PUK k.
     (b) $\mathbb{B}_t \leftarrow \mathbb{B}_{t-1} \cup \{\langle k, \mathbf{x}, \mathbf{y}\rangle \mid \mathbf{y} \in \mathbb{D}_t \backslash \mathbb{R}_t\}$

4. BatchApprove. Device $\mathbf{x}$ approves devices at time t, meaning it approves all non-revoked devices $\mathbf{y}$ such that $d_t(\mathbf{x}) < d_t(\mathbf{y})$.

   - Preconditions:
     (a) $\mathbf{x} \in \mathbb{D}_{t-1}$
     (b) $\mathbf{x} \notin \mathbb{R}_{t-1}$
   - Effects:
     (a) $\mathbb{A}_t \leftarrow \mathbb{A}_{t-1} \cup \{\langle \mathbf{x}, \mathbf{y}\rangle \mid \mathbf{y} \in \mathbb{D}_t \backslash \mathbb{R}_t, d_t(\mathbf{x}) < d_t(\mathbf{y})\}$
     (b) $\mathbb{B}_t \leftarrow \mathbb{B}_{t-1} \cup \{\langle k, \mathbf{x}, \mathbf{y}\rangle \mid \mathbf{y} \in \mathbb{D}_t \backslash \mathbb{R}_t, d_t(\mathbf{x}) < d_t(\mathbf{y}), k \in p_{t-1}(\mathbf{x})\}$

The thinking behind the batch approval operation is that a user should always revoke all devices when necessary, and therefore should be approving all the devices that can be approved whenever they approve any device.

DeviceKeyRotate is not explicitly modeled here, as it would have the same preconditions and effects of a PerUserKeyRotate link.

Note that $\mathbb{A}_t$ defines an undirected graph on the set $\mathbb{D}_t$ (where elements of $\mathbb{A}_t$ are the edges).

Definition B.4. We denote with $e_t(\mathbf{x}) \subset \mathbb{D}_t$ the connected component of $\mathbf{x}$ in the graph defined on $\mathbb{D}_t$ by $\mathbb{A}_t$. Let $v_t : \mathbb{D}_t \to \mathbb{N}$ be the function defined as:

$$v_t(\mathbf{x}) = \min \{d_t(\mathbf{y}) \mid \mathbf{y} \in e_t(\mathbf{x})\}$$

We can define an equivalence relation between two devices with respect to a device family state $\mathcal{F}_t$ (which boils down to being part of the same connected component):

Definition B.5. Given a device family $\mathcal{F}_t$, $\forall \mathbf{x}, \mathbf{y} \in \mathbb{D}_t$ define $\mathbf{x} \equiv_t \mathbf{y}$ iff $v_t(\mathbf{x}) = v_t(\mathbf{y})$.

Note that, while revoked devices cannot perform new approvals, the approvals they make before being revoked are still considered part of the graph. For example, consider a family with 3 devices and the following transitions:

1. **a** is provisioned
2. **b** is provisioned
3. **a** approves **b**
4. **c** is provisioned
5. **b** approves **c**

At this point ($t = 5$), we have: $v_5(\mathbf{a}) = v_5(\mathbf{b}) = v_5(\mathbf{c}) = 1$. Then (at $t = 6$) **b** self-revokes. $v_6(\mathbf{c})$ remains at 1, even though the only path from **a** to **c** at $t = 6$ crosses **b**, a revoked device.

It's worth noting that by definition of v_t and the equivalence $\equiv_t$, the idea of approval is bidirectional. That is, if Alice provisions **x** then **y**, then approves **y** with **x**, then **x** and **y** are in the same equivalence class, even though she never approved **x** with **y**. We think this is a useful simplification. The rationale is that when **y** was added, Alice had access to the list of her devices on **y** which includes **x**, so we assume she implicitly approved **x** with **y**, since she didn't revoke it when she had a chance.

B.1 A Claim about Device Equivalence Classes

We have an important claim to flesh out: that all devices in the same equivalence class know the same PerUserKey secrets. This allows us to treat them the same throughout the UI, either for the purposes of propagating trust, or for the purposes of propagating secrets.

Theorem B.1. *For any device family, and any time t, $\forall \mathbf{x}, \mathbf{y} \in \mathbb{D}_t \backslash \mathbb{R}_t$, if $\mathbf{x} \equiv_t \mathbf{y}$ then $p_t(\mathbf{x}) = p_t(\mathbf{y})$.*

Before we get to the proof, it helps to proves some simpler lemmas about the structure of device families given our transition rules in Definition B.3.

First, a simple observation:

Lemma B.2. *For any device family, any time t, $\forall \mathbf{x}, \mathbf{y} \in \mathbb{D}_t \backslash \mathbb{R}_t$, if $\langle \mathbf{x}, \mathbf{y} \rangle \in \mathbb{A}_t$ then $v_t(\mathbf{x}) = v_t(\mathbf{y})$.*

Proof. If if $\langle \mathbf{x}, \mathbf{y} \rangle \in \mathbb{A}_t$, then the two are connected, and thus $e_t(\mathbf{x}) = e_t(\mathbf{y})$, from which we have $v_t(\mathbf{x}) = v_t(\mathbf{y})$. ■

The second lemma states that older devices know strictly more PerUserKey private halves than newer devices. For devices, $\mathbf{a}, \mathbf{b}, \mathbf{c}$ and $\mathbf{d}$, provisioned in that order, it is clear to see that the later devices will always box for the earlier devices. But keep in mind that when $\mathbf{b}$ approves new devices, it never sends PerUserKeys back to earlier devices. So we must formally show that, for instance, $\mathbf{c}$ approving $\mathbf{d}$ does not transmit PerUserKeys from $\mathbf{a}$ that $\mathbf{b}$ didn't know about.

Lemma B.3. *For any device family, at any time t, $\forall \mathbf{x}, \mathbf{y} \in \mathbb{D}_t \backslash \mathbb{R}_t$, if $d_t(\mathbf{x}) < d_t(\mathbf{y})$ then $p_t(\mathbf{y}) \subseteq p_t(\mathbf{x})$.*

Proof. The proof is by induction over time t. For the base case, for any device family containing less than two devices (which includes any family at time $t = 0$), the claim is trivially true. For the inductive case, assume the lemma is true for time $t - 1$. Take arbitrary $\mathbf{x}, \mathbf{y} \in \mathbb{D}_t \backslash \mathbb{R}_t$, such that $d_t(\mathbf{x}) < d_t(\mathbf{y})$. Consider the 4 possible state transitions:

1. DeviceAdd. A new device $\mathbf{z}$ is introduced at time t, meaning $d_t(\mathbf{z}) = t$. Note that it cannot be $\mathbf{z} = \mathbf{x}$ as $d_t(\mathbf{x}) < d_t(\mathbf{y}) \leq t$.

   If $\mathbf{z} = \mathbf{y}$, then let k be the PUK generated at Effect 1c. $\mathbb{B}_t$ gets $\langle k, \mathbf{y}, \mathbf{y} \rangle$ and $\langle k, \mathbf{y}, \mathbf{x} \rangle$ as a result of Effect 1d. Thus, $p_t(\mathbf{y}) = \{k\}$ and $k \in p_t(\mathbf{x})$, which proves $p_t(\mathbf{y}) \subseteq p_t(\mathbf{x})$.

   If instead $\mathbf{z} \notin \{\mathbf{x}, \mathbf{y}\}$, it must be that $d_t(\mathbf{x}) < d_t(\mathbf{y}) < d_t(\mathbf{z})$. By inductive assumption, $p_{t-1}(\mathbf{y}) \subseteq p_{t-1}(\mathbf{x})$. As before, let k be the PUK that is introduced at Effect 1c. At Effect 1d, $\mathbb{B}_t$ is augmented with $\langle k, \mathbf{z}, \mathbf{y} \rangle$ and $\langle k, \mathbf{z}, \mathbf{x} \rangle$, and boxes for other devices. Thus, $p_t(\mathbf{x}) = p_{t-1}(\mathbf{x}) \cup \{k\}$, and $p_t(\mathbf{y}) = p_{t-1}(\mathbf{y}) \cup \{k\}$. Combining with the inductive assumption, it follows that $p_t(\mathbf{y}) \subseteq p_t(\mathbf{x})$.

2. DeviceRevoke and PerUserKeyRotate follow the same logic as device provisioning, so we leave out the argument for brevity.

3. BatchApprove is the interesting case. Device $\mathbf{z}$ approves all younger devices at time t. We consider three disjoint subcases:

   (a) $d_t(\mathbf{z}) \leq d_t(\mathbf{x}) < d_t(\mathbf{y})$. As a result of Effect 4b, we get that $p_t(\mathbf{x}) = p_{t-1}(\mathbf{x}) \cup p_t(\mathbf{z})$ and $p_t(\mathbf{y}) = p_{t-1}(\mathbf{y}) \cup p_t(\mathbf{z})$. By inductive assumption we have that $p_{t-1}(\mathbf{y}) \subseteq p_{t-1}(\mathbf{x})$. Combining these three statements proves $p_t(\mathbf{y}) \subseteq p_t(\mathbf{x})$.

   (b) $d_t(\mathbf{x}) < d_t(\mathbf{z}) < d_t(\mathbf{y})$. Here we apply the inductive assumption to $\mathbf{x}$ and $\mathbf{z}$, giving us that $p_{t-1}(\mathbf{z}) \subseteq p_{t-1}(\mathbf{x})$. Effect 4b doesn't change $p_{t-1}(\mathbf{x})$ so therefore $p_{t-1}(\mathbf{x}) = p_t(\mathbf{x})$. Chaining these set relations together gives us that $p_{t-1}(\mathbf{z}) \subseteq p_t(\mathbf{x})$. Next, apply the inductive assumption to $\mathbf{z}$ and $\mathbf{y}$, giving us that $p_{t-1}(\mathbf{y}) \subseteq p_{t-1}(\mathbf{z})$. By Effect 4b, $p_t(\mathbf{y}) = p_{t-1}(\mathbf{y}) \cup p_{t-1}(\mathbf{z})$. By basic set theory, it is still the case that $p_t(\mathbf{y}) \subseteq p_{t-1}(\mathbf{z})$. By transitivity, $p_t(\mathbf{y}) \subseteq p_t(\mathbf{x})$.

(c) $d_t(\mathbf{x}) < d_t(\mathbf{y}) \leq d_t(\mathbf{z})$. Due to Effect 4a, devices $\mathbf{y}$ and $\mathbf{x}$ do not receive any new keys, i.e. $p_t(\mathbf{x}) = p_{t-1}(\mathbf{x})$ and $p_t(\mathbf{y}) = p_{t-1}(\mathbf{y})$. Combining with the inductive assumption $p_{t-1}(\mathbf{y}) \subseteq p_{t-1}(\mathbf{x})$ proves the subcase. ■

The next lemma establishes that older devices will always be grouped into older equivalence classes:

Lemma B.4. *For any device family $\mathcal{F}$, any time t, $\forall \mathbf{x}, \mathbf{y} \in \mathbb{D}_t \setminus \mathbb{R}_t$, if $d_t(\mathbf{x}) < d_t(\mathbf{y})$ then $v_t(\mathbf{x}) \leq v_t(\mathbf{y})$.*

Proof. Let $\mathbf{k} \in e_t(\mathbf{y})$ be such that $d_t(\mathbf{k})$ is minimal (in particular, $v_t(\mathbf{y}) = d_t(\mathbf{k})$). We have 3 cases.

Consider the case $d_t(\mathbf{k}) < d_t(\mathbf{x})$ (which implies $\mathbf{k} \neq \mathbf{y}$, as $d_t(\mathbf{x}) < d_t(\mathbf{y})$). Consider a path from $\mathbf{k}$ to $\mathbf{y}$ along edges in $\mathbb{A}_t$. There must be some $\langle \mathbf{a}, \mathbf{b} \rangle \in \mathbb{A}_t$ along this path such that $d_t(\mathbf{a}) < d_t(\mathbf{x}) \leq d_t(\mathbf{b})$. This implies that $\langle \mathbf{a}, \mathbf{x} \rangle \in \mathbb{A}_t$ (as $\mathbf{a}$ must have approved $\mathbf{x}$ when it approved $\mathbf{b}$), which means $\mathbf{x} \in e_t(\mathbf{y})$, and therefore $v_t(\mathbf{y}) = v_t(\mathbf{x})$.

If $d_t(\mathbf{k}) = d_t(\mathbf{x})$, then $\mathbf{x} = \mathbf{k} \in e_t(\mathbf{y})$ and $v_t(\mathbf{y}) = v_t(\mathbf{x})$.

Finally, if $d_t(\mathbf{x}) < d_t(\mathbf{k})$, we have that by definition $v_t(\mathbf{x}) \leq d_t(\mathbf{x})$ and thus $v_t(\mathbf{x}) \leq d_t(\mathbf{x}) < d_t(\mathbf{k}) = v_t(\mathbf{y})$. ■

In the next lemma we show that a device performing an approval transition doesn't change its own equivalence class.

Lemma B.5. *For any device family, any time t, $\forall \mathbf{x} \in \mathbb{D}_t \setminus \mathbb{R}_t$, if device $\mathbf{x}$ performs approval at time t, then $v_t(\mathbf{x}) = v_{t-1}(\mathbf{x})$.*

Proof. Assume for contradiction there exists a device family, and a time t such that $\mathbf{x}$ does a device approval transition at time t and $v_t(\mathbf{x}) < v_{t-1}(\mathbf{x})$. Pick $\mathbf{i}$ to be minimal in $\mathbf{x}$'s equivalence class at time t, i.e. such that $d_t(\mathbf{i}) = v_t(\mathbf{x})$, and consider a path (without repeated nodes) from $\mathbf{i}$ to $\mathbf{x}$. Since this path does not exist in $\mathbb{A}_{t-1}$ (else it would be $v_{t-1}(\mathbf{x}) \leq d_t(\mathbf{i})$ which leads to a contradiction), any such path must contain one of the edges $\langle \mathbf{x}, \mathbf{b} \rangle \in \mathbb{A}_t \setminus \mathbb{A}_{t-1}$ (with $d_t(\mathbf{x}) < d_t(\mathbf{b})$) added due to Effect 4a in the transition at time t. Since the path does not have repeated edges, the sub-path between $\mathbf{b}$ and $\mathbf{i}$ must be composed of edges in $\mathbb{A}_{t-1}$, from which we derive $v_{t-1}(\mathbf{b}) = d_{t-1}(\mathbf{i})$. Putting it altogether, at time $t-1$ we have that $d_{t-1}(\mathbf{x}) < d_{t-1}(\mathbf{b})$ but $v_{t-1}(\mathbf{x}) > v_t(\mathbf{x}) = d_{t-1}(\mathbf{i}) = v_{t-1}(\mathbf{b})$ which contradicts Lemma B.4. ■

Additionally, a device performing an approval cannot change the equivalence classes of older devices.

Lemma B.6. *For any device family, any time t, $\forall \mathbf{x}, \mathbf{y} \in \mathbb{D}_t \setminus \mathbb{R}_t$ such that $d_{t-1}(\mathbf{x}) < d_{t-1}(\mathbf{y})$, if $\mathbf{y}$ runs approval at time t then $v_t(\mathbf{x}) = v_{t-1}(\mathbf{x})$.*

Proof. The proof is similar to the one of the previous lemma. Assume for contradiction there exists a device family, and a time t such that $\mathbf{y}$ does a device approval transition

at time t and $v_t(\mathbf{x}) < v_{t-1}(\mathbf{x})$ for some device $\mathbf{x}$ such that $d_t(\mathbf{x}) < d_t(\mathbf{y})$. Pick $\mathbf{i}$ to be minimal in $\mathbf{x}$'s equivalence class at time t, i.e. such that $d_t(\mathbf{i}) = v_t(\mathbf{x})$, and consider a path (without repeated nodes) from $\mathbf{i}$ to $\mathbf{x}$. Since this path does not exist in $\mathbb{A}_{t-1}$ (else it would be $v_{t-1}(\mathbf{x}) \leq d_t(\mathbf{i})$ which leads to a contradiction), any such path must contain one of the edges $\langle \mathbf{y}, \mathbf{b} \rangle \in \mathbb{A}_t \setminus \mathbb{A}_{t-1}$ (with $d_t(\mathbf{x}) < d_t(\mathbf{b})$) added due to Effect 4a in the transition at time t. Since the path does not have repeated edges, the sub-path between $\mathbf{b}$ and $\mathbf{i}$ must be composed of edges in $\mathbb{A}_{t-1}$, from which we derive $v_{t-1}(\mathbf{b}) = d_{t-1}(\mathbf{i})$. Putting it altogether, at time $t-1$ we have that $d_{t-1}(\mathbf{x}) < d_{t-1}(\mathbf{y}) < d_{t-1}(\mathbf{b})$ but $v_{t-1}(\mathbf{x}) > v_t(\mathbf{x}) = d_{t-1}(\mathbf{i}) = v_{t-1}(\mathbf{b})$ which contradicts Lemma B.4. ■

Finally, we can prove Theorem B.1. The proof is by induction over t. It is trivially true at time $t = 0$, since there is only the null equivalence class. Assume it is true for time $t-1$, and we prove true tor time t. We proceed case-wise, reasoning over the four transition types that could explain the transition from $t-1$ to t.

And now to the case-wise analysis:

1. DeviceAdd. A new device $\mathbf{z}$ is introduced at time t. In this case, since $\mathbf{x} \equiv_t \mathbf{y}$, $\mathbf{z}$ cannot be neither $\mathbf{x}$ nor $\mathbf{y}$, and also $\mathbf{x} \equiv_{t-1} \mathbf{y}$. We need to prove $p_t(\mathbf{x}) = p_t(\mathbf{y})$, but without loss of generality, we can prove one inclusion, and argue the other follows by symmetry. Given $k \in p_t(\mathbf{x})$, we need to prove $k \in p_t(\mathbf{y})$. We consider these subcases:
   (a) $k \in p_{t-1}(\mathbf{x})$. Then by inductive assumption, $k \in p_{t-1}(\mathbf{y})$. Since the set $\mathbb{B}_t$ only grows over time, it follows that $k \in p_t(\mathbf{y})$, which proves the case.
   (b) $k \notin p_{t-1}(\mathbf{x})$. That is, there does not exist a $\mathbf{w}$ such that $\langle k, \mathbf{w}, \mathbf{x} \rangle \in \mathbb{B}_{t-1}$. Then it follows that at Effect 1d above, such a member was introduced. Recall that $\mathbf{x}, \mathbf{y} \in \mathbb{D}_t \backslash \mathbb{R}_t$ because the equivalence relation is only defined over devices in $\mathbb{D}_t \backslash \mathbb{R}_t$. The loop in Effect 1d is over all devices in $\mathbb{D}_t \backslash \mathbb{R}_t$, so $\mathbf{x}$ and $\mathbf{y}$ were both included. Thus, it follows that $\langle k, \mathbf{w}, \mathbf{y} \rangle \in \mathbb{B}_t$, and therefore that $k \in p_t(\mathbf{y})$.
2. DeviceRevoke. This case follows much like Case 1 (Device Provisioning), just above. In considering the two subcases $k \in p_{t-1}(\mathbf{x})$ is argued the same way. In the second subcase, $k \notin p_{t-1}(\mathbf{x})$. Then the $\langle k, \mathbf{w}, \mathbf{y} \rangle$ triple must have been introduced at Effect 2c, and by similar argument as above $\langle k, \mathbf{w}, \mathbf{y} \rangle \in \mathbb{B}_t$, and therefore $k \in p_t(\mathbf{y})$.
3. PerUserKeyRotate. This case follows from the same reasoning as the other two cases above (provisioning and revocation).
4. BatchApprove. This is the most interesting case. Let $\mathbf{w}$ be the device that's doing the approval at time t. Without loss of generality, assume that $d_t(\mathbf{x}) < d_t(\mathbf{y})$. The three cases to consider is how $d_t(\mathbf{w})$ is interwoven with $d_t(\mathbf{x})$ and $d_t(\mathbf{y})$.
   (a) $d_t(\mathbf{w}) \leq d_t(\mathbf{x}) < d_t(\mathbf{y})$. By Lemma B.3, $p_{t-1}(\mathbf{x}) \subseteq p_{t-1}(\mathbf{w})$ and $p_{t-1}(\mathbf{y}) \subseteq p_{t-1}(\mathbf{w})$. As a result of Effect 4b, $\mathbf{x}$ and $\mathbf{y}$ will learn about all of the PerUserKeys that $\mathbf{w}$ knows. Combining with the prior observation, $p_t(\mathbf{x}) = p_t(\mathbf{y}) = p_t(\mathbf{w})$.
   (b) $d_t(\mathbf{x}) < d_t(\mathbf{w}) < d_t(\mathbf{y})$. This implies $v_t(\mathbf{x}) \leq v_t(\mathbf{w}) \leq v_t(\mathbf{y})$ by Lemma B.4, and since by assumption $\mathbf{x} \equiv_t \mathbf{y}$, it must be $v_t(\mathbf{x}) = v_t(\mathbf{w}) = v_t(\mathbf{y})$. By Lemmas B.5 and B.6, $v_t(\mathbf{x}) = v_{t-1}(\mathbf{x})$ and $v_t(\mathbf{w}) = v_{t-1}(\mathbf{w})$, and by transitivity, $v_{t-1}(\mathbf{x}) = v_{t-1}(\mathbf{w})$, and therefore $\mathbf{x} \equiv_{t-1} \mathbf{w}$. We apply the inductive assumption to deduce

that $p_{t-1}(\mathbf{x}) = p_{t-1}(\mathbf{w})$. Again $p_{t-1}(\mathbf{x})$ isn't affected by $\mathbf{w}$'s approving newer devices, so $p_{t-1}(\mathbf{x}) = p_t(\mathbf{x})$ and therefore $p_t(\mathbf{x}) = p_{t-1}(\mathbf{w})$ by transitivity. By Lemma B.3, we know that $p_{t-1}(\mathbf{y}) \subseteq p_{t-1}(\mathbf{w})$. By Effect 4b, $\mathbf{y}$ gets all of $\mathbf{w}$'s keys, and $p_{t-1}(\mathbf{w})$ is unchanged, so this gives $p_t(\mathbf{y}) = p_t(\mathbf{w}) = p_{t-1}(\mathbf{w})$. Chaining set equality, $p_t(\mathbf{x}) = p_t(\mathbf{y})$, which proves the case.

(c) $d_t(\mathbf{x}) < d_t(\mathbf{y}) \leq d_t(\mathbf{w})$. By assumption $v_t(\mathbf{x}) = v_t(\mathbf{y})$, and by Lemmas B.5 and B.6, $v_{t-1}(\mathbf{x}) = v_t(\mathbf{x})$ and $v_{t-1}(\mathbf{y}) = v_t(\mathbf{y})$. Chaining, $v_{t-1}(\mathbf{x}) = v_{t-1}(\mathbf{y})$ which means $\mathbf{x} \equiv_{t-1} \mathbf{y}$. Applying the inductive assumption, $p_{t-1}(\mathbf{x}) = p_{t-1}(\mathbf{y})$. Device $\mathbf{w}$ running approval has no effect on $p_{t-1}(\mathbf{x})$ or $p_{t-1}(\mathbf{y})$. That is, $p_{t-1}(\mathbf{x}) = p_t(\mathbf{x})$ and $p_{t-1}(\mathbf{y}) = p_t(\mathbf{y})$. Chaining, $p_t(\mathbf{x}) = p_t(\mathbf{y})$ which proves it. ■

C Cake-AES

Cake-AES is a general-purpose symmetric encryption mode built on top of 256-bit AES-GCM and SHA-512 with the following features:

- Support for incremental encryption of long messages (theoretically, up to 2^{109} bytes)
- Support for incremental, authenticated, random-access decryption of long messages
- A single long-lived key can encrypt a very high number of ciphertexts, by using large 24-byte nonces
- Nonces are internally generated, reducing the likelihood of nonce misuse
- Ciphertexts commit to both the key and (indirectly) the message

By contrast, AES-GCM and ChaCha20-Poly1305 both use 12-byte nonces, and have a message size limit of 64 GiB and 256 GiB respectively. While both support random-access decryption, the full message must be decrypted to be authenticated.

Cake-AES private keys are 32-byte strings generated uniformly at random. At a high level, Cake-AES derives a one-time key by hashing the private key with the nonce, breaks the plaintext into 16 KiB chunks, and encrypts each chunk with AES-GCM and a position-based nonce. Below, we describe in more depth how to encrypt and decrypt data using Cake-AES.

C.1 Encryption

- Generate a random 24-byte nonce.
- Concatenate the caller's 32-byte key and the random nonce and hash them with SHA-512. Use the first 32 bytes of the hash as a one-time key, and the second 32 bytes as a key commitment.

- Split the message into 16 KiB chunks, with a short final chunk that may be empty.
- Encrypt each chunk with AES-GCM using the one-time key. For the AES-GCM nonce, use the little-endian encoding of either twice the chunk index (for non-final chunks) or twice the chunk index plus one (for the final chunk).
- Concatenate the random nonce, the key commitment, and all the encrypted chunks to produce the ciphertext.

C.2 Decryption

- Read the random nonce and the key commitment from the front of the ciphertext.
- Hash the caller's 32-byte key and the random nonce with SHA-512. The first 32 bytes of the hash are the one-time key. Assert that the second 32 bytes match the key commitment.
- Split the remainder of the ciphertext into 16400-byte chunks (16 KiB plus 16 bytes for the GCM tag), with a short final chunk.
- Decrypt each chunk with AES-GCM using the one-time key and the same chunk nonce schedule as above.
- Concatenate all the plaintext chunks to form the original message. Or, since each chunk is independently authenticated, return chunks of plaintext to the caller as a stream.

C.3 Random-Access Decryption

- If the one-time key is not yet cached:
  - Read the random nonce and the key commitment from the front of the stream, derive the one-time key, and verify the key commitment as above.
  - Cache the one-time key.
- If the authentic plaintext length is not yet cached:
  - Get the apparent ciphertext length from the underlying stream. For example, with a ciphertext saved on disk, this would be the file size. This value may be inauthentic, and we need to authenticate it.
  - Compute the apparent length of the final ciphertext chunk. This is the total length, minus the random nonce and commitment lengths (56), modulo 16400. Subtract that length from the total to give the apparent start offset of the final ciphertext chunk.
  - Seek the underlying stream to that start offset.
  - Read the final ciphertext chunk and decrypt it with AES-GCM using the one-time key and the appropriate final chunk nonce.

  - If decryption succeeds, the ciphertext length is authentic. Compute the authentic plaintext length by subtracting the random nonce and commitment lengths, plus 16 bytes times the number of chunks (the GCM tags), from the ciphertext length.
  - Cache the authentic plaintext length.

- If the plaintext seek target is beyond the authentic plaintext length, arrange to return EOF to the caller for any subsequent reads.
- Otherwise, seek the underlying stream to the start of the ciphertext chunk corresponding to the caller's plaintext seek target. To account for the seek offset within that chunk, arrange to skip the appropriate number of bytes after the next decryption.