E2E Encryption for Zoom Meetings

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], 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

October 5, 2021
Version 3.1

Contents

1 Introduction

Hundreds of millions of participants join Zoom Meetings each day. They use Zoom to learn among classmates scattered by recent events, to connect with friends and family, to collaborate with colleagues and, in some cases, to discuss critical matters of state. Zoom users deserve excellent security, and Zoom is working to provide these protections in a transparent and peer-reviewed process. This document, mindful of practical constraints, proposes major security and privacy enhancements for Zoom.

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 Background and Current System

Zoom Meetings currently use encryption to protect identity, data for meeting setup, and meeting contents. 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" we include all of these various forms of packaging. Crucially, these are systems to which we can deploy cryptographic software. Zoom Meetings also supports web browsers through a combination of WebRTC and custom code. If enabled, Zoom meetings support the use of clients not controlled by Zoom, namely phones using the public switched telephone network (PSTN) and room systems supporting SIP and H.323.

In the meeting setting (as opposed to webinars), Zoom supports up to 1,000 simultaneous users. When a Zoom client gains entry to a Zoom meeting, it gets a 256-bit per-meeting key created by the Zoom server, which retains the key to distribute it to participants as they join. In the version of Zoom's meeting encryption protocol released on May 30, 2020, this per-meeting key is used to derive a per-stream key by combining the per-meeting key with a non-secret stream ID using an HMAC function. Each stream key is used to encrypt audio/video (UDP) packets using AES in GCM mode, with each client emitting one or more uniquely-identified streams. 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. There is no mechanism to re-key a meeting.

Videoconferencing systems in which the server relies on plaintext access to the meeting content to perform operations such as multiplexing would be exceptionally difficult to secure end-to-end. In this design, we take advantage of the fact that the Zoom servers do not require any access to meeting content, allowing end-to-end security at exceptionally large scale.

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 current design provides confidentiality and authenticity for all Zoom data streams by providing encryption between Zoom client endpoints. However, it does not provide end-to-end key management. In the current implementation, a passive adversary who can monitor Zoom's server infrastructure and who has access to the memory of the relevant Zoom servers may be able to defeat encryption. The adversary can observe the shared meeting key (MK), derive session keys, and decrypt all meeting data. Zoom's current setup, as well as virtually every other cloud product, relies on securing that 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 Zoom infrastructure.

1.2 Goals and Threat Model

This document proposes upgrades to Zoom that achieve end-to-end security against a range of powerful adversaries. In particular, we consider the following classes of adversaries:

Outsiders: Individuals who are not part of Zoom's trusted infrastructure, and do not have access to non-public meeting access control information (e.g., meeting passwords, IDs, SSO systems). These attackers may monitor, intercept, and modify network traffic, but do not have access to Zoom infrastructure.

Meeting participants: Participants who can access a meeting, because they know meeting's ID and password or exercise other qualifying credentials.

Insiders: Those who develop and maintain Zoom's server infrastructure and its cloud providers.

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

Confidentiality: Only authorized meeting participants should have access to meeting audio and video streams. People removed from a meeting should have no further access after their expulsions.

Integrity: Those who are not allowed into a meeting should have no ability to corrupt the content of a meeting.

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

We note that the current Zoom Meeting system has many highly-effective server-driven security mechanisms that are orthogonal to this proposal's concerns, and therefore remain unchanged.

1.3 Limitations

To achieve these objectives would be an important improvement to Zoom's overall security and would give Zoom's users additional assurances that their meetings are secure along the axes they care most about. However, as with any security program, there are limitations to our approach.

First, we intend to 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, meetings may become vulnerable because of attacks on their infrastructure.

Second, there are certain classes of attack and threats that we deem out of scope, including:

In-meeting impersonation attacks: A malicious but otherwise authorized meeting participant colluding with a malicious server can masquerade as another authorized meeting participant.

Metadata and traffic analysis: Even for end-to-end encrypted meetings, insiders and outsiders can learn details about meeting duration, meeting bandwidth, data streaming patterns, participant lists 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 guarantees we can make. Zoom relies on extensive analyses by independent third party auditors to reassure customers in this domain.

Third, we note that Zoom has a rich feature set and works on multiple platforms. Some of these features and use cases might be incompatible with strong cryptographic processes. Consider, for instance, dial-in phones or SIP/H.323 devices, which 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. E2E security of the type contemplated by this paper is not possible in meetings that support these legacy standards.

Fourth, users can access Zoom meetings through their web browser, and without installing Zoom's client. Supporting web users poses certain challenges: secure, long-term storage for cryptographic private keys might be unavailable; and worse, malicious web servers could feed backdoored source code to web users with little chance for discovery. We intend to participate in the web standards development process to facilitate the creation of browsers upon which we could offer dependable E2E security.

Finally, although we provide a simplification of the meeting protocol for one-to-one Zoom Phone calls in Section 3.14, the current document does not outline a solution for Zoom's webinar product, which supports meetings with up to 50,000 participants in a setting where many participate in a receive-only mode. Encryption for this setting will require a slightly different solution, and will be adapted from the current proposal once it is stable. Nor does

this document outline a solution for the Zoom Chat product, which will require a different design that supports the asynchronous and persistent aspects of chat versus synchronous, ephemeral video conferencing.

1.4 Outline

This proposal lays out a long-term roadmap for E2E security in Zoom in four phases. The first phase is an upgrade of the meeting key exchange protocol to use public-key cryptography, hiding all secret keys from the server. The next three phases harden the notion of a user's identity—even across multiple devices—to help maintain server honesty in the key exchange and to give hosts better information when allowing or disallowing participation in a meeting. We provide the most detail for the earliest stages. We surmise that the specifics of later phases will change with more implementation and deployment experience.

2 Roadmap

We propose a preliminary, incrementally-deployable four-phase roadmap.

2.1 Phase I: Client Key Management

In the first phase, we will roll out public key management, where every Zoom application generates and manages its own long-lived public/private key pairs; those private keys are known only to the client. From here, we will upgrade session key negotiation so that the clients can generate and exchange session keys without needing to trust the server. In this phase, a malicious party could still inject an unwanted public key into this exchange. We offer "meeting leader security codes" as an advanced feature, so motivated users can verify public keys. The security to be achieved here will approximate those of Apple's FaceTime and iMessage products.

The key improvement in Phase I is that a server adversary must now become active (rather than passive) to break the protocol. In Phase I, we will support native Zoom clients and Zoom Rooms. We will not support Web browsers, PSTN dial-in, and other legacy devices. There also will be no support for "Join Before Host", Cloud Recording, and some other Zoom features.

2.2 Phase II: Identity

In the first phase, clients will trust Zoom to accurately map usernames to public keys. In Phase II, we will introduce two parallel mechanisms for users to track each other's identities without trusting Zoom's servers. For users who authenticate to Zoom via Single-Sign-On (SSO), we will allow the SSO IDP (Identity Provider) to sign a binding of a Zoom public

key to an SSO identity, and to plumb this identity through to the UI. Unless the SSO or the IDP has a flaw, Zoom cannot fake this identity. Second, we allow users to track contacts' keys across meetings. This way, the UI can surface warnings if a user joins a meeting with a new public key.

2.3 Phase III: Transparency Tree

In the third phase, we will implement a mechanism that forces Zoom servers (and SSO providers) to sign and immutably store any keys that Zoom claims belong to a specific user, forcing Zoom to provide a consistent reply to all clients about these claims. Each client will periodically audit the keys that are being advertised for their own account and surface new additions to the user. Additionally, auditor systems can routinely verify and sound the alarm on any inconsistencies in their purview. In this scenario, if Zoom were to lie about Alice's keys (say, in order to join a meeting which Alice is invited to), it would have to lie to everyone in a detectable way. We will obtain these guarantees by building a "transparency tree," similar to those used in Certificate Transparency [12] and Keybase [2].

During this phase we will also provide the capability for meeting leaders to "upgrade" a meeting to end-to-end encrypted once it has begun, provided that all attendees are using the necessary client versions and incompatible features are not in use. Such incompatible features include PSTN dial-in and SIP/H.323 room systems. Meetings that cannot be upgraded will have the option grayed-out.

We also re-enable "Join Before Host" mode, and are exploring options to re-introduce a safer version of the Cloud Recording feature.

2.4 Phase IV: Real-Time Security

Consider this hypothetical attack against the Phase III design: a malicious Zoom server introduces a new "ghost" device for Bob, a user who does not have their IDP vouch for their identity. The attacker, using this fake new device, starts a meeting with Alice. Alice sees a new device for Bob but does not check the key fingerprint. After the fact, Bob can catch the server's malfeasance, but only after the attacker tricked Alice into divulging important information. The transparency tree encourages a "trust but verify" stance, where intrusions cannot be covered up. In Phase IV, we look to the future where Bob should sign new devices with existing devices, use an SSO IDP to reinforce device additions, or delegate to his local IT manager. Until one of these conditions is met, Alice will look askance at Bob's new devices.

3 Phase I: Client Key Management

Phase I builds out public-key driven session key negotiation for Zoom Meetings. Let's first dive into how Zoom Meetings works and then propose changes.

3.1 Current Design

Each standard Zoom meeting is an interaction with up to 1,000 participants. Meetings are identified externally by a short meeting identifier.

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 participant must possess the meeting ID as a precondition for joining a meeting. Current meeting IDs are short identifiers that must be known by the Zoom infrastructure to enable routing of data between meeting participants.

In the current system, access control to meetings is implemented via several mechanisms:

- 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 retain these server-enforced access control features; they are largely orthogonal to the E2E encryption design aside from where described.

3.2 Meeting UI Changes

The meeting setup interface will feature a new checkbox: "End-to-End Security." This bit is persisted across the scheduling system, and cannot be unset once the meeting starts. When checked, the behavior of the meeting changes in several key ways:

- The "Enable Join Before Host" checkbox becomes grayed out and deselected.
- All participating clients must run the official Zoom client software; those on dial-ins, web browsers, or legacy Zoom-enabled devices are locked out of the meeting.
- The Cloud Recording feature becomes disabled.

Once the meeting starts, there are other important UI changes:

- All participants will receive a clear indication of the security level of every meeting.
- They can see a "meeting leader security code" that they can use to verify that no one's connection to the meeting was intercepted. The host can read this code out loud, and all participants can check that their clients display the same code.

3.3 A Basic E2E Key Agreement Proposal

All meeting content sent between Zoom clients is currently encrypted using a "meeting key" that is distributed by the Zoom infrastructure. This key is pluralized into a set of per-client/per-stream encryption keys, which are then used to encrypt A/V and chat streams sent to other clients via Zoom's infrastructure. After the initial key sharing (and excluding value-added services such as Cloud Recording and PSTN) Zoom's infrastructure servers do not require knowledge of this key.

The goal of the new design, therefore, is simply to eliminate Zoom's role in distributing this initial shared meeting key material, and to shift this responsibility to the participating Zoom clients.

In the revised approach, all keys will be generated and distributed between individual authorized meeting participants that run the Zoom client software. 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 a meeting host (e.g., to support abuse reporting.)

3.4 System Components

The system assumes the following components:

Identity management system. The revised 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 overview requires that, at all times, one authorized Zoom client will be present in a meeting and considered the meeting "leader". This

client will have the responsibility of generating the shared meeting key, authorizing new meeting participants, kicking out unwanted participants, and distributing keys. For Phase I, this leader will be the meeting host, and Zoom servers will select a replacement host (giving preference to co-hosts) if the current host leaves. In future phases, we will relax this assumption (and therefore re-enable "Join Before Host").

3.5 Cryptographic Algorithms

All meeting data sent over UDP gets encrypted with AES in GCM mode [9]. Key derivation uses the HKDF algorithm [11]. 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 [8], as detailed below.

3.5.1 Signing

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

- Sign.KeyGen generates a keypair (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`).

3.5.2 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 keypair $(pk_{\mathsf{Box}}, sk_{\mathsf{Box}})$

To generate a keypair:

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 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 keypair 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/Poly-1305 over the plaintext M with the symmetric key K, the associated data D, and the nonce 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 keypair 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.

3.6 Long-term Key Management

When user i signs up, or upgrades their Zoom application to the first version that supports E2E as described in this proposal, their client generates a long-term signing keypair:

$$(IVK_i, ISK_i) \leftarrow \mathsf{Sign.KeyGen}()$$

The Zoom client posts the mapping $\langle (i, \mathsf{deviceID}) \rightarrow IVK_i \rangle$ to the server, signed with Sign.Sign under ISK_i. This self-signed binding becomes available to those who join user i in meetings.

The client will persist this keypair indefinitely on this device and secure ISK_i using whatever mechanisms the local hardware and operating system provide. Of course, ISK_i never leaves the device and must be excluded from any cloud backups.

A device may lose its long-term key after an OS reinstall, a disk corruption, an app reinstall on mobile, and so on. In this case, it appears to the system as a new device and goes through the provisioning process as a new device would.

The specifics for client storage of long-term keys are detailed in Section 3.11.

3.7 Join/Leave Protocol flow

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 3.10, 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 1 describes the basic flow of a leader admitting a participant into the meeting.

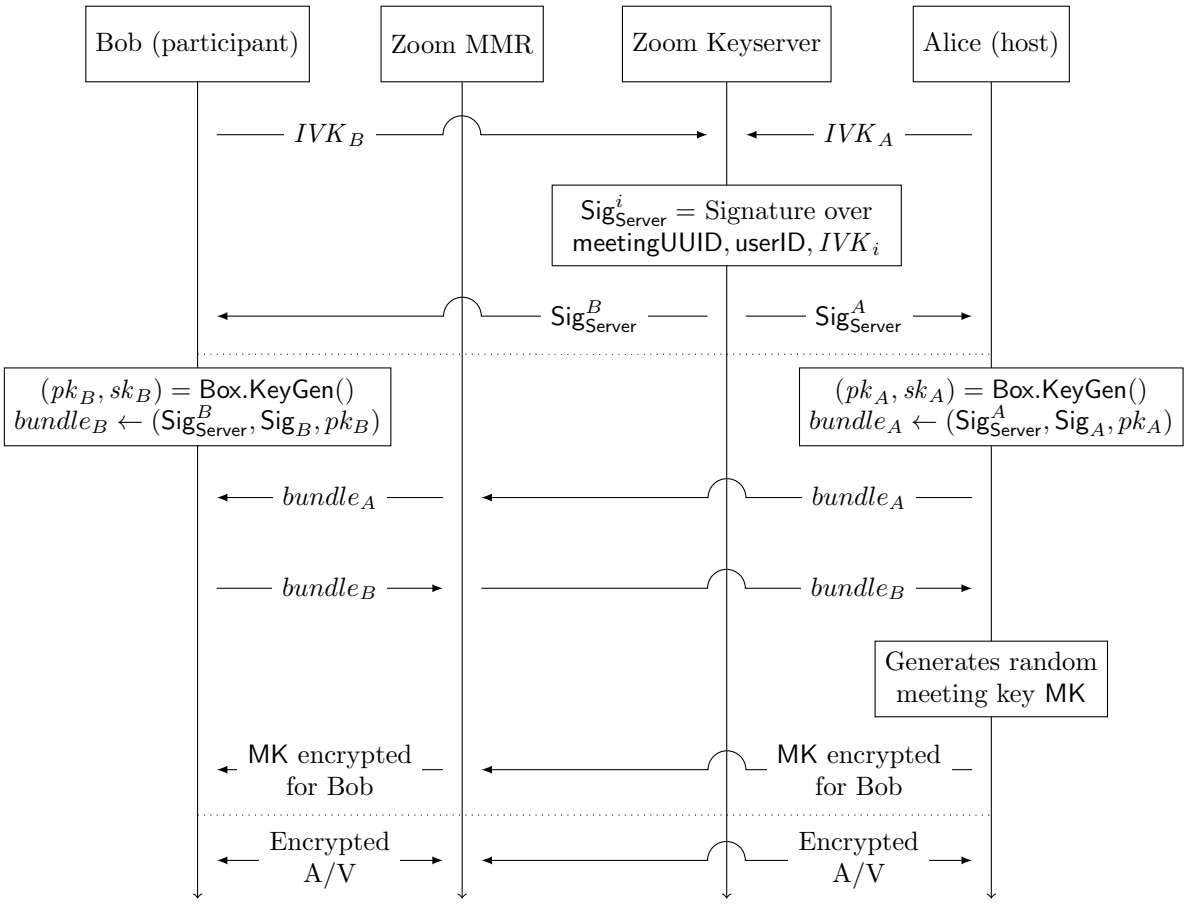

Figure 1: Protocol flow diagram for a leader accepting a participant into the meeting

3.7.1 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, deviceID, and IVK are authorized.

We'll use certificate pinning to strengthen the security of the server signature. Zoom clients will ship with a DigiCert root certificate and they will only trust certificates authorized for a specific Zoom domain via a certificate chain originating from the 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 will not be completed in the initial Phase I release (see Appendix A).

3.7.2 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 keypair: $(pk_i, sk_i) = \mathsf{Box.KeyGen}()$.
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 $\mathsf{Binding}_i \leftarrow (\mathsf{meetingID} \| \mathsf{meetingUUID} \| i \| \mathsf{deviceID} \| IVK_i \| pk_i)$.
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.

3.7.3 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 3.7.6.

3.7.4 Participant Join (Leader)

Given a leader ℓ and a participant i joining meeting meetingID on deviceID, 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 $\mathsf{Binding}_i \leftarrow (\mathsf{meetingID} \| \mathsf{meetingUUID} \| i \| \mathsf{deviceID} \| IVK_i \| pk_i)$.
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".

3.7.5 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 $\mathsf{Binding}_\ell \leftarrow (\mathsf{meetingID} \| \mathsf{meetingUUID} \| \ell \| \mathsf{deviceID} \| IVK_\ell \| pk_\ell)$.
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.

3.7.6 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)". Participants should not immediately encrypt using the new meeting key; they should wait about 2 seconds, to ensure all participants smoothly transition over. Each participant ensures mkSeqNum in the ciphertext sent by the leader is greater than the previously known mkSeqNum; otherwise the key rotation is ignored.

The leader should trigger a rekey whenever a participant enters or leaves the meeting, but not more than every 15 seconds (to prevent thrashing). As an important security measure, users joining the meeting never get to see keys that are more than 15 seconds old, since otherwise they could view encrypted in-meeting chats deep into the meeting's past. Similarly, users who leave a meeting might be able to decrypt meeting content sent in the next 15 seconds (or up to a couple of minutes if the server is suppressing messages to prevent the leader from rotating the key). But likewise this rekeying strategy shouldn't delay a user from joining a meeting.

It's important to note each MK is independently generated, so knowing the previous MK provides no information about the subsequent MK'.

3.7.7 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 $\mathsf{deviceID}$, as well as their display name. For users who have left the meeting, the LPL tracks only their userID, $\mathsf{deviceID}$, 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. It is possible that race conditions during leader changes (caused by a bad network or a malicious server) could prevent some meeting participants from being reflected in the "left" section of the participant list.

In Phase 1, the participant list only identifies users through their self-selected display name and profile picture. Changes to participant's display names are relayed by the server to the leader, who includes them in the LPL, so 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. In the following phases, we will introduce a strong notion of identities to 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 of 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 signalling channel at designated "heartbeat intervals":

$$\mathsf{Sign.Sign}\left(ISK_\ell, \mathsf{Context}, (\mathsf{Binding}_\ell \| \mathsf{SHA256}(LPL_v) \| v \| t \| \mathsf{mkSeqNum})\right)$$

Where $\mathsf{Context}$ is `"Zoombase1-ClientOnly-Sig-LeaderParticipantList"`, t increments on every send, v increments whenever the LPL changes, and $\mathsf{mkSeqNum}$ increments on every MK rotation.

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 signalling 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.

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 $(\mathsf{userID}, \mathsf{deviceID})$ 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 $\mathsf{deviceID}$ must have the same IVK_i, but not necessarily the same pk_i. These guarantees persist only over the course of a single meeting.

3.7.8 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 a participant from learning that the host has locked a meeting by withholding the relevant link and selecting a new host within 100 seconds. This would result in either a meeting partition or warnings about the unlocking for any participant who had received a link that locked the meeting.

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 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).

3.7.9 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.

3.8 Meeting Leader Security Code

If all participants can verify the authenticity of the leader's public key (IVK_ℓ), they are safe from "meddler-in-the-middle attacks" (MitM)[1]. 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

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

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[2] 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."

3.9 Changes to symmetric encryption

Symmetric encryption in Zoom meetings will use AES-GCM with unique per-stream keys. As in the current design, all keys will be derived using a secure key derivation function (KDF) from a per-meeting Meeting Key (MK). The meeting leader may rotate MK throughout the meeting.

All encrypted UDP packets are prefaced with the 4-byte mkSeqNum, so participants know which version of MK to decrypt with.

3.10 Security Properties

The Identity Management System, Bulletin Board and Signaling Channel as enumerated in Section 3.4 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 the design in Section 1.3, 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 3.1. 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 participants 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).

[2] 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.

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). Detecting unauthorized participants can be challenging in large meetings or in certain views, such as when the leader is sharing their screen. In the following phases, 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.

A leader's view of the "active" participants list shows everyone who is in the meeting (i.e., everyone who has been given current meeting key, or the next one if a key rotation is in progress). If the host observes the same user leaving the meeting four or more times, the participant list will reflect the number of observed leaves. This alert helps participants notice when an attacker repeatedly joins and leaves a meeting rapidly, which would allow the attacker to learn most of the meeting keys but remain in the "left" column for the majority of the meeting. Due to the distributed nature of the system, this view of the participant list can be slightly out of date, both for the leader and for the other participants, but will eventually converge if there is a long enough span without participant churn. Some race conditions, such as participants quickly joining and leaving a meeting while the meeting leader is changing, might also cause participants who left a meeting to not be reflected in the "left" section of the participant list. This is necessary to guarantee a smooth meeting experience under poor network conditions. Heartbeat messages ensure that a key is rotated in a timely manner after an unwelcome participant has been removed from the meeting.

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 re-joins) the meeting, or whenever the meeting leader security code changes as indicated by a UI notification.

3.11 Local Key Security

Long-term keys are stored in the local operating system's keychain, but with some added protection for two Zoom users using the same OS account. Before storing keys in the local OS-provided keychain, we will encrypt them using a server-synchronized key-wrapping key. When a user deprovisions their device, they will delete the keychain entry and the server will delete its 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.

We use the committing AEAD scheme $\mathsf{CtE1}$ [10] to prevent the server from supplying malicious key-wrapping keys. On provisioning, after generating ISK_i, the client:

1. Generates a 32-byte random string KWK and requests the server to store it persistently associated with the user.
2. Defines $\mathsf{Context} \leftarrow$ `"Zoombase-1-ClientOnly-KDF-SecretStore"`.

3. Computes $C \leftarrow \mathsf{CtE1\text{-}Enc}(\mathsf{K}{=}\mathsf{KWK}, \mathsf{H}{=}\mathsf{Context}, \mathsf{M}{=}ISK_i)$, where H is the associated data parameter for the underlying AEAD, and stores it in the system keychain.

For CtE1, we will use $\mathsf{HMAC}_{\mathsf{SHA256}}$ as the commitment function and `libsodium` [8]'s `crypto_aead_chacha20poly1305_ietf` for AEAD.

We want to emphasize that this feature will not protect against an insider who also has access to a user's device (in particular, 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.

3.12 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.

3.13 Guest Users

Users can join video meetings in guest mode, in which they will generate fresh long-term keys for every meeting. This prevents other participants from tracing them across meetings by noticing when a long-term key is reused. Guest users include users who are not logged in.

3.14 E2E 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, the current Zoom Phone design encrypts data streams 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 will also deploy end-to-end encryption for Zoom Phone.

In designing E2EE for Zoom Phone, we have 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.

Initially, we will only support E2EE for calls between two Zoom Phone clients in the same account, which greatly simplifies the protocol.

3.14.1 Join/Leave Protocol

We use the same `libsodium` cryptographic primitives and key management system as Zoom Meetings (described in Sections 3.5 and 3.6). 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 3.7.1). They also generate per-call ephemeral DH keys (pk_i, sk_i), sign them with their device keys (similar to Section 3.7.2), 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. 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.

3.14.2 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 3.8), 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.

3.15 Areas to Improve in Phase I

Phase I is geared toward a quick deployment and providing building blocks for future development phases. As such, there are some potential attacks we do not fully cover here:

Meddler-in-the-Middle. The meeting leader security code and phone security code mechanisms above are countermeasures for the classic "meddler in the middle attack", wherein Bob isn't actually connecting to Zoom, he's connecting to Eve who is proxying his communications. But our solution isn't perfect. First off, it can be defeated by deep fake technology; second, it's clunky from a UX perspective; and third, in meetings the leader may have to perform the ceremony multiple times with any users joining the meeting late. We could work toward addressing all of these concerns, but future phases of development provide better solutions by building strong notions of identity.

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. We'll fix this problem with better identity guarantees in Phase II.

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.

The upshot here is that in Phase II, we look to further strengthen meetings through better identity.

4 Phase II: Identity

In Phase II, we will introduce the concept of *identity* and use it to guarantee that only authorized participants will be able to join a meeting. 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 or there is a meddler-in-the-middle, encryption will not be sufficient to protect her. Zoom will give meeting leaders more helpful and trustworthy information to evaluate join requests and kick questionable users out of a meeting.

This section details changes to 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.

4.1 Identity at Zoom Before Phase II

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) which they can use to join meetings and use Zoom services.

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 government cloud for U.S. government employees and contractors, as well as separate white-labeled private Zoom instances, each with their own cloud. Users can join meetings that are hosted in a different cloud from their own.

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.

Individuals do not need to sign in as a Zoom user in order to join a meeting, unless it is configured otherwise. They can join a meeting by clicking a link or by entering the meeting ID and password in the app.

At the time of writing, Zoom gives the host and meeting participants limited information about the identities of the other participants. Aside from the audio and video streams that each individual participant might share, the participants are identified through a profile picture and a display name. This display name can be freely chosen and modified over the course of a meeting (both by the user and by the meeting host) as well as across different meetings. In some cases, account admins can restrict their users to an approved display name, but this is enforced by the Zoom servers and cannot be verified by clients.

Zoom provides some mechanisms to enforce access control in a meeting, such as 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 the host enough information to make a decision about whether to admit an attendee from the waiting room.

4.2 Changes in Phase II

The identity of a Zoom user will consist 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 their meeting partners. Second, each user's identity will include the set of devices (and their cryptographic keys) that are controlled by that user. We describe a device model which 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 includes a mechanism to securely share secret keys between all of a user's devices, which can be used to sync encrypted data. To allow a user's devices to securely communicate with each other, each device will now generate an additional keypair for encryption at the same time it generates its signing keypair as described in Section 3.6.

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

Devices will remember the users and devices they've seen in meetings and will provide warnings to users when they meet with a device they haven't met with before, in order to prevent certain kinds of impersonation attacks.

We also introduce an extension of the OpenID Connect protocol that IDPs can use to make claims about a user's identity that can be verified by other users. If an account's IDP supports this protocol, even Zoom insiders cannot impersonate a user in the account, unless the IDP or one of that user's devices is also compromised.

Finally, there will be several accompanying changes to the user interface, including a device approval process.

4.3 Displaying Identity

This section describes changes to how we display the identity of a Zoom user to others. In Phase II, these changes will mainly pertain to Zoom meetings, but they will eventually be expanded to other Zoom services such as Zoom Chat.

Because cryptographic keys are not easy to read, compare, or keep track of, we will 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
3. An email address, which can be used to distinguish individual users within the account

In Phase II, only users in accounts with identity providers will have identifiers shown next to their display name in meetings. In Phase III, we'll extend this functionality for users in accounts without an identity provider. Here are a few examples of how a user's identifiers would be displayed in-meeting:

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 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 the same device has been used across different meetings, even when the display name is altered.
Richard Roe	As in Phase I, users can join meetings as guests and display no identifying information to other users. Since they generate fresh long-term keys for each meeting, the aforementioned tracing attack is not possible.
Mike Doe (`mike.doe@example.com`)	In Phase III, we will be able to show email addresses for users like Mike whose accounts don't have an ADN or an IDP.

4.3.1 Identifying Accounts

Accounts on Zoom can be optionally identified using a domain name, which we will 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. We will allow internationalized domain names (IDNs) to be used as ADNs, but to prevent homograph attacks, the UI will show the Punycode representation by default and the rendered domain name only on mouse hover.

An account admin may choose one of the account's existing Associated Domains as the ADN, or Zoom can provide a new dedicated subdomain. Associated Domains is an existing Zoom feature where a Zoom account can be associated with multiple domain names. To

do so, the account admin has to prove ownership of the domain name to Zoom, e.g., by adding a specific DNS record, adding a header tag to the home page or by hosting a file at a specific location on the domain.

Only a single account at a time can use a domain name as their ADN, and account admins are allowed to change their ADN if needed.

4.3.2 Identifying Users

Zoom users will also be identified with email addresses. Most users already have an email address associated with their account; moreover emails (unlike names) are unique, and they sometimes represent people better than the legal name held in a company's HR system.

Users will be able to change the email associated with their account. 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 account, then people who used to meet with `support@example.com` when it was associated with Alice's account 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.

In Phase II, email addresses are treated as secondary to the account domain name, which allows account admins to delegate to a single identity provider the ability to attest to the identity of all users within their account, regardless of the email provider they use. In Phase III, we will also guarantee that each email is associated with a single Zoom user at a time, so that even accounts without an ADN or IDP can have identity guarantees.

4.4 Multi-Device Support

Zoom users often have several devices that they use Zoom on: their work computer, personal computer, mobile phone, and so on. In Phase I, each of these devices stores long-term signing keys (called *device keys*) which it uses to authenticate key exchanges at the start of video meetings. In Phase II, we will 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 will be used to join meetings and 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 their keys cannot be used to join meetings or sign new statements.

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 which 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 their meeting partners, 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 meetings 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 meetings 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 they had to recheck the 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 this trust to Alice's new mobile phone's key.

BatchApprove links 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

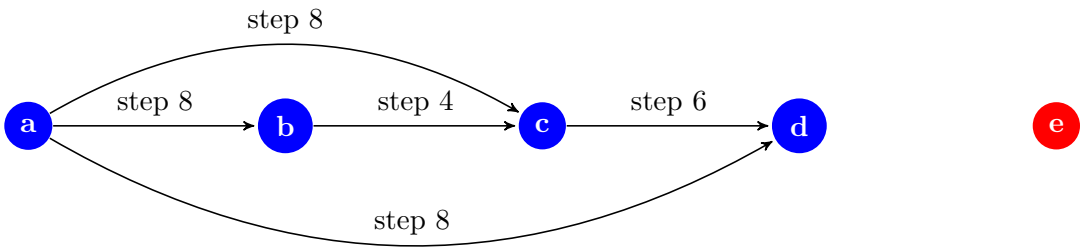

Figure 2: Device approval graph, where nodes are devices and edges are approvals.

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 his meeting partners, now know not to trust statements signed by **e**'s device key. Figure 2 summarizes Bob's trust graph after these steps.

4.4.1 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. PUKs include both symmetric keys and asymmetric encryption keys.

Symmetric PUKs can facilitate syncing encrypted data between a user's devices. Devices use the latest per-user key to encrypt all content, but the 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 PUKs 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 message for Bob using his asymmetric PUK, Bob can still read the message on devices added afterwards, as long as the device has been approved by an earlier one.

In each DeviceAdd or DeviceRevoke operation, devices generate a new seed and encrypt the seed for each previous unrevoked device using its device keys: devices implicitly trust all older devices (otherwise, they would revoke them). Each seed is associated with a number called a *PUK generation* that starts at 1 and increments every time the seed rotates. Note that all devices within an approval class share the same set of PUK seeds.

To generate a set of PUKs, devices:

1. Generate a new 32-byte secret seed
2. Use HKDF on the seed to generate two different 32-byte keys:
   (a) private X25519, context `"Zoombase-2-ClientOnly-KDF-PerUserX25519"`
   (b) 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.

4.5 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. In Phase III, we will add a transparency layer to ensure that the Zoom server must present the same information about sigchains to all users.

4.5.1 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 which modify an object (the sigchain state). For a user sigchain, the sigchain state would contain the list of active devices, 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 will 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.

4.5.2 Overview of Sigchain Types

Zoom devices, users, and accounts are each internally identified by unique immutable identifiers called deviceID, userID and accountID respectively. 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, for each domain name, the accountID to which the domain is associated.
5. Membership sigchains keep track, 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. For efficiency reasons, if this account only has a single user and doesn't have an ADN, then that user's sigchain will not mention their accountID, and there will be no corresponding account or membership sigchains until the account either gets another user or an ADN.

4.5.3 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 4.3, users can set and change the email addresses that will be displayed in the meeting UI. 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. Users will be notified when their account changes.

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
   }),
  "ed25519PublicKey": "ce8564...",
  "x25519PublicKey": "ad7913...",

  "perUserX25519PublicKey": "c2cce1...",

  "emailChange": ...,
  "accountChange": ...,

  "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. 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 key. 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.

If the DeviceAdd link is the first link in a user's sigchain, it might 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.

The user can also decide to revoke any undesired devices at the same time they are adding a new device, 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 by ed25519PublicKey.

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.

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 key and PUKs. This operation keeps the same deviceID but chooses new signing and encryption keys, as well as a new set of PUKs. After others see this link, they will only accept signatures and ciphertexts from the new public keys for this device.

This link is signed by the device's previous public key in addition to the new key.

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 4.4.

PerUserKeyRotate. If a device notices that the last PUK was generated by a device revoking itself, or there was a revocation from the website after the last 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.

4.5.4 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 DeviceAdd link).

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

4.5.5 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
  })
}
```

Links in account sigchains do not require any signatures.

4.5.6 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.

4.5.7 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 Phase III 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.

4.6 Highlighting Untrusted Devices with Contact Sync

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.

4.7 Attesting Users' Identities through External Identity Providers

Accounts that have an ADN and an identity provider (IDP) that supports our extension of the OIDC protocol will be able to have the IDP vouch for the identity of their users in a way that other meeting participants 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 their set of devices and keys

4.7.1 Associating Accounts with Identity Providers

In order to associate an account to its IDP, we will rely on TLS. Accounts with ADNs will host a JSON object at the root of the subdomain `idp-config.well-known.` of the ADN, e.g., `https://idp-config.well-known.example.org`.

The JSON object contains a field with key `us.zoom.idp.<CLOUD_NAME>` whose value points to the unique domain that the IDP has reserved for this account. For example, if `example.org` is using `generic-idp.com` as their identity provider and has an account hosted on the Zoom commercial cloud, the object might look like:

```
{
    "us.zoom.idp.commercial": "examplecorp.generic-idp.com"
}
```

Specifying the cloud in the field name allows to support ADNs configuring different IDPs for different clouds. Note that since accounts are expected to change their IDP rarely, clients can cache this mapping aggressively.

If the endpoint does not return a valid response, clients will assume there is no IDP and not display any identifiers for this user. Because Zoom will proxy client connections to ADNs to avoid leaking the IP addresses of Zoom clients to external parties, a malicious Zoom server (or network attackers) can convince clients that a certain account is not using an IDP by e.g., returning a timeout error or a DNS NXDOMAIN message. We believe that this is the right tradeoff for the following reasons:

1. Configuring the IDP via subdomains offers flexibility for the account admins. For example, the account admin can delegate the `idp-config.well-known` subdomain to the IDP, which can obtain a TLS certificate for the subdomain and host the required JSON file. The only technical action required by the account admin would be adding a DNS record for the subdomain.
2. In Phase III, our transparency layer will ensure that if the server tampers with an account's ADN or IDP, the misbehavior will be detected through auditing of the account and ADN sigchains.

4.7.2 IDP Attestations

IDP attestations will be 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 will 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. IDPs will keep 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 ID tokens about a specific user identity can be validated by other users.

Our modified OIDC ID token (which will also be referred 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

  "name": "Alice Henderson", // optional
  "email": "alice@example.com", // optional

  "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 domain that associates the IDP with the account (in our example, as returned by `https://example.org/.well-known/idp-config`). `name` and `email` are marked as optional to allow users to certify that they are part of a specific account without disclosing exactly which member they are. `iat` and `exp` define the validity of the token.

The `zoom-identity-snapshot` field encodes the state of a user's identity. As explained in Section 4.5, clients keep track of these states using sigchains. An identity snapshot will encode the user sigchain tail (which itself commits to the userID); however, the identity provider can treat this attribute as an opaque string and does not need to check its validity.

4.7.3 Updating Snapshots

Users who successfully sign into Zoom using OAuth 2.0 receive an access token from the IDP which can be used to access protected resources. This access token will also be used to read and update their own `zoom-identity-snapshot` attribute. To ensure that only an authorized user on a Zoom native client is able to update the identity snapshot stored by the IDP, the Identity Snapshot protocol requires IDPs to:

1. Introduce a new OAuth scope `update-zoom-identity` which is required to update the snapshot.
2. Only issue access tokens with the `update-zoom-identity` scope for requests that use PKCE and have set the redirect URI to a fixed custom URI intended to refer to the native Zoom desktop and mobile applications: `us.zoom://idp_auth`.

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.

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.

Whenever devices make updates to their user sigchain, they submit the latest sigchain tail to their IDP. In the event where a device successfully makes a sigchain update with the Zoom server but fails to update the IDP, the next device that comes online will notice the new sigchain links and update the IDP itself, but only after the user has reviewed the new links for potentially malicious device additions.

4.7.4 Validating IDP Attestations

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

1. Verify that the IDP in the `iss` field of the JWT matches the JSON hosted at the ADN's `idp-config.well-known.` subdomain. This ensures that the IDP is authorized to sign on behalf of the account ADN (as specified in Section 4.7.1).
2. Use OpenID Connect Discovery (e.g., make a request to `https://examplecorp.generic-idp.com/.well-known/openid-configuration`) to ensure that the key used to sign the JWT is valid.
3. Validate the JWT, including checking its signature and expiration date.
4. Validate that the email and snapshot match the user sigchain provided by the Zoom server. When in-meeting, users will accept attestations that do not cover the latest sigchain tail as long as the new links since the IDP's snapshot do not revoke the device currently being used in the meeting. If the new links change the user's email or account, then the user will be shown without identifiers.

Note that the fetched attestation may be shared with and validated by anyone, so it doesn't include the `aud` field.

4.8 Changes to the Client

Phase II includes new UI features as well as changes to client behavior during meetings.

4.8.1 Device Management Changes

Device management is now backed by the user's sigchain. Upon visiting the device list, clients will 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 will indicate active devices (which can be used to join E2EE meetings) and revoked devices (which can no longer be used). It may also contain devices added prior to Phase II that are not represented in the sigchain. Users will be able to revoke devices from this view. If a device realizes that it is revoked (either via a server-trust notification or by playing back its own sigchain), it will delete all private ephemeral and long-term keys as well as sensitive data, then log itself out.

When provisioning Zoom on a new device, users review the device list and mark any that are unrecognized, lost, stolen, or no longer used. Such devices are revoked at the same time the new device is added to the sigchain.

After provisioning a new device, users get 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 or account.

These notifications are powered by the sigchain, so the Zoom server does not have the ability to suppress them in order to hide a malicious device addition or email change.

4.8.2 Changes to Meeting Join/Leave Flow

When joining a meeting, devices use the device key specified in their user sigchain as the long-term signing key IVK_i for the protocol. The server signature as described in Section 3.7.1 now also includes the sigchain tails for the corresponding user, email, account, and ADN sigchains.

In the "Participant Key Generation" procedure of Section 3.7.2, the signed $\mathsf{Binding}_i$ will be extended to include the corresponding user, email, account, and ADN sigchains. This ensures that the server cannot equivocate the user's own account. If the account uses an IDP, the client will fetch and include an attestation from the IDP in the binding as well.

In a meeting, Alice's client verifies Bob's sigchains and IDP attestation (if applicable) 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, and IDP attestation. Alice's client verifies the server signature of the binding, checks that Bob's latest sigchain is consistent with any previous retrievals of Bob's sigchain, and verifies the IDP attestation by connecting to the ADN and verifying the IDP signature. Meeting participants' clients may perform these checks asynchronously during a meeting, but the meeting may be configured to require that the host completes this verification before the host performs the key exchange with a joining participant. For example, requiring this verification before key exchange can help increase the security of meetings that are restricted to users in specified accounts. Details about changes to how users' identities are displayed in the meeting room, participant list, and waiting room are described in Sections 4.3 and 4.6.

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.

4.9 Security Properties

Users in accounts with an IDP that supports our protocol receive particularly strong security guarantees. Because clients rely the account's ADN (and not the Zoom server) to determine the IDP, a Zoom insider cannot impersonate these users unless the IDP or TLS itself is compromised. The Zoom proxy or network attackers can potentially trick a user into believing that another user's account's ADN does not have an associated IDP, but in this case, the user will appear without an email address or ADN.

For all users, including those without an IDP or those whose IDP has been compromised, previous meeting partners get Contact Sync warnings regarding new, potentially malicious devices.

The Zoom server cannot force devices to forget identity updates like device revocations: when receiving sigchains, devices make sure the server cannot "rollback" a sigchain to a previous point in time.

In addition to the identifiers displayed in the user interface, our solution provides some limited extra information about a user to their meeting participants: 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). 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 sigchain statements are timestamped, time correlations between different statements might be exploited to infer, for example, that two users swapped their email addresses. While this information will not be displayed to the meeting participants by the Zoom app, 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: they are provided as needed to other meeting participants during the meeting. Even if Alice meets with Bob on one day, the Zoom server will not tell Bob about any updates to Alice's sigchain afterwards until they are in a meeting again.

We stress that 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 4.6, 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.

4.10 Areas to Improve in Phase II

While Phase II provides significantly stronger security guarantees compared to Phase I, there are still attacks it won't be able to prevent.

Though Contact Sync warnings improve Zoom users' security, alert fatigue can make the warnings less useful. If meeting partners regularly fail to approve new devices with earlier ones, or if the server does not properly sync meeting records, 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.

While devices ensure that the server cannot rollback a sigchain that the device has seen before, this doesn't prevent "forks" of sigchains across different devices. For example, a compromised server, in order to impersonate Bob, could add a fake device to Bob's sigchain when Alice requests it at the start of a meeting. When Bob requests his own sigchain to view his device list, the server removes the device addition link so Bob doesn't see the malicious device. If Alice and Bob later join a video meeting, this forking attack would be revealed as users sign over their view of the sigchain tail, but such a meeting is not guaranteed to occur.

Because of the potential for forking attacks in Phase II, we are unable to offer users in accounts without IDPs the ability to display their email address and ADNs in meetings, and we don't offer account admins the ability to exhaustively audit which users are in their account.

In Phase III, we'll introduce a transparency layer that will address these shortcomings.

5 Phase III: Transparency Tree

In the third phase, we will expand the authentication guarantees 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 security codes, as introduced in Phase I) that potentially have to be performed out-of-band. Instead, our plan detects equivocation by the Zoom servers and identity providers while minimizing active checking by the user.

In Phase III, we will force the Zoom server to provide the same mapping between user

accounts and public keys to all clients, to sign such a mapping, and to be 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 can then routinely verify that the server's mapping is consistent over time.

Thus the key fingerprint comparisons from the prior two phases 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.

5.1 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 [12] limits the damage that a compromised certificate authority can do by signing fake TLS certificates. It does so by requiring that all the signed x509 certificates have to 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 [13] 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 will build on prior work to design a new mechanism tailored to Zoom's use cases: the Zoom Transparency Tree (ZTT). The ZTT will be backed by a 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 Phase II) about another user's identity or their keys, this statement will be accompanied by an inclusion proof in the ZTT.

5.2 Integration Details

5.2.1 ZTT Auditing

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

Clients will query the auditors to ensure that their view of the ZTT's commitment 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 (as elsewhere in this protocol).

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.

5.2.2 Provisioning

When provisioning a new device, the client will ensure 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.

5.2.3 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.

5.2.4 Joining a Meeting or Accepting a Join Request

Because we only trust keys stored within the ZTT, users in a meeting will verify that each others' public keys are included in the ZTT, before proceeding with key exchange as in Phase I. If the verification fails, the client will fail to join the meeting.

5.2.5 Contact List Updates

The contact lists that users accumulate in Phase II are now also 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.

5.3 Areas to Improve in Phase III

Though an equivocating server will 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 the clients.

6 Phase IV: Real-Time Security

In Phase III, we described a mechanism for users to detect sigchain corruptions after-the-fact. For instance, an attacker adds a new device onto Alice's sigchain to gain access to her meeting without triggering a "new device" warning. The meeting happens and the attacker gleans the secret. Alice can detect the attack after the fact, since the attacker had to commit the malicious change to Alice's sigchain for the attack to work, but we can do better.

We dub a final refinement to this proposal: "real-time security," meaning Alice can prevent unauthorized device additions from happening in the first place. Users whose identities are vouched for by their SSO provider already have good protections here, since the IDP and Zoom would have to be colluding to authorize a new device. We extend similar protections to other uses in this phase.

In Phase IV, we propose an upgrade to user device provisioning. Users can login to new devices as before: via SSO, OAuth, or simple username and password. Recall from above that new devices get two or three signatures: a self signature, a signature from Zoom, and one from the IDP if available.

Further validations of the device are possible. Users can sign new devices with existing devices, 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.

In this phase, devices with just two signatures can appear "degraded." These degraded devices will trigger warnings in the UI, or would be excluded from meetings marked "high security."

7 Conclusion

We have proposed a roadmap for bringing end-to-end encryption technology to Zoom Meetings (as that term is best understood by security experts). At a high level, the approach is simple: use public key cryptography to distribute a session key to a meeting's participants; and provide increasingly stronger bindings between public keys and user identities. However, the devil is in the details, as user identity across multiple devices is a challenging problem, and has user experience implications. We proposed a phased deployment of E2E security, with each successive stage giving stronger protections. The

crux of the proposal are: (1) delegated authority to SSOs where applicable; (2) signed chains of cryptographic statements (sigchains) for user devices and contact lists; and (3) a privacy-preserving "transparency tree" to tie them tightly together.

We will continue to incorporate feedback and publish subsequent drafts as we refine, implement and deploy these ideas.

7.1 Acknowledgements

We thank Deirdre Connolly, Adriaan De Vos, Yevgeniy Dodis, Takanori Isobe, Ryoma Ito, 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.

[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] Frank Denis. The sodium cryptography library, Jun 2013.

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

[10] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. Cryptology ePrint Archive, Report 2017/664, 2017. `https://eprint.iacr.org/2017/664`.

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

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

[13] 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.

A Release Schedule

Our phases will be rolled out incrementally. This appendix details which Zoom client versions include the features described in this document.

Version 5.4.0 Phase I, except for Locked Meetings (Section 3.7.8), Server Key Certificate Chains (Section 3.7.1) and Zoom Phone (Section 3.14).

Version 5.6.0 Locked Meetings (Section 3.7.8).

Version 5.7.0 Server Key Certificate Chains (Section 3.7.1).

B Understanding Multiple Devices

As detailed in Section 4.4, 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 $t = 6$) $\mathbf{b}$ self-revokes. $v_6(\mathbf{c})$ remains at 1, even though the only path from $\mathbf{a}$ to $\mathbf{c}$ at $t = 6$ crosses $\mathbf{b}$, 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 $\mathbf{x}$ then $\mathbf{y}$, then approves $\mathbf{y}$ with $\mathbf{x}$, then $\mathbf{x}$ and $\mathbf{y}$ are in the same equivalence class, even though she never approved $\mathbf{x}$ with $\mathbf{y}$. We think this is a useful simplification. The rationale is that when $\mathbf{y}$ was added, Alice had access to the list of her devices on $\mathbf{y}$ which includes $\mathbf{x}$, so we assume she implicitly approved $\mathbf{x}$ with $\mathbf{y}$, 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 assuption 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, $\forall \mathbf{x} \in \mathbb{D}_t \backslash \mathbb{R}_t$, 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, $\forall \mathbf{x}, \mathbf{y} \in \mathbb{D}_t \backslash \mathbb{R}_t$ 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 casewise, reasoning over the four transition types that could explain the transition from $t-1$ to t.

And now to the casewise 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. ■