E2E Encryption for Zoom Meetings

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

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

May 22, 2020
Version 1

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 guarantees, 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 upgrades for Zoom.

We are at the beginning of 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 Zoom's servers, which retain the key to distribute it to participants as they join. In the version of Zoom's meeting encryption protocol set for release 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, but it does not provide "true" end-to-end (E2E) encryption as understood by security experts due to the lack of 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, 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 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. A malicious Zoom server in theory has the ability to swap mappings on-the-fly and to trick participants into entering a meeting with imposters. 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 [10] 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, SIP/H.323 room systems and cloud recordings. Meetings that cannot be upgraded will have the option grayed-out.

We also re-enable “Join Before Host” mode.

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

Each 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 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 reliable 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 will fall back to the co-host with the lowest user ID if the current host leaves. In future phases, we will relax this assumption (and therefore re-enable "Join Before Host").

3.5 Cryptographic Algorithms and Standards

In an ideal world, all public key cryptographic operations could happen via Diffie-Hellman over Curve25519 [3], and EdDSA over Ed25519 [4]. Relative to others, this curve and algorithm family have shown a consistent track record for resilience to common cryptographic attacks and implementation mistakes. These algorithms are currently in review for FIPS certification but, unfortunately, are not currently approved. Therefore, in some cases (like government uses that require FIPS certification) we must fall back to FIPS-approved algorithms, like ECDSA and ECDH over curve P-384. The protocol below has support for both algorithm families, and for now "doubles up" all public key operations. This technique eliminates error-prone branching. Once certification succeeds, we can safely "no-op" the operations over P-384.

We define several standard cryptographic schemes, and then introduce new constructions built from these schemes.

Signature schemes. Our algorithms make use of two signing primitives: (1) ECDSA as specified in [12], implemented over the P-384 elliptic curve, and (2) EdDSA as implemented in [8] using the Ed25519 elliptic curve. We describe these schemes using the following algorithms:

$\mathsf{EdDSA.KeyGen}() \rightarrow (vk_{\mathsf{EdDSA}}, sk_{\mathsf{EdDSA}})$. Generates an EdDSA keypair.
$\mathsf{EdDSA.Sign}(sk_{\mathsf{EdDSA}}, M) \rightarrow \mathsf{Sig}_{\mathsf{EdDSA}}$. Computes an EdDSA signature.
$\mathsf{EdDSA.Verify}(vk_{\mathsf{EdDSA}}, M, \mathsf{Sig}_{\mathsf{EdDSA}}) \rightarrow (\mathsf{true}, \mathsf{false})$. Verifies an EdDSA signature.

$\mathsf{ECDSA.KeyGen}() \rightarrow (vk_{\mathsf{ECDSA}}, sk_{\mathsf{ECDSA}})$. Generates an ECDSA keypair.
$\mathsf{ECDSA.Sign}(sk_{\mathsf{ECDSA}}, M) \rightarrow \mathsf{Sig}_{\mathsf{ECDSA}}$. Computes an ECDSA signature.
$\mathsf{ECDSA.Verify}(vk_{\mathsf{ECDSA}}, M, \mathsf{Sig}_{\mathsf{ECDSA}}) \rightarrow (\mathsf{true}, \mathsf{false})$. Verifies an ECDSA signature.

Diffie-Hellman primitives. We additionally specify Diffie-Hellman algorithms for P-384 (FIPS) and Curve25519 [3]. In the case of P-384, we use the standard base points and key establishment techniques defined in the appropriate NIST standards [13].

$\mathsf{GenCurve25519Keypair}() \rightarrow (pk_{\mathsf{25519}}, sk_{\mathsf{25519}})$. Generates a Curve25519 Diffie-Hellman keypair.
$\mathsf{GenFIPSKeypair}() \rightarrow (pk_{\mathsf{FIPS}}, sk_{\mathsf{FIPS}})$. Generates a FIPS Diffie-Hellman keypair.
$\mathsf{DH}_{\mathsf{25519}}(sk^{\mathsf{S}}_{\mathsf{25519}}, pk^{\mathsf{R}}_{\mathsf{25519}}) \rightarrow SS_{\mathsf{25519}}$. Given public and secret Curve25519 keys, outputs a shared secret.
$\mathsf{DH}_{\mathsf{FIPS}}(sk^{\mathsf{S}}_{\mathsf{FIPS}}, pk^{\mathsf{R}}_{\mathsf{FIPS}}) \rightarrow SS_{\mathsf{FIPS}}$. Given public and secret FIPS keys, outputs a shared secret.

We use AES in GCM mode [7] for all symmetric encryption. This is defined as follows:

$\mathsf{AES\text{-}GCM\text{-}ENC}(K, IV, M) \rightarrow C$. On input a key, IV, and message, outputs a ciphertext.
$\mathsf{AES\text{-}GCM\text{-}DEC}(K, IV, C) \rightarrow M$. On input a key, IV, and ciphertext, outputs a message or a distinguished error.

For key derivation, we employ the HKDF algorithm [9].

Composite signature primitives. Our protocols employ a dual signature primitive $\mathsf{DualSign}$ that combines both ECDSA and EdDSA. This scheme consists of three algorithms defined as follows:

$\mathsf{DualSign.KeyGen}$

Input: None

Output: $vk_{\mathsf{DualSign}}, sk_{\mathsf{DualSign}}$

To generate a new keypair:

1. Run $(vk_{\mathsf{EdDSA}}, sk_{\mathsf{EdDSA}}) \leftarrow \mathsf{EdDSA.KeyGen}()$.
2. Run $(vk_{\mathsf{ECDSA}}, sk_{\mathsf{ECDSA}}) \leftarrow \mathsf{ECDSA.KeyGen}()$.
3. Return $(vk_{\mathsf{DualSign}}, sk_{\mathsf{DualSign}}) \leftarrow ((vk_{\mathsf{EdDSA}}, vk_{\mathsf{ECDSA}}), (sk_{\mathsf{EdDSA}}, sk_{\mathsf{ECDSA}}))$.

$\mathsf{DualSign.Sign}$

Input: a message M

Output: a signature $\mathsf{Sig}_{\mathsf{DualSign}}$

This algorithm outputs an "attached" signature that contains the message contents. To generate an attached signature:

1. Compute $\mathsf{Sig}_{\mathsf{EdDSA}} \leftarrow \mathsf{EdDSA.Sign}(sk_{\mathsf{EdDSA}}, M)$.
2. Compute $\mathsf{Sig}_{\mathsf{ECDSA}} \leftarrow \mathsf{ECDSA.Sign}(sk_{\mathsf{ECDSA}}, M)$.
3. Return $\mathsf{Sig}_{\mathsf{DualSign}} \leftarrow (M, \mathsf{Sig}_{\mathsf{EdDSA}}, \mathsf{Sig}_{\mathsf{ECDSA}})$.

$\mathsf{DualSign.Verify}$

Input: a verification key vk_{DualSign} and an attached signature $\mathsf{Sig}_{\mathsf{DualSign}}$

Output: a result $(\mathsf{true}, \mathsf{false})$

To verify a signature:

1. Parse vk_{DualSign} as $((vk_{\mathsf{EdDSA}}, vk_{\mathsf{ECDSA}}), (sk_{\mathsf{EdDSA}}, sk_{\mathsf{ECDSA}}))$.
2. Parse $\mathsf{Sig}_{\mathsf{DualSign}}$ as $(M, \mathsf{Sig}_{\mathsf{EdDSA}}, \mathsf{Sig}_{\mathsf{ECDSA}}))$.
3. Compute $r_1 \leftarrow \mathsf{EdDSA.Verify}(vk_{\mathsf{EdDSA}}, M, \mathsf{Sig}_{\mathsf{EdDSA}})$.
4. Compute $r_2 \leftarrow \mathsf{ECDSA.Verify}(vk_{\mathsf{ECDSA}}, M, \mathsf{Sig}_{\mathsf{ECDSA}})$.
5. Return true if $r_1 = \mathsf{true}$ and $r_2 = \mathsf{true}$. Otherwise return false.

Authenticated Public-Key Encryption. Our algorithms employ an IND-CCA-secure authenticated public-key encryption scheme Box, similar to NaCL [5]'s `crypto_box` construction. A key difference is our algorithm should be FIPS-compliant, using NIST P-384 for Diffie-Hellman, AES-GCM for a symmetric cipher, and HDKF for key derivation. We additionally include a Diffie-Hellman key share over Curve25519.

$\mathsf{Box.KeyGen}$

Input: None

Output: an encryption keypair $(pk_{\mathsf{Box}}, sk_{\mathsf{Box}})$

To generate a keypair:

1. Compute $(pk_{\mathsf{25519}}, sk_{\mathsf{25519}}) \leftarrow \mathsf{GenCurve25519Keypair}()$.
2. Compute $(pk_{\mathsf{FIPS}}, sk_{\mathsf{FIPS}}) \leftarrow \mathsf{GenFIPSKeypair}()$.
3. Return $(pk_{\mathsf{Box}}, sk_{\mathsf{Box}}) \leftarrow ((pk_{\mathsf{25519}}, pk_{\mathsf{FIPS}}), (sk_{\mathsf{25519}}, sk_{\mathsf{FIPS}}))$.

$\underline{\mathsf{Box.Enc}}$

Input: Sender's secret key $sk^{\mathsf{S}}_{\mathsf{Box}}$ and receiver's public key $pk^{\mathsf{R}}_{\mathsf{Box}}$, metadata Meta, and a message $M \in \{0,1\}^*$.

Output: a ciphertext C

To encrypt:

1. Parse $sk^{\mathsf{S}}_{\mathsf{Box}}$ as $(sk^{\mathsf{S}}_{\mathsf{25519}}, sk^{\mathsf{S}}_{\mathsf{FIPS}})$.
2. Parse $pk^{\mathsf{R}}_{\mathsf{Box}}$ as $(pk^{\mathsf{R}}_{\mathsf{25519}}, pk^{\mathsf{R}}_{\mathsf{FIPS}})$.
3. Compute $SS_{\mathsf{25519}} \leftarrow \mathsf{DH}_{\mathsf{25519}}(sk^{\mathsf{S}}_{\mathsf{25519}}, pk^{\mathsf{R}}_{\mathsf{25519}})$.
4. Compute $SS_{\mathsf{FIPS}} \leftarrow \mathsf{DH}_{\mathsf{FIPS}}(sk^{\mathsf{S}}_{\mathsf{FIPS}}, pk^{\mathsf{R}}_{\mathsf{FIPS}})$.
5. Generate a 256-bit random string $\mathsf{RandomNonce}$.
6. Compute $K \leftarrow \mathsf{HKDF}(SS_{\mathsf{25519}} \| SS_{\mathsf{FIPS}}, pk^{\mathsf{R}}_{\mathsf{Box}} \| \mathsf{Meta} \| \mathsf{RandomNonce})$.
7. Compute $C' \leftarrow \mathsf{AES\text{-}GCM\text{-}ENC}(K, \vec{0}, M)$.
8. Output $C \leftarrow (C', \mathsf{RandomNonce})$.

$\underline{\mathsf{Box.Dec}}$

Input: Receiver's secret key $sk^{\mathsf{R}}_{\mathsf{Box}}$ and sender's public key $pk^{\mathsf{S}}_{\mathsf{Box}}$, metadata Meta and a ciphertext C.

Output: a message M, or error

To decrypt:

1. Parse $pk^{\mathsf{S}}_{\mathsf{Box}}$ as $(pk^{\mathsf{S}}_{\mathsf{25519}}, pk^{\mathsf{S}}_{\mathsf{FIPS}})$.
2. Parse $sk^{\mathsf{R}}_{\mathsf{Box}}$ as $(sk^{\mathsf{R}}_{\mathsf{25519}}, sk^{\mathsf{R}}_{\mathsf{FIPS}})$. Recompute $pk^{\mathsf{R}}_{\mathsf{Box}}$ from this value.
3. Parse C as $(C', \mathsf{RandomNonce})$.
4. Compute $SS_{\mathsf{25519}} \leftarrow \mathsf{DH}_{\mathsf{25519}}(sk^{\mathsf{R}}_{\mathsf{25519}}, pk^{\mathsf{S}}_{\mathsf{25519}})$.
5. Compute $SS_{\mathsf{FIPS}} \leftarrow \mathsf{DH}_{\mathsf{FIPS}}(sk^{\mathsf{R}}_{\mathsf{FIPS}}, pk^{\mathsf{S}}_{\mathsf{FIPS}})$.
6. Compute $K \leftarrow \mathsf{HKDF}(SS_{\mathsf{25519}} \| SS_{\mathsf{FIPS}}, pk^{\mathsf{R}}_{\mathsf{Box}} \| \mathsf{Meta} \| \mathsf{RandomNonce})$.
7. Compute $M \leftarrow \mathsf{AES\text{-}GCM\text{-}DEC}(K, \vec{0}, C')$. If decryption fails, output error. Otherwise output M.

3.6 Long-term Key Management

When users i signs up, or upgrades their Zoom applications to the first version that supports E2E as described in this proposal, their clients will generate a long-term signing keypair:

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

The Zoom client posts the mapping $\langle (i, \mathsf{deviceID}) \rightarrow IVK_i \rangle$ to the server, signed with $\mathsf{DualSign.Sign}$ 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.

3.7 Join/Leave Protocol flow

We assume each meeting is identified by its unique $\mathsf{meetingID}$, 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. We make the case below that a malicious server that sends stale messages from a previous meeting incarnation can at best deny service, which it can do regardless.

3.7.1 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 $\mathsf{meetingUUID}$ 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. Computes $\mathsf{Sig}_i \leftarrow \mathsf{DualSign.Sign}(ISK_i, \texttt{"Zoom00EPubKeys\textbackslash 0"} \| \mathsf{binding}_i)$.
5. Stores sk_i for the duration of the meeting.
6. Posts Sig_i to the bulletin board, so that all participants can see it.

3.7.2 Leader Join

When the leader joins the meeting $\mathsf{meetingID}$, it:

1. Fetches $\mathsf{meetingUUID}$ from the Zoom infrastructure.
2. Generates a symmetric 32-byte seed mk using a secure random number generator.
3. Computes the meeting key as $\mathsf{MK} \leftarrow \mathsf{HKDF}(mk, \texttt{"Zoom00SKey\textbackslash 0"} \| \mathsf{meetingID} \| \mathsf{meetingUUID})$.
4. Gets the full list of participants I from the MMR.
5. For each participant $i \in I$, it runs the "Participant Join (Leader)" subroutine for i.

3.7.3 Participant Join (Leader)

Given a leader ℓ and a participant i joining meeting $\mathsf{meetingID}$ on $\mathsf{deviceID}$, the leader:

1. Fetches IVK_i from the key server.
2. Fetches Sig_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. Verifies the signature Sig_i as $\mathsf{DualSign.Verify}(IVK_i, \texttt{"Zoom00EPubKeys\textbackslash 0"} \| \mathsf{binding}_i)$.
5. If verification fails, it aborts.
6. Computes $\mathsf{Meta} \leftarrow (\texttt{"Zoom00SDKey\textbackslash 0"} \| \mathsf{meetingID} \| \mathsf{meetingUUID} \| \ell \| i)$.
7. Computes $C \leftarrow \mathsf{Box.Enc}(sk_\ell, pk_i, \mathsf{Meta}, mk)$.
8. Posts (i, C) to the "bulletin board".

3.8 Participant Join (Non-Leader)

When participant i joins meeting $\mathsf{meetingID}$, it performs the reverse of the above procedure:

1. Fetches IVK_ℓ from the Key server for the leader ℓ.
2. Fetches Sig_ℓ from the meeting's "bulletin board".
3. Fetches (i, C_i) from the "bulletin board".
4. Fetches the $\mathsf{meetingUUID}$ from the server.
5. Verifies Sig_ℓ as $\mathsf{DualSign.Verify}(IVK_\ell, \texttt{"Zoom00EPubKeys\textbackslash 0"} \| \mathsf{binding}_i)$.
6. Computes $\mathsf{Meta} \leftarrow (\texttt{"Zoom00SDKey\textbackslash 0"} \| \mathsf{meetingID} \| \mathsf{meetingUUID} \| \ell \| i)$.
7. Decrypts $mk \leftarrow \mathsf{Box.Dec}(sk_i, pk_\ell, \mathsf{Meta}, C)$.
8. If decryption outputs error, it aborts.
9. Computes the meeting key as $\mathsf{MK} \leftarrow \mathsf{HKDF}(mk, \texttt{"Zoom00SKey\textbackslash 0"} \| \mathsf{meetingID} \| \mathsf{meetingUUID})$.

Now all participants have access to the shared meeting key MK, and can encrypt and decrypt meeting streams accordingly.

3.8.1 Key Rotation

At any point later in the meeting, the leader can generate a new 32-byte value mk'. The leader performs steps 7-8 of "Participant Join (Leader)" for all participants, with the updated mk' value. All participants see the rekey signal on their signaling channel, and perform steps 7-9 of "Participant Join (Non-Loader)". Participants should not immediately encrypt using the new meeting key; they should wait about 15 seconds, to ensure all participants smoothly transition over.

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. 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'.

Each mk has an associated sequence number $\mathsf{mkSeqNum}$, starting at 0 and incrementing whenever it rolls over. The leader sends this sequence number along with the participant list heartbeat below.

3.8.2 Authenticated Participant Lists

A meeting leader maintains a "leader participant list" (LPL), tabulating all users they have added to and removed from the meeting. Every time this list changes, the leader increments the counter v and posts the mapping $\langle v \rightarrow LPL_v \rangle$ to the bulletin board. They broadcast a signature over this list whenever membership changes, and at designated "heartbeat intervals":

$$\mathsf{DualSign}(ISK_\ell, (\texttt{"Zoom00LPL\textbackslash 0"} \| \mathsf{binding}_\ell \| \mathsf{SHA256}(LPL_v) \| v \| t \| \mathsf{mkSeqNum}))$$

Where t increments on every send, v increments whenever the LPL changes, and $\mathsf{mkSeqNum}$ increments on every mk rotation.

The other participants should know this list, 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 control channel. Heartbeats should go out at least every 10 seconds. All participants observe and verify these heartbeats, and if they fail to receive four heartbeats in a row, they should drop out of the meeting.

3.8.3 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.

3.8.4 Meeting Replay Attack

What happens if Alice, Bob and Charlie meet at meeting m two days in a row, and a malicious Zoom server mixes bulletin board messages from the two meetings? In particular, what if Alice is the leader and the server presents Bob's previous ephemeral key? Alice doesn't realize it's Bob's previous key, so she'll obediently encrypt for it in step 4 of "Participant Join (Leader)" using her new ephemeral key. This will result in a message that Bob can't decrypt, since he'll be using his new ephemeral private key, having throw his old ephemeral private key away at the end of the last meeting. The malicious server can't roll back the whole conversation, since Alice, Bob and Charlie know to use their new ephemeral keys; they couldn't even use their old ones if they tried.

3.9 Meeting 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 security code" in the security tab:

$$\mathsf{Wordify}(\mathsf{Hash128}(\texttt{"Zoom00MSecCode\textbackslash 0"}||IVK_\ell))$$

Hash128 is a standard hash function like SHA-256, but truncated to 128 bits. The Wordify function converts a binary hash to a human-readable and internationalization-friendly code. Crucially, every Zoom client in the meeting independently computes these values on 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 security code, and 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.

If deep fake technology 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.

We considered other approaches to the meeting 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."

3.10 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 bottom 8 bits of mkSeqNum, so participants know which version of MK to decrypt with. Wrapping is not a security concern, since users can trust the participant list heartbeats for the full mkSeqNum.

3.11 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,

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

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

3.12 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 "security code" proposal above is a countermeasure 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, users joining the meeting late must request the ceremony be reperformed. 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 start introducing the concept of identity and use it to guarantee that only authorized participants will be able to join a meeting. Meeting leaders will have more helpful information when evaluating join requests and deciding to kick questionable users out of the meeting. We build up the necessary machinery to securely expose identity information to clients.

4.1 Building an Identity

First we cover building up an identity, and then we discuss how it's used in the UI.

4.1.1 Provisioning

When a user first logs in on a new device, or first upgrades a device to new Zoom software, their client generates a statement binding the user's identity to the device's public key. For users who sign into Zoom directly or via OAuth, these data consist of email and organization (if not a personal account). For those who authenticate via SSO identity providers, this data will also include identity information the IDP provides. Several signatures accompany this binding:

1. **Always:** A signature by Zoom's servers.
2. **If Applicable:** a signature by the user's SSO IDP, made in a way that Zoom's server doesn't proxy the signature request. The data provided in the identity provider's signature must match the data in the identity statement, and verifying clients must check correspondence.
3. **Always:** A self-signature with the user's current key.
4. **Starting in Phase IV:** A signature by one of the user's previously authenticated devices.

The signatures should happen in this order, and later signatures should be computed over the statement body and previous signatures.

The client uploads the statement and accompanying signatures to Zoom's servers, which can relay these statements later when users join meetings.

4.1.2 Revoking Devices

When a user uninstalls Zoom on a device, or erases, throws away, or loses a device, they should sign a statement saying that the device is revoked. This way, if anyone shows up later using this device, other meeting participants can cast a suspicious eye.

The revoke statement simply says that a public key corresponding to a previously provisioned Zoom device is no longer valid. The same signatures as above apply, but any currently valid user device can issue "self-signature", not just the device being revoked.

When a user revokes a device, they do not invalidate all signatures made by the device prior to the revocation. Rather, they prevent the device from making signatures going forward. This simple plan does not cover the case in which Alice lost her device 2 weeks ago, realizes today, and wants to revoke all the signatures made in the interim. To cover that case, we might eventually include a mechanism to revoke individual signatures.

4.1.3 Signature Chains

A user makes many signatures pertaining to their identity; one for each device provisioned, and one for each device revoked. As such, these signatures comprise a natural sequence of events, and clients should have protections against malicious servers reordering them or omitting them (especially in the case of revocations).

Hence, the signed statements for a given user each contains a monotonically increasing sequence number. In addition, each signed statement will contain a hash of the statement with the previous sequence number. These hashes bind the statements related to the same identity to each other, forming a "hash chain" of signed statements which we refer to as a *sigchain.*

The server coordinates previous hashes and sequence numbering at this phase in the proposal. Users on SSOs have reasonable guarantees that malicious servers aren't corrupting this process, since they have independent signatures on their chain links. Users without SSOs don't get that additional protection, but we will add better sequencing guarantees in Phase III.

4.1.4 Hiding Personal Details

An important point to cover here is that the statements themselves must be structured in a way that they can be selectively deleted in the future without preventing the whole statement from passing a signature verification. It's best to build up an intuition here by example. Consider a statement of the form:

```
{
  "userID": "ebc0d2ecd5b761c2d873c34e15c1dc8c6eb6904432557c0605e465b7bf5265",
  "deviceName": "Jane's iPhone 11 Pro",
  "publicKey": "f27768d032c8578ff759309a09b1ffa1f623b2bc58f",
  "sequenceNumber": 10,
  "prev": "484ad7913d81ce856455d6de94e79896ad4731713ebb0580f051cc2cce1d14ae"
}
```

Imagine Jane later regrets the fact she named her device the way she did. She can revoke the device, but the various signatures computed over this statement should still be verifiable for auditing Jane's account. A simple cryptographic commitment can satisfy all of these requirements. Jane's statement instead should be of the form:

```
{
  "userIDg: "ebc0d2ecd5b761c2d873c34e15c1dc8c6eb6904432557c0605e465b7bf5265",
  "deviceName": HMAC(r, "Jane's iPhone 11 Pro"),
  "publicKey": "f27768d032c8578ff759309a09b1ffa1f623b2bc58f",
  "sequenceNumber": 10,
  "prev": "484ad7913d81ce856455d6de94e79896ad4731713ebb0580f051cc2cce1d14ae"
}
```

Where r is a random 32-byte value, and the value `"Jane's iPhone 11 Pro"` is stored on Zoom's servers (in plaintext). When Jane wants this information to be available, the server makes r and `"Jane's iPhone 11 Pro"` known. When she revokes the statement, the server throws away r and the associated plaintext. Hence, the statement is still verifiable, and the `publicKey` is still knowable, even after Jane revokes the statement and hides the personal details. Furthermore, these commitments allow Zoom to make these details selectively available. She can see her own device names on her settings screens, while other users examining her identity cannot. But all can verify her signature over the committed data.

In the rest of this document, we hide this design detail, but it's an important one for designing a secure, privacy-preserving system with immutable building blocks like signatures.

4.1.5 Signed Contact List Updates

Whenever Alice completes a meeting with Bob and Charlie, she should make a note of their public keys and the states of their identity, so she can flag if they later show up on new (potentially faked) devices. She should also make a note of whether they were kicked out of the meeting.

After leaving a meeting or a meeting ends, a user constructs a list of mappings for every user in the group:

$$\langle \mathsf{userID} \ \rightarrow \ (\mathsf{identitySigchainTail}, \mathsf{meetingStatus})\rangle$$

Where $\mathsf{identitySigchainTail}$ is the tail of the user's sigchain that describes their identity and devices, and the $\mathsf{meetingStatus}$ contains information about whether or not the user was evicted from the meeting, as well as how long the users were in the same meeting together, and how many other users were also participating. There might not be much to glean from a very large meeting, or a meeting in which two users only overlapped for a short time. The user signs this statement with their current device.

As above, these meeting status updates constitute a sigchain, so users can securely port this data across devices. This particular sigchain will grow large since it updates after every meeting. It should employ compression and checkpointing, so that users don't have to download and replay lots of data on their devices.

4.2 User Interface

This phase of the proposal includes some new UI features. First, a user's client will receive notifications about any new devices added to their account.

Device list management is now backed by the user's sigchain. The Zoom client will highlight the device additions and removals that have been authorized with user signatures (versus any that predate the system). In this view, users will also be able to trigger a signed device revocation. A device that knows it's being revoked should delete all private ephemeral and long-term keys, then log the user out.

Other changes are visible wherever users are displayed, primarily in the meeting screen, in the participants list, and in the waiting room. For users who use an identity provider to vouch for their identity (such as SSOs), the UI can show the identity that their IDP has vouched for. And recall, these identities are signed by the IDP, and therefore Zoom cannot tamper with them.

For any two users Alice and Bob, Alice's client can put a note on Bob's in-meeting profile if Alice doesn't recognize Bob's keys (recall from above that Alice is now keeping a record of Bob's public keys). From Alice's perspective, if any user in her meeting is joining using a public key that she doesn't recognize, the whole meeting gets a label to that effect, and a security interface displays which users are joining the meeting using new keys. Those new keys may be a result of Bob using a new device, of having reinstalled the application on a previous device, or of an attacker proxying and snooping on Bob's connection. So this indicator is not a security alarm but rather an indication that Alice and Bob may wish to check each other's meeting security code, as described in Phase I.

4.3 Areas to Improve in Phase II

One area to improve when moving beyond Phase II might be alert fatigue, since we don't have a good way to distinguish between Bob legitimately adding a new device and a malicious server inserting one on Bob's behalf, though this is significantly less noisy than in Phase I. Furthermore, a malicious server could potentially return an outdated or inconsistent view user sigchains. There is some protection against arbitrary server behavior in that users cross-sign each other's sigchains when they update their contact lists, but we can do better.

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 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 [10] 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 [6] and CONIKS [11] 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.

5.2.6 Multiple Trees

In this setting, we anticipate the need for multiple ZTTs, one for each organization, and a global tree computed over all the organization-specific trees. This tiered architecture has several advantages: it allows organizations who want independence from each other to keep their data disentangled; it enables sensitivity to local data protection laws, as different trees can have different parameters; it improves scalability; it better mirrors Zoom's current server-side infrastructure; it anticipates a future federated architecture; and it simplifies audits, since auditors can focus on the global tree and their organization's tree.

We anticipate a downside from this approach from our experience at Keybase: sometimes it is necessary to coordinate updates to different parts of the tree to ensure strict "happens before" relationships across leaves. For instance, Keybase guarantees that if a user signs on behalf of a team, and then revokes that key, the signature happens provably before the revocation.

The multiple tree architecture might make such constraints impossible if across different independent trees, but we anticipate that slight relaxations of certain security requirements will smooth the way.

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, singing 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 look forward to feedback from all stakeholders about all aspects of this proposal through the public review process, and will publish subsequent drafts as we refine, implement and deploy these ideas.

References

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

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

[3] 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.

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

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

[6] 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.

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

[8] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017.

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

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

[11] 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.

[12] National Institute of Standards and Technology. FIPS Pub 186-4 Digital Signature Standard (DSS). Available at https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf, 2013.

[13] National Institute of Standards and Technology. NIST Special Publication 800-56A Recommendations for Pairwise Key Establishment, Revision 3. Available at https://csrc.nist.gov/CSRC/media/Publications/sp/800-56a/rev-3/draft/documents/sp800-56ar3-draft.pdf, 2018.