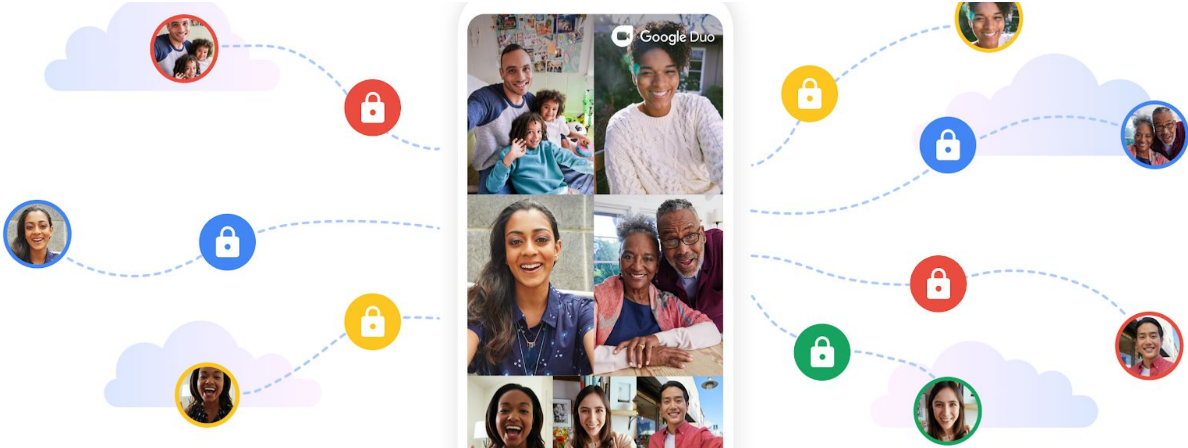
Google Duo End-to-End Encryption Overview

Technical Paper

Emad Omara
Communications Security Lead

June 2020

Version 1.0



A high-level technical overview of end-to-end encryption in Duo

INTRODUCTION **2**
User Registration **3**
Messaging **3**
Session Setup 3
Messages Encryption 4
Media Messages Encryption 5
Groups and Multi-Device 6
Receipts 6
State Recovery 7
Calling **7**
One-to-One 7
Group 8
Key management 8
Call Setup 8
Key Rotation 9
Media Encryption 9
Conclusion **11**

INTRODUCTION

Ensuring secure communication is one of Duo's core principles. We specifically designed Duo calls and media messages to support end-to-end encryption (E2EE), which means that only the media sender and their desired recipients can access the audio and video being exchanged. In this paper we give a high level technical overview of how E2EE works in Duo for mobile, tablet, web and Nest Smart Display clients, starting with account setup and 1:1 messaging, and expanding to cover calling in 1:1 and group scenarios.

Duo uses the Signal[1] protocol as the core E2EE protocol for messaging and group calling setup. In the following two sections, we explain the basics of the Signal protocol and how it is being used by Duo.

User Registration

When a user signs up for Duo on a new device (mobile, tablet, web, or a Nest Smart Display client), the Duo client generates the following key pairs, using the Curve25519 algorithm:

- Identity Key: A long-term key pair associated with the user on this device
- Signed Prekey: A single key pair, with the public key signed by the Identity Key
- Unsigned Prekeys: A set of unsigned key pairs

The keys are generated using the BoringSSL RAND_bytes secure random function. The public parts of these keys are uploaded to a Google key server, while the private parts never leave the device.

When establishing a secure session, the Duo client will request the identity key, signed prekey, and one unsigned prekey associated with the desired remote participant from the Google key server. Duo then uses the retrieved keys to derive the initial secrets for the secure session, in accordance with the Signal protocol.

Once an unsigned prekey is handed out, it is deleted from the key server. When the key server detects that a client is running low on prekeys (because almost all of them have been handed out), it sends a message to the Duo client to generate and upload a new set of one-time prekeys, and a new signed prekey.

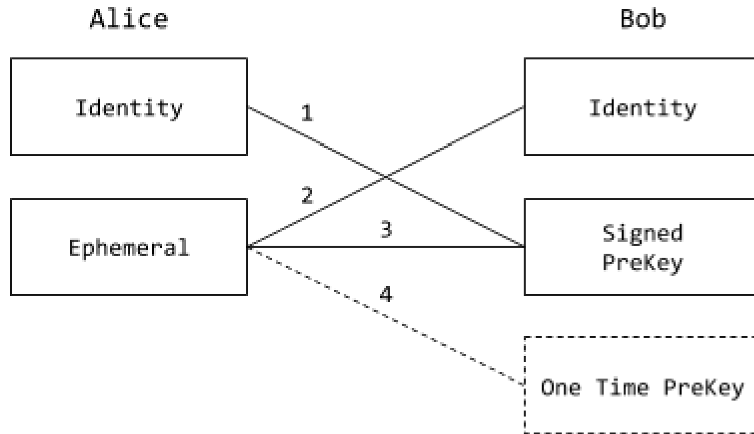
Messaging

In addition to video and audio calling, Duo supports sending and receiving media messages (photo, video and audio messages) and reactions. These messages are end-to-end encrypted via a framework built on top of the Signal protocol described above.

In this next section, we explain how this works, starting with basic one-to-one message delivery and then expanding to cover media and group messages.

Session Setup

Signal sessions are established as long-lived sessions between pairs of devices. When establishing a secure session, the device that initiates the session generates an ephemeral Curve25519 key pair, requests the other device's identity key, signed prekey, and one one-time prekey (if available) from the key server, and then performs the X3DH[2] algorithm using ECDH-X25519 to derive the session root key.



Alice sets up a session with Bob using their key pairs and the X3DH algorithm

```
A1 = ECDH-X25519(Sender_Identity, Recipient_SignedPreKey)
A2 = ECDH-X25519(Sender_Ephemeral, Recipient_IdentityKey)
A3 = ECDH-X25519(Sender_Ephemeral, Recipient_SignedPreKey)
A4 = ECDH-X25519(Sender_Ephemeral, Recipient_OneTimePreKey)

RootKey = HKDF-SHA256(A1||A2||A3||A4, "", "WhisperText", 64)
```

Before sending the first message in the session, the sender generates another Curve25519 keypair, the ratchet key. The sender uses this key to ratchet forward the root key, creating a new root key and a chain key:

```
A = ECDH-X25519(Sender_RatchetKey, Receiver_SignedPreKey)

RootKey[32], ChainKey[32]=
HKDF-SHA256(A, PreviousRootKey, "WhisperRatchet", 64)
```

The sender's first message will include their identity, ephemeral, and ratchet public keys. The recipient, who must have the private key material for their identity key, signed prekey, and one-time prekey, will be able to compute all the same agreements as the sender. The recipient also deletes the private key of the one-time prekey from their local storage.

The state of the secure session and root key are persisted on the local device, so future communications between these two devices don't require the establishment of a new session.

Message Encryption Basics

For one-to-one messages, we use the unmodified Signal protocol, where each message is encrypted using a different message key. This key is derived from the chain key:

```
Seed = HMAC-SHA256(ChainKey, 0x01)

MessageKey[32], MACKey[32], IV[16] =
HKDF-SHA256(Seed, "", "WhisperMessageKeys", 80)
```

Signal uses AES-256-CBC with PKCS#7 padding for message encryption. The encrypted message is stored in a protocol buffer along with other session states. An 8-byte MAC is computed over the serialized protocol buffer to create the final message:

```
Ciphertext ← AES-CBC(MessageKey, IV, Message)

MAC[8] =
HMAC-SHA256(MACKey,SenderIdPublicKey || ReceiverIdPublicKey || Ciphertext)

MessageToSend = Ciphertext || MAC
```

For stronger security properties (Forward Secrecy & Post-Compromise Security), Signal uses the Double-Ratcheting[3] algorithm, in which each time a client sends a message, they ratchet forward their chain key and delete the previous one:

```
NewChainKey = HMAC-SHA256(PreviousChainKey, 0x02)
```

Also, each message will include a Curve25519 key pair, called the "ratchet key." Whenever Alice sees a new ratchet key from Bob, Alice will derive a new root+chain key and create a new ratchet key for herself that she'll send with subsequent messages (and vice-versa). Senders are always one key ahead of the newest ratchet key they've seen used.

Media Message Encryption

For efficiency reasons, Duo doesn't send its audio and video messages directly through a Signal secure session. Instead, the sender client generates an ephemeral, one-time 32-byte key material KM for each media message. From KM it derives the following:

```
Key = HKDF-SHA256(KM, "", "MediaEncryptionKey", 32)

IV = HKDF-SHA256(KM, "", "MediaEncryptionIv" ,16)
```

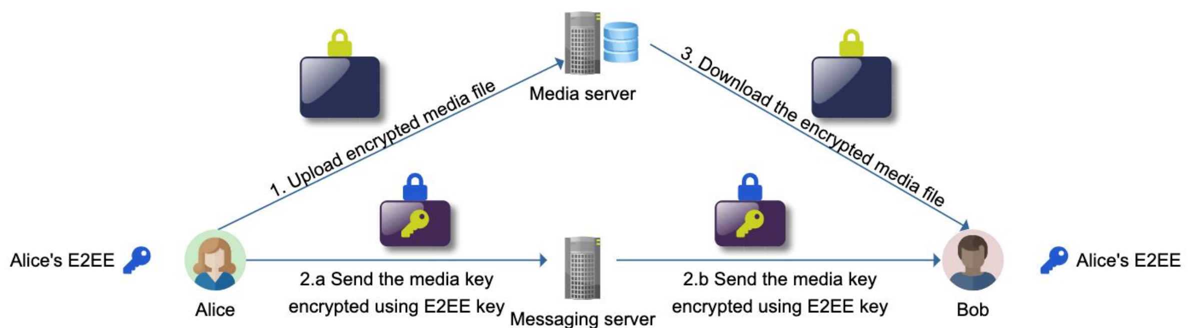
The media content M is then encrypted with Key and IV using AES-CTR 256 and produces ciphertext C. The Duo client also creates a 32-byte digest of the message using SHA256:

```
C = AES-CTR(Key, IV, M)

Digest = SHA256(C, 32)
```

The client then uploads the encrypted media blob C to a Google media storage server and retrieves a URL for it. This URL will be encrypted along with the key material KM and message digest, using the Signal Protocol as described above. When the receiver gets the message, it does the following:

1. Decrypts the message to retrieve the URL, KM, and message digest
2. Downloads the encrypted blob C from the Google media storage server
3. Verifies the digest, dropping the message if verification fails
4. Derives Key and IV from KM
5. Decrypts C to get the original content M



Alice sending encrypted video message to Bob

Groups and Multi-Device

If a message has a single recipient with a single device (aka one-to-one), the message is encrypted using Signal as described above. However, if the recipient has more than one device or it is a group message, we encrypt it differently using a per-message One-Time-Key (OTK), which we built on top of Signal for multi-device and group conversations. This increases efficiency and allows us to maintain the same security properties of Signal one-to-one messages.

When sending using a OTK, the sender client generates 32 bytes of one-time key material, KM. From KM it derives the following:

```
OTKEncKey = HKDF-SHA256(KM, "", "OTKEncryptionKey", 32)

OTKEncIV = HKDF-SHA256(KM, "", "OTKEncryptionIV", 12)
```

The message is encrypted a single time using the OTK and the ChaCha20-Poly1305 AEAD scheme, and then the encrypted message and KM are encrypted N times using Signal one-to-one sessions for each of the N recipients. The Duo client sends the resultant encrypted keys and message to the Duo service, which distributes it to each recipient. Upon receipt, the recipient decrypts KM via the Signal session and uses it to derive OTK and recover the original message.

This One-Time-Key approach achieves “Forward Secrecy” by relaying on the Signal session to encrypt the key and “Post-Compromise Security” by using a different key for each message.

Receipts

Duo clients send two types of receipts:

- Successful delivery receipts
  These receipts are E2EE using Signal secure sessions as described above. Encrypting these receipts not only helps with protecting user privacy, but it also triggers the Double-Ratcheting[2] algorithm immediately, ensuring the next message sent after a received receipt uses a fresh key.
- Decryption error receipts
  These receipts indicate a received message cannot be decrypted, and usually occur when the

recipient loses the session state due to storage corruption or a reinstall of the app. Because there are no longer any shared encryption keys, these receipts are not E2EE.

State Recovery

When a client receives a message they can't decrypt, it sends a decryption error receipt back to the sender. Upon receiving this receipt, the sender performs the following steps to securely retransmit the message:

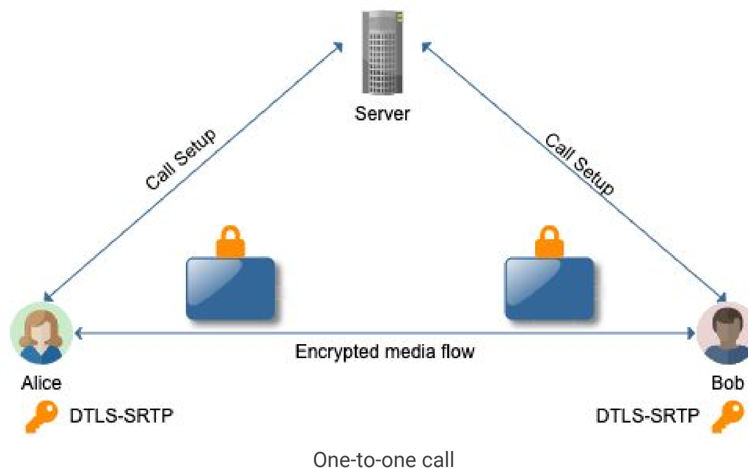
1. Verifies it actually sent a message to this recipient.
   The sending client stores the hashes of all outgoing messages with their message id and list of recipients, and clears them when they receive successful receipts from each recipient.
2. If the verification fails, aborts.
3. Deletes the existing Signal session state.
4. Re-establishes the session by requesting a new prekey for the recipient
5. Encrypts the message using the new session and sends it again.

Calling

One-to-One

Duo uses WebRTC[4] for its media stack, which natively supports end-to-end encryption for one-to-one calls using DTLS-SRTP[5]. DTLS (Datagram Transport Layer Security) works similarly to TLS, but over UDP, to establish a secure connection between the two parties in the call. This secure connection is used to derive a master encryption key, which is supplied to SRTP (Secure Real-time Transport Protocol) to derive the actual media encryption and authentication keys for each party.

When using DTLS-SRTP, all derived keys and the master key are short-lived, in-memory keys, used only during the call and not persisted in storage. Google servers don't have access to them.



One-to-one call

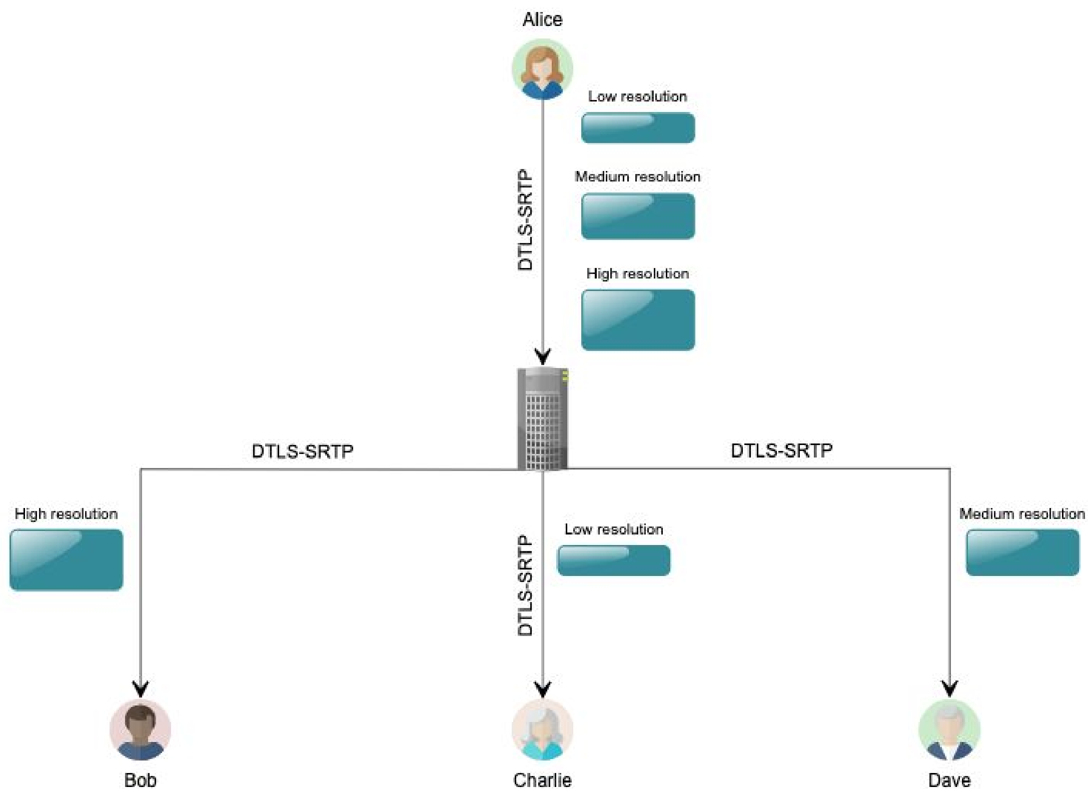
One-to-one calls are point-to-point, meaning there is no central server to route media. While a relay server may be used when the media cannot be sent point-to-point (for example, when a network

firewall doesn't allow direct IP access), the media remains end-to-end encrypted, as the server simply relays the encrypted packets at the IP level.

Group

Group calling is much more complex than one-to-one calls, because different participants on the call could have different Internet connection speeds, and we want to ensure they all can receive the right video quality for their connection. This is done by having the Duo client send multiple encrypted media streams, each with different quality levels, to a central media server that decides which stream should go to each participant.

Although DTLS-SRTP is a well-known standard for efficiently securing point-to-point connections, there is no similar standard securing media end-to-end in group calling.



Alice's media flow in a non-E2EE group call, shown for explanatory purposes only.
The media is encrypted with DTLS-SRTP, but only between client and server.

Key management

The media in Duo group calls is E2EE using a per-client ephemeral one-time key. These keys are exchanged with every other group member, even if they have not joined the call yet.

Call Setup

When preparing to start a group call, the initiator creates 32 bytes of ephemeral one-time key material, SK, and encrypts it to all other group members using Signal sessions and the OTK message format described above. When other clients receive this message from the initiator, they also create their own SK and send it to all other group members. This happens before the user actually starts the call, to minimize latency.

These key materials are short-lived, in-memory keys, used only during the call and not persisted in storage. Google servers don't have access to them.

Key Rotation

Clients rotate their SK when group membership changes during the call. For a “Member join” operation, a new key is derived from the previous key (ratcheting) and the new key is sent to the new user.

```
SK = HKDF-SHA256(PreviousSK, "", "SFrameRatchetKey", 32)
```

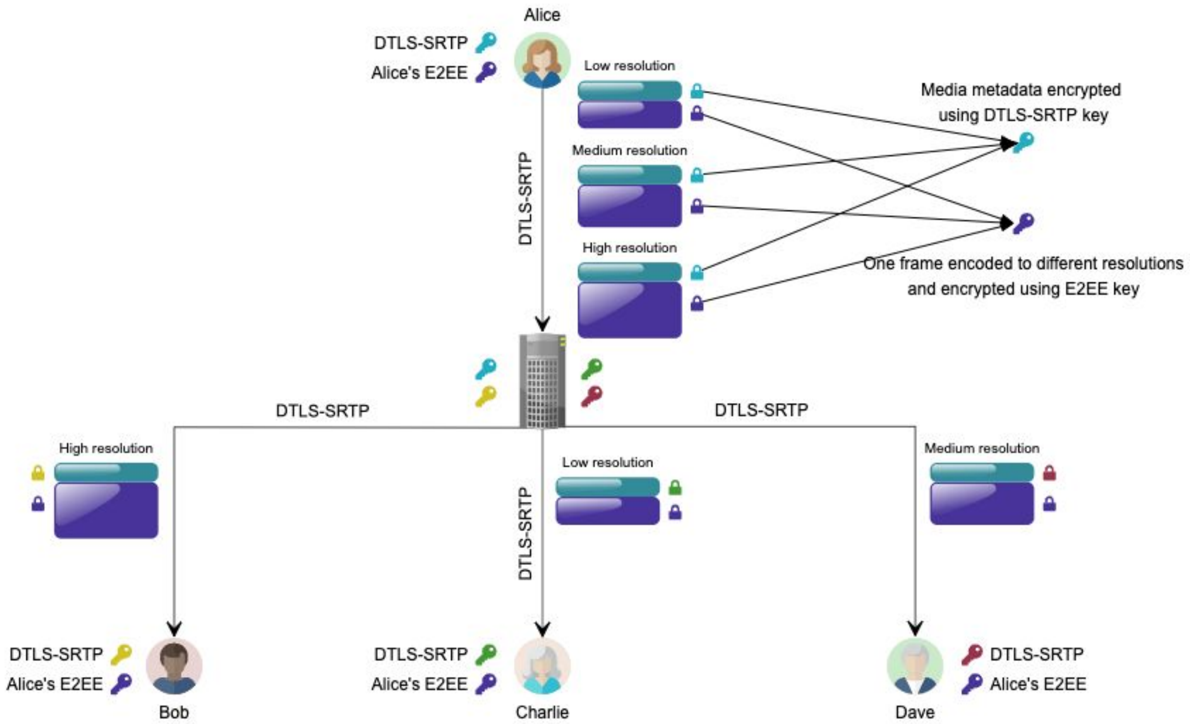
A “Member leave” operation requires all participants to create a new SK and exchange it with all other group members (similar to the call setup process). The new keys are not used immediately for encryption. Instead, they are activated when one of the following condition occurs:

1. The sender client receives successful receipts from all other group members, or
2. After 5 seconds from the key generation.

Rotating keys in this manner ensures that newly added participants cannot access call media from before they were added, and removed participants cannot access call media that was sent after they were removed.

Media Encryption

In order to efficiently protect call media end-to-end, we developed a new mechanism called SFrame[6]. With SFrame, the frames emitted by the audio or video encoder are immediately encrypted using the E2EE keys described above. The encrypted frames are then split into packets and sent to the server over a secure DTLS-SRTP channel. Because the audio and video media content is end-to-end encrypted, the server is only able to see the media metadata (e.g., video resolution, frame rate, etc.) needed by the server to pick the right media quality for each recipient, and cannot access the actual audio and video.



Alice’s media flow to the group: The server has access to the media metadata, but can’t access the actual media because it does not have the E2EE key

SFrame takes the sender key material (SK) as input and derives multiple keys:

```
SFrameEncKey = HKDF-SHA256(SK, "", "SFrameEncryptionKey", 16)

SFrameAuthKey = HKDF-SHA256(SK, "", "SFrameAuthenticationKey" ,32)

SFrameSaltKey = HKDF-SHA256(SK, "", "SFrameSaltKey" ,16)

IV = FrameCounter XOR SFrameSaltKey
```

It then uses this key to encrypt and authenticate the media frame using AES-CTR 128 and HMAC:

```
Ciphertext = AES-CTR(SFrameEncKey, IV, frame)

MAC = HMAC-SHA256(SFrameAuthKey,Ciphertext || FrameMetadata)
```

After the frame is encrypted, a SFramePayload is constructed using the encrypted frame, MAC and FrameMetadata:

```
SFramePayload = Ciphertext || MAC || FrameMetadata
```

SFramePayload is passed back into the WebRTC media stack to continue its normal flow, where it is split into packets that are then encrypted again using DTLS-SRTP before being transmitted to the media server.

Upon receiving all the packets for a frame, the media server will reconstruct the SFramePayload, but it can only access the frame metadata portion, and not the content of the encrypted media frame. After it processes the frame, it again splits it into packets, and re-encrypts each packet to the intended recipient client using DTLS-SRTP.

Finally, the recipient client receives all the packets for the frame and reconstructs SFramePayload, derives the same keys from the sender key material, and validates the MAC, dropping the frame if the verification process fails. It then decrypts the frame using the derived keys to retrieve the original media frame, which is supplied back to the WebRTC media stack for decoding and processing.

Conclusion

Duo has been designed from the start to have a strong focus on security without compromising the quality of the experience or simplicity of the application. It utilizes end-to-end-encryption throughout to protect user communications, and as a result Google servers do not have access to encryption keys and can't decrypt user messages and calls.

With the increasing importance of video calling for personal communication, end-to-end encryption helps ensure your communications stay private and secure.

REFERENCES

1. Signal Protocol [here]
2. X3DH [here]
3. Double Ratcheting [here]
4. WebRTC Security Overview [here]
5. DTLS-SRTP [here]
6. SFrame [here]