DRAFT UNDER EMBARGO UNTIL 11/19/20 at 7AM PT

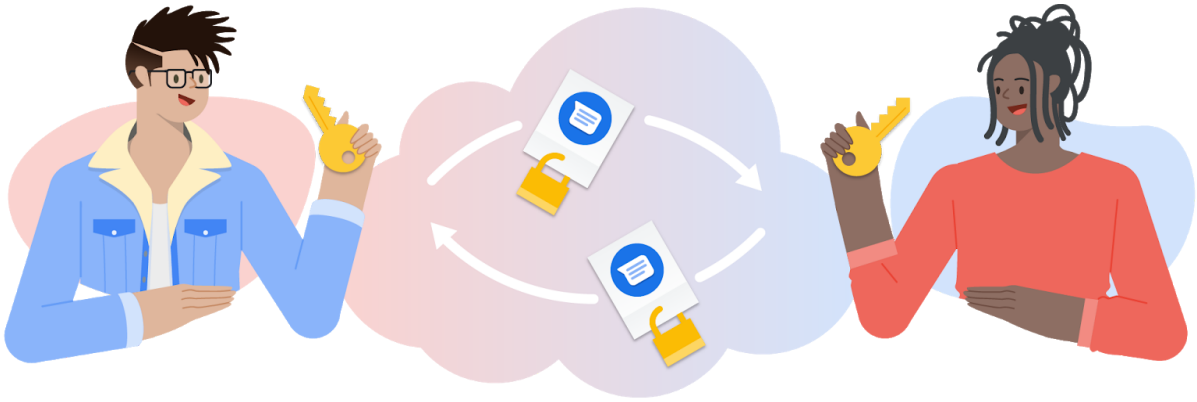
Messages End-to-End Encryption Overview

Technical Paper

Emad Omara
Communications Security Lead

November 2020

Version 1.0



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

Introduction

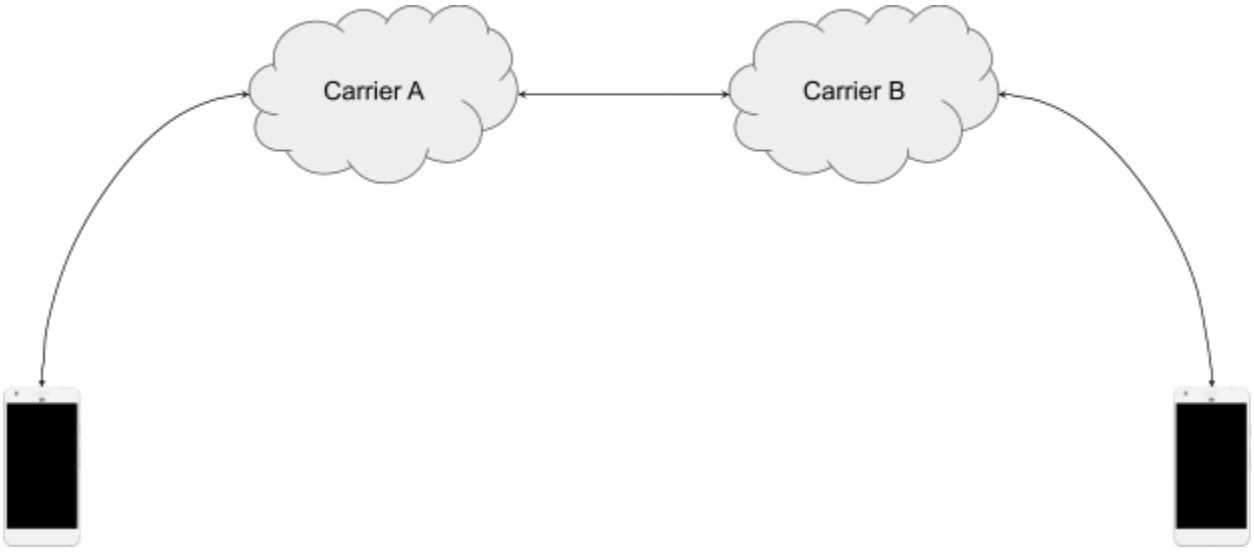
Rich Communication Services (RCS) is designed to improve users' experience and security over Short Message Service (SMS)/Multimedia Messaging Service (MMS), and we've invested in making Messages by Google a modern and globally available RCS & SMS/MMS messaging app for Android phones. While RCS messages are already a big improvement over SMS/MMS in terms of security, we

wanted to take it a step further and add end-to-end encryption (E2EE) to Messages clients, so no one else – including Google servers or a third-party server – can access your conversations as they travel between your phone and the phone you message.

Background & RCS Ecosystem

RCS uses a set of standard internet protocols like Session Initiation Protocol (SIP)[1] to establish a connection between two clients through a central messaging server, then uses this connection to exchange the messages using Message Session Relay Protocol (MSRP)[2]. In some RCS deployments this server is hosted by the carrier and in other deployments the server is hosted by Jibe Mobile from Google.

In situations where the two clients are not on the same carrier network, they're connected through multiple servers, one from each carrier. The connection between the RCS client app and the carrier's messaging server is encrypted using Transport Layer Security (TLS). Server to server communications are also encrypted.



Threat Model

With end-to-end encryption, users' message content can only be accessed by the clients in the conversation. The message servers, either hosted by Google or by the carrier, should continue to route the messages, but they must not be able to access the contents. This helps to protect against:

- **A malicious or compromised server**
  If an attacker compromises the messaging server, they shouldn't be able to intercept, modify or replay the message contents. The best they can do is to access the messages metadata or perform a denial of service (DoS) attack by dropping the messages.

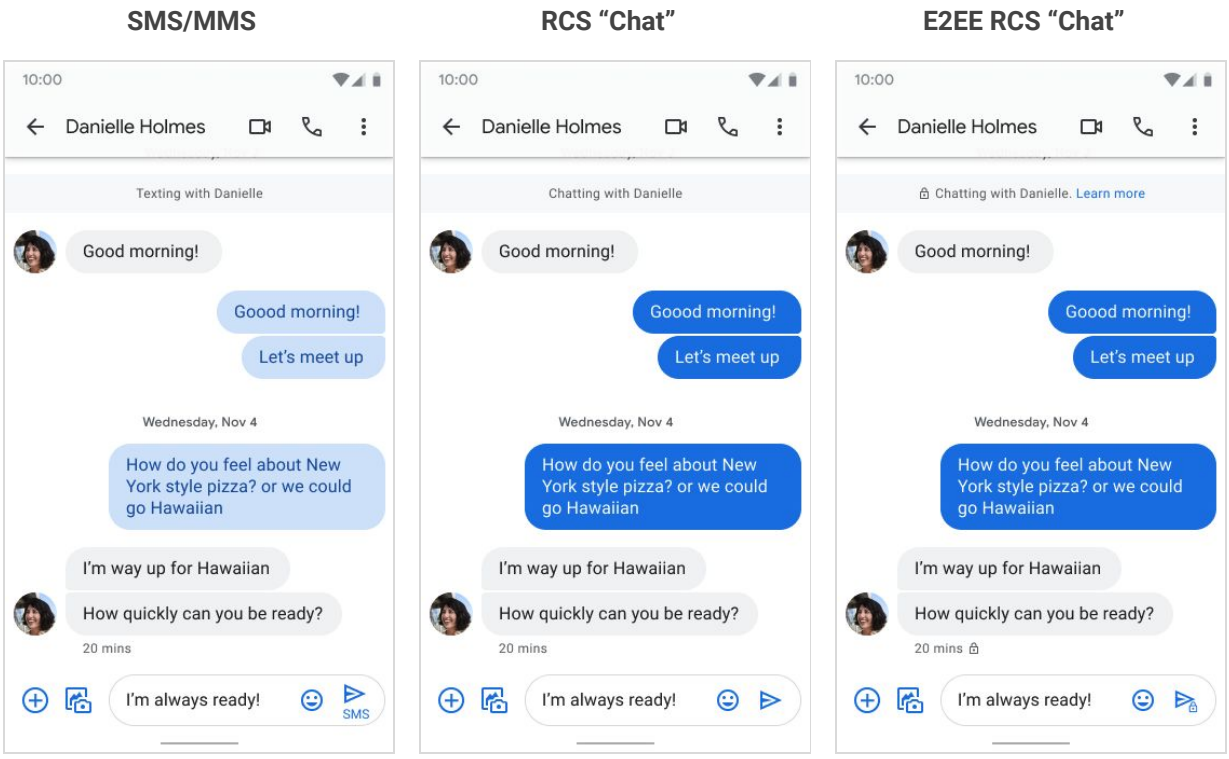
- **A malicious user**
  Users can only decrypt messages addressed to them. They can't impersonate other users without being detected.

Goals

- **Confidentiality**
  Only clients in the conversation can access the message content.
- **Integrity**
  The encrypted message can't be modified in transit without being detected.
- **Authentication**
  A malicious client can't spoof another client phone number and send encrypted messages without being detected.
- **Forward Secrecy & Post-Compromise Security**
  If one message happens to be compromised, the security of all past and future messages is still maintained.
- **User Experience**
  Provide a consistent user experience.

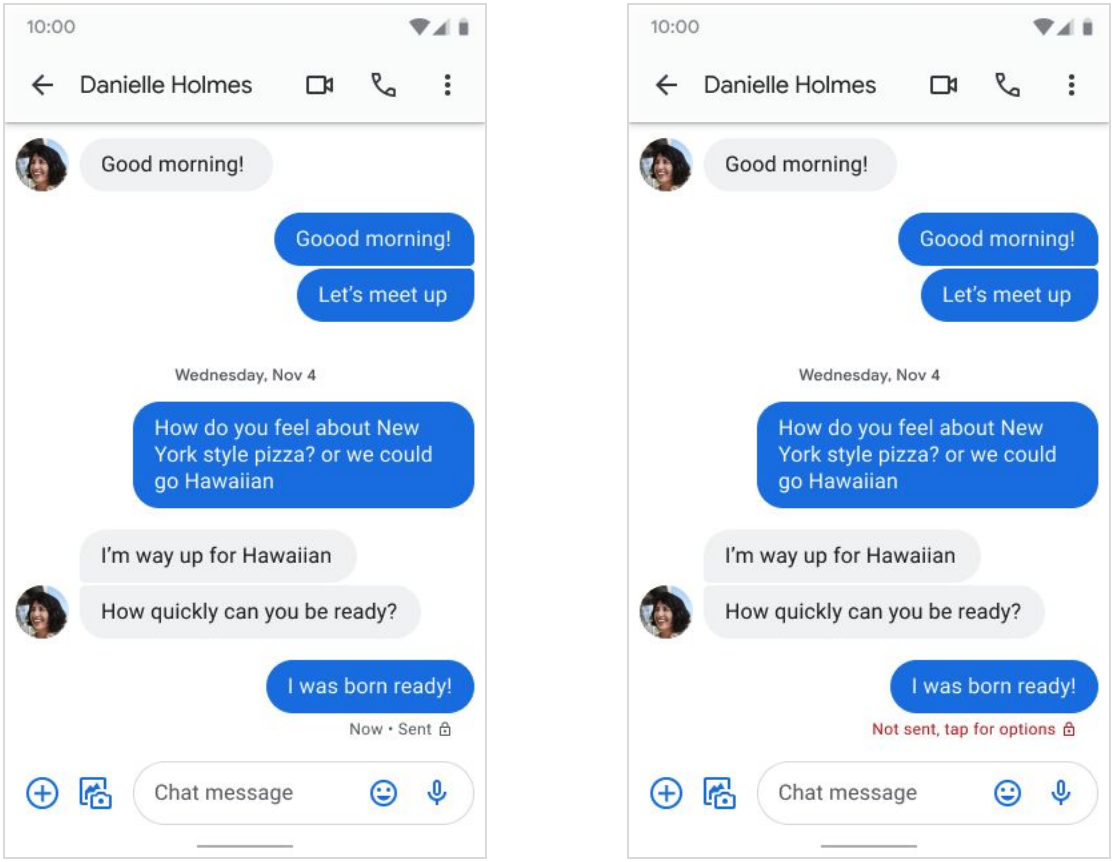
UI Changes

Starting with one-to-one conversations, all RCS chat messages will be E2EE if both clients have the latest version of Messages. Conversations already automatically upgrade from SMS to RCS when eligible and now they will upgrade to end-to-end encrypted when eligible. Messages already differentiates between RCS and SMS/MMS messages by using different colors. To signify when the conversation is end-to-end encrypted, we added lock icons on the send button in the compose field and next to the timestamp of the message.



SMS/MMS Fallback

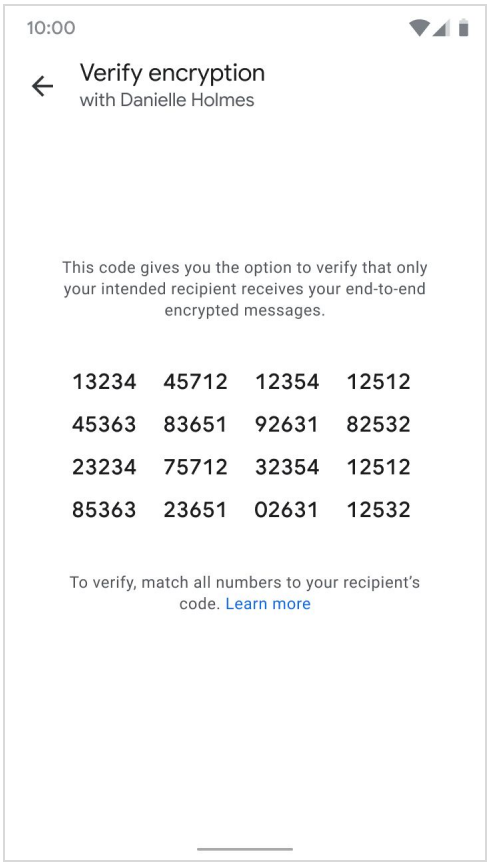
One of the many benefits of default messaging is that users can connect over cellular or data networks. To improve both the security and experience for users, we’ve upgraded the default so messages, when eligible, will send over RCS and not automatically fall back to SMS/MMS. If the recipient is offline, these messages will be stored in the server and delivered when the recipient is back online, but the sender can choose to resend as SMS/MMS instead.



A sent end-to-end encrypted message (left) and if the message fails to send (right)

Identity Verification

For additional assurance that the conversation is E2EE, users can verify they are getting the right public key for the other side of the conversation by comparing the verification code, which is the fingerprint of the two identity keys.
This step is only needed when the users message for the first time or receive a new identity key due to app reinstall or switching to a new device.



The fingerprint is a 60-digit numeric string generated by doing 512 iterations of SHA512 for both identity keys.

We will continue to iterate on this screen to make it easier for users to compare these codes in the future.

E2EE in Messages

Starting with one-to-one conversations, all RCS chat messages will be E2EE if both clients have the latest version of Messages. The following section will cover the technical details for how Messages implemented E2EE for RCS.

Signal Protocol

Messages uses the Signal Protocol[3] to build E2EE for RCS messages, which allows us to achieve all of the security properties mentioned earlier.

When the client is set up, it generates the following Curve25519 key pairs:

- Identity key: A long-term key pair associated with the user on this device

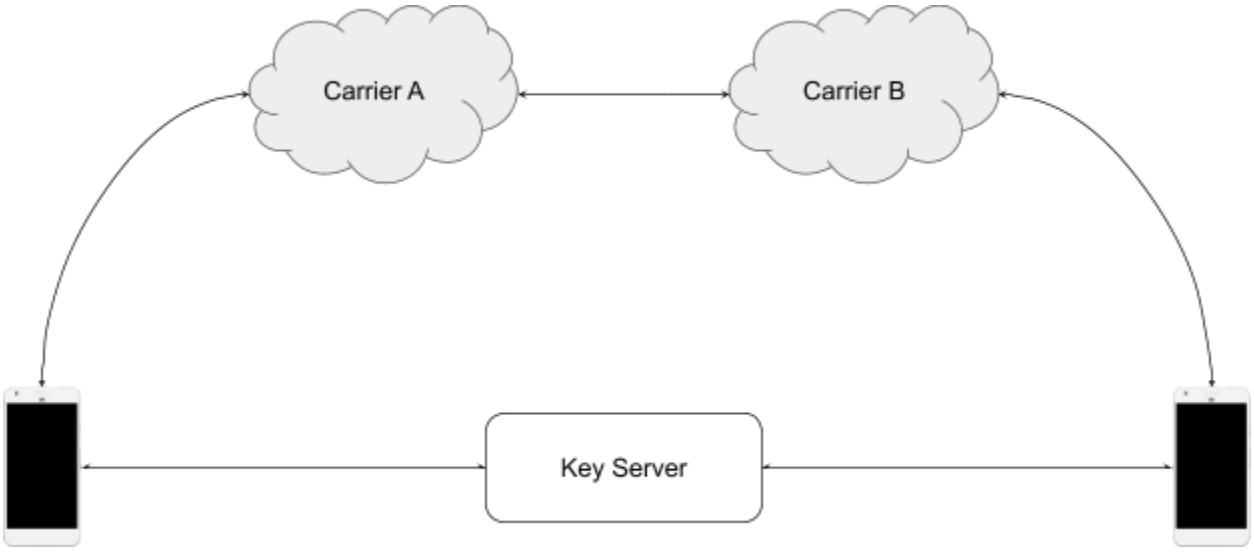
- Signed prekey: A medium-term key pair, with the public key signed by the Identity Key
- Unsigned prekeys: A set of short-term key pairs

These 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.

Signal uses these keys to set up the E2EE session between two clients using the triple Diffie-Hellman algorithm (X3DH)[4] and derives 256 bit root key material from them. The root key is used to derive another 256 bit key material called chain key. From the chain key, it derives the message encryption key, authentication key, and initialization vector. Every message is encrypted with different keys as the chain key and root keys change using the Double-Ratcheting[5] algorithm.

Key Server

In order to store and exchange user public keys like identity keys and prekeys, we need to have a central key server. Unlike the message server, this server is always hosted by Google to store users' public keys.



When the client turns RCS on for the first time, it also registers with the key server and uploads the public key parts that are used for session setup. For example, when Alice talks to Bob for the first time, her client requests the following information from the key server:

- Bob's identity key
- Bob's signed prekey
- [Optional] one of Bob's prekeys

The server deletes the returned prekey from its storage. Alice uses these keys in the triple Diffie-Hellman algorithm to derive the session keys.

When the key server detects that one of the clients is low on prekeys, it sends a push notification for that client to generate and upload a new set of prekeys and a new signed prekey.

Messages Encryption

Once a secure session is established using remote client prekeys, Signal derives the following values:

- 256 bit AES encryption key
- 256 bit MAC key
- 128 bit initialization vector

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. A 64 bit MAC is computed over the serialized protocol buffer using HMAC-SHA256 to create the final message payload. After each encryption, Signal advances the chain key to achieve forward secrecy.

The encrypted payload is encoded in Base64 and added to the MSRP packet while keeping everything else, like sender & recipient info and timestamp, in plaintext so the message server can continue to route messages properly.

```
content-type: message/cpim
...
To: +12074651140
From: +12079102224
DateTime: 2020-11-18T18:13:07.861Z

Content-Length: 215
Content-Type: application/vnd.google.rcs.encrypted; charset=utf-8

CnYIARJyMwohBdeTpErO+LvfI9dvQt0IJlm44sd+6bmIBa4ca4kjNj4KEAAYACJA5d1VbsJgz2QCy
bdpzVml0pMWVzfYXCJZu/KbuRidycuHMfQ3pAbalGNQeZfVo7EpG3GtEKeXJypXLtRSeolSNLtRUg
akGL7PEiQwZjRmYmYzMy1mYzdkLTRmOWMtYTA0MC00NjQyNDE3MjExY2Y=
```

E2EE RCS message

We also introduced a new content type “rcs.encrypted” for E2EE messages, so the recipient client can handle it properly.

Typing indicators and delivery and read receipts are also E2EE using the same format, however we introduced a new flag in the metadata to indicate the message class (i.e ephemeral) to the server. For example, we don’t want the server to send push notifications for these types of messages.

Attachment Encryption

Attachments in one-to-one conversations are also covered by E2EE. In RCS, an attachment is uploaded to the carrier content store, then a link to that file along with other metadata, like filename and size, are sent in the RCS message in XML payload. With E2EE, we divided this into steps:

- **File encryption**
  Before the sender client uploads the file, it generates a fresh 256 bit random key material from which it derives 256 bit symmetric encryption key and 128 bit initialization vector which are used to encrypt the file using AES-CTR 256. Then it generates a 256 bit digest using SHA256. The encrypted file is then uploaded to the carrier content store.
- **Message encryption**
  The file metadata, key material, and digest are encrypted using the Signal session and the same prior message format.

The recipient client does the following operation:

1. Decrypts the message to retrieve the file metadata, key material, and message digest
2. Downloads the encrypted file from the content store
3. Verifies the digest, dropping the message if verification fails
4. Derives the encryption key and IV from the key material
5. Decrypts the file

Since the actual RCS message payload is E2EE, the message server can't differentiate between regular messages and attachments anymore.

Session Recovery

If the recipient client loses its local state, which includes the current session keys, for any reason, such as app reinstall or switching to a new phone, and receives a message that's encrypted using the old session keys, the recipient client will fail to decrypt that message. The recipient sends an error control message to the sender which automatically reestablishes the session by requesting a new prekey from the keys server, then re-encrypts and sends the message.

Web Client

Messages for web also supports E2EE conversations by using the phone to send and receive the messages. The client establishes a secure connection with the phone by scanning a shared key encoded in a QR code. The QR code contains two keys generated using WebCrypto:

- 256 bit symmetric encryption key
- 256 bit authentication key

Messages between the web client and phone are encrypted using AES-CTR with a random 128 bit initialization vector and authenticated using HMAC-SHA256. The Google server can't access the contents sent between the web client and the phone.

Storage & Access

Android Messages Database

As part of an open, interoperable ecosystem, SMS and RCS messages are stored in Android's local on-device messages database[6] to allow seamless integration with companion applications such as reading messages aloud in your car or sending from your home device or watch. Only applications with SMS permissions can access this database, and users can manage this permission from the Android settings. Messages also uses this central database to include SMS and RCS messages in Android system backup, so messages transfer to a new client or device. Starting from Android P, the Android system backup is end-to-end encrypted with a secret key derived from the user's lock screen PIN/pattern/passcode[7] so Google servers can't access them.

To keep the same user experience and avoid losing messages when users move to a new device, E2EE RCS messages will continue to be stored in the device's messages database and accessible to apps with SMS permissions. We will work across the ecosystem to further reduce how apps access E2EE messages and we are working on improving how these messages are stored and accessible to other applications without compromising the companion messaging experience.

Notifications

In addition to SMS permissions, other apps may have access to end-to-end encrypted messages through incoming messaging notifications. This type of access can be managed by the user on a per app basis through SMS permissions or notification access.

Limitations

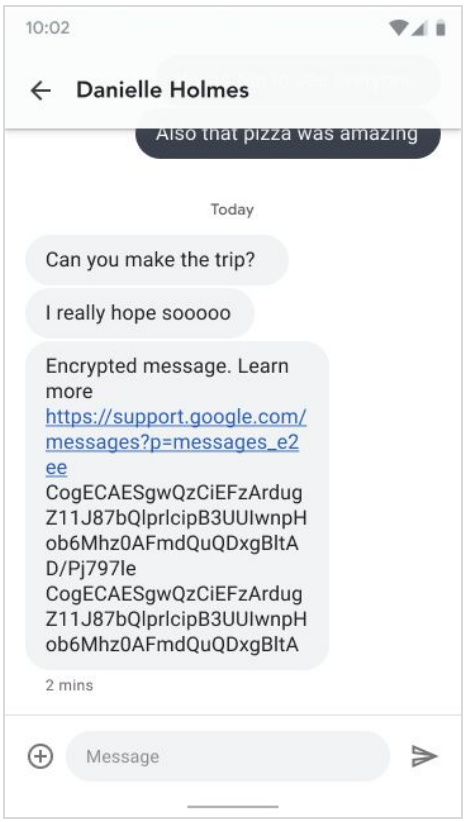
Unlike traditional messaging apps where the service provider controls the client and server, RCS is a federated protocol implemented by other clients and servers. We made sure that E2EE messages will continue to be delivered though other RCS servers without breaking the user experience, but this comes with some challenges and limitations that we will continue working to overcome.

Third Party RCS Client

E2EE is implemented in the Messages client, so both clients of the conversations must use Messages, otherwise the conversation becomes unencrypted RCS. In rare situations where the conversation starts as E2EE, then one of the clients migrates to a different RCS client, Messages doesn't detect that immediately, so if the Messages user sends a new message, it's still E2EE, however the recipient client renders the encrypted base64 payload.

To improve this situation, we append "Encrypted message…" prefix to the encrypted payload and link to the documentation to explain to the recipient user what is happening. Also, this situation should only happen for the first message, as the Messages client recovers when it receives a plaintext

delivery receipt and downgrades the conversation to unencrypted RCS. It doesn't automatically resend the problematic encrypted message.



This situation could also happen if the recipient client is using an older version of Messages that doesn't support E2EE.

Conclusion

E2EE in Messages has been a big challenge, given how complex the RCS ecosystem is. We managed to build solutions for most of these challenges and we will continue to build solutions for the rest of them and bring E2EE to more features in Messages in order to improve our users' privacy and security.

REFERENCES

1. Session Initiation Protocol [here]
2. Message Session Relay Protocol [here]
3. Signal Protocol [here]

4. X3DH [here]
5. Double Ratcheting [here]
6. Android Messages Database [here]
7. Android E2EE backup [here]