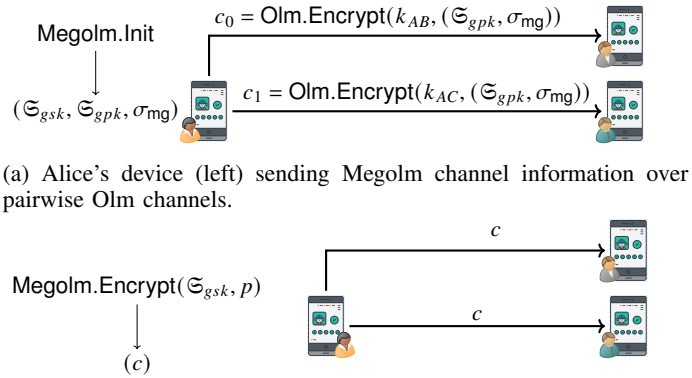
Practically-exploitable Cryptographic Vulnerabilities in Matrix

Martin R. Albrecht*, Sofía Celi†, Benjamin Dowling‡ and Daniel Jones*
*Information Security Group, Royal Holloway, University of London, {martin.albrecht,dan.jones}@rhul.ac.uk
† Brave Software, cherenkov@riseup.net
‡ Security of Advanced Systems Group, University of Sheffield, b.dowling@sheffield.ac.uk

***Abstract*—We report several practically-exploitable cryptographic vulnerabilities in the Matrix standard for federated real-time communication and its flagship client and prototype implementation, Element. These, together, invalidate the confidentiality and authentication guarantees claimed by Matrix against a malicious server. This is despite Matrix' cryptographic routines being constructed from well-known and -studied cryptographic building blocks. On the one hand, one of our attacks proceeds by chaining three attacks to achieve a full authentication and confidentiality break. On the other hand, the vulnerabilities we exploit differ in their nature (insecure by design, protocol confusion, lack of domain separation, implementation bugs) and are distributed broadly across the different subprotocols and libraries that make up the cryptographic core of Matrix. Together, these vulnerabilities highlight the need for a systematic and formal analysis of the cryptography in the Matrix standard.**



(a) Alice's device (left) sending Megolm channel information over pairwise Olm channels.

(b) Alice's device (left) sending payload data over the unidirectional Megolm channel.

Fig. 1: Alice establishes a Megolm channel (a) and sends a ciphertext (b).

I. Introduction

Matrix [1] is an open standard and communication protocol roughly aiming to do for real-time communication what SMTP does for email. In particular, the specification defines a federated communication protocol allowing clients, with accounts on different Matrix servers (their *homeservers*), to exchange messages across the entire ecosystem. Since this setting inherently involves untrusted third party servers, the specification enables end-to-end encryption by default.[1]

While Matrix' federated nature makes it difficult to assess how widely it is used, several notable organisations and institutions have adopted it or announced plans to do so. For example, both KDE and Mozilla announced plans to switch their internal communications to Matrix in 2019; the Fourth Estate announced its plans to build an encrypted messenger for journalists and news organisations based on Matrix in 2021; the French government announced plans to create their own instant messaging app – *Tchap* – based on Matrix which was released in 2019; the German ministry of defence launched *BwMessenger* – for use in internal, official (and classified) communication – based on Matrix in 2020 with a view to move over other parts of the German government; the German healthcare system announced its plans to adopt Matrix in 2021. In March 2021, matrix.org – the most popular Matrix server – announced that there are 28 million global visible accounts. The Element (see below) website claims +60M Matrix users.

The most popular implementations of the Matrix server and client are *Synapse* and *Element* (Desktop, Android, iOS, Web) respectively. While the security guarantees of Matrix and these popular implementations have received attention from the information security practitioners community – e.g. CVE-2022-31052, CVE-2022-23597, CVE-2021-41281, CVE-2021-39163, CVE-2021-39164, CVE-2021-32659, CVE-2021-32622 and CVE-2021-29471 – its bespoke cryptographic protocol has not received an in-depth treatment from the cryptographic (academic or practitioner) community. That is, while Matrix uses TLS to secure the communication between clients and servers and between servers (for federation), end-to-end encryption is realised using a custom cryptographic protocol called *Megolm* which extends *Olm* to support group chat (see below). Since in Matrix every chat is a group chat, including 1-on-1 chats (as users have different devices), the study of its Megolm group messaging protocol is central to understand its security guarantees. We begin by describing the Matrix cryptographic protocol.

A. Matrix Overview

In Matrix a **user** may have several devices (e.g. a phone and a laptop). For a user Alice with identifier A, we refer to their

[1] In addition to standard security considerations such as breaches or lack of trust in a single-server setting.

ith device as (A, i) with device identifier $D_{A,i}$. Each user has an account with a **homeserver**, which allocates their user and device identifiers. There are many homeservers, i.e. the protocol is federated, but for our purposes it suffices to think of the network of homeservers as a single such homeserver that facilitates communication.

A **room** is a collection of devices communicating in a single conversation. Each unique pair of devices in a room shares an **Olm** channel [2], used to share and exchange channel establishment information for the devices' Megolm channels [3]. The Olm protocol is an implementation of a modified 3DH key exchange protocol [4][2] and the Signal Double Ratchet algorithm [6], [1]. Olm plaintexts exchanged between devices are not visible to users, and are used to manage Megolm channels as in Fig. 1. Out-of-band verification allows users to verify that the received (device-specific) Olm public keys are actually owned by that device.

Each **Megolm** channel [3] is a unidirectional channel, used to send payload information from one device to all other devices in the room. The composition of unidirectional Megolm channels enables groups of devices to communicate in a single conversation. These unidirectional Megolm channels are used to exchange all instant messages (those entered and seen by users). All conversations are implemented as group messaging, even with only two devices present. We use the terms *Megolm channel* and *Megolm session* interchangeably, but prefer the latter to emphasise a party's view or state of a channel.

After channel establishment, senders (and receivers) symmetrically ratchet (via the bespoke **Megolm Ratchet** [3]) the shared secret state forward after each message sent (resp. received) by the unidirectional channel, aiming to achieve forward secrecy. We note, however, that the specification allows implementations to keep old copies of the ratchet on the receiving side [3], [7] – something which `matrix-js-sdk` does – and that this invalidates forward secrecy guarantees. In addition, senders can periodically generate a new (and independent) Megolm secret state, and send it to the receiving devices in the room via Olm, thus aiming to achieve some form of post-compromise security.

Each device has a unique cryptographic identity (with long-term signing keys). Matrix optionally allows users to verify and sign each others identities and devices. The **Cross-Signing** module defines cryptographic identities for users and their devices (each consisting of one or more Ed25519 [8] key pairs). These are linked with one another using Ed25519 signatures, as the result of verification through the **Verification Framework**. It provides protocols for users to verify other users, and their own devices, using an out-of-band channel. It provides two protocols: **Short Authentication String** (SAS) verification and QR code verification.

By default, a user's secret cross-signing keys are generated and stored on the first device for which they login. However, it is important for a user to be able to recover their cross-signing identity if they lose access to this device (or simply log out).

The cross-signing module uses the **Secure Secret Storage and Sharing** (SSSS) module to store and backup users' secret keys. SSSS enables users to backup secrets to their homeserver (encrypted using a recovery passphrase) as well as to share those secrets with their verified devices (over the Olm protocol). The SSSS module provides a generic facility for storing and sharing user secrets.

In addition to SSSS, Matrix offers two modules to enable devices to backup and share Megolm sessions specifically. The **Key Request** protocol provides a means for devices to request and share copies of inbound Megolm sessions with each other. The **Server-side Megolm Backups** module enables devices to backup copies of inbound Megolm sessions on the server. This allows a user's new device to gain access to old messages the user has access to, even if no other devices are online (that could otherwise distribute the sessions using the Key Request protocol). The recovery key used to decrypt these backups is itself stored and shared using the SSSS module.

B. Prior Work

Cryptanalysis: An audit of the Olm and Megolm protocols (along with their reference implementation) was performed by NCC Group in 2016 [7]; this audit found a number of security issues that have now been fixed or recorded as limitations in the protocol specification [3], [2]. Since then, several further cryptographic vulnerabilities have been reported, e.g. CVE-2021-34813 and CVE-2021-40824. Wong reported a vulnerability in the out-of-band verification provided by the Short Authentication String (SAS) protocol in Matrix [9, Chapter 11]. These suggest that further study is needed to assess the resistance of Matrix to cryptanalytic attacks. In 2022 Matrix started a series of audits of their (future) core libraries [10], [11].

Formal analysis: As mentioned above, Olm is a modified implementation of the Signal protocol, which itself has received multiple formal analyses over the last seven years [12], [13], [14]. Further, the Megolm protocol shares its architecture with the *Sender Keys* variant of the Signal protocol [15]. This variant is also used to implement group messaging in WhatsApp. For this reason, existing analysis of these protocols will be relevant to Megolm; [16] provides one such example. However, none of these works cover Olm or Megolm itself.

C. Contributions

We report several practically-exploitable vulnerabilities in the end-to-end encryption in the Matrix standard and describe proof-of-concept attacks exploiting these vulnerabilities. When relying on implementation specific behaviour, these attacks target the Matrix standard as implemented by

[2] 3DH key exchange is a pre-cursor to the Extended Triple Diffie-Hellman (X3DH) key exchange protocol [5].

the `matrix-react-sdk` and `matrix-js-sdk` libraries.[3] These libraries provide the basis for the aforementioned Element flagship client.

We are primarily interested in a setting where encrypted messaging and verification are enabled, i.e. in the presence of the strongest protections offered by the protocol. Furthermore, all attacks require cooperation of the homeserver. This is a natural threat model to consider, given that end-to-end encryption aims to provide protections against such untrusted third parties. As mentioned above, for ease of exposure, we assume a single homeserver in this work. We report the following vulnerabilities and attacks:

a) Trivial confidentiality break: In Section III we start by reporting two trivial attacks breaking confidentiality of Megolm channels, the central object in Matrix secure messaging, thus breaking confidentiality of user messages. These attacks exploit the homeserver's control over the list of users and/or devices in a room. The attacks differ in whether they target the list of users in a room or the list of devices of a user. We note that this attack does not break confidentiality of the underlying Olm sessions, nor does it break the cryptographic guarantees of individual Megolm sessions. Instead, we utilise the control that homeservers have over the room participants to force target clients into sharing decryption keys with devices under the attacker's control. These attacks exploit issues in the specification.

b) Attack against out-of-band verification: In Section IV we report an attack on out-of-band verification in Matrix. This enables an attacker to convince a target to cryptographically sign (and thus verify) a cross-signing identity controlled by the attacker. This attack exploits a lack of domain separation between device identifiers and users' master signing keys. The attack enables a mallory-in-the-middle (MITM) attack breaking confidentiality and authenticity of the underlying Olm channels (and thus also Megolm channels). This attack exploits an insecure implementation choice permitted by the specification which does not enforce domain separation.

c) Semi-trusted impersonation: In Section V we report on an impersonation attack against Megolm by which attackers achieve the same level of authentication as keys honestly *forwarded* through the Key Request protocol (cf. Section II-H). Since the Key Request protocol does not provide the same security guarantees as non-forwarded keys, messages decrypted using forwarded keys are flagged as less trustworthy in the UI. Whilst Matrix clients restrict who they share keys with, no verification is implemented on who accepts key shares from. Our attack exploits this lack of verification in order to send attacker controlled Megolm sessions to a target device, claiming they belong to a session of the device they wish to impersonate. The attacker can then send messages to the target device using these sessions, which will authenticate the messages as coming from the device being impersonated. This exploits an implementation bug supported by a lack of guidance on processing incoming key shares in the spec.

d) Trusted impersonation: In Section VI-B we report on a second impersonation attack against Megolm which builds on the first. Here, we exploit protocol confusion, whereby message types expected to originate from an Olm channel will be accepted when sent over a Megolm channel. Briefly, the attack proceeds by initiating a new Megolm session over the Megolm channel established through the previous attack (Section V). This *inner* shared Megolm session inherits its *sender* from the outer, forged Megolm session, but without inheriting its *forwarded* status. Thus, this attack allows an attacker to *upgrade* the level of trust enjoyed by the key material sent by the attacker such that no indication is given in the UI that a user should treat it with caution. That is, this attack convinces a target device of the validity of the impersonated session to a stronger degree than legitimate execution of the Key Request protocol. As a consequence, an attacker can outperform a legitimate party in convincing a target device of the validity of a sending identity. This exploits an implementation bug aided by the overall design of cryptographic processing in `matrix-js-sdk`.

e) Impersonation to confidentiality break: In Section VI-C we report on a confidentiality break against Megolm which builds upon the semi-trusted impersonation attack in Section V. Upon completing out-of-band self-verification, the newly verified device will use the SSSS protocol to request a copy of the key used for server-side Megolm backups from the verifying device. The Olm/Megolm protocol confusion in Section VI-A (also exploited in Section VI-B) can be exploited by an attacker to impersonate a trusted device and reply to the request, setting the Megolm backup key used by the newly verified device.[4] The device will accept this key and proceed to backup inbound Megolm sessions to the homeserver. The attacker and colluding homeserver are then able to decrypt the backups, giving them access to the plaintext of every Megolm message the target device has access to. This exploits an implementation bug.

f) IND-CCA break: AES-CTR is used for encryption in both the SSSS protocol and in symmetric Megolm Key Backups. However, the initialisation vector (IV) for AES-CTR is not included in the message authentication code (MAC). A similar issue exists when attachments are shared. This can be exploited to break the IND-CCA security of the underlying encryption scheme: an adversary is able to decrypt a challenge ciphertext by querying encryption and decryption oracles, without

[3]Our analysis is based on, and our proof-of-concept attacks tested against, Element Web at commit `#479d4bf` with `matrix-react-sdk` at commit `#59b9d1e` and `matrix-js-sdk` at commit `#4721aa1`. We note that while these SDKs are still the default, Matrix is in the process of transitioning to `matrix-rust-sdk` [17].

[4]That is, the attack in Section VI-C exploits the same class of vulnerability as Section VI-B, but a different instance of this vulnerability.

requesting decryption of the challenge ciphertext directly. However, in practice we do not know how to instantiate this attack and thus it, in contrast to those mentioned so far, is only of theoretical interest. See Appendix A. This is an issue in the Matrix specification.

In summary, we found that Matrix and its flagship client Element as deployed provide neither authentication nor confidentiality against homeservers that actively attack the protocol, i.e. its end-to-end encryption falls short of the security guarantees expected from it.

D. Disclosure

We disclosed our attacks to the Matrix developers between 20 May 2022 and 6 July 2022. They acknowledge these as vulnerabilities except for one of our attacks on confidentiality (discussed in Section III-C) which they consider as an accepted risk (but aim to mitigate regardless). We coordinated a public vulnerability disclosure for the 28 September 2022, to coincide with the first set of countermeasures. These should provide immediate fixes (to varying degrees) for the attacks in Sections III to VI. At the time of public disclosure, the Matrix specification and Element will not be vulnerable to the attack against out-of-band verification (Section IV), the semi-trusted impersonation attack (Section V), the trusted impersonation attack (Section VI-B) and the impersonation to confidentiality attack (Section VI-C). A second set of countermeasures is currently in the design phase, which aim to provide complete fixes for every vulnerability in this work.

In particular, the attacks concerning homeserver control of room membership and user's device lists (Section III) will not be fixed at the time of disclosure. However, a new local per-room setting will be added alongside the disclosure in order to mitigate the homeserver's control of user device lists. In the long-term, the Matrix developers plan to develop fixes for both of these attacks (detailed in Section III-C). A fix for the IND-CCA break (in Appendix A) will also be distributed at a later date. Since the IND-CCA break appears not to be practically exploitable, this should not affect users.

To aid readability, throughout the remainder of this work, we use the present tense to refer to vulnerabilities and behaviours in Matrix clients or servers even if these have since been addressed in response to our vulnerability disclosures.

E. Scope

We discuss the scope of this work:

First, we exclusively considered the Matrix specification and the Matrix flagship client Element. There are other clients available, see e.g. [18], some of which also use the `matrix-js-sdk` and some of which do not, but we did not investigate to what extent these other clients are vulnerable to our attacks or variants thereof. For the avoidance of doubt, any implementation specific behaviour reported throughout exclusively refers to `matrix-js-sdk` and Element, even when we write "Matrix".

Second, in this work we focus on authentication and confidentiality, i.e. the two most fundamental security properties provided by cryptography, without the need for a client secret compromise. Matrix aims to provide stronger notions of security such as forward secrecy (i.e. before a client secret compromise), post-compromise security (i.e. eventually after a client secret compromise) and deniability. Given our results against more fundamental security goals, we consider such more advanced notions of security out of scope.

Third, our attacks target Matrix in the setting where every device and user have performed out-of-band verification. In this ideal scenario, the client interface will display a warning next to messages that are unencrypted, or cannot be cryptographically linked to the claimed sender. From the perspective of an attacker, this is the most challenging and thus interesting setting. However, when a user has not been verified, the Element client will no longer display such warnings. The Matrix specification does not enforce that messages in encrypted rooms are indeed encrypted. This renders impersonation attacks trivial: the attacker, in collusion with the homeserver, simply sends an unencrypted message with a forged sender. As such, even when the issues described in this work are fixed, clients operating in this non-ideal setting do not offer any cryptographic authentication guarantees.

II. Preliminaries

We write *semi-trusted* for messages that are accepted but displayed with a warning and *trusted* for messages that are accepted (without warning).

A. Algorithms

We reference the following algorithms:

- $\mathsf{sort}(x_1, x_2, \ldots, x_n)$ returns a sorted copy of the list $[x_1, x_2, \ldots, x_n]$.
- $\mathsf{HMAC\text{-}SHA\text{-}256}(k, m)$ is a Hash-based Message Authentication Code (HMAC) constructed with the SHA-256 [19] hash function taking as input a key k and message m [20]. Matrix truncates HMAC outputs to 64 bits[5], which contrasts with the HMAC RFC [21] which recommends at least 128 bits for SHA-256 (no less than half the length of the hash output) and not less than 80 bits.
- $\mathsf{HKDF\text{-}SHA\text{-}256}(s, k, c, \ell)$ is a Hash-based Key Derivation Function (HKDF) constructed with SHA-256 where s is the salt, k is the secret key material, c is the info/context and ℓ is the output length in bytes [22], [23].
- $\mathsf{AES}(k, m)$ is AES [24] taking a key k and a message block m of size 128 bits. Matrix uses AES-256, i.e. keys of length 256 bits.
- $\mathsf{AES\text{-}CTR}(iv, k, m)$ is AES in counter (CTR) mode [25]

[5]This issue was noted in an audit[11] and will be fixed in a future version of the Olm and Megolm specifications.

where iv is the nonce, k is an AES encryption key and m is a message.

• AES-CBC(iv, k, m) is AES in cipher block chaining (CBC) mode [25] where iv is the nonce, k is an AES encryption key and m is a message. Matrix uses PKCS7 [26] padding to split plaintexts into blocks for CBC mode in the Olm and Megolm protocols, as well as the asymmetric Megolm backup scheme.

B. Message Types

Matrix supports a number of message types. Messages with the type `m.room.message` are instant messages entered and seen by users. Encrypted messages consist of an inner plaintext message (with its own message type) that has been encrypted and placed in an outer message structure with the type `m.room.encrypted`. The outer structure of an encrypted message is an unauthenticated wrapper, specifying the encryption algorithm used. We introduce additional message types in this text as needed.

Messages are sent either to a particular room, in which case they will be distributed by homeservers to the devices of all users in the room, or to a particular device. In the latter case, these are known as *to-device* messages.

C. Users, Identities and Cross-Signing

Upon registration, each **user** is allocated a user identifier A of the form `@localpart:domain` by their homeserver. Similarly, when a new **device** logs in with account credentials, the homeserver allocates device identifier $D_{A,i}$. The device then generates its keys (a) Device Fingerprint/Signing Key $(ask_{A,i}, apk_{A,i})$, and (b) Olm Key Bundle $(isk_{A,i}, ipk_{A,i}, esk_{A,i}, epk_{A,i}, fsk_{A,i}, fpk_{A,i})$. It registers them with the homeserver (as a bundle self-signed with $ask_{A,i}$). The tuple $(esk_{A,i}, epk_{A,i})$ represents the pre-key bundle consisting of one or more key pairs. The tuple $(fsk_{A,i}, fpk_{A,i})$ represents a bundle of one or more fallback key pairs [27].

The cross-signing module [28] provides support for cryptographic user identities. It defines three sets of cryptographic keys for each user in the form of Ed25519 digital signature key pairs: (a) Master Keys (msk_A, mpk_A); (b) User-signing Keys (usk_A, upk_A); (c) Self/Device-signing Keys (ssk_A, spk_A).

These key pairs are generated on the device where cross-signing is setup. The master key mpk_A signs both the self-signing key ssk_A and the user-signing key upk_A. The self-signing key is used to sign a user's own devices, while the user-signing key is used to sign other users' master keys. These signatures are created and distributed by the homeserver.

When logging in to a new device, users are prompted to (optionally) verify it (as described in Section II-D). An existing device (with control of the user's cross-signing keys) and the new device then mutually verify each other out-of-band. Once verification is complete, the new device's keys $(apk_{A,i}, ipk_{A,i})$ are signed by the user's self-signing key ssk_A. The existing device then distributes the cross-signing secrets to the new device using the secret sharing functionality of SSSS (Section II-E). This process is referred to as *self-verification*.

Additionally, when two users are communicating with one another, they may perform an out-of-band verification between two of their devices (each device should have a copy of their respective user's cross-signing keys). Once verification is complete, each user will sign the other's master key mpk with their user-signing key usk. This process is referred to as *cross-signing*.

Together, these enable pairs of users to verify each other's identities, then rely on the other user to verify each of their own devices. See Fig. 3.

In this work, the term *sender identity* refers to the combination of the user's cross-signing keys and a set of long-term device keys (apk, ipk).

Key		Description
msk_A	mpk_A	Master signing key for user A
usk_A	upk_A	User signing key for user A
ssk_A	spk_A	Self-signing key for user A
$ask_{A,i}$	$apk_{A,i}$	Fingerprint/signing key for A's ith device
$isk_{A,i}$	$ipk_{A,i}$	Olm identity key for A's ith device
$esk_{A,i}$	$epk_{A,i}$	Olm ephemeral pre-keys for A's ith device
$fsk_{A,i}$	$fpk_{A,i}$	Olm fallback keys for A's ith device

Fig. 2: Summary of the keys used in Matrix.

D. Out-of-band Verification

The Matrix standard defines an out-of-band verification framework allowing users to verify themselves [1]. This functionality enables users to ensure that the cryptographic identity they are communicating with correctly maps to the intended user. This is intended to prevent mallory-in-the-middle attacks in cases of *first use* in contrast to a *trust-on-first-use* approach.

Matrix defines multiple out-of-band verification protocols within this framework. Element defaults to using the **QR Code Verification Protocol** when the device has a camera, and a **Short Authentication String** (SAS) protocol in all other cases. In this work, we refer to users and devices that have gone through this process as having been *verified*.

Short Authentication String Protocol: We briefly describe Matrix' SAS protocol, focusing on the parts relevant to our attack in Section IV. The SAS protocol builds upon the ZRTP key agreement handshake [29]. It uses an ephemeral X25519 key exchange to compute a shared secret. Any attempts to modify the connection between the two parties should result in them computing different shared secrets. To detect this, the parties compare their shared secrets through an authenticated out-of-band channel. They then share their cryptographic identities, using the shared secret for verification. A detailed description of the protocol can be found in Figures 11 and 12.

Once the shared secret has been generated, each party compiles a list of the keys they wish to have signed into an

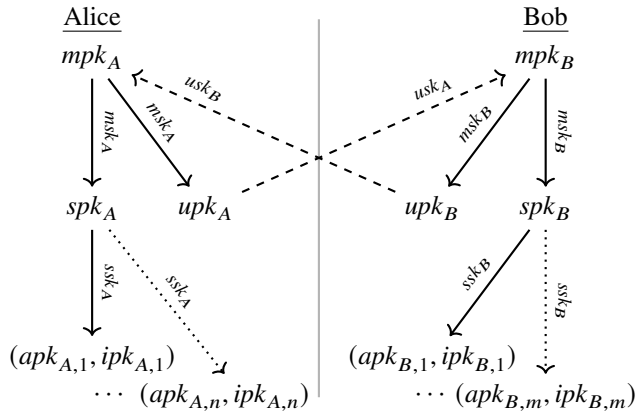


Fig. 3: An example of the long-term key hierarchy for two users, Alice and Bob, and each of their devices [1]. Each arrow denotes a signature. Dashed arrows denote signatures resulting from an out-of-band verification between Alice and Bob. Dotted arrows denote signatures resulting from an out-of-band verification between a user cross-signing session and a device (self-verification). Since the cross-signing session exists on the first device where cross-signing is enabled, the first device identity signed by each user is not the result of an out-of-band verification. Diagram based on [1, #cross-signing].

`m.key.verification.mac` message (Fig. 4). The shared secret is used to compute a message authentication code (MAC) for each key, calculated over its public part and details from the SAS protocol execution. A second MAC is computed over a list of key identifiers, corresponding to the list of keys for which MACs have been included. These MACs are added to the message, and ensure that only parties in possession of the shared secret can request keys for signing.

Out-of-band verification is used in two cases. (1) Two users are verifying each other: The protocol is executed between a device from each user (each of which holds the user's secret cross-signing keys), and each include their master cross-signing key *mpk* in the `m.key.verification.mac`, which the other device will sign using their user signing key *usk*. (2) A user is verifying one of their own devices: The protocol is executed between the verifying device (which holds the cross-signing secret keys) and the new device. The device being verified uses the `m.key.verification.mac` message to send their device identity key apk to the verifying device. The verifying device uses the device self-signing key *ssk* to sign the new device's identity key apk and Olm identity key *ipk*.

E. Secure Secret Storage and Sharing

The Secure Secret Storage and Sharing (SSSS) module enables devices to share secrets with the user's other trusted devices [1]. It provides two sets of functionality: (1) Backup secret key material to the server, encrypted symmetrically using a key generated from a master passphrase. (2) Distribute these secrets to other devices via a request-response protocol, similar to the Key Request protocol described in Section II-H.

This module is used to backup user cross-signing secrets and the recovery key for server-side Megolm keys to the server, as well as to distribute them between a user's devices.

We briefly describe the secret sharing functionality of SSSS. Upon completion of self-verification, the newly verified device will request copies of the user's secret cross-signing keys and secret Megolm backup recovery key from the verifying device. An unencrypted `m.secret.request` message is sent for each secret they are requesting. When the verifying device receives the request, it will check that the requesting device's identity is verified, then reply with the requested secret in an Olm encrypted, to-device `m.secret.send` message. When receiving `m.secret.send` messages, `matrix-js-sdk` requires that the response is encrypted and that the Olm identity key ipk used for encryption matches the device identifier they sent the request to.

F. Megolm Sessions

Megolm **sessions** consist of the following components: (a) Group signing and verification keys (gsk, gpk) using Ed25519; (b) Megolm ratchet R and (c) Megolm ratchet index i. While the nature of the ratchet in Megolm deviates from Olm and Signal, and is one of the innovations of the Megolm protocol, its details do not matter for our purposes. Thus, we may simply think of (i, R) as some symmetric key material. The group verification key gpk is used as the unique identifier for sessions. As such, it is also referred to as the *session identifier*. Megolm sessions are also referred to as the *room key*. A Megolm session is split into an *outbound* and *inbound session*. The inbound session allows the holder to decrypt and verify messages that were encrypted and signed by the outbound session. The outbound session contains $\mathfrak{S}_{gsk} := (i, R, gsk)$ while the inbound session contains $\mathfrak{S}_{gpk} := (i, R, gpk)$. Messages are encrypted using AES in CBC mode with HMAC applied to the ciphertext, so it is a encrypt-then-MAC construction. This authenticated ciphertext is then signed with gsk and the result sent to the homeserver for distribution. Figure 10 describes how Megolm sessions are initiated, as well as how they are used to encrypt and decrypt new messages.

G. Distributing Megolm Sessions

When a session is created, the inbound session is distributed to the other devices in the room. It is first encrypted and then signed with the group signing key gsk, wrapped inside an `m.room_key` message, and then sent over separate pairwise Olm channels between the session creator and each receiving device. To distribute these Olm ciphertexts to individual devices, their wrapper is formatted as a to-device message which the homeserver will deliver to the correct device.

Since the Megolm session has been sent over an Olm channel, the receiving device is able to cryptographically link

the Olm identity (see below) of the sending device with the session. When a message is verified, decrypted and displayed, participants use these keys (associated with the Megolm session used to decrypt it) to identify the cryptographic identity of the sender.

H. Key Request Protocol

There are cases where a device should have access to an inbound Megolm session, but missed its initial distribution. For example, when an existing member of the room adds a new device, the latter should be given access to all Megolm sessions created since the device joined that room.

To solve this, the Matrix standard defines a Key Request protocol, which allows devices in a group to request inbound Megolm sessions they need (the secret keys they are missing), and for devices to share them; where permitted.

The protocol starts by the requesting client sending a `m.room_key_request` to-device message to each device they are requesting from. When receiving `m.room_key_request` messages, each device must determine whether it should share the key with the requesting device. Devices may only share Megolm sessions with other devices of the same user or, (if they are the session owner) to devices which they have sent the session to in the past. Additionally, when sharing sessions with devices of the same user, those with cross-signing enabled may only share sessions with other verified devices. To fulfill a request, the sharing device packages their copy of the inbound Megolm session in *session sharing format* [3] (without the signature σ_{mg}) inside an Olm encrypted, to-device `m.forwarded_room_key` message [1].

I. Server-side Megolm Backups

The Matrix standard provides a mechanism for devices to backup copies of inbound Megolm sessions to the server. These backups are shared across different devices of the same user, enabling new devices to access Megolm sessions when the user's other devices are not online (and are, thus, unable to forward session keys through the Key Request protocol).

A backup configuration, specifying the backup scheme and key to use, is stored on the homeserver. Clients will trust this configuration if the field signifying the key to use has been signed with the user's master cross-signing key *msk*, or if the client already has a copy of the secret part of the key.

1) Asymmetric Megolm Key Backups: For setup, the client generates an X25519 [30] recovery key pair. The secret part of the recovery key pair is either generated from a user-supplied passphrase, or encrypted and backed up to the server using SSSS (described in Section II-E). The public key is signed by the user's master cross-signing master key *msk*, then uploaded to the homeserver as part of the backup configuration.

To backup a Megolm session, each device fetches the backup configuration from the homeserver. Clients trust the configuration if they possess the private part of the key, or if the public key has been signed by *msk*. Next, they generate a shared secret by performing DH key exchange with the recovery key and an ephemeral X25519 key pair. This shared secret is fed into HKDF-SHA-256 to generate an IV and key material for authenticated encryption using AES-CBC followed by HMAC-SHA-256, used to encrypt the inbound Megolm session. The resulting ciphertext is uploaded to the homeserver (with the public part of the ephemeral key).

An asymmetric encryption scheme such as this does not authenticate the party that has created the backup [31]. This opens clients to a impersonation attack (detailed in Appendix B).

2) Symmetric Megolm Backups: MSC 3270 [31] introduces an alternative scheme for encrypting server-side backups that does not have such a shortcoming. The scheme uses a shared secret to encrypt backups, since only devices in possession of the secret can create and decrypt such backups. Thus, clients do not necessarily mark sessions they receive through symmetric server-side backups as untrusted.

For setup, the client generates a secret from either a user-supplied passphrase or a secret stored with SSSS.

To backup a Megolm session, each device fetches the backup configuration from the homeserver. Clients trust the configuration if they possess the key, or if the key has been signed by *msk*. To encrypt the session, the shared secret is fed into HKDF-SHA-256 to generate key material for authenticated encryption using AES-CTR and HMAC-SHA-256 (using a randomly generated IV).

Whilst the asymmetric scheme remains the default method of encrypting Megolm backups (at the time of writing), a client will use this scheme if directed to by a trusted backup configuration.

III. Homeserver Control of Room Membership

In this section we consider the control that the homeserver has over metadata such as room membership, and the device list of a user. We consider two attacks that compromise confidentiality guarantees. The attacks in this section are trivial because no cryptographic protection exists by design.

A. Room Members

The Matrix standard allows roles and permissions to be assigned to users in a room. Amongst other things, these roles and permissions control which users are allowed to manage the room membership. However, room management messages (even in end-to-end encrypted rooms) are neither encrypted, checked for integrity nor cryptographically authenticated. A malicious homeserver can forge room management messages to appear as if they are from users with permission to change room membership, simulating the process of a new user

being invited then joining the room. Thus, the homeserver has control of the member list also for encrypted rooms.

While the specification does not require mitigations, the Element client exhibits some behaviour meant to mitigate such attacks. That is, when a user is added to a room, this will be displayed as an event in the timeline, and is thus detectable by users. However, we stress that such a detection requires careful manual membership list inspection from users and that to participants, this event appears as a legitimate group membership event. In particular, in sufficiently big rooms such an event is likely to go unnoticed by users. This immediately compromises the confidentiality of the room.

In environments where cross-signing and verification are enabled, adding a new *unverified* user adds a warning to the room to indicate that unverified devices are present. However, it is possible for a homeserver to add a verified user to rooms without changing the security properties of the room. This allows a colluding homeserver and verified user to eavesdrop on rooms not intended for them. In other words, the warning regarding unverified devices is independent to whether the device is intended to participate in the specific room. Finally we note that users may, of course, simply ignore warnings.

B. Device List

Each user has a list of devices associated with their account. This list is controlled by the homeserver. It exists in parallel to (and independently of) the cross-signing/verification system, which provides a cryptographically controlled list of devices for a user.

A malicious homeserver may create their own device that can then be added to the device list of an existing user in a room they wish to eavesdrop in. Whenever a device in the room next sends a message, they will share their Megolm session with the homesever-controlled device. The homeserver will then be able to decrypt future messages.

Again, the Element client exhibits some behaviour meant to mitigate such attacks. In environments where cross-signing and verification are enabled, adding an unverified device to the user's list of devices will alert their existing sessions to start the verification process. To avoid the notification, the homeserver can present two different versions of the device list depending on the user requesting it. When a user requests their own device list, the homeserver does not include the unverified device. When a different user requests the list, the homeserver includes an unverified device that they control. The target users' devices will not be aware that a new, unverified device has been added to their account. Therefore, their clients will not present the verification dialog.

Nevertheless, adding an unverified device to the room will add a warning indicator to the room. But the same caveats as in Section III-A for this control apply.

C. Remediation

At the time of public disclosure, the Matrix developers plan to mitigate the homeserver's control of the **device** list by implementing a per-room local setting to prevent sharing keys with unverified devices.[6] This high-assurance setting will stop the attack described in Section III-B, since clients will refuse to interact with the unverified devices added by a malicious homeserver.

In the long-term, the Matrix developers plan to require all devices to be verified before a user can participate in end-to-end encrypted conversations. This will be accompanied by a trust-on-first-use (TOFU) scheme that allows users to trust a user's master cross-signing key on their first interaction (without out-of-band verification). The user's cross-signing identity is then fixed by the client and cannot be overwritten by the homeserver.

The developers consider the fact that the homeserver controls **room membership** as a risk they accept as part of their threat model. Whilst they do not plan to include any countermeasures for this attack at the time of disclosure, they are developing a solution to target this stronger threat model. A brief summary of their design follows. When inviting a user to join the room, the inviting user must include the master cross-signing key of the new user in a signed message. In doing this, the transcript of invites form a tree of signatures, rooted in the room's creation event. This solution is currently in the design phase.[7] The rollout of such a solution will be made practical by the long-term fixes for the device list attack.

IV. Key/Device Identifier Confusion in SAS

In this section we describe an attack against the SAS protocol for out-of-band verification. In this attack, a malicious homeserver tricks parties executing the SAS protocol into signing cross-signing identities it controls, rather than their own. This enables the homeserver to perform an active MITM attack against users.

A. Vulnerability

In the SAS protocol, two parties compute a shared secret, then compare this shared secret through an authenticated out-of-band channel. Once established, the shared secret is used as the MAC key for an `m.key.verification.mac` message, containing the cryptographic identities for the other party to sign. In our attack, the homeserver is able to trick each party into including a homeserver controlled key inside their message. In effect, requesting the other device to sign a homeserver controlled identity, rather than their own.

Recall that in the context of cross-signing, out-of-band verification is used in two cases. First, where two users' devices verify each other; and, second, where a user verifies a new device (self-verification). However,

[6]Element provides a setting to "Never send encrypted messages to unverified sessions from this session". We suggest this is made the default behaviour. However, this does not prevent attacks against authentication.

[7]In [32] a solution to a similar problem is detailed (with the additional requirement that the membership list is kept private).

the SAS.SendMAC (Fig. 12) algorithm, which generates `m.key.verification.mac` messages (Fig. 4), does not distinguish between these two cases. It always sends both keys.

```
{"mac": {"ed25519:<device_id>":
   SAS.CalcMAC(k, apk, c || "ed25519:<device_id>"),
 "ed25519:<mpk>":
   SAS.CalcMAC(k, mpk, c || "ed25519:<mpk>"),},
"keys": SAS.CalcMAC}(
  k, sort("ed25519:<device_id>", "ed25519:<mpk>"),
  c || "KEY_IDS")
```

Fig. 4: The format of an `m.key.verification.mac` message for a user with cross-signing setup.

```
{"mac": {"ed25519:<mpk'>":
   SAS.CalcMAC(k, apk, c || "ed25519:<mpk'>"),
  "ed25519:<mpk>":
   SAS.CalcMAC(k, mpk, c || "ed25519:<mpk>"),},
 "keys": SAS.CalcMAC(
   k, sort("ed25519:<mpk'>", "ed25519:<mpk>"),
   c || "KEY_IDS")}
```

Fig. 5: An `m.key.verification.mac` message generated by a user with cross-signing master verification key mpk, long-term device key apk and device identifier mpk' (which is also the master verification key of a homeserver controlled cross-signing identity). Whilst the two entries in the `mac` dictionary could be distinguished by the differing second argument given to SAS.CalcMAC, SAS.VerifyMAC interprets the first entry as a device, and then passes it to SAS.SignDevice which interprets it as a cross-signing identity.

1) Cross-signing keys as devices: Within `matrix-js-sdk` and Synapse, cross-signing identities are sometimes treated as devices. The same is true in the SAS protocol, as noted in the Client-Server API specification [1].[8] Thus, in some cases the string x in `ed25519:<x>` is interpreted as a device identifier, and in others it is interpreted as a cross-signing master verification key.

Since the homeserver allocates device identifiers, it is able to generate a string that is both a valid device identifier and a valid cross-signing master verification key. In this attack, the homeserver generates a cross-signing identity for the user they would like to impersonate, then sets this as the device identifier for the user's first device. Figure 5 demonstrates the format of such a message.

2) Processing of `m.key.verification.mac`: When processing `m.key.verification.mac` messages, `matrix-js-sdk` handles the aforementioned ambiguity inconsistently.

Referring to Fig. 5, mpk' is both a valid device identifier and cross-signing master verification key. However, the MAC that has been calculated includes the device identity key of the device it maps to. It can only pass the MAC verification as a device, not as a cross-signing master keys.

When verifying the MACs in the message, SAS.VerifyMAC (Fig. 12) first checks if the string is a device identifier, then checks if it maps to a cross-signing identity. The entry for mpk' passes verification as if it were a request for a device to be verified.

When fulfilling the signing request, SAS.SignDevice (Fig. 12) first checks if the string maps to a cross-signing identity, then checks whether it is a device identifier. If the processing device has an entry for mpk' in its cross-signing directory, it will sign the user and cross-signing identity then upload it to the homeserver.

B. Attack

Consider a setting with two users: Alice A and Bob B, each with device $D_{A,1}$ and $D_{B,1}$ respectively. Additionally, they are both registered to a malicious homeserver, whose aim is to compromise their out-of-band verification with the SAS protocol. The attack proceeds as follows:

1) When Alice A registers their account with the homeserver, the homeserver generates a parallel cross-signing identity with verification keys (mpk'_A, spk'_A, upk'_A).
2) When Alice A logs in for the first time, the homeserver sets the device identifier $D_{A,1} \leftarrow mpk'_A$.
3) When Bob B logs in for the first time, the homeserver sets $D_{B,1}$ as normal.
4) Alice and Bob each setup their own cross-signing identities with verification keys (mpk_A, spk_A, upk_A) and (mpk_B, spk_B, upk_B) respectively. They upload these to the homeserver.
5) The homeserver proceeds to present two versions of the cross-signing state:
   a) When Alice requests their own cross-signing information, they are presented with the version they uploaded (mpk_A, spk_A, upk_A).
   b) When Bob requests Alice's cross-signing information, they are presented with the version generated by the malicious homeserver (mpk'_A, spk'_A, upk'_A).
6) Alice and Bob perform an out-of-band verification using the SAS protocol. At the end, they exchange `m.key.verification.mac` messages containing their cryptographic identity (for signing). Figure 5 shows the structure of the message Alice sends. $D_{B,1}$ processes it as follows:
   a) SAS.VerifyMAC interprets the entry for mpk'_A as a request for device verification. It fetches the expected device identity key $apk_{A,1}$, then calculates a matching MAC. The device identity key pulled from the homeserver is legitimate, and matches the one used by Alice's device to

[8] "Verification methods can be used to verify a users master key by using the master public key, encoded using unpadded base64, as the device ID, and treating it as a normal device. For example, if Alice and Bob verify each other using SAS, Alices m.key.verification.mac message to Bob may include "ed25519:alices+master+public+key": "alices+master+public+key" in the mac property. Servers therefore must ensure that device IDs will not collide with cross-signing public keys."

generate the MAC. Thus, the message passes verification.

b) SAS.SignDevice interprets the entry for mpk'_A as a request for cross-signing verification. This is because the homeserver has led Bob's client to believe that Alice's cross-signing identity is mpk'_A.

7) Bob cross-signs the homeserver controlled identity for Alice, and uploads the signature to the homeserver to distribute to their other devices.

We implemented this attack and report that it succeeds in practice. We expect that this attack can be performed in parallel against Bob, such that Alice signs a homeserver controlled cross-signing identity for Bob (mpk'_B, spk'_B, upk'_B). From this point onwards, the homeserver can generate their own device identities. These device identities can create Olm connections with Alice as if they were a verified device of Bob, and vice versa. This results in a compromise of all Olm sessions between the two users (for which compromise of Megolm sessions follows).

C. Limitations

This attack exploits a specific issue with cross-signing between users' devices and does not work when a user is verifying two of their own devices. This means that a malicious homeserver cannot compromise Olm connections between devices of the same user.

D. Remediation

Alongside public disclosure, the Matrix developers will ensure that master cross-signing keys and device identifiers are not confused for one another. To do this, they plan to consistently process identifiers, checking whether they are key or device identifier in the same order throughout the codebase. We recommend that this order is formalised in the Matrix specification (with the security implications of a mistake explained).

The attack is made possible due to a lack of domain separation between device identifiers and cross-signing keys. In the long-term, the Matrix developers plan to separate the format for device identifiers and Ed25519 keys.

Additionally, we recommend that the use of device identifiers in `m.key.verification.mac` messages during self-verification is not necessary. Replacing these with the device's identity key would serve the same purpose, and reduce the need to include homeserver-controlled information during the processing of these messages.

V. Semi-trusted Impersonation against Megolm Authentication

In this section we describe an attack that allows a malicious device and homeserver to impersonate other (target) devices. The malicious device and the target devices may or may not belong to the same user. Whilst this attack causes a warning to be shown next to the message, messages sent using legitimately forwarded room keys display the same warning, thus this attack achieves the same level of trust as legitimately forwarded room keys. In Section VI-B we will build on this attack to achieve an impersonation that produces trusted messages.

A. Vulnerability

Clients using `matrix-js-sdk` will accept valid encrypted `m.forwarded_room_key` messages regardless of whether they requested the keys.[9] Further, whilst the key sharing restrictions mentioned in Section II-H are implemented on the sharing side, they are not implemented on the receiving side of the protocol: clients will accept forwarded room keys from any device as long as the message is encrypted.[10] This enables a malicious device to push Megolm sessions to other devices (potentially even overwriting a legitimate session with an illegitimate one).

Unlike sessions received via `m.room_key` messages, those received via `m.forwarded_room_key` messages do not have a cryptographic link between the session owner and the session. Instead, they have a cryptographic link between the forwarding party and the session. To track the trust level a device has in a particular Megolm session, the Key Request protocol associates the session with a list of Olm identity keys that have forwarded it (through the `forwarding_curve25519_key_chain` field of the message, which contains a list of Curve25519 keys through which this session was forwarded). Whenever a device receives a Megolm session through a `m.forwarded_room_key` message, they append the Olm identity key and long-term fingerprint key associated with it to its forwarding key chain. This allows for a key to be forwarded from, for example, Alice's first phone to a second one, and from the latter to a third, and so on. Provided the initial device trusts every other device in the list to honestly forward sessions, they can trust the Megolm session.

No such check is implemented.[11] When a message is decrypted by a Megolm session with a non-empty `forwarding_curve25519_key_chain` list, it is displayed with the same warning message regardless of which device forwarded it.

B. Attack

Consider a setting with three users: Alice A, Bob B and Claire C. Each user has a single logged-in device: $D_{A,1}$, $D_{B,1}$, and

[9] `MegolmDecryption.onRoomKeyEvent` in `matrix-js-sdk` (commit `#4721aa1`) does not check whether a matching room key request has been made before processing a `m.forwarded_room_key` message.

[10] `MegolmDecryption.onRoomKeyEvent` in `matrix-js-sdk` (commit `#4721aa1`) uses `event.senderKey` to ensure the message was encrypted. However, it will accept `m.forwarded_room_key` messages from any user or device.

[11] See `getEventEncryptionInfo` in `matrix-js-sdk` (`#4721aa1`).

$D_{C,1}$. Let G = `room_id` identify a room consisting of Alice and Bob's devices. To impersonate Bob $D_{B,1}$ to Alice $D_{A,1}$ in G, Claire $D_{C,1}$ can:

1) Generate a new Megolm session inbound/outbound pair (i, R, gpk), (i, R, gsk), for which $D_{C,1}$ keeps the outbound session.
2) Construct an `m.forwarded_room_key` event to share the Megolm session, and set `sender_key` and `sender_claimed_ed25519_key` to the Olm identity key and long-term fingerprint key of $D_{B,1}$ respectively.
3) Setup an Olm channel between Claire $D_{C,1}$ and Alice $D_{A,1}$. Use the channel to encrypt the `m.forwarded_room_key` message.
4) The ciphertext is wrapped with metadata and formatted as a to-device message, to be sent to Alice's device $D_{A,1}$.

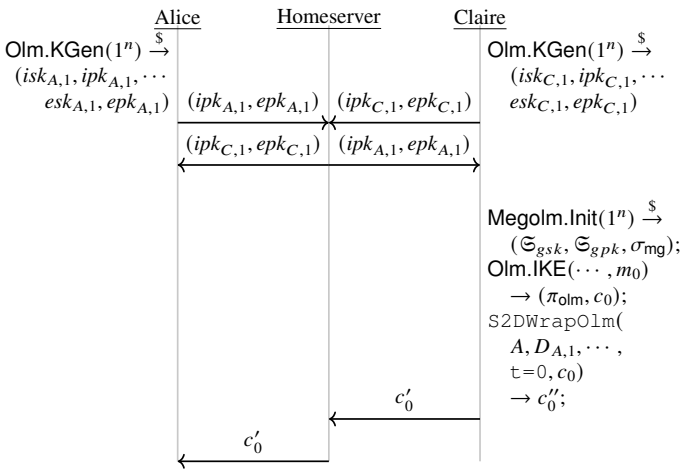


Fig. 6: The impersonation attack described in Section V-B. The function `S2DWrapOlm` wraps an Olm encrypted ciphertext with the appropriate metadata to be sent as a to-device message. We let m_0 be `m.forwarded_room_key`$(G, gpk, ver, i, R, ipk_{B,1}, apk_{B,1})$.

We give the attack flow in Fig. 6. When Alice's $D_{A,i}$ device receives this message, it will accept the forwarded key and store it associated with the Olm identity key $ipk_{B,1}$ and long-term fingerprint key $apk_{B,1}$ of $D_{B,1}$. Additionally, it will add the Olm identity key of $D_{C,1}$ to the forwarded key chain associated with this session.

At the end of this process, Claire's device $D_{C,1}$ knows the gsk of a Megolm session, i.e. knows the outbound session secrets for an inbound session, that Alice's device $D_{A,1}$ believes is owned by Bob's device $D_{B,1}$. Using this session, $D_{C,1}$ may send messages to $D_{A,1}$ that will be displayed in the user interface as having come from $D_{B,1}$.

When determining the user that sent a message, each device starts by looking at the `sender` field of the ciphertext wrapper. The device then ensures that the cryptographic identity used to send the message matches the value in this field. The `sender` field is set by the homeserver when a device uploads a new message, using the user identifier associated with that device. This attack, therefore, requires a colluding homeserver to forge the `sender` field of Megolm messages.

The messages will be semi-trusted, displayed with a warning message. However, as mentioned above, this message is also displayed alongside messages decrypted with legitimately forwarded Megolm sessions. We implemented this attack and report that is succeeds in practice.

C. Remediation

At the time of public disclosure, clients will only process `m.forwarded_room_key` messages in response to previously issued requests. Since client's correctly request keys from trusted device's only, this aims to ensure that they only accept them from trusted devices, as well.[12]

In the long-term, the Matrix developers plan to discontinue the use of `m.forwarded_room_key` messages, replacing all instances with `m.room_key`. This will ensure that the inbound Megolm session being shared is always bound to gsk with a signature.

Further, clients will further restrict the devices they will accept forwarded room keys from to only those beging forwarded by verified devices of the same user. This is an additional restriction over the current policy, whereby the owner of the session may also share it regardless of their verification status.

VI. Trusted Impersonation and Confidentiality Breaks against Megolm

In this section we describe two attacks that, together, compromise the authenticity and confidentiality of Megolm channels. In Section II, we described how pairwise Olm channels are used by the Megolm and Key Request protocols, as well as the SSSS modules's secret sharing functionality. These protocols maintain their security by relying on the Olm protocol and its connection to user's cross-signing identities. However, a lack of verification in the implementation allows such messages to also be sent over Megolm. The following two attacks send protocol messages intended to be sent over Olm channels, over semi-trusted Megolm sessions instead (using the attack described in Section V). In doing so, the attacker is able to impersonate a trusted device in order to place secrets on the target device.

A. Vulnerability

1) Protocol confusion: The Matrix and Megolm specifications requires that `m.room_key` and `m.secret.send` messages are sent over encrypted Olm channels. Whilst

[12]Matrix optionally allows sharing room history with new members. This is implemented by sharing previous inbound Megolm sessions with the new member. The fix described will not apply to such messages, though the resulting sessions will be marked as untrusted in the code and user interface. The Matrix developers are working on a long-term countermeasure with stronger guarantees.

```
{"messages": {
  "<receiver_user_id>": {
   "<receiver_device_id>": {
    "algorithm": "m.megolm.v1.aes-sha2",
    "sender_key": sender_ipk, "session_id": gpk,
    "room_id": room_id, "ciphertext": ciphertext}}}
```

(a) Message sent by the device.

```
{"type": "m.room.encrypted",
 "sender": sender_user_id
 "content": {
  "algorithm": "m.megolm.v1.aes-sha2",
  "sender_key": sender_ipk, "session_id": gpk,
  "room_id": room_id, "ciphertext": ciphertext}}
```

(b) Message as forwarded by homeserver.

Fig. 7: Figure 7a shows the format of a Megolm encrypted to-device message as it is sent to the homeserver (the message type, `m.room.encrypted`, is encoded in the URL). The homeserver will split up to-device messages, collate them by device, then redistribute them as a list of messages in the format seen in Figure 7b. Synapse does not preserve the `room_id` field of messages when converting between the two formats shown here, and thus requires a colluding homeserver to enable the protocol confusion. The `sender` field is added by the homeserver and, similarly, requires collusion from the homeserver to set it to the attacker's desired value. `sender_key` and `session_id` are used by the receiving device to locate the Megolm session with which to decrypt the ciphertext.

the specifications do not include this requirement for `m.forwarded_room_key` messages, we believe this is the intended behaviour. These messages are used for sending key material in the Megolm, Key Request, SSSS secret sharing protocols (respectively). However, whilst the handler for these incoming messages ensures they have been encrypted, it does not explicitly check which algorithm they were encrypted with. It is therefore possible to encrypt `m.room_key` messages using Megolm rather than Olm, provided they are distributed via to-device messaging (cf. Fig. 7).

2) Inheriting sender identity: When Megolm messages are decrypted, they inherit the sender identity associated with the Megolm session used for decryption. Similarly, when Megolm sessions are received through an `m.room_key` message, they inherit the sender identity of the encrypted channel they were sent over. In the expected case, this will be the sender identity of the Olm channel it was sent over. When sending `m.room_key` messages over a Megolm session, it will inherit the sender identity that the Megolm session inherited (inherited from the encrypted channel where it was sent).

B. Trusted Impersonation Attack against Megolm Authentication

We describe an attack that allows a malicious device and colluding homeserver to impersonate any device. The attacker uses the forwarded Megolm session from the attack in Section V to deliver a second Megolm session to the target device that is indistinguishable from a legitimate session (as sent via an `m.room_key` message). No warning is shown alongside messages encrypted with the second Megolm session, i.e. its messages are trusted.

The inheritance strategy described above (Section VI-A2) maintains the cryptographic link between the Olm channel first used to send a Megolm session, and any subsequent Megolm sessions sent over it.

However, `m.forwarded_room_key` messages allow attackers to insert Megolm sessions with an associated sender identity that does not have this (or any) cryptographic link. If the adversary then sends an `m.room_key` message over the forwarded Megolm session, this latest session inherits the presumed sender identity, regardless of whether that sender identity has been cryptographically verified or not. Thus, such a message "upgrades" the presumed validation of the key material from unknown to verified.

The attack starts after the attack in Section V-B has concluded, where Claire's device $D_{C,k}$ has forwarded the Megolm session $\mathfrak{S}_{gpk}$ to Alice's device $D_{A,i}$ in order to impersonate Bob's device $D_{B,j}$. Claire's device $D_{C,k}$ proceeds as follows:

1) Generate a new Megolm session pair, $\mathfrak{S}'_{gpk}$, $\mathfrak{S}'_{gsk}$, for which $D_{C,k}$ keeps the outbound session: $\mathfrak{S}'_{gsk}$.
2) Construct an `m.room_key` message (Fig. 9a) to share the inbound session $\mathfrak{S}'_{gpk}$.
3) Encrypt the `m.room_key` message using the previously constructed outbound Megolm session, $\mathfrak{S}_{gsk}$, from Section V-B.
4) Construct a Megolm encrypted, to-device message wrapper (Fig. 9b) to distribute the ciphertext from the previous step to Alice's device $D_{A,i}$.

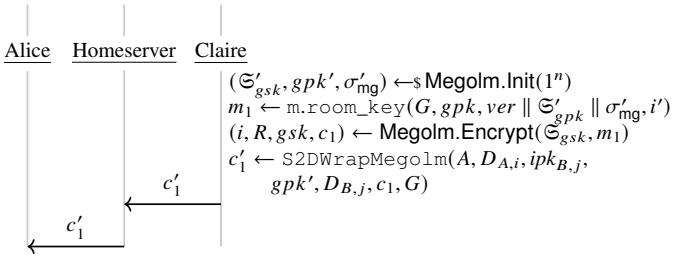


Fig. 8: The impersonation attack described Section VI-B. This diagram continues on from the sequence diagram in Fig. 6, which itself describes the attack in Section V-B. The function `m.room_key` generates a message of the same name (Fig. 9a describes its structure). The function `S2DWrapMegolm` wraps a Megolm encrypted ciphertext with the appropriate metadata to be sent as a to-device message (Fig. 9b describes its structure).

We illustrate this attack in Fig. 8. When Alice's device $D_{A,i}$ receives the second Megolm session $\mathfrak{S}'_{gpk}$, the new session

will be saved as if it were sent via an Olm channel with Bob's device $D_{B,i}$ sender identity.

Claire's device $D_{C,k}$ now has control over gsk' of an outbound Megolm session that Alice's device $D_{A,i}$ believes is owned by Bob's device $D_{B,j}$. In contrast with the attack in Section V-B, there is no evidence presented to the receiver that this Megolm session was forwarded.

$D_{C,k}$ may send messages to $D_{A,i}$ using this session that will be displayed in the user interface as coming from $D_{B,j}$ (without any warnings in the user interface). As before, this requires collaboration with the homeserver to forge the `sender` field of the Megolm messages. We implemented this attack and report that it succeeds in practice.

C. Adversary Controlled Megolm Backup Key

We now describe an attack whereby a malicious homeserver is able to set the secret key used by target devices when encrypting inbound Megolm sessions for backup on the homeserver. This enables the homeserver decrypt the backups, compromising the confidentiality of messages sent in those sessions.

In brief, the attack proceeds as follows. When newly verified devices request a copy of the Megolm backup key through SSSS secret sharing, the homeserver impersonates a trusted device (using the Olm/Megolm protocol confusion) and responds with an attacker-controlled backup key. From this point onwards, the target device will backup their inbound Megolm sessions to the homeserver, encrypted with a homeserver controlled key. Thus, this attack extends the authentication attack of Section V-B to additionally break confidentiality.

It is expected that `m.secret.send` messages are encrypted with the Olm protocol; however, as in Section VI-B, it is possible to send a to-device `m.secret.send` message encrypted with Megolm. When the receiving client decrypts the message, it will inherit the Olm identity key ipk associated with the Megolm session.[13]

In this attack, the adversary generates and sends a Megolm session using a `m.forwarded_room_key` message with the claimed Olm device identity of a verified device belonging to the same user as the target device. The adversary then uses this Megolm session to send a `m.secret.send` with secrets they control.

Consider a setting with a homeserver H that controls a colluding user account Bob B and device $D_{B,1}$. The target user Alice A, has two devices $D_{A,1}$ and $D_{A,2}$. $D_{A,1}$ is the first of Alice's devices, and thus possesses both the cross-signing secrets (msk_A, usk_A, ssk_A) and a key for server-side Megolm backups. Alice's second device, $D_{A,2}$, is the result of a recent login from a new client. Whilst it has received a copy of Alice's public cross-signing identity through the homeserver, it does not yet have access to the cross-signing secrets. The attack proceeds as follows:

1) H generates a symmetric key $k_H \leftarrow\$ \{0,1\}^{256}$. It constructs a backup configuration signifying the use of symmetric server-side key backups[14] with key k_H (the key is not signed). The homeserver will present this backup configuration to Alice's second device, $D_{A,2}$.
2) Due to the lack of signature, $D_{A,2}$ will not trust this key and will not enable backups.
3) Alice completes out-of-band verification between $D_{A,1}$ and $D_{A,2}$.
4) $D_{A,2}$ sends an `m.secret.request` message to $D_{A,1}$ requesting the key for server-side Megolm backups.
5) H does not distribute this request message to $D_{A,1}$.
6) H and $D_{B,1}$ perform the attack in Section V-B against $D_{A,2}$, giving them control over a Megolm session that $D_{A,2}$ believes originally came from $D_{A,1}$.
7) H uses this Megolm session to respond to $D_{A,2}$'s request, sending an `m.secret.send` message containing the attacker-controlled k_H.
8) $D_{A,2}$ receives and decrypts the message, storing k_H as the secret key for server-side Megolm backups.
9) $D_{A,2}$ will now trust the homeserver's backup configuration, since $D_{A,2}$'s private Megolm backup key matches the key identifier in the configuration.
10) $D_{A,2}$ will enable server-side Megolm backups, uploading inbound Megolm sessions to the homeserver H (encrypted using the homeserver-controlled key k_H).

D. Remediation

At the time of public disclosure, clients will ensure that encrypted to-device messages use the Olm protocol only. Such a fix should prevent the exploited protocol confusion.

However, we remark that this protocol confusion is aided by the overall layout of the Matrix SDK where cryptographic functionality is spread across different sub-libraries rather than being contained in a small, easily auditable core.

In particular, Element currently implements verification of message authentication not at decryption time but at display time. However, for messages that are never displayed but silently affect the state of the client this means that no verification of the sender is ever triggered. This makes it impossible for clients to establish the level of trust they have in different parts of their state.

VII. Discussion

We presented six attacks that together invalidate the fundamental security promises made by Matrix' end-to-end encryption against a malicious server. In particular, the version of Matrix as implemented in Element and analysed here neither provides confidentiality nor authentication against such an attacker.

[13] `getEventEncryptionInfo` in `matrix-js-sdk` (commit `#4721aa1`) uses `event.senderKey` both to ensure the message was encrypted and to check the identity of the sender.

[14] We tested this attack against the symmetric backup scheme, but we expect it to also work against the asymmetric scheme.

```
{"type": "m.room_key",
 "room_id": room_id,
 "content": {
  "algorithm": "m.megolm.v1.aes-sha2",
  "room_id": room_id, "session_id": gpk',
  "session_key":
   ver || i' || R' || gpk' || \
   sign(gsk', ver || i' || R' || gpk'),
  "chain_index": i'}}
```

(a) Plaintext message.

```
{"messages": {
  "<alice_user_id>": {
   "<alice_device_id>": {
    "algorithm": "m.megolm.v1.aes-sha2",
    "sender_key": bob_ipk,
    "session_id": gpk,
    "device_id": bob_device_id,
    "ciphertext": ciphertext,
    "room_id": room_id}}}}
```

(b) To-device message wrapper.

Fig. 9: This message allows Claire $D_{A,i}$ to impersonate Bob $D_{B,j}$ to Alice $D_{A,i}$. It works by placing a Megolm session into Alices device using an existing Megolm session that is already associated with Bobs sender identity. The plaintext in Figure 9a is encrypted using the forwarded Megolms session from the previous attack $\mathfrak{S}_{gpk}$. In Figure 9b, the session identifier is gpk to match the session used for encryption, not the session we are sending. This allows the second Megolm session to be sent using an `m.room_key` message, without being marked as a forwarded session.

On the one hand, some of our attacks highlight a rich attack surface by "chaining" different attacks to achieve their goals. In particular, we compose (1) the "weak" authentication break in Section V which exploits missing verifications, (2) a stronger authentication break in Section VI-B which exploits a protocol confusion aided by the design choice to check cryptographic properties at display rather than receiving time, and (3) a MITM attack that breaks confidentiality by convincing a target to use an adversary controlled key as backup in Section VI-C. On the other hand, our attacks are well distributed across the different parts of the overall cryptographic core of the Matrix protocol and implementation. In particular, we show (1) that Matrix offers no confidentiality guarantees by design against a malicious homeserver which may trivially add new users and devices to a room in Section III, (2) that an identifier confusion in a separate protocol allows to break authentication and thus confidentiality even for the lowest level Olm channels in Section IV, and (3) that the key backup scheme in yet another subsystem does not achieve formal IND-CCA security in Appendix A.

Besides the observed implementation and specification errors, these vulnerabilities highlight a lack of a unified and formal approach to security guarantees in Matrix. Rather, the specification and implementations seem to have grown "organically" with new sub-protocols adding new functionalities and thus inadvertently subverting the security guarantees of the core protocol. This suggests that, besides fixing the specific vulnerabilities reported here, Matrix/Megolm will need to receive a formal security analysis to establish confidence in the design.

We finish by reiterating a point already made in the introduction, namely that our attacks are against a setting where Matrix aims to provide the strongest guarantees, i.e. where every device and user have performed out-of-band verification. If this condition is not satisfied, even for one device or user, then "all bets are off" and e.g. impersonation becomes trivial. While Element already supports the option of refusing to send messages to unverified devices it does not reject messages from such devices. Thus, unless the client-side option is provided to reject all communication from unverified devices or rooms with such devices within them, Matrix will not provide a secure chat environment regardless of cryptographic guarantees provided for verified devices.

Acknowledgements

The work of Jones was supported by the EPSRC and the UK Government as part of the Centre for Doctoral Training in Cyber Security for the Everyday at Royal Holloway, University of London (EP/S021817/1).

References

[1] The Matrix.org Foundation, "Client-Server API (unstable)," May 2021. [Online]. Available: https://spec.matrix.org/unstable/client-server-api/
[2] ——, "Olm: A Cryptographic Ratchet." [Online]. Available: https://gitlab.matrix.org/matrix-org/olm/-/raw/master/docs/olm.md
[3] ——, "Megolm group ratchet." [Online]. Available: https://gitlab.matrix.org/matrix-org/olm/-/raw/master/docs/megolm.md
[4] M. Marlinspike, "Simplifying OTR deniability." Jul. 2013. [Online]. Available: https://signal.org/blog/simplifying-otr-deniability/
[5] ——, "The X3DH Key Agreement Protocol," Nov. 2016, revision 1.
[6] ——, "The Double Ratchet Algorithm," Nov. 2016. [Online]. Available: https://signal.org/docs/specifications/doubleratchet/
[7] J. Meredith and A. Balducci, "Matrix Olm Cryptographic Review," NCC Group, Tech. Rep., Nov. 2016, version 2.0.
[8] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, Sep. 2012.
[9] D. Wong, *Real-world Cryptography*. Manning Publications., 2021.
[10] M. Hodgson, "Independent public audit of Vodozemac, a native rust reference implementation of Matrix end-to-end encryption." [Online]. Available: https://matrix.org/blog/2022/05/16/independent-public-audit-of-vodozemac-a-native-rust-reference-implementation-of-matrix-end-to-end-encryption
[11] Anna Kaplan, Ann-Christine Kycler, Denis Kolegov, Jan Winkelmann, and Rai Yang, "Vodozemac Security Audit Report," Least Authority, Tech. Rep., Mar. 2022. [Online]. Available: https://matrix.org/media/Least%20Authority%20-%20Matrix%20vodozemac%20Final%20Audit%20Report.pdf
[12] T. Frosch, C. Mainka, C. Bader, F. Bergsma, J. Schwenk, and T. Holz, "How secure is TextSecure?" in *IEEE European Symposium on Security and Privacy, EuroS&P 2016*, 2016, pp. 457–472.
[13] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A formal security analysis of the signal messaging protocol," *Journal of Cryptology*, vol. 33, no. 4, pp. 1914–1983, Oct. 2020.

[14] J. Alwen, S. Coretti, and Y. Dodis, "The double ratchet: Security notions, proofs, and modularization for the Signal protocol," in *Advances in Cryptology – EUROCRYPT 2019, Part I*, ser. Lecture Notes in Computer Science, Y. Ishai and V. Rijmen, Eds., vol. 11476. Darmstadt, Germany: Springer, Heidelberg, Germany, May 19–23, 2019, pp. 129–158.
[15] M. Marlinspike, "Private Group Messaging," May 2014. [Online]. Available: https://signal.org/blog/private-groups/
[16] P. Rösler, C. Mainka, and J. Schwenk, "More is less: On the end-to-end security of group chats in signal, whatsapp, and threema," in *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*. IEEE, 2018, pp. 415–429. [Online]. Available: https://doi.org/10.1109/EuroSP.2018.00036
[17] The Matrix.org Foundation, "A new hope: matrix-rust-sdk." [Online]. Available: https://matrix.org/blog/2021/12/22/the-mega-matrix-holiday-special-2021#a-new-hope-matrix-rust-sdk
[18] ——, "Clients Matrix." [Online]. Available: https://matrix.org/clients-matrix/
[19] "Secure hash standard," National Institute of Standards and Technology, NIST FIPS PUB 180-2, U.S. Department of Commerce, Aug. 2002.
[20] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," IETF Internet Request for Comments 2104, Feb. 1997.
[21] ——, "RFC 2104: HMAC: keyed-hashing for message authentication," 1997. [Online]. Available: https://doi.org/10.17487/RFC2104
[22] H. Krawczyk, "Cryptographic extraction and key derivation: The HKDF scheme," in *Advances in Cryptology – CRYPTO 2010*, ser. Lecture Notes in Computer Science, T. Rabin, Ed., vol. 6223. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 15–19, 2010, pp. 631–648.
[23] H. Krawczyk and P. Eronen, "RFC 5869: HMAC-based Extract-and-Expand Key Derivation Function (HKDF)," 2010. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5869
[24] "Advanced Encryption Standard (AES)," National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, Nov. 2001.
[25] M. Dworkin, "Recommendation for Block Cipher Modes of Operation: Methods and Techniques," National Institute of Standards and Technology, Tech. Rep. NIST Special Publication (SP) 800-38A, Dec. 2001. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-38a/final
[26] R. Housley, "RFC 5652: Cryptographic Message Syntax (CMS)," Sep. 2009. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5652
[27] The Matrix.org Foundation, "MSC2732: Olm fallback keys," Jun. 2021. [Online]. Available: https://github.com/matrix-org/matrix-spec-proposals/pull/2732
[28] ——, "Implementing more advanced e2ee features, such as cross-signing," 2021. [Online]. Available: https://matrix.org/docs/guides/implementing-more-advanced-e-2-ee-features-such-as-cross-signing
[29] P. Zimmermann, E. A. Johnston, and J. Callas, "RTP: Media Path Key Agreement for Unicast Secure RTP," Apr. 2011. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6189
[30] D. J. Bernstein, "Curve25519: New Diffie-Hellman speed records," in *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*, ser. Lecture Notes in Computer Science, M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., vol. 3958. New York, NY, USA: Springer, Heidelberg, Germany, Apr. 24–26, 2006, pp. 207–228.
[31] The Matrix.org Foundation, "MSC3270: Symmetric megolm backup," Jul. 2021. [Online]. Available: https://github.com/matrix-org/matrix-spec-proposals/pull/3270
[32] M. Chase, T. Perrin, and G. Zaverucha, "The signal private group system and anonymous credentials supporting efficient verifiable encryption," in *ACM CCS 2020: 27th Conference on Computer and Communications Security*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds. Virtual Event, USA: ACM Press, Nov. 9–13, 2020, pp. 1445–1459.

Appendix

A. IND-CCA Attack on Backups

Here we break the formal IND-CCA security of symmetric Megolm Key Backups and the Secure Secret Storage and Sharing protocol.

Megolm.Init(1^n)

$i \leftarrow 0$
$R \leftarrow \{0,1\}^{1024}$
$(gsk, gpk) \leftarrow \mathsf{Ed25519.KGen}(1^n)$
$\mathsf{ver} \leftarrow \texttt{0x03}$
$\sigma_{\mathsf{mg}} \leftarrow \mathsf{Ed25519.Sign}(gsk, (\mathsf{ver}, i, R, gpk))$
$\mathfrak{S}_{gsk} \leftarrow (\mathsf{ver}, i, R, gsk)$
$\mathfrak{S}_{gpk} \leftarrow (\mathsf{ver}, i, R, gpk)$
return $\mathfrak{S}_{gsk}, \mathfrak{S}_{gpk}, \sigma_{\mathsf{mg}}$

Megolm.Encrypt$(\mathfrak{S}_{gsk}, m)$

$(\mathsf{ver}, i, R, gsk) \leftarrow \mathfrak{S}_{gsk}$
$k_e \parallel k_h \parallel k_{iv} \leftarrow \mathsf{HKDF}(0, R, \text{“}\texttt{MEGOLM\_KEYS}\text{”}, 80)$
$c \leftarrow \mathsf{AES\text{-}CBC}(k_e, k_{iv}, m)$
$\tau \leftarrow \mathsf{HMAC}(k_h, (\mathsf{ver}, i, c))[0:8]$
$\sigma \leftarrow \mathsf{Ed25519.Sign}(gsk, (\mathsf{ver}, i, c, \tau))$
$c' \leftarrow (\mathsf{ver}, i, c, \tau, \sigma)$
$i, R \leftarrow \mathsf{MegolmRatchet.Advance}(i, R)$
$\mathfrak{S}_{gsk} \leftarrow (\mathsf{ver}, i, R, gsk)$
return $(\mathfrak{S}_{gsk}, c')$

Megolm.Decrypt$(\mathfrak{S}_{gpk}, c)$

$(\mathsf{ver}, i, R, gpk) \leftarrow \mathfrak{S}_{gpk}$
$(\mathsf{ver}', i', c', \tau, \sigma) \leftarrow c$
if $!\mathsf{Ed25519.Verify}(gpk, \sigma, (\mathsf{ver}, i', c', \tau))$ **then**
  return $(\mathfrak{S}_{gpk}, \perp)$
$(i, R) \leftarrow \mathsf{MegolmRatchet.Advance}^{i'-i}(i, R)$
$k_e \parallel k_h \parallel k_{iv} \leftarrow \mathsf{HKDF}(0, R', \text{“}\texttt{MEGOLM\_KEYS}\text{”}, 80)$
if $\tau \neq \mathsf{HMAC}(k_h, (\mathsf{ver}, i, c')[0:8]$ **then**
  return $(\mathfrak{S}_{gpk}, \perp)$
$m \leftarrow \mathsf{AES\text{-}CBC.Dec}(k_e, k_{iv}, c')$
$\mathfrak{S}_{gpk} \leftarrow (\mathsf{ver}, i, R, gpk)$
return $(\mathfrak{S}_{gpk}, m)$

Fig. 10: The Megolm protocol consists of three algorithms, Megolm = (Megolm.Init, Megolm.Encrypt, Megolm.Decrypt). The $\mathsf{MegolmRatchet.Advance}(i, R)$ algorithm takes the Megolm ratchet R and index i, and symmetrically advances it.

1) Vulnerability: Matrix uses an encrypt-then-MAC encryption scheme composing AES-CTR with HMAC-SHA-256. Recall that AES-CTR proceeds by encrypting a series of blocks $iv, iv \oplus 1, iv \oplus 2, \ldots$ and XORing the result onto the message blocks m_i to produce the ciphertext blocks c_i. The full ciphertext is $iv \| c_0 \| c_1 \| \ldots$. However, the iv is not covered by the authentication tag produced by HMAC-SHA-256.[15]

[15] A similar issue exists for attachments which are shared out of band in encrypted form [1, #sending-encrypted-attachments]. Here the hash shared over Megolm (which takes the role of the MAC) does not include the iv. Since the iv itself is also shared over Megolm and thus implicitly authenticated, we do not see a way to exploit this behaviour.

2) Attack: The IND-CCA attack proceeds as follows: Let $c*$ be some challenge ciphertext for either some message m_0 or m_1 of length 128 bits:

$$c* := iv \| \mathsf{AES}(k_0, iv) \oplus m_b \| \mathsf{HMAC}(k_1, \mathsf{AES}(k_0, iv) \oplus m_b).$$

The adversary requests an encryption of zero from the encryption oracle and receives

$$c := iv' \| \mathsf{AES}(k_0, iv') \oplus 0 \| \mathsf{HMAC}\left(k_1, \mathsf{AES}(k_0, iv') \oplus 0\right)$$

for some iv'. Finally, the adversary requests a decryption of

$$c^+ := iv \| \mathsf{AES}(k_0, iv') \oplus 0 \| \mathsf{HMAC}\left(k_1, \mathsf{AES}(k_0, iv') \oplus 0\right)$$

from the decryption oracle. Note that the MAC verifies correctly, and so the adversary will receive '$t := \mathsf{AES}(k_0, iv') \oplus \mathsf{AES}(k_0, iv) \oplus 0$. Given that the adversary already knows $\mathsf{AES}(k_0, iv')$ it can now recover '$\mathsf{AES}(k_0, iv) = t \oplus \mathsf{AES}(k_0, iv')$' and thus decrypt the challenge $c*$.

3) Limitations: In principle, in the Matrix setting, forwarding keys to the target and observing the resulting backups on the homeserver provides an encryption oracle. Similarly, modifying a backup on the homeserver then requesting the resulting key provides a decryption oracle.

However, in practice, this attack is not exploitable as far as we can see for two reasons. First, the use of per-session keys means the scope of the decryption oracle is limited to the per-session key of sessions the attacker already has access to. Second, any modified ciphertexts will likely be invalid JSON structures and fail to parse correctly (preventing the decryption oracle from sharing the plaintext with the adversary).

4) Remediation: The Matrix developers plan to include the iv alongside the ciphertext when calculating the MAC. Similarly, clients will include the iv of encrypted attachments inside Megolm ciphertexts alongside their SHA-256 and encryption key. Since this issue is not currently practically exploitable in Matrix, these fixes will not be made available at the time of disclosure.

B. Attacks enabled by the Asymmetric Megolm Backup scheme

Since this scheme does not authenticate the party that creates the backups, it provides an alternative means to perform a semi-trusted impersonation attack. In this attack, a malicious homeserver can generate Megolm sessions under their control, encrypt them for the recovery key and share them with a target user as backups [31].

For this reason, sessions acquired through asymmetric server-side backups are marked as untrusted internally, and messages decrypted using such sessions are accompanied with a warning. This issue is inherent to the design of the scheme and is a known issue to the protocol designers [31], motivating the development of the symmetric scheme described in Section II-I2.

Nonetheless, we note that this attack provides an alternative first step in performing the attacks described in Section VI.

C. Megolm Protocol

We give the main algorithms used in Megolm in Fig. 10.

D. Short Authentication String (SAS) Protocol

Figures 11 and 12 describe the SAS protocol for out-of-band verification. When clients source keys from their homeserver, we represent this through a call to the algorithm HS.QueryKey. It takes as input a string representing the key type, followed by a series of indices to identify the particular key. For example, HS.QueryKey("apk", A, i) returns $apk_{A,i}$.

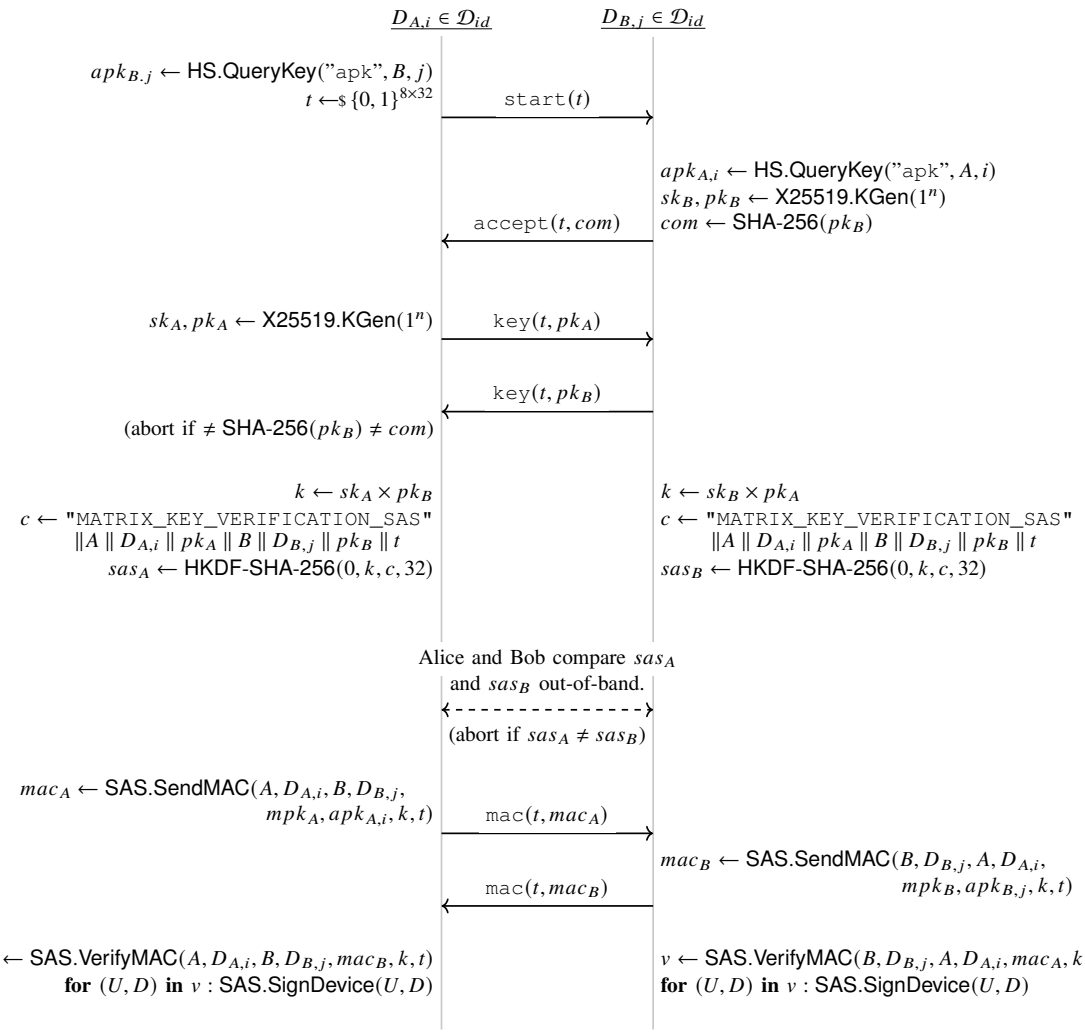


Fig. 11: A sequence diagram summarising the SAS protocol. The contents of `m.key.verification.mac` are described in Fig. 4, while pseudocode descriptions of SAS.SendMAC, SAS.VerifyMAC and SAS.SignDevice are in Fig. 12. Message types in the above diagram have had the prefix `m.key.verification.` removed.

SAS.CalcMAC(k, m, c)

$k' \leftarrow$ HKDF-SHA-256$(0, k, c, 32)$
$mac \leftarrow$ HMAC-SHA-256(k', m)
return mac

SAS.SendMAC$(U, D, U', D', mpk, apk, k, t)$

$c \leftarrow$ "MATRIX_KEY_VERIFICATION_SAS" $\backslash$
$\quad \| U \| D \| U' \| D' \| t$
$id_{\text{dev}} \leftarrow$ "ed25519:" $\| D$
$mac_{\text{dev}} \leftarrow$ SAS.CalcMAC$(k, apk, c \| id_{\text{dev}})$
$id_{\text{cs}} \leftarrow$ "ed25519:" $\| mpk$
$mac_{\text{cs}} \leftarrow$ SAS.CalcMAC$(k, mpk, c \| id_{\text{cs}})$
$ms \leftarrow ((id_{\text{dev}}, mac_{\text{dev}}), (id_{\text{cs}}, mac_{\text{cs}}))$
$ks \leftarrow$ SAS.CalcMAC$(k,$ sort$(id_{\text{dev}}, id_{\text{cs}}), c \|$ "KEY_IDS")
return (ms, ks)

SAS.SignDevice(U, D)

/ Check whether D is a cross-signing identity
$mpk \leftarrow$ HS.QueryKey("mpk", U)
if $D = mpk$ **then**
  return UserVerified(U, mpk)
/ Otherwise, D refers to a device
else
  return DeviceVerified(U, D)

SAS.VerifyMAC$(U, D, U', D', mac, k, t)$

$((id_{\text{dev}}, mac_{\text{dev}}), (id_{\text{cs}}, mac_{\text{cs}}), ks) \leftarrow mac$
$c \leftarrow$ "MATRIX_KEY_VERIFICATION_SAS" $\backslash$
$\quad \| U' \| D' \| U \| D \| t$
$ks' \leftarrow$ SAS.CalcMAC$(k,$ sort$(id_{\text{dev}}, id_{\text{cs}}), c \|$ "KEY_IDS")
if $(ks' \neq ks)$ **then**
  return $\emptyset$
$v \leftarrow \emptyset$
for (id, mac) **in** $((id_{\text{dev}}, mac_{\text{dev}}), (id_{\text{cs}}, mac_{\text{cs}}))$
  "ed25519:" $\| D' \leftarrow id$
  / Check if this is a device verification request
  $apk \leftarrow$ HS.QueryKey("apk", U', D')
  if $apk \neq \perp$ **then**
    $D'' \leftarrow x$
    if $mac =$ SAS.CalcMAC$(k, apk, c \| id)$ **then**
      $v \leftarrow v \cup \{(U', D'')\}$
  / Check if this is a cross-signing verification request
  elseif $(x =$ HS.QueryKey("mpk", $U)$
      $\cap\, mac =$ SAS.CalcMAC$(k, x, c \| id))$ **then**
    $mpk \leftarrow x$
    $v \leftarrow v \cup \{(U', mpk)\}$
return v

Fig. 12: Algorithms to generate, verify and process `m.key.verification.mac` messages in the SAS protocol. UserVerified signs the given user's master cross-signing key with the current device's user-signing key. Similarly, DeviceVerified signs the given device's fingerprint and Olm identity keys with the current device's self-signing key. These signatures are uploaded to the homeserver.