Device-Oriented Group Messaging: A Formal Cryptographic Analysis of Matrix' Core

Martin R. Albrecht, Benjamin Dowling† and Daniel Jones‡*
**King's College London, martin.albrecht@kcl.ac.uk*
† *Security of Advanced Systems Group, University of Sheffield, b.dowling@sheffield.ac.uk*
‡*Information Security Group, Royal Holloway, University of London, dan.jones@rhul.ac.uk*

***Abstract*—Focusing on its cryptographic core, we provide the first formal description of the Matrix secure group messaging protocol. Observing that no existing secure messaging model in the literature captures the relationships (and *shared state*) between users, their devices and the groups they are a part of, we introduce the Device-Oriented Group Messaging model to capture these key characteristics of the Matrix protocol. Utilising our new formalism, we determine that Matrix achieves the basic security notions of confidentiality and authentication, provided it introduces authenticated group membership. On the other hand, while the state sharing functionality in Matrix conflicts with advanced security notions in the literature – forward and post-compromise security – it enables features such as history sharing and account recovery, provoking broader questions about how such security notions should be conceptualised.**

1. Introduction

Matrix [1] is an open standard for interoperable, federated, real-time communication over the Internet. It consists of a number of specifications which, together, define a federated secure group messaging (SGM) protocol enabling clients, with accounts on different Matrix servers, to exchange messages.

Matrix has seen wide adoption across the private and public sectors. It is used in operation (to varying degrees) by governmental organisations in Germany, France, Sweden, and Luxembourg. In particular, the German ministry of defence and healthcare system both use Matrix in the field. Matrix is a popular choice within the free and open-source software (FOSS) ecosystem, with Mozilla, KDE and the FOSDEM 2022 conference all using it. A number of existing applications have developed Matrix integrations including the email client Mozilla Thunderbird, the forum software Discourse and the enterprise messaging platform Rocket.Chat. Overall, Matrix reportedly has over 80 million users. The IETF's More Instant Messaging Interoperability working group is currently discussing Matrix in their work specifying a set of mechanisms to make messaging applications interoperable.

The Matrix specification enables end-to-end encryption by default. While Matrix uses Transport Layer Security (TLS) to secure communication between clients and servers (and between servers for federation), *end-to-end encryption* is realised using a bespoke cryptographic protocol called Megolm which extends the pairwise protocol Olm to support group chat. Every chat in Matrix is a group chat, including 1-on-1 chats. Thus, the study of its group messaging protocol is central to understanding its security guarantees.

1.1. Prior Work

Cryptanalysis. An audit of the Olm and Megolm protocols (along with their implementations) was performed by NCC Group in 2016 [2]; this audit found a number of security issues that have since been fixed or recorded as limitations [3], [4]. Since then, several further cryptographic vulnerabilities have been reported, e.g. in CVE-2021-34813, CVE-2021-40824 and [5, Chapter 11]. Moreover, several practically exploitable vulnerabilities in both the Matrix specification and the flagship client *Element* were recently reported [6]. The vulnerabilities found varied in their nature and were distributed broadly across the subprotocols and libraries that make up the cryptographic core of Matrix and Element. In 2022, Matrix started a series of audits of their (future) core libraries [7], [8]. However, these vulnerabilities highlight the need for a formal and thus rigorous cryptographic analysis of the Matrix protocol, going beyond audits.

Formal analysis. Both Olm and Megolm build on existing work, with Olm being a modified implementation of the Signal protocol [4], [9] and Megolm sharing its architecture with the *Sender Keys* variant of Signal [10]. As such, existing analysis of these protocols will be relevant. The Signal protocol has received multiple analyses over the years, e.g. [11], [12], [13], while existing analysis of Sender Keys based protocols is sparse [14]. In concurrent and independent work, Balbás, Collins and Gajland provide a security analysis and proof for Sender Keys as it is used in WhatsApp [15].[1] Nonetheless, no prior work currently provides a security analysis and proof for Olm, Megolm or their composition.

The majority of works that examine group messaging protocols focus on the underlying group key exchange

1. A preliminary version appeared as [16].

(GKE) protocol, as this captures the cryptographic core of secure group messaging protocols, see [17], [18], [19], [20], [21], and for a systematisation of knowledge (SoK) of this area see [22]. Overall, [23] comes closest to our work, introducing SGM as an independent primitive and a model to assess its security. However, this model is insufficient for analysing subtle interactions in Matrix, as its design distinguishes between secrets known by *different devices owned by the same user* whilst allowing these secrets to be shared among those devices according to certain policies. Previous models for group messaging such as [14], [23], do not distinguish between users and their devices. For an SoK on multi-device secure messaging protocols (including high-level descriptions of a number of Matrix' key features) in real-world protocols, see [24].

1.2. Contributions

Recent vulnerabilities [6] in Matrix motivate a comprehensive formal analysis of the Matrix protocol.

Contribution 1. *Section 2 gives the first formal description of the Matrix protocol's cryptographic core.* [2]

Our description covers Olm, Megolm, Matrix' *Cross-Signing Framework* and its *Key Request Protocol*.[3]

Next, noting that the vulnerabilities discovered in [6] relied on cross-protocol interactions, we introduce a monolithic model for SGM protocols to capture the various sub-protocols involved in modern messaging and their interaction.

Contribution 2. *Section 3 introduces the Device-Oriented Group Messaging (DOGM) model for SGM protocols that captures the messaging functionality, group management and user/device identity management together. Additionally, the State Share sub-protocol captures state sharing among devices (as well as backup and recovery).*

Our model enables capturing various security boundaries and trade-offs between users and devices. Our adversarial model captures a diverse range of temporal compromises and full control over the network. In addition, the DOGM experiment does not provide a trusted map from identity to public key (as is standard in the literature [17], [20]). Rather, the challenger allows the implementing protocol to set up the requisite public-key infrastructure (PKI), then expects the protocol to correctly use it. [4]To avoid over-specialising the DOGM model to Matrix, we encode protocol-specific security requirements in CONF and AUTH predicates. These predicates track and define the cases where confidentiality and authentication are expected to be broken by the adversary: they capture "trivial wins" in the security game.[5]

Contribution 3. *Section 4 states and proves Matrix' confidentiality and authentication guarantees, defined through the predicates* MTXCONF *and* MTXAUTH*, under the adversarial model defined by the DOGM security experiment and against any PPT adversary.*

Our proof is in the random oracle model (ROM) and assumes that (a) the Gap Diffie-Hellman (Gap-DH) [26] assumption holds on Curve25519 (b) Ed25519 provides strong unforgeability under chosen message attacks (SUF-CMA) [27], (c) HKDF is a secure key derivation function (KDF) [28], (d) MgRatchet (see below) is a secure fast-forwardable pseudorandom generator (FF-PRG) [29], and (e) AES-CBC and HMAC (as combined in both Mg and OlmAEAD) is a secure authenticated encryption with associated data (AEAD) scheme.

In particular, the MTXCONF and MTXAUTH security predicates (and the trivial attacks they encode) show that compromising users and compromising their devices enable distinct attacks in Matrix. We discuss these predicates, and map them back to intuitive scenarios, in Section 4.1. These form the basis of a broader discussion in Section 5, where we discuss their impact on Matrix in its usage context.

1.3. Scope

We analyse Matrix in a computational rather than a symbolic model to capture the often subtle interplay of cryptographic components at a byte rather than a symbol level. Indeed, recent work has shown that we must not assume that the cryptographic components used in Matrix are perfect [6]. This more in-depth approach comes at the cost of breadth.

Backups and out-of-band verification. We do not cover Matrix' backup solutions for cryptographic key materials (neither Server-side Megolm Backups nor SSSS). Neither does this analysis cover the constituent protocols of the verification framework (such as the SAS protocol [6, Fig. 11 and 12]). This is unfortunate, since these components were the source of several recent vulnerabilities [6]. We hope that our work will enable future work that analyses these components in depth.

Olm channels. Whilst we do model the security of the Olm channel itself, we do not model compromises of its session

2. While the work of [6] focuses its description on the parts of Matrix relevant to the attacks, our description abstracts out the implementation details that may be necessary to fully describe attacks.

3. The description is a synthesis of the Matrix Client-Server API [1], Olm [4], [9] and Megolm [3] specifications, the implementation guide [25], source code of the `matrix-react-sdk` and `matrix-js-sdk` libraries provided by the Matrix foundation and that of the flagship client, Element.

4. Sessions must then use their own keys (and signatures distributed through the adversary) to determine their trust in the keys of other users or devices. The adversary may also create their own users and devices without the challenger's help; such parties are not verified, and not tracked by the security predicates. Our analysis will demonstrate such attacks will not work.

5. For example, in Matrix if the adversary gains access to a Megolm session secret, it is expected that it can decrypt all future messages in that session (until a replacement session is created). Winning the experiment through one of these ciphertexts is considered a "trivial win" and is captured in the MTXCONF predicate. These trivial wins are protocol-specific, however, and encode the security guarantees it provides.

state (only compromise of the device keys used to authenticate new channels). We do this because Matrix uses Olm in a manner that (partially) undermines its post-compromise security (PCS) guarantees.[6] As such, our approach captures forward-secrecy of Olm channels, and is motivated by the existing body of work on Olm and Signal mentioned above, but does not capture post-compromise security. Thus, we do not model exact nature of interactions with Megolm due to *Olm channel compromises*. See [15] for a detailed analysis of similar interactions between Sender Keys and the underlying Signal channels (as they are used in WhatsApp).

Implementation issues. While we analyse Matrix in a computational model, this does not cover implementation issues and side-channels. This is despite this work having to rely on implementation specific behaviour in the flagship Element client to establish a formal model of Matrix' intended behaviour, since the specification is sparse in some places.

Monolithic analysis. Our analysis treats the Matrix protocol as one monolithic whole, rather than analysing each subprotocol in isolation before considering their composition, as in [13].

1.4. Limitations

Group membership. As it stands, Matrix lacks cryptographic authentication for (recipient) group membership [6]. We capture this limitation by modelling these attacks as *trivial wins*; for example, if an adversary manages to add a corrupted device to a session to break its confidentiality. We note, however, that our model does allow us to capture authenticated group membership and thus allows us to model planned future versions of Matrix where this vulnerability is addressed. We also note that, more generally, work on modelling group administration [20], [23], [30] and out-of-band authentication [31] has only recently picked up pace.

Verification of identities. Our security statements are only valid when out-of-band verification is enforced, despite this not being required by the Matrix specification [1].[7]

Tightness. Our model captures *adaptive compromise* of secrets at the session and device level, motivated by recent attacks [6] and work [32] that demonstrated the limitations of current models of PCS. However, in order for the challenger to indistinguishably swap out key material and ciphertexts with the output of other security experiments (such as those output by the SUF-CMA challenger), they must correctly guess which keys will be compromised by the attacker at the start of the experiment. As such, we face the *commitment problem* [17], [33] and must guess the session/query to plant our challenges into. This proof technique allows us to derive a security result in the face of such a strong adversary, with the caveat that we abort the experiment whenever one of our guesses was incorrect. As a result, we obtain a loose bound on the security of Matrix against adaptive attackers (rather than a tighter bound against a non-adaptive attacker).

6. Matrix allows multiple active Olm channels between pairs of clients. Access to device keys, therefore, enables attackers to establish new trusted Olm channels (regardless of whether *existing channels* have recovered).

7. Element allows communication with unverified users/devices by default (with warnings in the user interface). An optional setting can disable this behaviour. However, note that such behaviour is common among deployments of secure messaging protocols.

2. Secure Messaging in the Matrix Standard

The Matrix standard defines the interaction between three types of entity: homeservers, users and devices. Homeservers provide the primary point of contact into the Matrix network for the users they host. Each user has an account on a particular homeserver as well as a number of associated devices.

All conversations occur within a **room**, each of which is located within a particular **homeserver**. Users from different homeservers are able to converse across the network thanks to federation i.e. a user with an account on homeserver H_1 can join and converse in rooms located at homeserver H_2 if H_1 and H_2 federate. For the purpose of modelling end-to-end encryption, we consider the network of federated homeservers as a single actor, H. We represent **users** with identifiers $U \in \mathcal{U}_{id}$ of the form `@username:homeserver.tld`. **Devices** also have their own identifiers, which we represent as $D^{U,i} \in \mathcal{D}_{id}$. Device identifiers are allocated by the homeserver and are expected to be unique within a single user. Our analysis indexes user devices in creation order with the tuple (U, i) to distinguish them without ambiguity.

The **Cross-Signing** module defines cryptographic identities for each user and their devices. It allows modelling relationships between these identities using digital signatures. The **Verification Framework** defines the **Short Authentication String (SAS)** and **QR code** verification protocols to allow verifying the cryptographic identities of users and devices out-of-band. In particular, two users can perform out-of-band verification. If the verification succeeds, they will sign each other's identities in a process known as *cross-signing*. Similarly, a user might perform out-of-band verification between two of their devices (one new and another that is already verified). If the verification succeeds, they will sign the new device's identity with the user's cross-signing identity to establish that the new device has been verified. This process is known as *self-verification*.

Matrix uses the **Secure Secret Storage and Sharing (SSSS)** module to gossip user-level secrets between verified devices, as well as to backup those secrets to the homeserver.

Together, cross-signing and self-verification provide a cryptographic link from real-world identities to users and their devices. The **Olm** protocol uses these cryptographic links to provide secure channels between pairs of devices. Olm uses a modified Triple Diffie-Hellman (3DH) [34] handshake for the initial key exchange, combining ephemeral keys with long-term device keys from the cross-signing module, followed by the Double Ratchet for

continuous key exchange. It provides a secure, underlying signalling layer between pairs of devices to the other sub-protocols in Matrix.

Matrix uses the **Megolm** protocol to protect the contents of conversations. Megolm provides a secure unidirectional channel between one sending device and many receiving devices. These channels use a symmetric ratchet, the **Megolm Ratchet**, to provide (optional) forward security that can be fast-forwarded. These unidirectional channels are composed together to form a group chat. Megolm uses pairwise Olm channels to distribute inbound Megolm sessions.

Separate to SSSS, the **Key Request** protocol allows devices to share inbound Megolm sessions over Olm after their initial distribution. This enables functionality such as history sharing. The **Server-side Megolm Backups** module enables a user's devices to backup encrypted copies of inbound Megolm sessions to the homeserver using a shared secret.

2.1. Cross-signing & Verification Framework

The cross-signing protocol consists of five algorithms, CS = (Init, SignUser, SignDevice, VerifyDevice), with the following syntax:

- $(\mathfrak{U}, st_{\mathsf{cs}}) \leftarrow\$ \ \mathsf{CS.Init}(1^\lambda, A)$ takes as input a security parameter and user identifier A. It initialises a cross-signing identity for A, returning state st_{cs} and a plaintext message with the user's public keys and signatures linking them together.
- $(st_{\mathsf{cs}}, u_B) \leftarrow \mathsf{CS.SignUser}(st_{\mathsf{cs}}, B, mpk_B)$ takes as input a cross-signing state st_{cs}, a user identity B and their master cross-signing identity mpk_B, and is executed after the user performs out-of-band verification with B. It returns an updated cross-signing state st_{cs} and a signed message t_B attesting that the user believes mpk_B is owned by user B.
- $(st_{\mathsf{cs}}, dt_{A,D}) \leftarrow \mathsf{CS.SignDevice}(st_{\mathsf{cs}}, A, D, dpk, ipk)$ takes as input the executing user's cross-signing state st_{cs}, user identifier A, and the device identifier D, device identity key dpk and Olm identity key ipk. This algorithm is executed after a user completes out-of-band verification with one of their devices (self-verification). It returns an updated cross-signing state st_{cs} and a signed message $dt_{A,D}$ to attest that D is one of user A's devices.
- $(B, E, dpk_*) \leftarrow \mathsf{CS.VerifyDevice}(st_{\mathsf{cs}}, ipk_*)$ takes as input cross-signing state st_{cs} and an Olm identity key ipk_*. It checks the signatures in st_{cs} to determine the key's user B and device E, and ensures that the calling user has performed out-of-band verification with B. If these checks pass, it returns the verified user B and device E of ipk_* alongside their cryptographic identity dpk_*. If not, it returns $(\perp, \perp, \perp)$.

We now give a brief description of the cross-signing protocol (detailed in Fig. 1).

User setup. Each user sets up an account with a particular homeserver, which allocates them a user identifier, A. Next, the user generates their *cross-signing keys* (as described by CS.Init). The *master key* (msk_A, mpk_A) serves as their long-term identity. It is used to sign the *user-signing key* and *self-signing key*. The *user-signing key* (usk_A, upk_A) signs other user's master keys. The *self-signing key* (ssk_A, spk_A) signs a user's own device keys. Together, these enable each pair of users to verify one another's identities once, then rely on the other user to verify their own devices (see Figure 2 and Table 1).

TABLE 1: Summary of the keys used in Matrix.

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
$dsk_{A,i}$	$dpk_{A,i}$	Fingerprint/signing key for A's ith device
$isk_{A,i}$	$ipk_{A,i}$	Olm identity key for A's ith device
$esk_{A,i,j}$	$epk_{A,i,j}$	jth ephemeral Olm pre-key for A's ith device
$fsk_{A,i,j}$	$fpk_{A,i,j}$	jth fallback Olm key for A's ith device

For example, when Alice adds a new device, they may use ssk_A to sign the new device's long-term identity keys $(dpk_{A,i}, ipk_{A,i})$. Alice's user-signing key usk_A may sign Bob's master key mpk_B after out-of-band verification, to signify that they believe Bob is in control of the master key msk_B. The public parts of these keys are uploaded to the homeserver and associated with the user account by the homeserver.

Before signing a user's keys (to indicate trust), the two users can verify their identity out-of-band (using the aforementioned verification framework). They then sign each other's master public key mpk with their own user signing key usk. We do not model the out-of-band verification process. Instead, we use CS.SignUser to simulate its result.

Device setup. When a new client logs in with their account credentials, the homeserver allocates a device identifier (denoted by $D^{A,i}$). The client then generates a cryptographic identity for this device to register it with the homeserver, creating a non-cryptographic association between the user identifier, device identifier and device keys. This identity includes: (1) Device Fingerprint/Signing Key $(dsk_{A,i}, dpk_{A,i})$ and (2) Olm Key Bundle $(isk_{A,i}, ipk_{A,i}, esk_{A,i}, epk_{A,i}, fsk_{A,i}, fpk_{A,i})$.

The tuple $(esk_{A,i}, epk_{A,i})$ is a *pre-key bundle*: ephemeral (single-use) keys distributed by the homeserver, allowing other devices to initiate a key exchange asynchronously as part of the Olm protocol (Section 2.2). The tuple $(fsk_{A,i}, fpk_{A,i})$ are bundles of one or more *fallback key pairs* [35]. The public parts of each of these keypairs are distributed to other devices through the homeserver (as a bundle self-signed with $dsk_{A,i}$). The algorithm Mtx.Reg (Section 2.5) details how the device identity and the self-signed device keys are created.

Once a new device has setup their device identity, an existing device (with possession of the user's secret cross-signing keys) can verify the new user's device out-of-band. This process is known as *self-verification*. Once verification is complete, the verifying device signs a bundle

$\mathsf{CS.Init}(1^\lambda, A)$

$(msk, mpk) \leftarrow \mathsf{DS.KGen}(1^\lambda)$
$(ssk, spk) \leftarrow \mathsf{DS.KGen}(1^\lambda)$
$(usk, upk) \leftarrow \mathsf{DS.KGen}(1^\lambda)$
$dt, ut \leftarrow \emptyset, \emptyset$
$m_m \leftarrow (mpk, \texttt{CsMstr}, A)$
$m_s \leftarrow (spk, \texttt{CsSelf}, A)$
$m_u \leftarrow (upk, \texttt{CsUser}, A)$
$\sigma_m \leftarrow \mathsf{DS.Sign}(msk, m_m)$
$\sigma_d \leftarrow \mathsf{DS.Sign}(msk, m_s)$
$\sigma_u \leftarrow \mathsf{DS.Sign}(msk, m_u)$
$\mathfrak{U} \leftarrow (m_m, \sigma_m, m_s, \sigma_d, m_u, \sigma_u)$
$st_{\mathsf{cs}} \leftarrow (A, msk, usk, ssk, dt, ut)$
return $(\mathfrak{U}, st_{\mathsf{cs}})$

$\mathsf{CS.SignUser}(st_{\mathsf{cs}}, B, mpk_B)$

$m_B \leftarrow (mpk_B, \texttt{CsMstr}, B)$
$\sigma_B \leftarrow \mathsf{DS.Sign}(st_{\mathsf{cs}}.usk, m_B)$
$st_{\mathsf{cs}}.ut[B] \leftarrow (m_B, \sigma_B)$
return $(st_{\mathsf{cs}}, st_{\mathsf{cs}}.ut[B])$

$\mathsf{CS.SignDevice}($
$st_{\mathsf{cs}}, A, D, dpk, ipk)$

$m_D \leftarrow (A, D, dpk, ipk, \texttt{OlmAlg})$
$\sigma_D \leftarrow \mathsf{DS.Sign}(st_{\mathsf{cs}}.ssk, m_D)$
$st_{\mathsf{cs}}.dt[ipk] \leftarrow (m_D, \sigma_D)$
return $(st_{\mathsf{cs}}, st_{\mathsf{cs}}.dt[ipk])$

$\mathsf{CS.VerifyDevice}(st_{\mathsf{cs}}, ipk_*)$

$(m_D, \sigma_D) \leftarrow st_{\mathsf{cs}}.dt[ipk_*]$
$(B, D, dpk, ipk, alg) \leftarrow m_D$
if $(ipk \neq ipk_*)$: **return** $(\perp, \perp)$
$(m_m, \sigma_m, m_s, \sigma_s, m_u, \sigma_u) \leftarrow \mathfrak{U}_B$
$(mpk_B, \texttt{CsMstr}, B) \leftarrow m_m$
$(spk_B, \texttt{CsSelf}, B) \leftarrow m_s$
if $!\mathsf{DS.Verify}(spk_B, \sigma_D, m_D)$: **return** $(\perp, \perp)$
if $!\mathsf{DS.Verify}(mpk_B, \sigma_s, m_s)$: **return** $(\perp, \perp)$
if $!\mathsf{DS.Verify}(mpk_B, \sigma_m, m_m)$: **return** $(\perp, \perp)$
if $A = B$: **return** (B, D, dpk)
$(m_B, \sigma_B) \leftarrow st_{\mathsf{cs}}.ut[B]$
if $(!\mathsf{DS.Verify}(st_{\mathsf{cs}}.\mathfrak{U}.upk, \sigma_B, m_B)$
$\vee\, m_B \neq (mpk_B, \texttt{CsMstr}, B))$: **return** $(\perp, \perp)$
return (B, D, dpk)

Figure 1: Pseudocode describing the Cross-Signing protocol.

containing the new device's long-term cryptographic keys $(dpk_{A,i}, ipk_{A,i})$ with the user's self-signing key ssk_A. The signature bundle is uploaded to the homeserver, creating a mapping from a user's cross-signing identity to the new device's cryptographic identity. Clients use the *verification framework* to perform the out-of-band verification. CS.SignDevice (Fig. 1) simulates the result of this process.

The combination of out-of-band verification between users (by verifying signatures of each other's mpk with usk) and user's self-verification of their own devices (via signatures of each device identity (dpk, ipk) with ssk) creates a chain of trust that can be followed to determine whether a given device really is controlled by a user. Figure 2 displays two users, their individual key hierarchies and the relation between them.

The algorithm $\mathsf{CS.VerifyDevice}(st_{\mathsf{cs}}, ipk_*)$ (Fig. 1) is run by user A to determine the user and device identifiers associated with the given device identity key ipk_*. Using the verifying user's cross-signing state, they ensure that ipk_* is the verified by the person they expect to be contacting. If a device passes these checks, it is considered trusted by the client and CS.VerifyDevice outputs the verified user identifier, device identifier and device key. We consider an idealised Matrix implementation that will only interact with trusted devices (as defined in CS.VerifyDevice).

We illustrate the sequence of algorithm calls and messages in Fig. 3.

2.2. The Olm Protocol

Olm is designed to create a secure channel between two devices. It aims to provide confidentiality, authenticity, forward secrecy (FS), PCS and deniability [4], [9].[8] Following the same broad design as the Signal protocol [34], [36], [37], Olm uses a modified 3DH [34] for the initial key exchange,

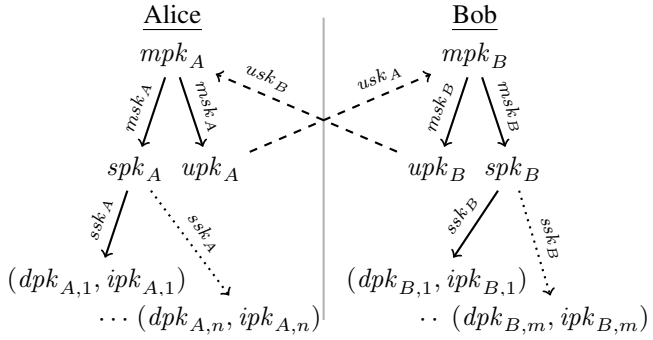

Figure 2: The long-term key hierarchy for two users, Alice and Bob, and each of their devices [1]. Each arrow denotes a signature. Dashed arrows denote signatures resulting from an out-of-band verification between Alice and Bob. Dotted arrows denote signatures resulting from an out-of-band verification 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].

which we refer to as Olm Triple-Diffie Hellman (O3DH) [1], [4], [9]. O3DH sets up the first epoch of the Double Ratchet protocol [37].[9] Olm consists of three algorithms, Olm = (KGen, Enc, Dec), with the following syntax.

- $(isk, esk, fsk), (ipk, epk, fpk) \leftarrow\$ \mathsf{Olm.KGen}(1^\lambda, n_e, n_f)$
- $(st_{\mathsf{olm}}, c) \leftarrow \mathsf{Olm.Enc}(st_{\mathsf{olm}}, m, isk_{A,i}, ipk_{B,j}, epk_{B,j,k})$
- $(st_{\mathsf{olm}}, m) \leftarrow \mathsf{Olm.Dec}(st_{\mathsf{olm}}, c, isk_{A,i}, esk_{A,i})$

We now give a brief description of the Olm protocol (detailed in Fig. 4). Refer to Fig. 5 to see how they interact.

8. Olm's initial key exchange, as we describe it in this section, provides deniability. However, since Matrix signs the ephemeral keys (see Mt×Olm in Fig. 9), this property is lost (in exchange for stronger FS) [9].

9. We combine our descriptions of the initial key exchange and Double Ratchet protocol within the Olm.Enc and Olm.Dec algorithms to reflect how the two intertwine in practice.

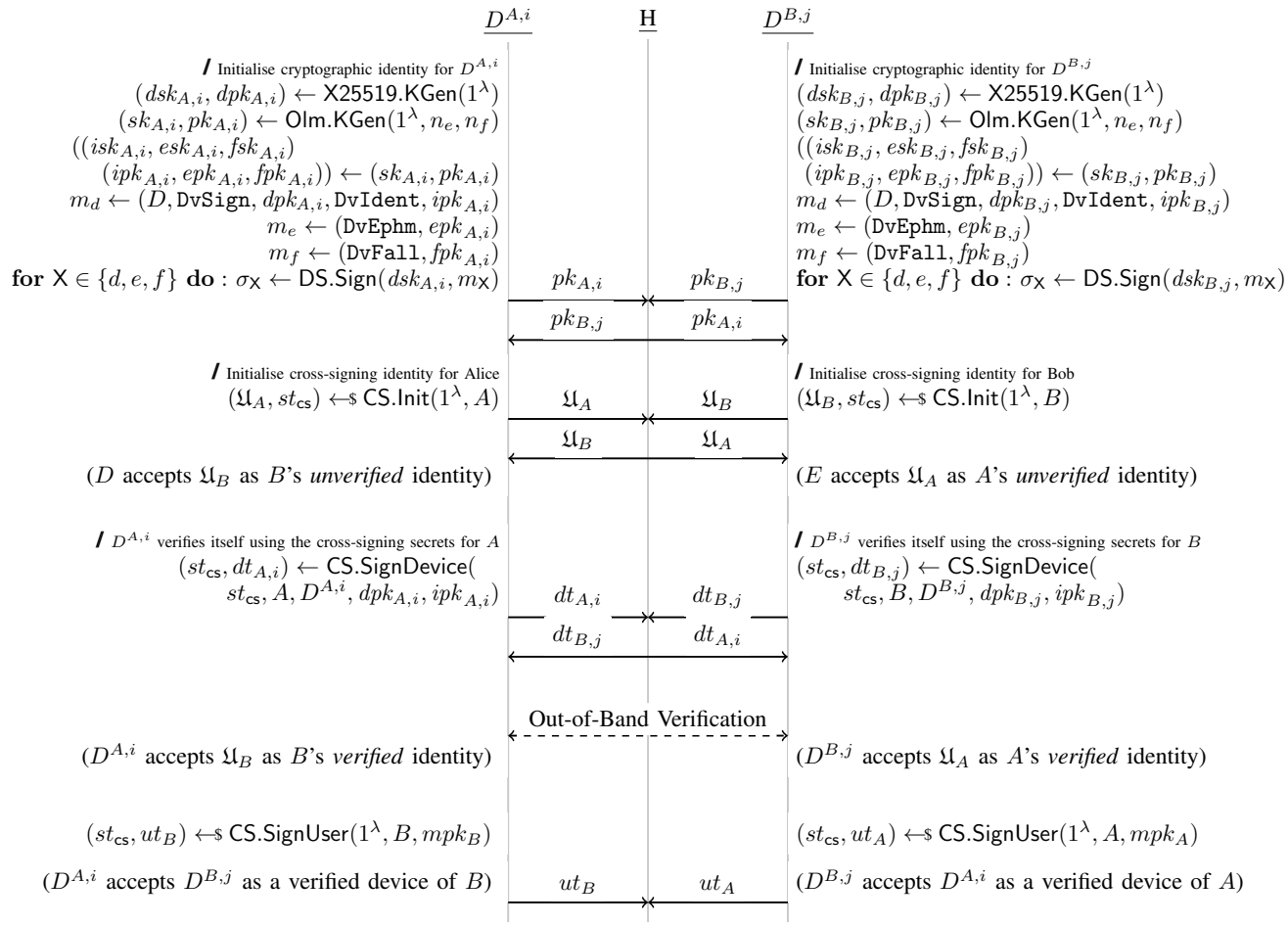

Figure 3: A demonstration of device setup, cross-signing setup and the *user verification* process between Alice A, Bob B and their two devices ($D^{A,i}$ and $D^{B,j}$ respectively). To start, both users setup each device, initialise their cross-signing setups and sign their first device. Since the device which initialises a user's cross-signing identity already has the secret keys, it does not need to perform self-verification with itself and can simply sign it's device identity. Next, each user distributes their device and cross-signing information through the homeserver. This identity is automatically accepted by the other user's devices but is marked as *unverified*. The two devices, $D^{A,i}$ and $D^{B,j}$, perform out-of-band user-to-user verification on behalf of Alice and Bob. Once complete, they each mark the other's cross-signing identity as *verified* and, by following the cross-signing signatures in $\mathfrak{D}$ may now mark the other device as *verified* as well. Note that we model Matrix as only accepting messages from verified devices, i.e. from this point onwards. However, in practice, Matrix would accept messages from unverified users but display a warning in the user interface.

Key generation. Each device, say $D^{A,i}$, maintains three sets of keys: 1) a long-term identity key pair (isk, ipk), 2) n_e ephemeral key pairs $\{(esk_{A,i,k}, epk_{A,i,k}) : 0 \leq k < n_e\}$, and 3) n_f fallback key pairs $\{(fsk_{A,i,k}, fpk_{A,i,k}) : 0 \leq k < n_f\}$.

The O3DH key exchange combines a long-term identity key and a single-use key from each device. The initiating device generates a single-use key and uses one ephemeral key from the receiving device (provided by the homeserver).[10] The fallback keys are used as a backup when no unused ephemeral keys available. $D^{A,i}$ generates these keys by executing $\mathsf{Olm.KGen}$, specifying the number of ephemeral and fallback key pairs they need. The public parts of these key pairs are distributed through the homeserver.

Session setup. We now describe the initial key exchange and how it sets up (and intertwines with) the first epoch of the Double Ratchet protocol (with epoch $p = 0$).

To initialise a session with Bob's device $D^{B,j}$, Alice's device $D^{A,i}$ first fetches Bob's keys from the homeserver, $ipk_{B,j}$ and $epk_{B,j,k}$. A fallback key $fpk_{B,j,k}$ is used if no ephemeral key pairs are available, in which case $epk_{B,j,k}$ is swapped with $fpk_{B,j,k}$ in the explanation that follows. Alice's device should verify the keys that the homeserver

10. Each ephemeral key pair should only be used once. Fallback keys may be used multiple times but should be replaced as soon as possible. Clients must enforce this themselves (rather than relying on the homeserver).

provided. Matrix achieves this with the cross-signing framework.

$D^{A,i}$ then executes Olm.Enc, providing the first message they would like to send, their private identity key and $D^{B,j}$'s keys. Olm.Enc initiates a 3DH key exchange between the two parties. The algorithm generates a single-use key pair, $(esk_{A,i}, epk_{A,i})$, to act as $D^{A,i}$'s ephemeral contribution to the key exchange. $D^{A,i}$ computes their side of the key exchange using $isk_{A,i}$, $esk_{A,i}$, $ipk_{B,j}$ and $epk_{B,j,k}$. The pre-computation of $epk_{B,j,k}$ allows $D^{A,i}$ to compute the shared secret without any interaction from $D^{B,j}$.

Now, $D^{A,i}$ uses the shared secret to compute the initial state of the Double Ratchet protocol. It derives the root key $rch_{0,0}$ and chain key $ck_{0,0}$ from the shared secret using HKDF-SHA-256. $D^{A,i}$ then uses $ck_{0,0}$ to derive key material for the first message. Messages sent when $D^{B,j}$ has not yet responded are *pre-key messages*. These additionally contain, as unauthenticated data wrapping the Olm ciphertext, the public key pairs $D^{B,j}$ needs to derive the shared secret for the first epoch: (1) $ipk_{A,i}$ (2) $epk_{A,i}$ (3) $epk_{B,j,k}$. Other than this, pre-key messages are encrypted as normal Olm messages (as in Section 2.2). As such, $D^{A,i}$ generates a new X25519 key pair and includes the public part as a contribution towards the root key for the next epoch.

When $D^{B,j}$ receives a pre-key message, it executes Olm.Dec which will calculate the shared secret then initialise the Double Ratchet protocol. $D^{A,i}$ may continue to encrypt new messages by ratcheting the chain key forward, which $D^{B,j}$ may similarly decrypt. The protocol progresses to the next epoch when $D^{B,j}$ replies.

Messaging. To encrypt a message $m_{p,q}$, the sending device runs Olm.Enc to produce a ciphertext $c_{p,q}$ that can later be decrypted by the receiving device using Olm.Dec. We address messages by the current epoch p and chain index q. An *epoch* represents an uninterrupted sequence of messages sent by a single device. Within a single epoch, the symmetric ratchet ck_q is progressed to ck_{q+1} after each message is encrypted. A new epoch starts when the receiving device of the current epoch replies. We now describe this process.

If $D^{B,j}$ sends a message in an epoch where they are the receiving party (i.e. in an epoch they did not initiate), they increment the epoch p, reset the chain index q to zero and progress the asymmetric ratchet. To do this, they compute the next root key rch_p and chain key $ck_{p,0}$ using the ratchet key rpk_{p-1} (provided in the most recent ciphertext sent by $D^{A,i}$) and a freshly generated ratchet key rsk_p. Otherwise, if $D^{B,j}$ sends a message where they are already the sending device (i.e. in an epoch they initiated) they simply increment the symmetric ratchet counter $q \leftarrow q + 1$ (noting that $D^{B,j}$ will have ratcheted forward the chain key to $ck_{p,q+1}$ after encrypting the previous message).

We now describe the process of encrypting (and decrypting) Olm messages. To encrypt $m_{p,q}$, $D^{B,j}$ derives fresh key material using the current chain key $ck_{p,q}$. This key material is provided to an AEAD scheme built from AES-CBC and HMAC-SHA-256 which derives the keys needed for each algorithm using HKDF. Each message includes the current Olm protocol version, the ratchet key generated by $D^{B,j}$ during encryption, and the current chain index q as authenticated data as part of the ciphertext $c_{p,q}$. Finally, $D^{B,j}$'s copy of the chain key is ratcheted forward, ready to encrypt the next message $m_{p,q+1}$.[11]

When $D^{A,i}$ receives $c_{p,q}$, they first check if it has initiated a new epoch (i.e. is this the first message they have received from $D^{B,j}$ since they last sent a message). If so, they update their copies of p and q, then use the ratchet key rpk_p in $c_{p,q}$ and their copy of rsk_{p-1} to generate the next root key rch_p and chain key $ck_{p,0}$. They then generate the key material needed to verify and decrypt the message, which they pass to OlmAEAD.Dec for decryption. Finally, $D^{A,i}$ ratchets their copy of the chain key forward, ready to decrypt the next message $m_{p,q+1}$.

The protocol proceeds with alternating epochs, each consisting of a sequence of messages from a single sender. Each new epoch is initiated by a reply from the recipient of the current epoch.

2.3. The Megolm Protocol

As described above, Megolm constructs a logical group conversation from the composition of many unidirectional channels (one for each sending party). It uses Olm channels between pairs of participants to setup and manage each of these Megolm sessions independently.

Each device generates their own Megolm session when they first send a message to the group. The session (i.e. their sender key) consists of two parts: an outbound session used to encrypt messages and an inbound session used to decrypt them. The inbound session is then distributed to each member of the group individually over their respective Olm channel.

It consists of four algorithms, Mg = (Init, Recv, Enc, Dec), with the following syntax:

- $\mathfrak{S}_{gsk}, \mathfrak{S}_{gpk}, \sigma_{\mathsf{mg}} \leftarrow\$ \mathsf{Mg.Init}(1^\lambda)$ takes as input a security parameter then generates a new Megolm session. It returns the outbound session, inbound session and a signature over the inbound session (created with the group signing key).
- $\mathfrak{S}_{gpk} \leftarrow \mathsf{Mg.Recv}(\mathfrak{S}_{gpk}, \sigma_{\mathsf{mg}})$ takes as input an inbound Megolm session $\mathfrak{S}_{gpk}$ and its signature σ_{mg}. It verifies the signature using the group verification key gpk contained within $\mathfrak{S}_{gpk}$. It returns $\mathfrak{S}_{gpk}$ if the signature is valid and $\perp$ if it is not.
- $\mathfrak{S}_{gsk}, c \leftarrow \mathsf{Mg.Enc}(\mathfrak{S}_{gsk}, m)$ takes as input an outbound Megolm session $\mathfrak{S}_{gsk}$ and a plaintext message m. It encrypts the message (using the ratchet R to generate symmetric key material) then signs the encrypted message with the group signing key gsk. The algorithm outputs an updated outbound session $\mathfrak{S}_{gsk}$ and the resulting ciphertext c. If the encryption fails, it returns an updated outbound session $\mathfrak{S}_{gsk}$ and $\perp$.
- $\mathfrak{S}_{gpk}, m \leftarrow \mathsf{Mg.Dec}(\mathfrak{S}_{gpk}, c)$ takes as input an inbound Megolm session $\mathfrak{S}_{gpk}$ and ciphertext c produced by Mg.Enc.

11. Old values of the root key rch, chain key ck, ratchet key (rsk, rpk), message keys mk and AEAD keys (k_e, k_h, k_{iv}) should be discarded to preserve FS and PCS.

$\mathsf{Olm.KGen}(1^\lambda, n_e, n_f)$

$(isk, ipk) \leftarrow\$ \mathsf{X25519.KGen}(1^\lambda)$
for $0 \le k < n_e$ **do** : $(esk_k, epk_k) \leftarrow\$ \mathsf{X25519.KGen}(1^\lambda)$
$(esk, epk) \leftarrow (esk_0, \ldots, esk_{n_e-1}\}, \{epk_0, \ldots, epk_{n_e-1})$
for $0 \le k < n_f$ **do** : $(fsk_k, fpk_k) \leftarrow\$ \mathsf{X25519.KGen}(1^\lambda)$
$(fsk, fpk) \leftarrow (fsk_0, \ldots, fsk_{n_f}\}, \{fpk_0, \ldots, fpk_{n_f-1})$
return $(isk, esk, fsk),\ (ipk, epk, fpk)$

$\mathsf{Olm.Enc}(st_{\mathsf{olm}}, m, isk, ipk^*, epk^*)$

if $(st_{\mathsf{olm}} = \bot)$: / session setup
  $\rho \leftarrow \mathtt{send}; p \leftarrow 0; q \leftarrow 0$
  $(esk, epk) \leftarrow\$ \mathsf{X25519.KGen}(1^\lambda)$
  $ms \leftarrow (epk^*)^{isk} \| (ipk^*)^{esk} \| (epk^*)^{esk}$
  $rch \| ck \leftarrow \mathsf{HKDF}(0, ms, \mathtt{OlmRch0})[0:64]$
  $(rsk, rpk) \leftarrow\$ \mathsf{X25519.KGen}(1^\lambda)$
else :
  $(\rho, p, q, rch, ck, rsk, rpk, rpk^*) \leftarrow st_{\mathsf{olm}}$
  if $(\rho = \mathtt{recv})$: / new epoch
    $\rho \leftarrow \mathtt{send}; p \leftarrow p+1; q \leftarrow 0$
    $(rsk, rpk) \leftarrow\$ \mathsf{X25519.KGen}(1^\lambda)$
    $rch \| ck \leftarrow \mathsf{HKDF}(rch, (rpk^*)^{rsk}, \mathtt{OlmRch})[0:64]$
  elseif $(\rho = \mathtt{send})$: $q \leftarrow q+1$ / existing epoch
$mk \leftarrow \mathsf{HMAC}(ck, \mathtt{0x01})$
$c \leftarrow \mathsf{OlmAEAD.Enc}(mk, \mathtt{OlmKeys}, (\mathtt{OlmVer}, rpk, q), m)$
$ck \leftarrow \mathsf{HMAC}(ck, \mathtt{0x02})$
if $(p = 0)$: $c \leftarrow (ipk, epk, epk^*) \| c$
$st_{\mathsf{olm}} \leftarrow (\rho, p, q, rch, ck, rsk, rpk, rpk^*)$
return (st_{olm}, c)

$\mathsf{Olm.Dec}(st_{\mathsf{olm}}, c, isk, esk)$

if $(st_{\mathsf{olm}} = \bot)$: / complete setup
  $\rho \leftarrow \mathtt{recv}; p \leftarrow 0; q \leftarrow 0$
  $(ipk^*, epk^*, epk, \mathtt{OlmVer}, rpk^*, q^*, x, \tau) \leftarrow c$
  $ms \leftarrow (ipk^*)^{esk} \| (epk^*)^{isk} \| (epk^*)^{esk}$
  $rch \| ck \leftarrow \mathsf{HKDF}(0, ms, \mathtt{OlmRch0})[0:64]$
else :
  $(\rho, p, q, rch, ck, rsk, rpk, rpk^*) \leftarrow st_{\mathsf{olm}}$
  if $(p = 0)$: $(ipk^*, epk^*, epk, \mathtt{OlmVer}, rpk^*, q^*, x, \tau) \leftarrow c$
  else : $(\mathtt{OlmVer}, rpk^*, q^*, x, \tau) \leftarrow c$
  if $(\rho = \mathtt{send})$ / new epoch
    $\rho \leftarrow \mathtt{recv}; p \leftarrow p+1; q \leftarrow 0$
    $rch \| ck \leftarrow \mathsf{HKDF}(rch, (rpk^*)^{rsk}, \mathtt{OlmRch})[0:64]$
    $(rsk, rpk) \leftarrow (\bot, \bot)$
  elseif $(\rho = \mathtt{recv})$ $q \leftarrow q+1$ / existing epoch
if $q \ne q^*$: **return** $(st_{\mathsf{olm}}, \bot)$
$mk \leftarrow \mathsf{HMAC}(ck, \mathtt{0x01})$
$ad, m \leftarrow \mathsf{OlmAEAD.Dec}(mk, \mathtt{OlmKeys}, c)$
if $(ad, m) = (\bot, \bot)$: **return** $(st_{\mathsf{olm}}, \bot)$
$ck \leftarrow \mathsf{HMAC}(ck, \mathtt{0x02})$
$st_{\mathsf{olm}} \leftarrow (\rho, p, q, rch, ck, rsk, rpk, rpk^*)$
return (st_{olm}, m)

Figure 4: Pseudocode describing the Olm protocol.

The algorithm checks the signature, then decrypts the message. It returns an updated inbound session $\mathfrak{S}_{gpk}$ and plaintext m if the decryption succeeds. If not, it returns an updated inbound session $\mathfrak{S}_{gpk}$ and $\bot$.

We describe each algorithm in the sections that follow (refer to Fig. 6 for a formal description).

Group initialisation and management. A Megolm session consists of the current message index i, the internal ratchet state R, and the group signing keypair (gsk, gpk).

A Megolm session can be either an *outbound* or *inbound session*. *Outbound sessions*, $\mathfrak{S}_{gsk} = (ver, i, R, gsk, gpk)$ are kept by the sending device and used to encrypt messages to the room. *Inbound sessions*, $\mathfrak{S}_{gpk} = (ver, i, R, gpk)$, allow other devices in the room to authenticate and decrypt these messages. The protocol version is stored alongside each session.

To begin a new session, the sending device executes Mg.Init, which outputs the inbound and outbound sessions separately. The inbound session is distributed individually to each device in the room using Olm in session sharing format[12]. When a device receives an inbound Megolm session, they use the Mg.Recv algorithm to verify its signature against the gpk it contains. If the verification succeeds, the algorithm outputs the inbound session. If not, it outputs $\bot$.

Messaging. To send a message, the sending device uses the ratchet to generate a fresh set of symmetric keys for authenticated encryption. It encrypts the message, appends an HMAC tag, then signs the authenticated ciphertext with the group signing key gsk. This ciphertext is sent to the homeserver which distributes it to devices in the group.

To decrypt a message, receiving devices first verify the signature with their copy of the group verification key gpk. If successful, they ratchet their local R forward to the index inside the message.[13] The receiving device then uses their copy of R to verify the MAC and decrypt the message.

We give a sequence diagram of an example execution of the Megolm protocol in Fig. 7.

Session rotation. The application layer may also rotate sessions during a conversation. For example, when a device leaves a group, the sending device generates a new session to ensure that the leaving device cannot decrypt future messages. Similarly, the specification suggests that Megolm sessions could be rotated regularly to enable PCS [3][14].

To rotate a Megolm session, the sending device simply executes Mg.Init (effectively resetting the message index to 0, then generating a new ratchet state and group signing

12. $\mathfrak{S}_{gpk}$ is output by Mg.Init in *session export* format [3]. When combined with its signature, $\mathfrak{S}_{gpk} \| \sigma_{\mathsf{mg}}$, it is said to be in *session sharing* format [3].

13. The Megolm specification recommends that sessions keep old copies of the ratchet state but, since this is optional behaviour, we subsume it into the Matrix protocol for a clearer description.

14. The PCS of an Olm channel relies on particular usage patterns and it is not necessarily the case that Megolm session rotation follows such patterns. The security model in [15] captures this relationship.

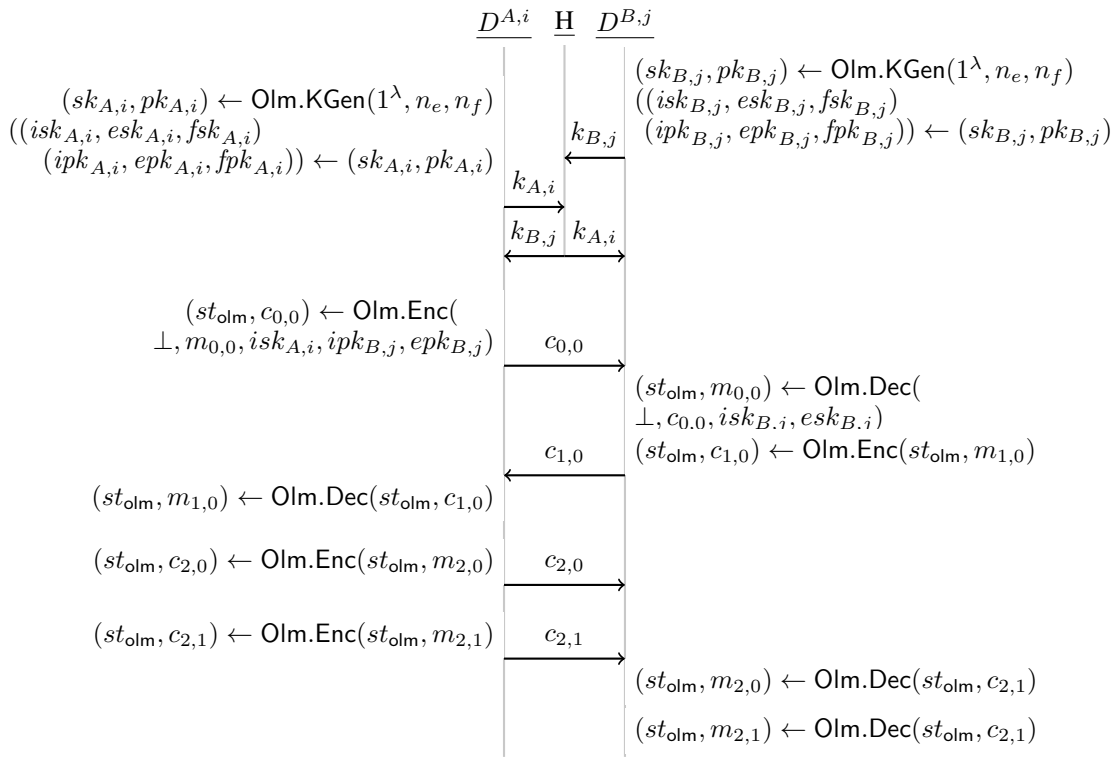

Figure 5: Example execution of the Olm protocol. In Matrix, the device keys distributed by the homeserver are signed with the device signing key dsk (although such behaviour is not mandated by the Olm protocol).

$\mathsf{Mg.Init}(1^\lambda)$

$(i, R) \leftarrow\!\$\ \mathsf{MgRatchet.Init}(1^\lambda)$
$(gsk, gpk) \leftarrow \mathsf{DS.KGen}(1^\lambda)$
$\sigma_{\mathsf{mg}} \leftarrow \mathsf{DS.Sign}(gsk, (\mathtt{0x01}, i, R, gpk))$
$\mathfrak{S}_{gsk} \leftarrow (\mathtt{0x01}, i, R, gsk, gpk)$
$\mathfrak{S}_{gpk} \leftarrow (ver, i, R, gpk)$
return $(\mathfrak{S}_{gsk}, \mathfrak{S}_{gpk}, \sigma_{\mathsf{mg}})$

$\mathsf{Mg.Recv}(\mathfrak{S}_{gpk}, \sigma_{\mathsf{mg}})$

$(ver, i, R, gpk) \leftarrow \mathfrak{S}_{gpk}$
if $\mathsf{DS.Verify}(gpk, \sigma_{\mathsf{mg}}, \mathfrak{S}_{gpk})$ **return** $\mathfrak{S}_{gpk}$
else : **return** $\perp$

$\mathsf{Mg.Enc}(\mathfrak{S}_{gsk}, m)$

$(ver, i, R, gsk, gpk) \leftarrow \mathfrak{S}_{gsk}$
$(i, R), (k_e \parallel k_h \parallel k_{iv})$
$\quad \leftarrow \mathsf{MgRatchet.Next}((i, R))$
$c \leftarrow \mathsf{AES\text{-}CBC.Enc}(k_{iv}, k_e, m)$
$\tau \leftarrow \mathsf{HMAC}(k_h, (ver, i, c))[0:8]$
$\sigma \leftarrow \mathsf{DS.Sign}(gsk, (ver, i, c, \tau))$
$c' \leftarrow (ver, i, c, \tau, \sigma)$
$\mathfrak{S}_{gsk} \leftarrow (ver, i, R, gsk, gpk)$
return $(\mathfrak{S}_{gsk}, c')$

$\mathsf{Mg.Dec}(\mathfrak{S}_{gpk}, c)$

$(ver, i, R, gpk) \leftarrow \mathfrak{S}_{gpk}$; $(ver', i', c', \tau, \sigma) \leftarrow c$
if $!\mathsf{DS.Verify}(gpk, \sigma, (ver, i', c', \tau))$:
return $(\mathfrak{S}_{gpk}, \perp)$
do $(i, R), k \leftarrow \mathsf{MgRatchet.Next}((i, R))$
until $i = i'$ // (see Fn. 13)
$(k_e \parallel k_h \parallel k_{iv}) \leftarrow k$
if $\tau \neq \mathsf{HMAC}(k_h, (ver, i, c'))[0:8]$:
return $(\mathfrak{S}_{gpk}, \perp)$
$m \leftarrow \mathsf{AES\text{-}CBC.Dec}(k_{iv}, k_e, c')$
$\mathfrak{S}_{gpk} \leftarrow (ver, i, R, gpk)$
return $(\mathfrak{S}_{gpk}, m)$

Figure 6: Pseudocode describing the Megolm protocol (see Section A for information on MgRatchet).

keypair). Next, they distribute the new inbound session $\mathfrak{S}'_{gpk}$ over Olm to the current set of group members. The public part of the Megolm signing key, gpk, is used to differentiate between (sub)sessions. We note that, in practice, clients such as Element keep a copy of old sessions and will accept messages from them. Indeed, the Key Request protocol mentioned below aims at sharing such old sessions.

2.4. Key Requests

There are a number of cases in Matrix where a device should have access to an inbound Megolm session, but missed its initial distribution. For example, when an existing member of a room adds a new device, the latter is expected to have access to all messages sent since the member first joined the room.

The key request protocol provides a solution to this problem. It allows devices in a group to request inbound Megolm sessions (the secret keys they are missing) and for devices (with possession of those sessions) to share them where permitted. Now, when an existing member of a room adds a new device, the new device may request old inbound Megolm sessions from the member's other devices.

The protocol consists of three algorithms, KS =

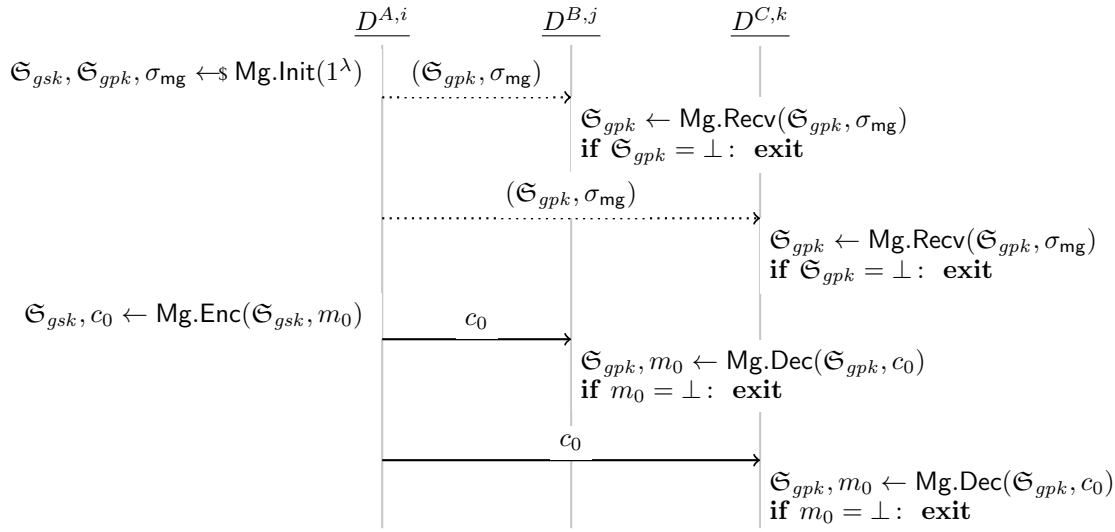

Figure 7: An example execution of the Megolm protocol. To start, Alice's sending device, $D^{A,i}$, generates a new session using Mg.Init. They keep their copy of $\mathfrak{S}_{gsk}$, then distribute the inbound session $\mathfrak{S}_{gpk}$ and its signature σ_{mg} (together in session sharing format) to Bob, $D^{B,j}$ and Claire, $D^{C,k}$. These messages are sent using Olm (such messages are represented with dotted arrows in the diagram). Bob and Claire both receive the inbound session and verify its signature. Now, Alice encrypts a message m_0 using their copy of the outbound session, generating a single ciphertext c_0 that Bob and Claire can both decrypt. All messages in this diagram implicitly pass through the homeserver (such that Alice may rely on the homeserver to distribute c_0 to all recepient devices).

$\mathsf{KS.Request}(A, D, ds, gpk_\dagger, ipk_\dagger, st_{\mathsf{cs}})$

$ds \leftarrow \{(E, dpk, ipk) \in ds : \mathsf{CS.VerifyDevice}(st_{\mathsf{cs}}, ipk) = (A, E, dpk)\}$
$m \leftarrow (\texttt{MsgReq}, \texttt{request}, A, D, \texttt{MgAlg}, gpk_\dagger, ipk_\dagger)$
return (ds, m)

$\mathsf{KS.Recover}(A, D, E, dpk_*, ipk_*, m, st_{\mathsf{cs}}, \vec{st}_{\mathsf{mg}})$

$(\texttt{MsgFwd}, \texttt{MgAlg}, gpk_\dagger, dpk_\dagger, ipk_\dagger, \mathfrak{S}_{gpk},) \leftarrow m$
if $\mathsf{CS.VerifyDevice}(st_{\mathsf{cs}}, ipk_*) \neq (A, E, dpk_*)$:
  return $(\textbf{false}, \vec{st}_{\mathsf{mg}})$
$\vec{st}_{\mathsf{mg}}[gpk_\dagger, ipk_\dagger] \leftarrow (\mathfrak{S}_{gpk}, dpk_\dagger, \textbf{true})$
return $(\textbf{true}, \vec{st}_{\mathsf{mg}})$

$\mathsf{KS.Share}(A, D, m, st_{\mathsf{cs}}, \vec{st}_{\mathsf{mg}})$

$(t, a, B, E, alg, gpk_\dagger, ipk_\dagger) \leftarrow m$
if $(t \neq \texttt{MsgReq} \vee a \neq \texttt{request}$
  $\vee A \neq B \vee D = E)$: **return** $(\bot, \bot, \bot)$
$\{ipk_E\} \leftarrow \{ipk : (A', E', \_, ipk, \_) \in st_{\mathsf{cs}}.dt \wedge A' = A \wedge E' = E\}$
$(A', E', dpk_E) \leftarrow \mathsf{CS.VerifyDevice}(st_{\mathsf{cs}}, ipk_E)$
if $(A', E') \neq (A, E)$: **return** $(\bot, \bot, \bot)$
$(\mathfrak{S}_{gpk}, dpk_\dagger, F) \leftarrow \vec{st}_{\mathsf{mg}}[gpk, ipk]$
if $\mathfrak{S}_{gpk} = \bot$: **return** $(\bot, \bot, \bot)$
$m \leftarrow (\texttt{MsgFwd}, \texttt{MgAlg}, gpk_\dagger, dpk_\dagger, ipk_\dagger, \mathfrak{S}_{gpk})$
return (dpk_E, ipk_E, m)

Figure 8: Pseudocode describing the Key Request protocol.

(Request, Share, Recover), with the following syntax.

- $(ds, m) \leftarrow \mathsf{KS.Request}(A, D, ds, gpk_\dagger, ipk_\dagger, st_{\mathsf{cs}})$ generates a set of key request messages. It takes as input the executing device's user identifier A, device identifier D, the set of A's devices ds, the group verification key of the Megolm session to request $gpk_\dagger$, the Olm identity key of the session owner $ipk_\dagger$ and the user's cross-signing state. Each item in ds should contain its device identifier, device signing key and Olm identity key. The algorithm will check whether each device in ds has been self-verified by the executing user. For each device that passes this check, a plaintext `MsgReq` message requesting the session identified by gpk. It returns a filtered list of devices ds and the message m.
- $(dpk_*, ipk_*, m) \leftarrow \mathsf{KS.Share}(A, D, m, st_{\mathsf{cs}}, \vec{st}_{\mathsf{mg}})$ takes as input the user and device identifiers, a key request message m, the executing device's cross-signing state st_{cs} and their inbound Megolm session store $\vec{st}_{\mathsf{mg}}$. After ensuring the requesting device is a verified device from the same user, it returns a plaintext m containing the requested key and the verified cryptographic identity ipk_E of the requesting device dpk_E. The caller must then encrypt this message over an Olm channel.
- $(t, \vec{st}_{\mathsf{mg}}) \leftarrow \mathsf{KS.Recover}(A, D, E, dpk_*, ipk_*, m, st_{\mathsf{cs}}, \vec{st}_{\mathsf{mg}})$ processes an incoming `MsgFwd` message and, if the sender is a verified device from the same user, saves the inbound Megolm session. It takes as input the executing device's user identifier A and device identifier D, the sending device's identifier E, the cryptographic identity of the device which sent the Olm message (dpk_E, ipk_E), the

decrypted Olm message m containing the key share, the executing device's cross-signing state st_{cs} and their Megolm sessions $\vec{st}_{\mathsf{mg}}$. If the Olm device that sent the message is verified and from the same user, the algorithm will save the session it has received in $\vec{st}_{\mathsf{mg}}$ then return $(\mathbf{true}, \vec{st}_{\mathsf{mg}})$. If the verification fails, it will return $(\mathbf{false}, \vec{st}_{\mathsf{mg}})$ where $\vec{st}_{\mathsf{mg}}$ is unmodified.

We now give a brief description of the Key Request protocol (detailed in Fig. 8).

Requesting keys. The protocol is triggered when a client receives a ciphertext for which they are missing the Megolm session needed to decrypt it. The client generates a `MsgReq` message containing the gpk of the session they are requesting access to and the ipk of its presumed owner (sourced from the metadata of the Megolm ciphertext). It is sent in plaintext to all of the user's verified devices. KS.Request in Fig. 8 describes the process of generating a `MsgReq` message in detail.

Sharing keys. When a device receives a `MsgReq` message, they first determine whether to share the key with the requesting device. Since the standard specifies that Megolm sessions may only be shared between trusted (i.e. verified) devices of the same user, they first verify the identity contained in the `MsgReq` message. If these checks pass, the sharing device packages their copy of the inbound Megolm session in *session export format* [3] inside a `MsgFwd` message [1] (alongside the claimed identity of the session owner) to be sent over Olm. In our formulation of the protocol, KS.Share returns the device identity, (dpk, ipk), of the intended recipeint for the caller to encrypt appropriately. Note that this is the only message in the key request protocol that is sent over Olm. KS.Share in Fig. 8 describes the process of processing a `MsgReq` message in detail.

Receiving keys. Upon receiving a `MsgFwd` message, the requesting client checks that it came from an Olm channel belonging to a verified device of the same user. If so, it is saved to the receiving device's Megolm session store. KS.Recover in Fig. 8 describes the process of receiving a `MsgFwd` message in detail.

2.5. The Matrix Secure Messaging Protocol

The Matrix protocol is a tuple of nine algorithms, Mtx = (Gen, Reg, Init, Recv, Add, Remove, Enc, Dec, ReqKey). These algorithms (detailed in Fig. 9) capture the interactions of a single group, its users and their devices.

User and device setup. The Mtx.Gen algorithm captures the initialisation of a user and their cryptographic identity (described exactly by CS.Init). Mtx.Reg captures the initialisation of a device and its cryptographic identity, then triggers the self-verification process. Before initialising a group, each user must initialise their cryptographic identity along with one or more devices. Devices can be added later but the Matrix protocol does not support device revocation.

Users perform out-of-band verification the first time they interact. This process is modelled via calls to the $\mathsf{VF.User}(st_{\mathsf{cs}}, B)$ algorithm which attempts an out-of-band verification with the device $D^{B,j}$. If successful, it returns the user's mpk_B which is then signed using the CS.SignUser algorithm. Self-verification of new devices is simulated during device registration. In practice, the device holding the cross-signing secrets will execute $\mathsf{VF.Device}(st_{\mathsf{cs}}, B, j)$ which, upon successful verification, returns $dpk_{B,j}$ and $ipk_{B,j}$. Only then will CS.SignDevice be executed and sign the device's identity.[15] See [6, Fig. 11 and 12] for a detailed description of the Short Authentication String (SAS) protocol.

Group initialisation. Mtx.Init is executed by each device in the group. Each device generates a Megolm session which they will use to send messages to the group. The Mtx.Add algorithm must then be executed by each device, for every other device, to initialise the group membership.[16]

Adding devices. To add a device, all existing member's devices execute Mtx.Add, sharing a copy of their inbound Megolm session state with the new device (over an Olm channel). If no Olm channel exists, a new one will be created (after checking whether that the device has been verified).

Removing devices. When a device is removed from the group, the Mtx.Remove algorithm is run by each device in the group. The generate a fresh Megolm session using Mg.Init, then share the resulting inbound Megolm session with the updated list.

In practice, these algorithms are only run when the device next sends a message to the group. This also allows batching of multiple membership changes into one operation, reducing the overall cost. Our description in Mtx.Add and Mtx.Remove does not capture this.

Adding and removing users. Matrix also allows the addition (and removal) of users from the group in a single operation. We capture this through the repeated application of Mtx.Add (and Mtx.Remove, resp.) for each of the user's devices.

Message ordering. Individual Olm and Megolm sessions can define a canonical message ordering on the sender and receiver sides through their key schedules. However, since Matrix allows multiple Olm channels to coexist between a single pair of devices, canonical message ordering in Olm and Megolm sessions does not translate to consistent canonical message ordering within Matrix conversations.[17]

15. In practice, out-of-band verification can be triggered by users at any time. This formalism captures the first time this process occurs in the protocol.

16. It is possible for different devices in the group to have different views of the group membership. This is possible in practice and in our formalism.

17. Our security analysis (and predicates) are defined in terms of the canonical ordering that are observed by the challenger in the experiment.

$\mathsf{Mtx.Reg}(1^\lambda, A, D, st_{\mathsf{cs}})$

$(\mathfrak{D}_{sk}, \mathfrak{D}) \leftarrow \mathsf{MtxOlm.KGen}($
$\quad 1^\lambda, A, D, 10, 1)$
$(st_{\mathsf{cs}}, dt) \leftarrow \mathsf{CS.SignDevice}($
$\quad st_{\mathsf{cs}}, A, D, \mathfrak{D}.dpk, \mathfrak{D}.ipk)$
return $(\mathfrak{D}, dt),\ (\mathfrak{D}_{sk}, st_{\mathsf{cs}})$

$\mathsf{Mtx.Init}(1^\lambda, A, D, \mathfrak{U}, \mathfrak{D}, st_{\mathsf{cs}}, \mathfrak{D}_{sk}, G)$

$\vec{st}_{\mathsf{mg}} \leftarrow []$; $\vec{st}_{\mathsf{olm}} \leftarrow []$
$CU \leftarrow \{A\}$; $CD \leftarrow \{(D, \mathfrak{D}.dpk, \mathfrak{D}.ipk)\}$
$(\mathfrak{S}_{gsk}, \mathfrak{S}_{gpk}, \sigma_{\mathsf{mg}}) \leftarrow\!\!\$\ \mathsf{Mg.Init}(1^\lambda)$
$\vec{st}_{\mathsf{mg}}[gpk, \mathfrak{D}.ipk] \leftarrow (\mathfrak{S}_{gpk}, \mathfrak{D}.dpk, \textbf{false})$
$st_{\mathsf{mt}} \leftarrow (G, A, D, CU, CD, \mathfrak{U}, \mathfrak{D}, st_{\mathsf{cs}},$
$\quad dsk, isk, esk, fsk, \mathfrak{S}_{gsk}, \vec{st}_{\mathsf{mg}}, \vec{st}_{\mathsf{olm}})$
return (st_{mt}, G)

$\mathsf{Mtx.Add}(st_{\mathsf{mt}}, B, E, dpk_*, ipk_*)$

if $(B \notin st_{\mathsf{mt}}.CU)$:
$\quad mpk^* \leftarrow \mathsf{VF.User}(B)$
$\quad$ **if** $(mpk^* = \perp)$: **return** $(st_{\mathsf{mt}}, \perp)$
$\quad (st_{\mathsf{mt}}.st_{\mathsf{cs}}, \_) \leftarrow$
$\quad\quad \mathsf{CS.SignUser}(st_{\mathsf{mt}}.st_{\mathsf{cs}}, B, mpk^*)$
$\quad st_{\mathsf{mt}}.CU \leftarrow st_{\mathsf{mt}}.CU \cup \{B\}$
$(B, E, dpk^*) \leftarrow \mathsf{CS.VerifyDevice}(ipk^*)$
if $(B, E) \in \{(\perp, \perp),$
$\quad (st_{\mathsf{mt}}.A, st_{\mathsf{mt}}.D)\}$: **return** $(\perp, \perp)$
$(ver, i, R, gsk, gpk) \leftarrow st_{\mathsf{mt}}.\mathfrak{S}_{gsk}$
$\mathfrak{S}_{gpk} \leftarrow (ver, i, R, gpk)$
$\sigma_{\mathsf{mg}} \leftarrow \mathsf{DS.Sign}(gsk, \mathfrak{S}_{gpk})$
$m_w \leftarrow (\mathtt{MgAlg}, G, \mathfrak{S}_{gpk}, \sigma_{\mathsf{mg}})$
$(st_{\mathsf{mt}}, c_w) \leftarrow \mathsf{MtxOlm.Enc}($
$\quad\quad st_{\mathsf{mt}}, dpk^*, ipk^*, m_w)$
$st_{\mathsf{mt}}.CD \leftarrow st_{\mathsf{mt}}.CD \cup \{(E, dpk^*, ipk^*)\}$
return (st_{mt}, c_w)

$\mathsf{Mtx.Remove}(st_{\mathsf{mt}}, B, E, dpk_*, ipk_*)$

$st_{\mathsf{mt}}.CD \leftarrow st_{\mathsf{mt}}.CD \setminus \{(E, dpk^*, ipk^*)\}$
if $(\nexists F : (B, F, \_, \_) \in st_{\mathsf{mt}}.CD)$:
$\quad st_{\mathsf{mt}}.CU \leftarrow st_{\mathsf{mt}}.CU \setminus \{B\}$
$(\mathfrak{S}_{gsk}, \mathfrak{S}_{gpk}, \sigma_{\mathsf{mg}}) \leftarrow\!\!\$\ \mathsf{Mg.Init}(1^\lambda)$
$st_{\mathsf{mt}}.\vec{st}_{\mathsf{mg}}[gpk, \mathfrak{D}.ipk] \leftarrow (\mathfrak{S}_{gpk}, \mathfrak{D}.dpk, \textbf{false})$
$st_{\mathsf{mt}}.\mathfrak{S}_{gsk} \leftarrow \mathfrak{S}_{gsk}$
$cs \leftarrow \{\}$; **for** (C, F) **in** $st_{\mathsf{mt}}.CD$
$\quad (st_{\mathsf{mt}}, c) \leftarrow \mathsf{Mtx.Add}(st_{\mathsf{mt}}, ipk_{C,F})$
$\quad cs \leftarrow cs \cup \{c\}$
return (st_{mt}, cs)

$\mathsf{Mtx.ReqKey}(st_{\mathsf{mt}}, gpk, ipk)$

$(ds, m) \leftarrow \mathsf{KS.Request}(st_{\mathsf{mt}}.A,$
$\quad st_{\mathsf{mt}}.D, st_{\mathsf{mt}}.CD, gpk, ipk, st_{\mathsf{mt}}.st_{\mathsf{cs}})$
return $(st_{\mathsf{mt}}, ds, m)$

$\mathsf{Mtx.Recv}(st_{\mathsf{mt}}, dpk_s, ipk_s, m)$

$(alg, G, \mathfrak{S}^*_{gpk}, \sigma_{\mathsf{mg}}) \leftarrow m$
if $(alg \neq \mathtt{MgAlg}) \vee (G \neq st_{\mathsf{mt}}.G)$
$\quad$: **return** $(st_{\mathsf{mt}}, \perp)$
$\mathfrak{S}^*_{gpk} \leftarrow \mathsf{Mg.Recv}(\mathfrak{S}^*_{gpk}, \sigma_{\mathsf{mg}})$
if $(\mathfrak{S}^*_{gpk} = \perp)$: **return** $(st_{\mathsf{mt}}, \perp)$
$(\mathfrak{S}_{gpk}, dpk_s', F) \leftarrow st_{\mathsf{mt}}.\vec{st}_{\mathsf{mg}}[\mathfrak{S}^*_{gpk}.gpk, ipk_s]$
if $(\mathfrak{S}_{gpk} \neq \perp) \wedge (\mathfrak{S}_{gpk}.i \leq \mathfrak{S}^*_{gpk}.i)$
$\quad$: **return** $(st_{\mathsf{mt}}, \perp)$
$st_{\mathsf{mt}}.\vec{st}_{\mathsf{mg}}[\mathfrak{S}^*_{gpk}.gpk, ipk_s] \leftarrow (\mathfrak{S}^*_{gpk}, dpk_s, F)$
return $(st_{\mathsf{mt}}, \textbf{true})$

$\mathsf{Mtx.Enc}(st_{\mathsf{mt}}, m)$

$ipk \leftarrow st_{\mathsf{mt}}.\mathfrak{D}.ipk$; $gpk \leftarrow st_{\mathsf{mt}}.\mathfrak{S}_{gsk}.gpk$
$(st_{\mathsf{mt}}.\mathfrak{S}_{gsk}, c) \leftarrow \mathsf{Mg.Enc}($
$\quad st_{\mathsf{mt}}.\mathfrak{S}_{gsk}, (\mathtt{MsgPln}, st_{\mathsf{mt}}.G, m))$
$c_w \leftarrow (\mathtt{MsgEnc}, \mathtt{MgAlg}, gpk, ipk, c)$
return (st_{mt}, c_w)

$\mathsf{Mtx.Dec}(st_{\mathsf{mt}}, c_w)$

$(t, a, c_w) \leftarrow c_w$
if $(t = \mathtt{MsgEnc}) \wedge (a = \mathtt{MgAlg})$:
$\quad (gpk_s, ipk_s, c) \leftarrow c_w$
$\quad (\mathfrak{S}_{gpk}, dpk_s, F) \leftarrow st_{\mathsf{mt}}.\vec{st}_{\mathsf{mg}}[gpk_s, ipk_s]$
$\quad$ **if** $(\mathfrak{S}_{gpk} = \perp)$: **return** $(\perp, \perp)$
$\quad \_, (t, G_s, m_w) \leftarrow \mathsf{Mg.Dec}(\mathfrak{S}_{gpk}, c)$
$\quad$ **if** $(G_s \neq st_{\mathsf{mt}}.G)$: $(\perp, \perp)$
$\quad (t, m) \leftarrow m_w$
$\quad$ **if** $(t = \mathtt{MsgPln})$:
$\quad\quad$ **return** (st_{mt}, m)
elseif $(t = \mathtt{MsgEnc}) \wedge (a = \mathtt{OlmAlg})$:
$\quad (st_{\mathsf{mt}}, ipk_s, m_w)$
$\quad\quad \leftarrow \mathsf{MtxOlm.Dec}(st_{\mathsf{mt}}, c_w)$
$\quad (B_{in}, A_{in}, dpk_{s,in}, dpk_{r,in}, t, m_w) \leftarrow m_w$
$\quad (B, D, dpk_s) \leftarrow \mathsf{CS.VerifyDevice}(ipk_s)$
$\quad$ **if** $(dpk_{r,in} \neq st_{\mathsf{mt}}.\mathfrak{D}.dpk \ \vee B_{in} \neq B \vee$
$\quad\quad dpk_{s,in} \neq dpk_s)$: **return** $(\perp, \perp)$
$\quad$ **if** $(t = \mathtt{MsgKey})$:
$\quad\quad$ **return** $\mathsf{Mtx.Recv}(st_{\mathsf{mt}}, dpk_s, ipk_s, m)$
$\quad$ **elseif** $(t = \mathtt{MsgFwd})$:
$\quad\quad (accept, \vec{st}_{\mathsf{mg}}) \leftarrow \mathsf{KS.Recover}($
$\quad\quad\quad st_{\mathsf{mt}}.A, st_{\mathsf{mt}}.D, D, dpk_s, ipk_s,$
$\quad\quad\quad m, st_{\mathsf{cs}}, st_{\mathsf{mt}}.\vec{st}_{\mathsf{mg}})$
$\quad\quad$ **if** $(accept)$: $st_{\mathsf{mt}}.\vec{st}_{\mathsf{mg}} \leftarrow \vec{st}_{\mathsf{mg}}$
$\quad\quad$ **return** $(st_{\mathsf{mt}}, accept)$
elseif $(t = \mathtt{MsgReq})$:
$\quad m \leftarrow c_w$
$\quad (dpk_s, ipk_s, m) \leftarrow \mathsf{KS.Share}($
$\quad\quad st_{\mathsf{mt}}.A, st_{\mathsf{mt}}.D, m, st_{\mathsf{mt}}.st_{\mathsf{cs}},$
$\quad\quad st_{\mathsf{mt}}.\vec{st}_{\mathsf{mg}})$
$\quad (st_{\mathsf{mt}}, c_w) \leftarrow \mathsf{MtxOlm.Enc}(st_{\mathsf{mt}},$
$\quad\quad dpk_s, ipk_s, m_w)$
$\quad$ **return** (st_{mt}, c_w)
return $(\perp, \perp)$

Figure 9: Pseudocode describing secure group messaging in Matrix.

Key sharing. When a device receives a ciphertext that it is unable to decrypt, they may initiate the key request protocol using the Mtx.ReqKey algorithm. The device executes $\mathsf{Mtx.ReqKey}(st_{\mathsf{mt}}, gpk, ipk)$ using the gpk and ipk from the ciphertext they failed to decrypt. This, in turn, starts an instance of the key request protocol by calling KS.Request.

3. Device-Oriented Group Messaging

A Device-Oriented Group Messaging (DOGM) protocol is a tuple of algorithms $\mathsf{DOGM} = (\mathsf{Gen}, \mathsf{Reg}, \mathsf{Init}, \mathsf{Add}, \mathsf{Remove}, \mathsf{Encrypt}, \mathsf{Decrypt})$. In addition, a group messaging protocol may have additional functionality for state recovery: StateShare. We define a DOGM protocol over sets $\mathcal{P}_k, \mathcal{S}_k, \mathcal{G}_{id}, \mathcal{D}_{id}, \mathcal{U}_{id}, \mathcal{ST}, \mathcal{C}, \mathcal{M}$ where $\mathcal{P}_k$ and $\mathcal{S}_k$ are the public and secret authenticator spaces, $\mathcal{G}_{id}, \mathcal{D}_{id}, \mathcal{U}_{id}$ are the group, device and user identifier spaces, respectively, $\mathcal{ST}$ is the space of sessions' local secret states, $\mathcal{C}$ is the space of protocol ciphertexts, and $\mathcal{M}$ is the plaintext message space.

Algorithms Gen and Reg generate static authentication values used across multiple protocol executions:

- $\mathsf{Gen} : \mathbb{N} \times \mathcal{U}_{id} \xrightarrow{\$} \mathcal{P}_k \times \mathcal{S}_k$ – takes as input a security parameter and user identifier and outputs public and secret authenticator values. In Matrix, this corresponds to calling $\mathsf{Mtx.Init} = \mathsf{CS.Init}$ outputting $(\mathfrak{U}, st_{\mathsf{cs}})$.
- $\mathsf{Reg} : \mathbb{N} \times \mathcal{U}_{id} \times \mathcal{D}_{id} \times \mathcal{S}_k \xrightarrow{\$} \mathcal{P}_k \times \mathcal{S}_k$ – takes as input a security parameter, user identifier, device identifier and secret authenticator value and outputs public and secret authenticator values. In Matrix, this corresponds to Mtx.Reg

$\mathsf{MtxOlm.KGen}(st_{\mathsf{mt}}, 1^\lambda, A, D, n_e, n_f)$

$(dsk, dpk) \leftarrow\!\$\ \mathsf{DS.KGen}(1^\lambda)$
$((isk, esk, fsk),\ (ipk, epk, fpk)) \setminus$
$\quad \leftarrow\!\$\ \mathsf{Olm.KGen}(1^\lambda, n_e, n_f)$
$m_d \leftarrow (D, \mathtt{DvSign}, dpk, \mathtt{DvIdent}, ipk)$
$m_e \leftarrow \{(\mathtt{DvEphm}, epk_k) : epk_k \in epk\}$
$m_f \leftarrow \{(\mathtt{DvFall}, fpk_k) : fpk_k \in fpk\}$
$\sigma_d \leftarrow \mathsf{DS.Sign}(dsk, m_d)$
$\sigma_e \leftarrow \{\mathsf{DS.Sign}(dsk, m_{e,k}) : m_{e,k} \in m_e\}$
$\sigma_f \leftarrow \{\mathsf{DS.Sign}(dsk, m_{f,k}) : m_{f,k} \in m_f\}$
$\mathfrak{D} \leftarrow (m_d, \sigma_d, m_e, \sigma_e, m_f, \sigma_f)$
$\mathfrak{D}_{sk} \leftarrow (dsk, isk, esk, fsk)$
return $(\mathfrak{D}_{sk}, \mathfrak{D})$

$\mathsf{MtxOlm.Enc}(st_{\mathsf{mt}}, dpk_r, ipk_r, m_w)$

if $(st_{\mathsf{mt}}.\vec{st}_{\mathsf{olm}}[ipk_r] = \perp)$: $t = 0$
  if !$\mathsf{DS.Verify}(dpk_r, \mathfrak{D}_r.\sigma_d, (D, \mathtt{DvSign},$
      $dpk_r, \mathtt{DvIdent}, ipk_r))$:
    return $(\perp, \perp)$
  if !$\mathsf{DS.Verify}(dpk_r, \mathfrak{D}_r.\sigma_{e,k},$
      $(\mathtt{DvEphm}, \mathfrak{D}_r.m_{e,k}.epk))$:
    return $(\perp, \perp)$
  $(epk_r, epk) \leftarrow (m_e.epk, \perp)$
  $st_{\mathsf{olm}} \leftarrow \mathsf{Olm.Enc}(\perp, m_w,$
    $st_{\mathsf{mt}}.isk, ipk_r, m_{e,k}.epk)$
else: $t = 1$
  $(epk_r, epk, st_{\mathsf{olm}})$
    $\leftarrow \mathsf{MRU}(st_{\mathsf{mt}}.\vec{st}_{\mathsf{olm}}[ipk_r])$
  $(st_{\mathsf{olm}}, c) \leftarrow \mathsf{Olm.Enc}(st_{\mathsf{olm}}, m_w)$
$st_{\mathsf{mt}}.\vec{st}_{\mathsf{olm}}[ipk_r][epk_r, epk] \leftarrow st_{\mathsf{olm}}$
$c_w \leftarrow (\mathtt{MsgEnc}, \mathtt{OlmAlg}, st_{\mathsf{mt}}.\mathfrak{D}.ipk, t, c)$
return (st_{mt}, c_w)

$\mathsf{MtxOlm.Dec}(st_{\mathsf{mt}}, c_w)$

$(ipk_s, t, c) \leftarrow c_w$
if $(t = 0)$:
  $(ipk'_s, epk_s, epk_k, v, rpk_s, q, x, \tau) \leftarrow c$
  if $(ipk'_s \neq ipk_s)$: **return** $(\perp, \perp)$
  $st_{\mathsf{olm}} \leftarrow st_{\mathsf{mt}}.\vec{st}_{\mathsf{olm}}[ipk_s][epk_s, epk_k]$
  if $(st_{\mathsf{olm}} = \perp)$:
    if $(q \neq 0)$: **return** $(\perp, \perp)$
    $(st_{\mathsf{olm}}, m_w) \leftarrow \mathsf{Olm.Dec}($
      $st_{\mathsf{olm}}, c, st_{\mathsf{mt}}.isk, st_{\mathsf{mt}}.esk_k)$
    $st_{\mathsf{mt}}.esk_k \leftarrow \perp$
  else: $(st_{\mathsf{olm}}, m_w) \leftarrow \mathsf{Olm.Dec}(st_{\mathsf{olm}}, c)$
elseif $(t = 1)$:
  for $(epk_s, epk_k, st_{\mathsf{olm}})$ **in** $st_{\mathsf{mt}}.\vec{st}_{\mathsf{olm}}[ipk_s]$
    $(st_{\mathsf{olm}}, m_w) \leftarrow \mathsf{Olm.Dec}(st_{\mathsf{olm}}, c)$
    if $(m_w \neq \perp)$ **break**
if $(m_w = \perp)$ **return** $(\perp, \perp)$
$st_{\mathsf{mt}}.\vec{st}_{\mathsf{olm}}[ipk_s][epk_s, epk_k] \leftarrow st_{\mathsf{olm}}$
return $(st_{\mathsf{mt}}, ipk_s, m_w)$

Figure 10: The MtxOlm algorithms form a wrapper around the Olm protocol, customised for Matrix' inclusion of signed ephemeral key pairs and multiple Olm channels.

outputting $(\mathfrak{D}, dt)$ and $(\mathfrak{D}_{sk}, st_{\mathsf{cs}})$.

The following set of algorithms define the group messaging protocol:

- Init : $\mathcal{U}_{id} \times \mathcal{D}_{id} \times \rho \times \mathcal{S}_k(\times\mathcal{G}_{id}) \xrightarrow{\$} \mathcal{ST} \times \mathcal{G}_{id}$ – takes as input a user identifier, a device identifier, a secret authenticator value, a role, and (optionally) group identifier when instantiating a session for an existing group. It outputs secret session state and the group identifier. In Matrix, this corresponds to first calling Mtx.Init outputting (st_{mt}, G) for the sender and calling Mtx.Add to add each receiving device. The final output is (st_{mt}, G). Receiving devices initialise the group by decrypting the Olm message with Mtx.Dec then passing the inbound session to Mtx.Recv.
- Add : $\mathcal{S}_k \times \mathcal{ST} \times \mathcal{U}_{id} \times \mathcal{D}_{id}(\times\mathcal{C}) \xrightarrow{\$} \mathcal{ST}(\times\mathcal{C}) \cup \{\perp\}$ – takes as input a secret authenticator, secret state, user identifier, device identifier and (potentially) a ciphertext, and outputs updated state and (potentially) a ciphertext, or failure $\perp$. In Matrix, this calls Mtx.Add.
- Remove : $\mathcal{S}_k \times \mathcal{ST} \times \mathcal{U}_{id} \times \mathcal{D}_{id}(\times\mathcal{C}) \to \mathcal{ST}(\times\mathcal{C}) \cup \{\perp\}$ – takes as input a secret authenticator, secret state, user identifier, device identifier and (potentially) a ciphertext. It outputs updated state and (potentially) a ciphertext, or failure $\perp$. In Matrix, this calls Mtx.Remove.
- Encrypt : $\mathcal{S}_k \times \mathcal{ST} \times \mathcal{M} \xrightarrow{\$} \mathcal{ST} \times \mathcal{C} \cup \{\perp\}$ – takes as input a secret authenticator value, secret state and a plaintext, and outputs an updated state and a ciphertext, or failure $\perp$. In Matrix, this encrypts plaintexts to the group via Mtx.Enc.
- Decrypt : $\mathcal{S}_k \times \mathcal{ST} \times \mathcal{C} \xrightarrow{\$} \mathcal{ST}(\times\mathcal{M}) \cup \{\perp\}$ – takes as input a secret authenticator, secret state and a ciphertext. It outputs an updated state and a plaintext message, or failure. In Matrix, this decrypts ciphertexts received by the session via Mtx.Dec.

We introduce the StateShare protocol that allows adding state sharing functionality to DOGM protocols:

- StateShare : $\mathcal{S}_k \times \mathcal{ST}(\times\mathcal{U}_{id})(\times\mathcal{D}_{id})(\times\mathcal{C})(\times\mathbb{N}) \xrightarrow{\$} \mathcal{ST}(\times\mathcal{C})$ – takes as input a secret authenticator, secret state and (optionally) a user identifier, device identifier, ciphertext and a session stage index. It outputs an updated state and (potentially) a ciphertext, or a failure state $\perp$. In Matrix, this uses Mtx.ReqKey to initiate the key request protocol.

3.1. Execution Environment

We now describe the DOGM execution environment. Consider $\mathsf{Exp}^{\mathsf{IND\text{-}CCA},\mathcal{A}}_{\mathsf{DOGM},n_P,n_D,n_I,n_S,n_M}(1^\lambda)$ played between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. The challenger $\mathcal{C}$ maintains a set of n_P parties $P_1, \ldots, P_{n_P} \in \mathcal{U}_{id}$ (representing users interacting with each other via the DOGM protocol). A maximum of n_D devices can be created for each party P_A, identified by $D^{A,1}, \ldots, D^{A,n_D} \in \mathcal{D}_{id}$. Each device can run n_I sessions of a probabilistic protocol DOGM, across n_S different stages, with each stage consisting of up to n_M messages. We use $\pi^s_{A,i}$ to refer both to the identifier of the s-th instance of the DOGM being run by A's device $D^{A,i}$ and the collection of per-session variables $\pi^s_{A,i}$ maintains:

- $A \in \mathcal{U}_{id}$ – the User identifier for this party.
- $D^{A,i} \in \mathcal{D}_{id}$ – the Device identifier for this party.
- $G \in \mathcal{G}_{id}$ – the Group identifier for this session.
- $\rho \in \{\mathtt{send}, \mathtt{recv}\}$ – the role of the party in the current session. Note that parties can be directed to act as either a `send` or `recv` in concurrent or subsequent sessions.
- $t \in \mathbb{N}$ – the current stage of the session.
- $z \in \mathbb{N}$ – the current message index of the current stage.

• $\alpha \in \{\perp, \texttt{active}, \texttt{reject}\}$ – the status of the session, initialised by $\perp$.
• $CU[0], \ldots, CU[n_P] \in \mathcal{U}_{id}$ – the current set of intended communication partners, where $CU[0]$ is the sending User.
• $CD[0,0], \ldots, CD[0,n_D], \ldots CD[n_P,n_D] \in \mathcal{D}_{id}$ – the current set of devices associated with the communication partners $CU[0] \ldots CU[n_P]$, where $CD[0]$ is the sending Device.
• $T[t,z] \in \mathcal{C}^* \cup \{\perp\}$ – the z-th message sent in the t-th stage sent or received by $\pi^s_{A,i}$. We use $|T[t]|$ as the shorthand for the first value z such that $T[t,z] = \perp$, or the number of messages accepted in the t-th stage.[18]
• $st \in \{0,1\}^*$ – any additional state used by the session during protocol execution.

Recall that DOGM is a tuple of algorithms $\mathsf{DOGM} = \{\mathsf{Gen}, \mathsf{Reg}, \mathsf{Init}, \mathsf{Add}, \mathsf{Remove}, \mathsf{Encrypt}, \mathsf{Decrypt}\}$ with additional functionality $\{\mathsf{StateShare}\}$.

The experiment begins with $\mathcal{C}$ running $\mathsf{DOGM.Gen}(1^\lambda, A)$ n_P times to generate a public key pair $(\mathsf{pk}_A, \mathsf{sk}_A)$ for each party $A \in \{P_1, \ldots, P_{n_P}\}$ and delivers all public-keys pk_A to $\mathcal{A}$. $\mathcal{A}$ can now issue $\mathsf{CorruptUser}$ queries listed in Section 3.2. After, $\mathcal{C}$ randomly samples a bit $b \xleftarrow{\$} \{0,1\}$ used to generate challenge ciphertexts, sets a flag $\mathsf{win} \leftarrow \texttt{false}$, and interacts with $\mathcal{A}$ via the queries listed in Section 3.2 (except $\mathsf{CorruptUser}$). $\mathcal{C}$ maintains challenge ciphertexts in the set $\mathbf{C}$ through the experiment.

Eventually, $\mathcal{A}$ terminates and outputs a guess b' of the challenger bit b. $\mathcal{A}$ wins the DOGM experiment if $b' = b$, and the confidentiality predicate CONF is satisfied. $\mathcal{A}$ may also win the DOGM experiment if win has been set to $\texttt{true}$ by a call to Decrypt. Before setting $\mathsf{win} \leftarrow \texttt{true}$ $\mathcal{C}$ checks that the experiment satisfies the authenticity predicate AUTH. To indicate that $\mathcal{A}$ has won, $\mathcal{C}$ immediately terminates the experiment and returns 1.

3.2. Adversarial Model

We now define how the adversary in the DOGM security experiment may interact with the challenger; in turn, describing how an attacker may interact with sessions of the Matrix protocol. These interactions define our adversarial model. The adversary is in complete control of the communication network – able to modify, inject, delete or delay messages – and may additionally *compromise* secrets at three levels:

1) Adaptive compromise of the current session state.
2) Adaptive compromise of a device's long-term keys.
3) Non-adaptive compromise of a user's long-term keys.

The first models state-compromising attacks (such as temporary device access). The latter two capture key misuse, as well as state-compromising attacks. This enables the model to capture a nuanced understanding of PCS and FS.

The adversary interacts with $\mathcal{C}$ via the queries below.

18. We note that this assumes that it is possible for the challenger to define a single global ordering of messages. However, this global ordering needs only exist for the model: it does not have to be possible for the receiving session to recover this ordering.

• $\mathsf{Create}(A,i) \rightarrow \{dpk_{A,i}, \perp\}$: allows $\mathcal{A}$ to create new devices. $\mathcal{C}$ creates a new device $D^{A,i}$ owned by A, and runs $\mathsf{DOGM.Reg}(1^\lambda, A, D^{A,i}, \mathsf{sk}_A) \xrightarrow{\$} (dpk_{A,i}, dsk_{A,i})$. If a device $D^{A,i}$ has already been created, $\mathcal{C}$ returns $\perp$, otherwise, $dpk_{A,i}$.

• $\mathsf{Init}(A,i,\rho,(G)) \rightarrow \{(s,(gid)),(\perp)\}$: allows $\mathcal{A}$ to initiate a session $\pi^s_{A,i}$ for device $D^{A,i}$ owned by A. It initiates $\pi^s_{A,i}$, i.e. $\mathcal{C}$ runs $\mathsf{DOGM.Init}(A, D^{A,i}, \rho, (\mathsf{sk}_A, dsk_{A,i}), (G)) \xrightarrow{\$} \pi^s_{A,i}$. If there already exists a session $\pi^s_{B,j}$ such that $\pi^s_{B,j}.\rho = \texttt{send}$, $\pi^s_{A,i}.G = \pi^s_{B,j}.G$ and $\pi^s_{A,i}.\rho = \texttt{recv}$, then $\pi^s_{A,i}.CU \leftarrow \pi^s_{B,j}.CU$ and $\pi^s_{A,i}.CD \leftarrow \pi^s_{B,j}.CD$.

• $\mathsf{AddMember}(A,i,s,B,j) \rightarrow \{c,\perp\}$: allows $\mathcal{A}$ to direct session $\pi^s_{A,i}$ to add a new device $D^{B,j}$ owned by party B to their group messaging session. $\mathcal{C}$ runs $\mathsf{DOGM.Add}((\mathsf{sk}_{A,i}, dsk_{A,i}), \pi^s_{A,i}.st, B, D^{B,j}, \perp) \rightarrow (\pi^s_{A,i}.st, c)$ and returns c to $\mathcal{A}$.

• $\mathsf{RemoveMember}(A,i,s,B,j) \rightarrow \{c,\perp\}$: allows $\mathcal{A}$ to direct session $\pi^s_{A,i}$ to remove device $D^{B,j}$ owned by party B from their group messaging session. $\mathcal{C}$ runs $\mathsf{DOGM.Remove}((\mathsf{sk}_{A,i}, dsk_{A,i}), \pi^s_{A,i}.st, B, D^{B,j}, \perp) \rightarrow (\pi^s_{A,i}.st, c)$ and returns c to $\mathcal{A}$.

• $\mathsf{CorruptUser}(A) \rightarrow \{\mathsf{sk}_A, \perp\}$: allows $\mathcal{A}$ access to the secret long-term key generated by a party A. $\mathcal{C}$ returns sk_A to $\mathcal{A}$.

• $\mathsf{CorruptDevice}(A,i) \rightarrow \{dsk_{A,i}, \perp\}$: allows $\mathcal{A}$ access to the secret long-term device key $dsk_{A,i}$ generated upon device creation. $\mathcal{C}$ returns $dsk_{A,i}$ to $\mathcal{A}$.

• $\mathsf{Compromise}(A,i,s) \rightarrow \{\pi^s_{A,i}, \perp\}$: allows $\mathcal{A}$ to compromise the current session state of $\pi^s_{A,i}$. [19]

• $\mathsf{Encrypt}(A,i,s,m_0,m_1) \rightarrow \{c,\perp\}$: allows $\mathcal{A}$ to encrypt messages from the session $\pi^s_{A,i}$, depending on the challenge bit b. Let $t \leftarrow \pi^s_{A,i}.t$ and $z \leftarrow |\pi^s_{A,i}.T[t]|$. If $|m_0| \neq |m_1|$, then $\mathcal{C}$ returns $\perp$. Else, $\mathcal{C}$ computes $\mathsf{DOGM.Encrypt}((\mathsf{sk}_A, dsk_{A,i}), \pi^s_{A,i}.st, m_b) \rightarrow (\pi^s_{A,i}.st', c)$ then adds $T[\pi^s_{A,i}.t, \pi^s_{A,i}.z] \leftarrow c$. When $m_0 \neq m_1$, then c is a challenge ciphertext and $\mathbf{C} \leftarrow c$. $\mathcal{C}$ returns c to $\mathcal{A}$.

• $\mathsf{Decrypt}(A,i,s,c) \rightarrow \{m',\perp\}$: allows $\mathcal{A}$ to direct session $\pi^s_{A,i}$ to process a message and receive its output. $\mathcal{C}$ computes $\mathsf{DOGM.Exec}((\mathsf{sk}_A, dpk_{A,i}), \pi^s_{A,i}.st, c) \rightarrow (\pi^s_{A,i}.st', m')$ (where $\mathsf{Exec} \in \{\mathsf{Add}, \mathsf{Remove}, \mathsf{Decrypt}, \mathsf{StateShare}\}$). This corresponds to $\mathcal{C}$ calling Mtx.Dec with the given ciphertext. Let $t^* \leftarrow \pi^s_{A,i}.t$ and $z^* \leftarrow |\pi^s_{A,i}.T[t]|$. We say that c was honest if and only if there exists a session $\pi^s_{B,j}$ where $\pi^s_{B,j}.\rho = \texttt{send}$ and $\pi^s_{A,i}.T[t^*,z^*] = \pi^s_{B,j}.T[t^*,z^*] = c$.[20] If c was decrypted by $\pi^s_{A,i}$ (i.e. $m' \neq \perp$), and c was not honest,

19. For Matrix, we model compromise of the current session state as giving the adversary access to all the Megolm session states that this protocol execution has access to. In particular, $\mathsf{Compromise}(A,i,s)$ provides the adversary with the contents of $\pi^s_{A,i}.st.\vec{st}_{\mathsf{mg}}$ and $\pi^s_{A,i}.st.\mathfrak{S}_{gsk}$. It does not provide access to the Olm sessions states in $\pi^s_{A,i}.st.\vec{st}_{\mathsf{olm}}$.

20. Since the challenger defines the canonical ordering in our analysis of Matrix, they check that the ratchet used by the receiver matches the one used to encrypt the message (and that the message index matches).

then $\mathcal{A}$ forged the ciphertext c. Thus, $\mathcal{C}$ sets win $\leftarrow$ `true` and returns 1. Otherwise, $\mathcal{C}$ sets $\pi^s_{A,i}.st \leftarrow \pi^s_{A,i}.st'$ and, if $c \notin \mathbf{C}$, returns m' to $\mathcal{A}$, otherwise $\mathcal{C}$ returns $\perp$.

• $\mathsf{StateShare}(A, i, s, t, B, j, c) \rightarrow \{c', \perp\}$: allows $\mathcal{A}$ to direct session $\pi^s_{A,i}$ to share their group state with session $\pi^s_{B,j}$ on device $D^{B,j}$ owned by party P_B. $\mathcal{C}$ runs $\mathsf{DOGM.StateShare}((\mathsf{sk}_A, dsk_{A,i}), \pi^s_{A,i}.st, B, j, c, t) \rightarrow (\pi^s_{A,i}, c')$ and c' is returned to $\mathcal{A}$.

3.3. Predicates

Since some adversarial queries allow the adversary to compromise secrets, this can lead to wins in the DOGM security experiment that should not necessarily be considered breaks of the protocol. In the case of Matrix, for example, the adversary could use the $\mathsf{Compromise}(A, i, s)$ query to directly access the ratchet R currently being used by $\pi^s_{A,i}$ to encrypt messages. If the adversary then calls $c \leftarrow \mathsf{Encrypt}(A, i, s, m_0, m_1)$, they may now use the ratchet R to decrypt the ciphertext c, recover m_b and return b, thus winning the experiment due to the correctness of Megolm.

DOGM provides two predicates, AUTH and CONF, that we use to capture trivial wins in the experiment. The AUTH predicate is checked before the game accepts an authentication win (as part of a Decrypt call). The CONF predicate is checked before the game accepts a confidentiality win (i.e. when the adversary correctly guesses b at the end of the game). If either of these fail (the predicates are not fulfilled), then the experiment is aborted.

This ensures the adversary cannot gain an advantage through expected protocol behaviour. These predicates encode the limitations of the analysed protocol and, thus, should be interpreted as part of the security result. In particular, these predicates work in tandem with the adversarial model in the DOGM security experiment: the former define the guarantees that the protocol achieves under the adversarial interactions defined by the latter. Note that DOGM protocols with different security properties will restrict $\mathcal{A}$ differently; we detail Matrix-specific predicates next.

4. Security Analysis of Matrix

Section 4.1 initiates our security analysis of the Matrix protocol by defining the MTXCONF and MTXAUTH predicates, accompanied by intuitive descriptions of the cases they cover. These predicates, in combination with the adversarial interactions defined in Section 3.2, specify the confidentiality and authentication guarantees of the Matrix protocol (that our results capture). Section 4.2 presents our primary security statement, Theorem 1, accompanied by a proof sketch.

4.1. Trivial Attacks in Matrix

We now enumerate the situations within the Matrix protocol where the adversary is expected to be able to decrypt or forge a ciphertext. We separate our analysis into two cases: wins resulting from 1) an expected confidentiality break, and 2) an expected authentication break. For each case, we describe a set of attacks against the protocol and the situations in which they could occur. These are formalised in MTXCONF and MTXAUTH, resp.

Some of the trivial attacks we discuss in this section are the result of the adversary directly compromising state needed to break the respective security goal. Others represent limitations of the security guarantees the Matrix protocol provides. For example, many exploit Matrix' KS feature, which essentially removes all boundaries between a user and their verified devices and between verified devices of a user. Other attacks exploit the virtual absence of PCS or FS guarantees (between user devices), since Matrix allows the establishment of fresh Olm channels using long-term authentication keys, and old key material is not deleted. In Section 5, we discuss the trade-offs these security predicates represent, and how they may be interpreted in a real-world context.

The first set of attacks target confidentiality, in the context where $\pi^s_{A,i}$ has encrypted the challenge ciphertext for which $\pi^s_{B,j}$ is an intended recipient, and are captured by MTXCONF. In particular, $c = \mathsf{Encrypt}(A, i, s, m_0, m_1)$ is a challenge ciphertext and gpk' is the matching public key held by the recipient, identifying the session that c corresponds to.

1) $\mathsf{CorruptUser}(B)$**:** If the recipient user, B, has had their long-term secrets compromised, the adversary can generate a new device identity, sign it, and request access to all the inbound Megolm sessions that B has access to (through the Key Request protocol). In particular, by assumption there exists some device $D^{B,j}$ that can decrypt c. Thus, using the long-term secrets recovered by corrupting B, the adversary creates and signs a new device $D^{B,k}$ under its control. It then establishes new pairwise Olm channels with $D^{B,j}$ of the user and submits a `MsgReq` message for gpk'.
2) $\mathsf{CorruptDevice}(B, k)$**:** Similarly, if the adversary gains access to the long-term secrets of one of B's devices (including when $k = j$), they can use these to initiate new Olm sessions with B's other devices then use the Key Request protocol to gain access to all the inbound Megolm sessions that B has access to. In particular, assume some device $D^{B,j}$ exists that can decrypt c. While $\mathsf{CorruptDevice}(B, k)$ does not recover the Olm states st_{olm} that $D^{B,k}$ uses to communicate with $D^{B,j}$, it recovers the long-term device secrets (dsk, isk) used to create fresh Olm channels between devices (Matrix allows multiple parallel Olm channels to coexist). At session setup $\mathsf{Olm.Enc}(st_{\mathsf{olm}}, m, isk, esk, ipk_{B,j}, epk_{B,j})$ takes esk which can be freshly generated and signed by dsk, and isk (both recovered by $\mathsf{CorruptDevice}(B, k)$) and public inputs. The attack then proceeds as for $\mathsf{CorruptUser}(B)$ by sending a `MsgReq` message for gpk'.
3) $\mathsf{Compromise}(B, j, s)$**:** If the adversary directly compromises the Megolm session state of an intended recipient, they can trivially decrypt the ciphertext. Explicitly, after running $\mathsf{Compromise}(B, j, s)$, the adversary is given

a copy of $\mathfrak{S}_{gpk}$ with the required key material to decrypt $c = \mathsf{Encrypt}(A, i, s, m_0, m_1)$.

Definition 1. *A* DOGM *security experiment fulfils the* MTXCONF *confidentiality predicate if, for all* $\mathsf{Encrypt}(A, i, s, m_0, m_1)$ *queries placed by the adversary during the experiment, where* $m_0 \neq m_1$, $\pi^s_{A,i}.t = t^*$ *and* $\pi^s_{A,i}.z = z^*$*, none of the following conditions are true:*

1) $\mathsf{CorruptUser}(B)$ *was issued for* $B \in \mathcal{U}_{id}$ *and where* $\exists k$ *s.t.* $\mathsf{CanDecrypt}(B, k, A, i, s, t^*, z^*) = \mathbf{true}$.

2) $\mathsf{CorruptDevice}(B, j)$ *was issued for* $B \in \pi^s_{A,i}.CU$ *and* $j \in [n_D]$*, where* $\exists k$ *s.t.* $\mathsf{CanDecrypt}(B, k, A, i, s, t^*, z^*) = \mathbf{true}$.

3) $\mathsf{Compromise}(B, j, s)$ *was issued where* $\mathsf{CanDecrypt}(B, j, A, i, s, t^*, z^*) = \mathbf{true}$

where $\mathsf{CanDecrypt}(B, j, A, i, s, t^*, z^*)$ *returns true whenever the session* $\pi^s_{B,j}$ *(with its current state at the time of* CanDecrypt *being called) is able to decrypt the message sent by* $\pi^s_{A,i}$ *at stage* t^* *and message index* z^**. Precisely,* $\mathsf{CanDecrypt}(B, j, A, i, s, t^*, z^*)$ *is true iff, for the session identified by* $\pi^s_{A,i}.T[t^*, z^*].gpk$*,* $\pi^s_{B,j}$*'s copy* $\pi^s_{B,j}.st.\vec{st}_{\mathsf{mg}}[gpk, ipk]$ *contains an index* z *less than or equal to* z^*.

The second set of attacks target authentication, in the context where $\pi^s_{A,i}$ accepts a forged ciphertext c claiming to be from $\pi^s_{B,j}$, and are captured by the MTXAUTH predicate. In particular, session $\pi^s_{A,i}$ has received a ciphertext c, and successfully processes it with $\mathsf{Decrypt}(A, i, s, c)$. However, the ciphertext c was not sent by a corresponding session $\pi^s_{B,j}$.

1) $\mathsf{CorruptUser}(A)$ **or** $\mathsf{CorruptDevice}(A, i')$**:** If user A has had their long-term secrets (or those of one of their devices, $D^{A,i'}$) compromised, the adversary can use the Key Request protocol to inject an inbound Megolm session under their control to $\pi^s_{A,i}$. The adversary can claim that this session is owned by another session $\pi^s_{B,j}$ with the only constraint being that $(B, j) \neq (A, i)$. In particular, assume that A is a receiver but not also the sender. We exploit that KS can inject arbitrary receiving $\mathfrak{S}_{gpk}$ sessions on devices of the same user. If $\mathsf{CorruptUser}(A)$ was called, $\mathcal{A}$ creates a new device, authenticates it with the compromised user key material and starts an Olm channel with device $D^{A,i}$ of A. If $\mathsf{CorruptDevice}(A, i')$ was called, $\mathcal{A}$ uses dsk, isk to start an Olm channel with $D^{A,i}$.

We now proceed in two cases. *Case 1* is either $\mathsf{CorruptUser}(A)$ or $\mathsf{CorruptDevice}(A, i')$ with $i \neq i'$. The attack proceeds by triggering device $D^{A,i}$ to request key material for session gpk in epoch t^* by sending a forged ciphertext which $D^{A,i}$ cannot decrypt nor authenticate: $D^{A,i}$ will then request the key material from its trusted peer devices. $\mathcal{A}$ responds with gpk, which $D^{A,i}$ accepts. *Case 2* is $\mathsf{CorruptDevice}(A, i')$ with $i = i'$. Here, $\mathcal{A}$ first calls $\mathsf{Create}(A, k)$ for some $k \neq i$, which triggers A creating an honest device. $\mathcal{A}$ then sends the forgery to $D^{A,k}$, which requests the missing key material for gpk, which $\mathcal{A}$ provides using the compromised credentials from $\mathsf{CorruptDevice}(A, i)$. Finally $\mathcal{A}$ sends the forgery to $D^{A,i}$ which will request gpk, which will be provided by the honest device $D^{A,k}$.

2) $\mathsf{CorruptUser}(B)$ **or** $\mathsf{CorruptDevice}(B, j)$**:** If $\mathcal{A}$ has knowledge of the sending user's long-term secrets (or those of one of their devices), they may have used these long-term secrets to initiate the epoch t^* with a $\mathfrak{S}_{gsk}$ they control.In the first case, where the user's long-term secrets have been corrupted, the adversary could have generated a new device (without the challenger's help) and used this device to create stage t^*. In the second case, where device $D^{B,j}$'s long-term secrets were corrupted before the initiation of stage t^*, the adversary could have created the Megolm session, initiated a new Olm channel using ephemeral keys signed with the device's long-term keys and distributed the inbound session to $\pi^s_{A,i}$. [21]

3) $\mathsf{Compromise}(B, j, s)$**:** If $\mathcal{A}$ has directly compromised the sending sessions, $\pi^s_{B,j}$, this compromise includes the outbound session secrets contained in $\mathfrak{S}_{gsk}$. This is an outbound Megolm session, allowing it to encrypt, MAC then sign the forged message. Since the symmetric key material is ratcheted forward after each encryption, access to the current outbound Megolm state does not allow the adversary to forge messages with a lower message index (unless they additionally have access to an earlier copy of the inbound or outbound session).

Definition 2. *A* DOGM *security experiment fulfils the* MTXAUTH *authentication predicate if, processing a* $\mathsf{Decrypt}(A, i, s, c)$ *query where* $\pi^s_{A,i}.t = t^*$ *and* $\pi^s_{A,i}.z = z^*$*, none of the following conditions are true:*

1) $\mathsf{CorruptUser}(A)$ *or* $\mathsf{CorruptDevice}(A, i')$ *was issued when* $\pi^s_{A,i}.t < t^*$ *and* $i' \in [n_D]$.

2) $\mathsf{CorruptUser}(B)$ *or* $\mathsf{CorruptDevice}(B, j)$ *was issued such that* $\pi^s_{A,i}.CU[0] = B$*,* $\pi^s_{A,i}.CD[0] = j$ *when* $\pi^s_{A,i}.t < t^*$.

3) $\mathsf{Compromise}(B, j, s)$ *was issued and* $\pi^s_{A,i}.CU[0] = B$*,* $\pi^s_{A,i}.CD[0] = j$ *when* $\pi^s_{B,j}.t = t^*$ *and at least one of the following is true:*

a) $\pi^s_{B,j}.z \leq z^*$.

b) $\exists\ C \in \mathcal{U}_{id}$ *s.t.* $\mathsf{CorruptUser}(C)$ *has been called and* $\exists\ k$ *s.t.* $\mathsf{CanDecrypt}(C, k, B, j, s, t^*, z^*) = \mathbf{true}$.

c) $\exists\ C \in \mathcal{U}_{id}, k \in \mathcal{D}_{id}$ *s.t.* $\mathsf{CorruptDevice}(C, k)$ *has been called and* $\exists\ l$ *s.t.* $\mathsf{CanDecrypt}(C, l, B, j, s, t^*, z^*) = \mathbf{true}$.

d) $\exists\ C \in \mathcal{U}_{id}, k \in \mathcal{D}_{id}$ *s.t.* $\mathsf{Compromise}(C, k, s)$ *has been called when* $\mathsf{CanDecrypt}(C, l, B, j, s, t^*, z^*) = \mathbf{true}$.

4.2. Result

Theorem 1. *Matrix (as described by* Mtx*) is* IND-CCA*-secure under authentication predicate* MTXAUTH *and confidentiality predicate* MTXCONF *in the random oracle model. That is, for any PPT algorithm* $\mathcal{A}$ *in the* DOGM *security experiment,* $\mathsf{Adv}^{\mathsf{DOGM},\mathcal{A},\mathsf{MTXAUTH},\mathsf{MTXCONF}}_{\mathsf{Mtx},n_P,n_D,n_S,n_M}(\lambda)$ *is negligible if the Gap-DH [26] assumption holds on Curve25519,*

21. Note that, when this predicate is applied, the call to Decrypt has ended the experiment. Thus, *any* calls to $\mathsf{CorruptUser}(B)$ or $\mathsf{CorruptDevice}(B, j)$ within the experiment must have occurred prior to the Decrypt call.

Ed25519 *is SUF-CMA secure [27],* HKDF *is a secure KDF [28],* MgRatchet *is a secure FF-PRG [29], and* AES-CBC *and* HMAC *(as combined in both* Mg *and* OlmAEAD*) is a secure AEAD scheme.*

We separate the proof into two cases. In Case 1, we consider the probability of the adversary winning the game through an authentication break. In Case 2, we consider the probability of the adversary winning the game through a confidentiality break. We bound the advantage of winning both cases and demonstrate that under certain assumptions, $\mathcal{A}$'s advantage of winning overall is negligible. A sketch proof for each case is provided below (Section B contains the complete proof).

Proof sketch of Case 1. **Game 0** is the standard DOGM game. In **Game 1** and **Game 2**, we guess the accepting session (A, i, s), accepting at stage T from believed device $D^{B,j}$ at a cost of $n_P^2 \cdot n_I \cdot n_D^2 \cdot n_S$. In **Game 3**, we introduce an abort event that triggers when a message is accepted that was not sent by the genuine partner, and bound the probability of this event in the remainder of the proof. In **Game 4** we replace the self-signing key ssk_B with the public key from a SUF-CMA challenger, preventing Olm Key Bundle forgeries by the SUF-CMA security of DS. We proceed similarly in **Game 5**, but for the self-signing key of A. In **Game 6**, we split our proof depending on which party initiated an Olm channel with the other. The proof proceeds identically, incurring a factor of two. In **Game 7**, we prevent Olm ephemeral key injections from $D^{A,i}$ to $D^{B,j}$, using the SUF-CMA security of DS by replacing $D^{A,i}$'s $dpk_{A,i}$ from SUF-CMA challenger. In **Game 8**, we rely on the Gap-DH assumption to argue that it is hard for $\mathcal{A}$ to query to $g^{isk_{B,j} \cdot esk_{A,i,k}}$ to a HKDF random oracle, rendering its output (indistinguishable from) uniformly random. In **Game 9**, **Game 10** and **Game 11** we replace the Olm master secret ms derived by $D^{A,i}$, the Olm ratchet state until the target Megolm ratchet is generated, and the encryption key used to encrypt the target Megolm ratchet with uniformly random values, each time relying on the KDF security of HKDF. In **Game 12**, we prevent $\pi^s_{A,i}$ from accepting a ciphertext containing the target Megolm session without being sent by B, bound by the auth security of OlmAEAD. Finally, in **Game 13**, we prevent $\pi^s_{A,i}$ from accepting a Megolm ciphertext that $D^{B,j}$ did not send, by replacing gpk with a public key from a SUF-CMA challenger.

Proof sketch of Case 2. **Game 0** is the standard DOGM game. **Game 1** reduces the number of Encrypt$(\pi^s_{A,i}, m_0, m_1)$ challenge calls (s.t. $m_0 \neq m_1$) to one using a hybrid argument at the cost of n_Q, the polynomial upper bound on the number of queries by $\mathcal{A}$. In **Game 2** we guess the session $\pi^s_{A,i}$ that generated the challenge ciphertext c_b at the cost of $n_P \cdot n_I \cdot n_D \cdot n_S$. **Game 3** introduces the abort event $abort_{\mathfrak{D},B}$ triggered if B receives a device key package $\mathfrak{D}_{A'}$ that was not generated by A' (for each uncorrupted party A'). This enables the self-signing key $ssk_{A'}$ of each party A' to be replaced with the public key of a SUF-CMA challenge. We reduce winning this game to the SUF-CMA security of Ed25519 for each uncorrupted party (bounded by n_P) and the other winning conditions denoted by Adv$(break_3)$, bounded by the following games. In **Game 4** we gain a factor of $|\pi^s_{A,i}.CU|$ since we will repeat the following games for all partners of $\pi^s_{A,i}.CU$. That is, **Game 5** to **11** consider the security of Olm channels between A and each of their communicating partners $B \in \pi^s_{A,i}.CU$. **Game 5** restricts the game to cases where B initiated the Olm channel between A and B. Since the analysis proceeds equivalently, at the cost of a factor of two. **Game 6** replaces A's $dpk_{A,i}$ with the public key from a SUF-CMA challenge. Now, in the remaining games, the device keys $(ipk_{A,i}, epk_{A,i,k})$ used by B to initiate the Olm channel are known to be the keys generated by A. In **Game 7**, we replace $ipk_{B,j}$ and $epk_{A,i}$ with a Diffie-Hellman pair output by a Gap-DH challenger and embed the Gap-DH problem into the computation of HKDF$(g^{isk_{B,j} \cdot esk} \| g^{esk_{B,j'} \cdot isk} \| g^{esk_{B,j'} \cdot esk})$. This allows us to conclude that ms is (indistinguishable from) uniformly random and independent of protocol execution. In **Games 8** and **9**, we replace each rch_l, ck_l with random values, relying on the KDF security of HKDF, until $D^{B,j}$ outputs the T-th Megolm session consisting of $(\mathfrak{S}_{gsk}, \mathfrak{S}_{gpk}, \sigma_{\mathsf{mg}})$. In **Game 10**, we conclude that the message key mk_l used to encrypt the message containing the Megolm session is (indistinguishable from) uniformly random and independent of protocol execution. This is done by replacing mk_l with the output of a KDF challenger to give $\hat{mk}_l$. In **Game 11**, we replace the Megolm inbound session $\mathfrak{S}_{gpk} = (ver, i, R, gpk)$ encrypted with OlmAEAD using $\hat{mk}_l$. In **Game 12**, we use the key indistinguishability of the Megolm ratchet to demonstrate that the key values output are uniformly random. Finally, **Game 13** utilises the confidentiality guarantee of an authenticated encryption challenger (replacing the output of Megolm's encryption routine using k) to prove that the adversary cannot distinguish the ciphertexts c_0 and c_1 without winning the confidentiality game of the authenticated encryption challenger.

Summary. Matrix achieves confidentiality and authenticity, for a message m sent in stage t, if the adversary's corruptions only affect users (or their devices) that (1) are never a member of the room, (2) not yet a member of the room or (3) stopped being a member of the room before the sender initialises stage t. Note that this result applies in the context of the DOGM security experiment defined in this paper and requires careful interpretation when applied to practical use. We discuss these caveats in Sections C.1 to C.3 and 5.

5. Discussion

Our formal security statements should be interpreted to mean that Matrix' core cryptographic components *can* achieve the stated security goals *after* the authentication vulnerabilities reported in [6] are fixed and formally analysed. In particular, we reiterate that it is currently trivial for a Matrix server to add a user as a recipient to a session or a unverified new device to a user. Both of these will be visible in the flagship *Element* client, but this behaviour is not

prevented (see Section 4.1). Furthermore, these guarantees would also *only* apply when composed with a formally analysed cryptographic backup solution, which is not captured by our model. Finally, our guarantees only cover when devices only communicate with other verified devices. This is our main take away for a general security audience.

Confidentiality and authentication between whom? Matrix guarantees the *confidentiality* of a message between the sender and the users the sender intended as recipients, i.e. the list of group members displayed in the client. This includes all of their verified devices: past, present and future. However, since there is no cryptographic control of the list of group members, the sender's list of intended recipients is controlled by the adversary.[22]

Matrix guarantees the *authenticity* of a message in that it ensures its integrity and correctly identifies the user, device and Megolm session it originated from. However, the Matrix protocol will decrypt any message for which it has been sent the necessary inbound Megolm session from a verified user and device (regardless of the sender's group membership status, for example).[23]

A lack of canonical message order limits guarantees. Our predicates define a canonical ordering of messages in a Matrix conversation. Specifically, the challenger indexes Matrix stages (and messages) based on the order they are generated by the sender session in the experiment. Thus, the guarantees we prove must be interpreted with respect to this canonical ordering. However, this session ordering does not map back intuitively to the message order that users might observe in their clients. This is due to Matrix' use of multiple Olm channels in parallel, making it impossible for receiving clients to determine a canonical ordering of Megolm sessions (without trusting in the server to provide one). For this reason, we suggest that future iterations of the protocol disallow multiple Olm channels between any single pair of devices. Section C.1 discusses this issue in more detail.

FS and PCS enable user management. It is worth discussing the nature of FS and PCS guarantees in Matrix. Typically in the context of secure messaging, FS ensures that messages sent *before* compromise remain secure [38], and PCS ensure that messages sent *after* compromise remain secure (assuming the adversary remains passive) [39]. Our trivial win conditions in Section 4.1 essentially establish that neither security goal, as defined in the cryptographic literature, is attained by Matrix when long-term secrets are compromised.[24] Indeed, once an adversary corrupts one device of a user, the key sharing feature allows an attacker to escalate that compromise to all other devices (and all of their sessions). Furthermore, the Matrix specification allows sessions to maintain old key materials, invalidating FS guarantees. We found it useful to think of the combined states of all of a user's devices as one big meta state that can be compromised in total and that offers essentially no PCS nor FS guarantees when long-term key material is compromised. Rather, the use of cryptographic techniques typically deployed to achieve FS and PCS serve the purpose of user management: when a new user joins and leaves a room, keys are updated.

These are our main take aways for applied cryptographers.

Conclusion. It is for these reasons that we avoid the standard terminology of "security proof" in this work, which can be misunderstood as a "seal of approval". That is, rather than stating that Matrix is *secure* (or not), this work establishes *what* security guarantees its core components can provide. This is, of course, always what a cryptographic security proof does, but we consider it imperative to stress here. Despite this, our analysis suggests that current efforts by the Matrix developers to remedy the attacks reported in [6], combined with our suggested improvements, may suffice to produce a secure protocol. These are our main takeaways for the Matrix developers.

Open problems. A central open problem is to analyse the authenticated group management currently in development for Matrix as well as both of its backup mechanisms (Server-side Megolm Backups and SSSS).

This work analyses Matrix within a single, monolithic security model and experiment. Taking a more modular approach, ideally reusing existing models and analysis[25], would provide further confidence in the overall design and its components. Together, such works could enable a more comprehensive model of secure group messaging.

Furthermore, while Matrix falls short of what the cryptographic literature expects, this is only partially due to avoidable design flaws (such as long-term authentication keys circumventing PCS guarantees of Olm channels) but partially also due to a deliberate design trade-off. That is, some of the FS/PCS guarantees we establish here match those intended by the Matrix designers who accept this behaviour in favour of utility in a chat context: making older messages available across devices. This design choice limits the authenticity guarantees of the protocol. Whether this trade-off is correct is a question that falls outside the expertise of cryptography in a narrow sense. Indeed, recent work has established that the FS/PCS guarantees provided by cryptography do not align well with the needs of some people reliant on secure messaging in a higher-risk environment [40]. Establishing what FS and PCS *should* be is an exciting area for future multidisciplinary work.

The difficulty of establishing a consistent global ordering

22. This is distinct from the control over group membership that is afforded to the adversary as part of their role in the experiment (given through access to the AddMember and RemoveMember adversarial queries).

23. This maps back to a particular sending session, such as $\pi^{s}_{B,j}$, within the DOGM experiment.

24. Our analysis shows that PCS can be achieved when only Megolm states are compromised. We do not cover Olm state compromises.

25. For example, those from the SGM and continuous group key agreement (CGKA) ecosystem.

of messages within Matrix poses an interesting question about attacks that are possible due to inconsistent message ordering in protocols that allow out-of-order decryption. Some group messaging protocols such as Message Layer Security [41] rely on the server to provide such a consistent global ordering: determining the impact of this ability from non-honest servers may be useful in formalising the security of such protocols.

Finally, it was necessary for us to model the security separation between users and devices in order to meaningfully capture how Matrix does not achieve this separation (for the most part). However, there exist many real-world configurations of SGM, each with their own set of differing trade-offs. We hope that the DOGM model proves useful in the analysis of such schemes, to capture the varying degrees of user-device separation, FS and PCS security guarantees that they can achieve. A natural open question is whether stronger security guarantees can be achieved by an alternative design whilst maintaining the same function requirements of Matrix.

Acknowledgements

We thank our anonymous reviewers for their helpful reviews, feedback and suggestions. D. 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," Jun. 2022, version: unstable. [Online]. Available: https://spec.matrix.org/unstable/client-server-api/

[2] J. Meredith and A. Balducci, "Matrix Olm Cryptographic Review," NCC Group, Tech. Rep., Nov. 2016, version 2.0. [Online]. Available: https://research.nccgroup.com/2016/11/01/public-report-matrix-olm-cryptographic-review/

[3] The Matrix.org Foundation, "Megolm group ratchet," May 2022. [Online]. Available: https://gitlab.matrix.org/matrix-org/olm/-/raw/master/docs/megolm.md

[4] ——, "Olm: A Cryptographic Ratchet," Nov. 2019. [Online]. Available: https://gitlab.matrix.org/matrix-org/olm/-/raw/master/docs/olm.md

[5] D. Wong, *Real-world Cryptography*. Manning Publications., 2021.

[6] M. R. Albrecht, S. Celi, B. Dowling, and D. Jones, "Practically-exploitable cryptographic vulnerabilities in Matrix," in *44th IEEE Symposium on Security and Privacy*, T. Ristenpart and P. Traynor, Eds., 2023.

[7] M. Hodgson, "Independent public audit of Vodozemac, a native rust reference implementation of Matrix end-to-end encryption," May 2022. [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

[8] 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/LeastAuthority-MatrixvodozemacFinalAuditReport.pdf

[9] The Matrix.org Foundation, "Signature keys and user identity in libolm," Nov. 2020. [Online]. Available: https://gitlab.matrix.org/matrix-org/olm/blob/master/docs/signing.md

[10] M. Marlinspike, "Private Group Messaging," May 2014. [Online]. Available: https://signal.org/blog/private-groups/

[11] 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.

[12] 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.

[13] 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.

[14] 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

[15] D. Balbás, D. Collins, and P. Gajland, "WhatsUpp with sender keys? Analysis, improvements and security proofs," in *ASIACRYPT 2023*, ser. LNCS. Springer, Heidelberg, Dec. 2023, to appear.

[16] ——, "Analysis and improvements of the sender keys protocol for group messaging," in *XVII Reunión española sobre criptología y seguridad de la información (RECSI)*, D. S. Renedo, Ed., 2022. [Online]. Available: https://arxiv.org/abs/2301.07045

[17] J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis, "Security analysis and improvements for the IETF MLS standard for group messaging," in *Advances in Cryptology – CRYPTO 2020, Part I*, ser. Lecture Notes in Computer Science, D. Micciancio and T. Ristenpart, Eds., vol. 12170. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 17–21, 2020, pp. 248–277.

[18] J. Alwen, S. Coretti, D. Jost, and M. Mularczyk, "Continuous group key agreement with active security," in *TCC 2020: 18th Theory of Cryptography Conference, Part II*, ser. Lecture Notes in Computer Science, R. Pass and K. Pietrzak, Eds., vol. 12551. Durham, NC, USA: Springer, Heidelberg, Germany, Nov. 16–19, 2020, pp. 261–290.

[19] J. Alwen, B. Auerbach, M. C. Noval, K. Klein, G. Pascual-Perez, K. Pietrzak, and M. Walter, "CoCoA: Concurrent continuous group key agreement," in *Advances in Cryptology – EUROCRYPT 2022, Part II*, ser. Lecture Notes in Computer Science, O. Dunkelman and S. Dziembowski, Eds., vol. 13276. Trondheim, Norway: Springer, Heidelberg, Germany, May 30 – Jun. 3, 2022, pp. 815–844.

[20] J. Alwen, D. Jost, and M. Mularczyk, "On the insider security of MLS," in *Advances in Cryptology – CRYPTO 2022, Part II*, ser. Lecture Notes in Computer Science, Y. Dodis and T. Shrimpton, Eds., vol. 13508. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 15–18, 2022, pp. 34–68.

[21] C. Brzuska, E. Cornelissen, and K. Kohbrok, "Security analysis of the MLS key derivation," in *2022 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 22–26, 2022, pp. 2535–2553.

[22] B. Poettering, P. Rösler, J. Schwenk, and D. Stebila, "SoK: Game-based security models for group key exchange," in *Topics in Cryptology – CT-RSA 2021*, ser. Lecture Notes in Computer Science, K. G. Paterson, Ed., vol. 12704. Virtual Event: Springer, Heidelberg, Germany, May 17–20, 2021, pp. 148–176.

[23] J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis, "Modular design of secure group messaging protocols and the security of MLS," in *ACM CCS 2021: 28th Conference on Computer and Communications Security*, G. Vigna and E. Shi, Eds. Virtual Event, Republic of Korea: ACM Press, Nov. 15–19, 2021, pp. 1463–1483.

[24] A. Dimeo, F. Gohla, D. Goßen, and N. Lockenvitz, "SoK: Multi-device secure instant messaging," Cryptology ePrint Archive, Report 2021/498, 2021, https://eprint.iacr.org/2021/498.

[25] The Matrix.org Foundation, "End-to-End Encryption implementation guide." [Online]. Available: https://matrix.org/docs/guides/end-to-end-encryption-implementation-guide

[26] T. Okamoto and D. Pointcheval, "The gap-problems: A new class of problems for the security of cryptographic schemes," in *PKC 2001: 4th International Workshop on Theory and Practice in Public Key Cryptography*, ser. Lecture Notes in Computer Science, K. Kim, Ed., vol. 1992. Cheju Island, South Korea: Springer, Heidelberg, Germany, Feb. 13–15, 2001, pp. 104–118.

[27] J. Brendel, C. Cremers, D. Jackson, and M. Zhao, "The provable security of Ed25519: Theory and practice," in *2021 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 24–27, 2021, pp. 1659–1676.

[28] 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.

[29] Y. Dodis, D. Jost, and H. Karthikeyan, "Forward-secure encryption with fast forwarding," in *TCC 2022: 20th Theory of Cryptography Conference, Part II*, ser. Lecture Notes in Computer Science, E. Kiltz and V. Vaikuntanathan, Eds., vol. 13748. Chicago, IL, USA: Springer, Heidelberg, Germany, Nov. 7–10, 2022, pp. 3–32.

[30] D. Balbás, D. Collins, and S. Vaudenay, "Cryptographic administration for secure group messaging," in *USENIX Security 2023: 33rd USENIX Security Symposium*. USENIX Association, 2023, to appear.

[31] B. Dowling and B. Hale, "Secure messaging authentication against active man-in-the-middle attacks," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021, pp. 54–70.

[32] C. Cremers, J. Fairoze, B. Kiesl, and A. Naska, "Clone detection in secure messaging: Improving post-compromise security in practice," 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. 1481–1495.

[33] J. Jaeger and N. Tyagi, "Handling adaptive compromise for practical encryption schemes," in *Advances in Cryptology – CRYPTO 2020, Part I*, ser. Lecture Notes in Computer Science, D. Micciancio and T. Ristenpart, Eds., vol. 12170. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 17–21, 2020, pp. 3–32.

[34] M. Marlinspike, "Simplifying OTR deniability." Jul. 2013. [Online]. Available: https://signal.org/blog/simplifying-otr-deniability/

[35] The Matrix.org Foundation, "MSC2732: Olm fallback keys," Jun. 2021. [Online]. Available: https://github.com/matrix-org/matrix-spec-proposals/pull/2732

[36] M. Marlinspike, "The X3DH Key Agreement Protocol," Nov. 2016, revision 1. [Online]. Available: https://signal.org/docs/specifications/x3dh/

[37] ——, "The Double Ratchet Algorithm," Nov. 2016. [Online]. Available: https://signal.org/docs/specifications/doubleratchet/

[38] M. Bellare, A. C. Singh, J. Jaeger, M. Nyayapati, and I. Stepanovs, "Ratcheted encryption and key exchange: The security of messaging," in *Advances in Cryptology – CRYPTO 2017, Part III*, ser. Lecture Notes in Computer Science, J. Katz and H. Shacham, Eds., vol. 10403. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 20–24, 2017, pp. 619–650.

[39] K. Cohn-Gordon, C. J. F. Cremers, and L. Garratt, "On post-compromise security," in *CSF 2016: IEEE 29th Computer Security Foundations Symposium*, M. Hicks and B. Köpf, Eds. Lisbon, Portugal: IEEE Computer Society Press, Jun. 27–1, 2016, pp. 164–178.

[40] M. R. Albrecht, J. Blasco, R. B. Jensen, and L. Mareková, "Collective information security in large-scale urban protests: the case of hong kong," in *USENIX Security 2021: 30th USENIX Security Symposium*, M. Bailey and R. Greenstadt, Eds. USENIX Association, Aug. 11–13, 2021, pp. 3363–3380.

[41] R. Barnes, B. Beurdouche, R. Robert, J. Millican, E. Omara, and K. Cohn-Gordon, "The Messaging Layer Security (MLS) Protocol," RFC 9420, Jul. 2023. [Online]. Available: https://www.rfc-editor.org/info/rfc9420

[42] 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.

[43] 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.

[44] "Secure hash standard," National Institute of Standards and Technology, NIST FIPS PUB 180-2, U.S. Department of Commerce, Aug. 2002.

[45] H. Krawczyk and P. Eronen, "RFC 5869: HMAC-based Extract-and-Expand Key Derivation Function (HKDF)," May 2010. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5869

[46] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," IETF Internet Request for Comments 2104, Feb. 1997.

[47] "Advanced Encryption Standard (AES)," National Institute of Standards and Technology, NIST FIPS PUB 197, U.S. Department of Commerce, Nov. 2001.

[48] 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

[49] R. Housley, "RFC 5652: Cryptographic Message Syntax (CMS)," Internet Engineering Task Force, Sep. 2009. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5652

Appendix A. Preliminaries

We reference the following primitives and algorithms:

- $\mathsf{X25519}$ is the Curve25519 [42] based DH key exchange.
- $\mathsf{Ed25519}$ is a $\mathsf{SUF\text{-}CMA}$-secure digital signature scheme [43], $\mathsf{DS} = (\mathsf{DS.KGen}, \mathsf{DS.Sign}, \mathsf{DS.Verify})$.
- $\mathsf{HKDF}(s, k, c)$ (or $\mathsf{HKDF\text{-}SHA\text{-}256}$) is a Hash-based Key Derivation Function constructed with SHA-256 [44] where s is the salt, k is the secret key material and c is the context [28], [45].
- $\mathsf{HMAC}(k, m)$ (or $\mathsf{HMAC\text{-}SHA\text{-}256}$) is a Hash-based Message Authentication Code constructed with SHA-256 [44] taking as input a key k and message m [46]. Matrix truncates HMAC outputs to 64 bits [6], [8] contrasting with the HMAC RFC [46] which recommends at least 128 bits for SHA-256.
- $\mathsf{MgRatchet} = (\mathsf{Init}, \mathsf{Next})$ is the Megolm ratchet where $\mathsf{MgRatchet.Init}$ takes a security parameter and outputs an initial state (i, R) and $\mathsf{MgRatchet.Next}$ takes a current state and outputs a new state and a key [3]. It can be seen as an implementation of an FF-PRG [29, Def. 2] that we assume

satisfies the security and correctness requirements in Def's 3 and 4 of [29].

• $\mathsf{AES}(k, m)$ is AES [47] taking a key k and a message block m of size 128 bits. Matrix uses AES-256, i.e. 256-bit keys.

• $\mathsf{AES\text{-}CBC}.\mathsf{Enc}(iv, k, m)$ and $\mathsf{AES\text{-}CBC}.\mathsf{Dec}(iv, k, c)$ is AES in cipher block chaining (CBC) mode [48] where iv is the nonce, k is an AES encryption key m is a message and c is a ciphertext. Matrix uses PKCS7 [49] padding to split plaintexts into blocks for CBC mode in the Olm and Megolm protocols, as well as the asymmetric Megolm backup scheme.

• MRU abstracts iteration over a sequence, ordered by most recent use.

Algorithms described in Section 2 may access the public keys (and cross-signing signature hierarchy) of *other* users and devices directly (through reference to $\mathfrak{U}_B$, $\mathfrak{D}_{B,E}$ or $ipk_{B,E}$, for example). This simulates fetching these values from an untrusted server using the given identifiers.

In Table 2 we map our pseudocode shorthand strings to those used in Matrix.

Appendix B. Security Analysis of Matrix

Proof. We begin by separating the proof into two cases and denote with $\mathsf{Adv}^{\mathsf{DOGM},\mathcal{A},C_\ell}_{\mathsf{Mtx},n_P,n_D,n_S,n_M}(\lambda)$ the advantage of the adversary winning the DOGM security game in Case ℓ. These correspond to 1) $\mathcal{A}$ has triggered the authentication win condition, 2) $\mathcal{A}$ has terminated the game and output a guess bit b.
Since $\mathsf{Adv}^{\mathsf{DOGM},\mathcal{A}}_{\mathsf{Mtx},n_P,n_D,n_S,n_M}(\lambda) \leq \mathsf{Adv}^{\mathsf{DOGM},\mathcal{A},C_1}_{\mathsf{Mtx},n_P,n_D,n_S,n_M}(\lambda) + \mathsf{Adv}^{\mathsf{DOGM},\mathcal{A},C_2}_{\mathsf{Mtx},n_P,n_D,n_S,n_M}(\lambda)$, we may bound the overall advantage of winning by considering each case in turn. In doing so, we demonstrate that under certain assumptions, $\mathcal{A}$'s advantage of winning overall is negligible.

Case 1: Adversary has triggered the authentication win condition. We treat the advantage of $\mathcal{A}$ in triggering the authentication win condition. We do this via the following sequence of games.

Game 0 This is the standard DOGM game in Case 1. Thus we have $\mathsf{Adv}^{\mathsf{DOGM},\mathcal{A},C_1}_{\mathsf{Mtx},n_P,n_D,n_S,n_M}(\lambda) \leq \mathsf{Adv}(break_0)$.

Game 1 In this game, we guess a session $\pi^s_{A,i}$ and stage T to be the first session that causes the authentication flag win to be set when $\pi^s_{A,i}.t = T$, and trigger an abort event if any other session other than $\pi^s_{A,i}$ causes $\mathsf{win} \leftarrow \mathtt{true}$. Specifically, at the beginning of the experiment we guess a tuple of values (A, i, s) and abort the game if the session processing a Decrypt call is $\pi^s_{B,j}$ such that $(B, j, s) \neq (A, i, s)$ and $\pi^s_{B,j}.t \neq T$. Thus: $\mathsf{Adv}(break_0) \leq n_P \cdot n_I \cdot n_D \cdot n_S \cdot \mathsf{Adv}(break_1)$.

Game 2 In this game, we guess a session $\pi^s_{B,j}$ to be the sending partner and device communicating with $\pi^s_{A,i}$, and trigger an abort event if our guess is incorrect. Specifically, at the beginning of the experiment we guess a tuple of values (B, j) and abort the game if $\pi^s_{A,i}.CU[0] \neq B$ and $\pi^s_{A,i}.CD[0] \neq j$. Thus: $\mathsf{Adv}(break_1) \leq n_P \cdot n_D \cdot \mathsf{Adv}(break_2)$.

Game 3 In this game, the challenger will abort if $\pi^s_{A,i}$ accepts a non-honest message. Specifically, we define an abort event $abort_{np}$ that triggers when $\pi^s_{A,i}$ sets $\mathsf{win} \leftarrow \mathtt{true}$. Thus, by the definition of the case $\mathsf{Adv}(break_2) = 0$. In what follows, we will bound the probability of $abort_{np}$. Thus: $\Pr[abort_{np}] \leq \mathsf{Adv}(break_3)$.

Game 4 In this game, we introduce an abort event $abort_{\mathfrak{D},B}$ that triggers if party A receives device keys $(dpk_{B,j}, ipk_{B,j}) \in dt_{B,j}$, but B did not generate $(dpk_{B,j}, ipk_{B,j})$. Based on our partner guess from **Game 2**, we replace at the beginning of the game spk_B with pk from a strongly unforgeable digital signature challenger $\mathcal{C}_{\mathsf{DS}}$. Whenever B is required to sign a device key package $dt_{B,j}.m_D$ using spk_B, B instead queries $dt_{B,j}.m_D$ to $\mathcal{C}_{\mathsf{DS}}$. Note that $dt_{B,j}.m_D = (B, j, dpk_{B,j}, ipk_{B,j}, \mathtt{OlmAlg})$. Thus, if party A receives key package $dt_{B,j}$ such that $\mathsf{DS.Verify}(spk_B, dt_{B,j}.m_D, dt_{B,j}.\sigma_D) = 1$, but $(dpk_{B,j}, ipk_{B,j})$ were not generated honestly by B (and thus would trigger the abort event $abort_{\mathfrak{D},B}$), then $dt_{B,j}.m_D, dt_{B,j}.\sigma_D$ is a valid forgery, and breaks the strong unforgeability of the digital signature scheme DS. Thus: $\mathsf{Adv}(break_3) \leq \mathsf{Adv}(break_4) + \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{DS}}(\mathcal{B})$.

Game 5 In this game, we introduce an abort event $abort_{dt,A}$ that triggers if party B receives device keys $(dpk_{A,i}, ipk_{A,i}) \in dt_{A,i}$, but A did not generate $(dpk_{A,i}, ipk_{A,i})$. Based on our guess from **Game 1**, we replace at the beginning of the game spk_A with pk from a strongly unforgeable digital signature challenger $\mathcal{C}_{\mathsf{DS}}$. Whenever A is required to sign a device key package $dt_{A,i}.m_D$ using spk_A, A instead queries $dt_{A,i}.m_D$ to $\mathcal{C}_{\mathsf{DS}}$. Thus, if party B receives a key package $dt_{A,i}$ such that $\mathsf{DS.Verify}(spk_A, dt.m_D, dt.\sigma_D) = 1$, but $(dpk_{A,i}, ipk_{A,i})$ were not generated honestly by A (and thus would trigger the abort event $abort_{dt,A}$), then $dt_{A,i}.m_D, dt_{A,i}.\sigma_D$ is a valid forgery, and breaks the strong unforgeability of the digital signature scheme DS. Thus: $\mathsf{Adv}(break_4) \leq \mathsf{Adv}(break_5) + \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{DS}}(\mathcal{B})$.

Game 6 In this game, we introduce an abort event $abort_{in}$ that triggers if device $D^{B,j}$ does not initiate the Olm channel between $D^{A,i}$ and $D^{B,j}$. We note that the proof proceeds identically in either case, up to a change in notation, hence the guess. Thus: $\mathsf{Adv}(break_5) \leq 2 \cdot \mathsf{Adv}(break_6)$.

Game 7 In this game, we introduce an abort event $abort_{\mathfrak{D}}$ that triggers if party B receives an Olm key package $\mathfrak{D}$, but A did not generate $(ipk_{A,i}, epk_{A,i,k}) \in \mathfrak{D}$. Based on our guess from **Game 1**, we replace (when A generates device i) $dpk_{A,i}$ with pk from a strongly unforgeable digital signature challenger $\mathcal{C}_{\mathsf{DS}}$. Whenever A is required to sign device keys, identity keys $dpk_{A,i}, ipk_{A,i} \in m_d$, ephemeral key $epk_{A,i,k} \in m_{e,k}$, or a fallback key $fpk_{A,i,k} \in m_{f,k}$ $\mathcal{C}$ instead queries these messages to $\mathcal{C}_{\mathsf{DS}}$. Thus, if party B receives a

$\mathsf{OlmAEAD.Enc}(k, i, ad, m)$

$k_e \parallel k_h \parallel k_{iv} \leftarrow \mathsf{HKDF}(0, k, i)[0:80]$
$x \leftarrow \mathsf{AES\text{-}CBC.Enc}(k_{iv}, k_e, m)$
$\tau \leftarrow \mathsf{HMAC}(k_h, ad \parallel x)[0:8]$
return $ad \parallel x \parallel \tau$

$\mathsf{OlmAEAD.Dec}(k, i, c)$

$k_e \parallel k_h \parallel k_{iv} \leftarrow \mathsf{HKDF}(0, k, i)[0:80]$
$ad \parallel x \parallel \tau \leftarrow c$
if $\tau \neq \mathsf{HMAC}(k_h, ad \parallel x)[0:8]$:
  return $(\perp, \perp)$
$m \leftarrow \mathsf{AES\text{-}CBC.Dec}(k_{iv}, k_e, x)$
return (ad, m)

Figure 11: $\mathsf{OlmAEAD} = (\mathsf{OlmAEAD.Enc}, \mathsf{OlmAEAD.Dec})$ scheme for AEAD.

key package $\mathfrak{D}$ such that $\mathsf{DS.Verify}(dpk_{A,i}, m_d, \sigma_d) = 1$ or $\mathsf{DS.Verify}(dpk_{A,i}, m_{e,k}, \sigma_{e,k}) = 1$, but $(ipk_{A,i}, epk_{A,i,k})$ were not generated honestly by A (and thus would trigger the abort event $abort_{\mathfrak{D}}$), then either m_d, σ_d or $m_{e,k}, \sigma_{e,k}$ is a valid forgery, and breaks the strong unforgeability of the digital signature scheme DS. Thus: $\mathsf{Adv}(break_6) \leq \mathsf{Adv}(break_7) + \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{DS}}(\mathcal{B})$.

Game 8 In this game, we introduce an abort event $abort_{gdh}$ that triggers if the adversary queries $g^{isk_{B,j} \cdot esk_{A,i,k}}$ to a HKDF random oracle. Specifically, we initialise a Gap-DH challenger $\mathcal{C}_{\mathsf{Gap\text{-}DH}}$ that outputs a Diffie-Hellman pair X, Y, which we embed into $ipk_{B,j}$ and $epk_{A,i,k}$ (AUTH ensures that $\mathcal{C}$ will not have to answer any query that leaks $isk_{B,j}$, nor $esk_{A,i,k}$).

We now turn to demonstrating how $\mathcal{C}$ may need to perform computations using $isk_{B,j}$, or $esk_{A,i,k}$. In the latter case, this is an ephemeral key that is explicitly only used once, so $\mathcal{C}$ will not need $esk_{A,i,k}$. In the case of $isk_{B,j}$, the challenger will instead pick random keys for the output ms instead of deriving them via $\mathsf{HKDF}(g^{isk_{B,j} \cdot esk} \| g^{esk_{B,j'} \cdot isk} \| g^{esk_{B,j'} \cdot esk})$. $\mathcal{C}$ maintains a list of all sessions in which random keys should have been substituted: this contains the random session keys and public keys that should have been used to compute each component of the master secret. $\mathcal{C}$ ensures that the key values used are consistent with any $g^{isk_{B,j} \cdot esk} \| g^{esk_{B,j'} \cdot isk} \| g^{esk_{B,j'} \cdot esk}$ queries that $\mathcal{A}$ makes to the random oracle. Before answering a random oracle query, $\mathcal{C}$ will go through each entry in the above list of sessions: for each entry, it uses the $\mathcal{C}_{\mathsf{Gap\text{-}DH}}$'s DDH oracle to check if the public keys that should have been used to compute each component of the master secret match the corresponding component $g^{isk \cdot esk}$, $g^{esk \cdot isk}$, $g^{esk \cdot esk}$ of the random oracle query. If all components, when queried in the DDH oracle, return 1, then $\mathcal{C}$ uses the randomly chosen keys from that list as the random oracle response, otherwise, $\mathcal{C}$ samples a new random value. Note that if $\mathcal{A}$ causes $(ipk_{B,j}, epk_{A,i,k}, g^{isk_{B,j} \cdot esk_{A,i,k}})$ to be queried and the response is 1, then $\mathcal{A}$ has solved the Gap-DH problem and $\mathcal{C}$ submits $g^{isk_{B,j} \cdot esk_{A,i,k}}$ to $\mathcal{C}_{\mathsf{Gap\text{-}DH}}$, which triggers $abort_{gdh}$. Finally, we note that as a result of this change, the value ms computed by $D^{A,i}$ and $D^{B,j}$ is uniformly random and independent of the protocol execution. Thus: $\mathsf{Adv}(break_7) \leq \mathsf{Adv}(break_8) + \mathsf{Adv}^{\mathsf{Gap\text{-}DH}}_{\mathsf{HKDF}}(\mathcal{B})$.

Game 9 In this game, we replace the computation of rch_0, ck_0 in the Olm session between $D^{A,i}$ and $D^{B,j}$ with uniformly random keys $\hat{rch}_0, \hat{ck}_0$. Specifically, when computing rch_0, ck_0, the challenger instead initialises a KDF challenger $\mathcal{C}_{\mathsf{kdf}}$, and queries $g^{esk_{A,i,k} \cdot rsk}$, and replaces ms with the output of said query. We note that by the **Game 8** ms is already a uniformly random and independent value, so this replacement is sound. We note that if the bit b sampled by $\mathcal{C}_{\mathsf{kdf}}$ is 0, then we are in **Game 8**, else we are in **Game 9**. Thus: $\mathsf{Adv}(break_8) \leq \mathsf{Adv}(break_9) + \mathsf{Adv}^{\mathsf{kdf}}_{\mathsf{KDF}}(\mathcal{B})$.

Game 10 In this game, we repeat the process of replacing the computation of rch_l, ck_l in the Olm session between $D^{A,i}$ and $D^{B,j}$ with uniformly random keys $\hat{rch}_l, \hat{ck}_l$ until $D^{B,j}$ produces the T-th Megolm Ratchet. Specifically, when computing rch_l, ck_l, the challenger instead initialises a KDF challenger $\mathcal{C}_{\mathsf{kdf}}$, queries $g^{rsk \cdot rsk}$, and replaces rch_l, ck_l with the output of said query. This replacement is sound since, by **Game 9**, $\hat{rch}_{i-1}$ is already a uniformly random and independent value. We note that if the bit b sampled by $\mathcal{C}_{\mathsf{kdf}}$ is 0, then we are in **Game 9**, else we are in **Game 10**. Thus: $\mathsf{Adv}(break_9) \leq \mathsf{Adv}(break_{10}) + k \cdot \mathsf{Adv}^{\mathsf{kdf}}_{\mathsf{KDF}}(\mathcal{B})$.

Game 11 In this game, we replace the computation of mk_l in the Olm session between $D^{A,i}$ and $D^{B,j}$ with uniformly random key $\hat{mk}_l$. Specifically, when computing mk_l, the challenger instead initialises a KDF challenger $\mathcal{C}_{\mathsf{kdf}}$, and queries 0x01, and replaces mk_l with the output of said query. We note that by the **Game 10** $\hat{ck}_l$ is already a uniformly random and independent value, so this replacement is sound. We note that if the bit b sampled by $\mathcal{C}_{\mathsf{kdf}}$ is 0, then we are in **Game 10**, else we are in **Game 11**. Thus: $\mathsf{Adv}(break_{10}) \leq \mathsf{Adv}(break_{11}) + \mathsf{Adv}^{\mathsf{kdf}}_{\mathsf{KDF}}(\mathcal{B})$.

Game 12 In this game, we introduce an abort event that triggers if $D^{A,i}$ decrypts an Olm ciphertext (keyed by $\hat{mk}_l$), and accepts a Megolm inbound session $\mathfrak{S}'_{gpk} = (ver, i, R, gpk)$ but the ciphertext was not output by $D^{B,j}$. Specifically, the challenger initialises an auth challenger $\mathcal{C}_{\mathsf{auth}}$, which the challenger queries when $D^{A,i}$ needs to encrypt or decrypt with $\hat{mk}_l$. The abort event only triggers if $\mathcal{A}$ can produce a valid ciphertext that decrypts under $\hat{mk}_l$, and we can submit the ciphertext to $\mathcal{C}_{\mathsf{auth}}$, breaking the auth security of the aead scheme. By **Game 11** $\hat{mk}_l$ is already uniformly random and independent and this replacement is sound. Any $\mathcal{A}$ that can trigger the abort event can be used by the challenger to break the auth security of aead. Thus: $\mathsf{Adv}(break_{11}) \leq \mathsf{Adv}(break_{12}) + \mathsf{Adv}^{\mathsf{auth}}_{\mathsf{aead}}(\mathcal{B})$.

Game 13 In this game, we introduce an abort event $abort_{auth}$ that triggers if $\pi^s_{A,i}$ decrypts a Megolm ciphertext $c' = (ver, i, c, \tau, \sigma)$, but c' was not output by $\pi^s_{B,j}$.

Specifically, whenever $D^{B,j}$ creates the Megolm inbound session $\mathfrak{S}_{gpk}$ encrypted under $\hat{mk}_l$, the challenge instead replaces gpk with pk from a strongly unforgeable digital signature challenger $\mathcal{C}_{\mathsf{DS}}$. Whenever $\pi^s_{B,j}$ is required to sign Megolm ciphertexts with gsk, $\pi^s_{B,j}$ instead queries the ciphertext to $\mathcal{C}_{\mathsf{DS}}$. Thus, if $\pi^s_{A,i}$ receives a Megolm ciphertext c' such that $\mathsf{DS.Verify}(gpk, (ver, i, c, \tau), \sigma) = 1$ but $c' = (ver, i, c, \tau, \sigma)$ was not generated honestly by $\pi^s_{B,j}$ (and thus would trigger the abort event $abort_{auth}$), then $(ver, i, c, \tau), \sigma$ is a valid forgery, and breaks the strong unforgeability of the digital signature scheme DS. Thus: $\mathsf{Adv}(break_{12}) \leq \mathsf{Adv}(break_{13}) + \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{DS}}(\mathcal{B})$.

Note that now $\mathcal{A}$ can never cause $\pi^s_{A,i}$ to accept a non-honest ciphertext in stage T, and thus $\mathsf{Adv}(break_{13}) = 0$.

Case 2: Adversary terminates and outputs a bit b. Here, we bound the probability that $\mathcal{A}$ correctly guesses the bit b via the following sequence of games.

Game 0 This is the standard DOGM game in Case 2. Thus we have $\mathsf{Adv}^{\mathsf{DOGM},\mathcal{A},C_2}_{\mathsf{Mtx},n_P,n_D,n_S,n_M}(\lambda) \leq \mathsf{Adv}(break_0)$.

Game 1 In this game, we transition to an adversary that only makes a single $\mathsf{Encrypt}(\pi^s_{A,i}, m_0, m_1)$ (where $m_0 \neq m_1$) query, via a hybrid argument. Since $\mathcal{A}$ is polynomially-bounded, then this introduces a factor upper-bounded by the number of queries that $\mathcal{A}$ can make, which we denote n_Q. Thus we find $\mathsf{Adv}(break_0) \leq n_Q \cdot \mathsf{Adv}(break_1)$.

Game 2 In this game, we guess a session $\pi^s_{A,i}$ and stage T such that $\mathsf{Encrypt}(A, i, s, m_0, m_1)$ (where $m_0 \neq m_1$) is called when $\pi^s_{A,i}.t = T$, and trigger an abort event if we are not correct. Specifically, at the beginning of the experiment we guess a tuple of values (A, i, s, T) and abort the game if any other query $\mathsf{Encrypt}(B, j, t, m_0, m_1)$ is called such that $(B, j, t) \neq (A, i, s)$ or $\pi^s_{A,i}.t \neq T$. Thus we find $\mathsf{Adv}(break_1) \leq n_P \cdot n_I \cdot n_D \cdot n_S \cdot \mathsf{Adv}(break_2)$.

Game 3 In this game, for each party A' that $\mathcal{A}$ did not $\mathsf{Corrupt}$ at the beginning of the experiment, we introduce an abort event $abort_{dt,B}$ that triggers if another party B receives a device key bundle $(dpk_{A',i'}, ipk_{A',i'}) \in dt_{A',i'}$, but A' did not generate $(dpk_{A',i'}, ipk_{A',i'})$. We replace at the beginning of the game (for each uncorrupted party A') $spk_{A'}$ with pk from a strongly unforgeable digital signature challenger $\mathcal{C}_{\mathsf{DS}}$. Whenever $\mathcal{C}$ is required to sign a device key package $dt_{A',i'}.m_D$ using $spk_{A'}$, $\mathcal{C}$ instead queries $dt_{A',i'}.m_D$ to $\mathcal{C}_{\mathsf{DS}}$. Thus, if party B receives a key package $dt_{A',i'}$ such that $\mathsf{DS.Verify}(spk_{A'}, dt_{A',i'}.m_D, dt_{A',i'}.\sigma_D) = 1$, but $(dpk_{A',i'}, ipk_{A',i'})$ were not generated honestly by A' (and thus would trigger the abort event $abort_{dt,B}$), then $dt_{A',i'}.m_D, dt_{A',i'}.\sigma_D$ is a valid forgery, and breaks the strong unforgeability of the digital signature scheme DS. There are at most n_P uncorrupted parties, and thus: $\mathsf{Adv}(break_2) \leq \mathsf{Adv}(break_3) + n_P \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{DS}}(\mathcal{B})$. We note that by the cleanness predicate CONF, that all parties that A encrypts to as a result of the $\mathsf{Encrypt}(A, i, s, m_0, m_1)$ must be uncorrupted and thus, as a result of **Game 3**, A has received all device public keys of their communicating partners $\pi^s_{A,i}.CD$ without modification.

Game 4 Note that from **Game 5** to **Game 11** we repeat the actions for each communicating partner of $\pi^s_{A,i}$, i.e. for each $B \in \pi^s_{A,i}.CU$. Let $n_U = |\pi^s_{A,i}.CU|$ be the number of communicating partners of $\pi^s_{A,i}$. Thus we have: $\mathsf{Adv}(break_3) \leq n_U \cdot \mathsf{Adv}(break_4)$.

Game 5 In this game, we introduce an abort event $abort_{in}$ that triggers if device $D^{B,j}$ does not initiate the Olm channel between $D^{A,i}$ and $D^{B,j}$. We note that the proof proceeds identically in either case, up to a change in notation. Thus: $\mathsf{Adv}(break_4) \leq 2 \cdot \mathsf{Adv}(break_5)$.

Game 6 In this game, we introduce an abort event $abort_{\mathfrak{D}}$ that triggers if party B receives an Olm key package $\mathfrak{D}$, but A did not generate $(ipk_{A,i}, epk_{A,i,k}) \in \mathfrak{D}$. Based on our guess from **Game 2**, we replace (when A generates device i) $dpk_{A,i}$ with pk from a strongly unforgeable digital signature challenger $\mathcal{C}_{\mathsf{DS}}$. Whenever A is required to sign device keys, identity keys $dpk_{A,i}, ipk_{A,i} \in m_d$, ephemeral key $epk_{A,i,k} \in m_{e,k}$, or a fallback key $fpk_{A,i,k} \in m_{f,k}$ $\mathcal{C}$ instead queries these messages to $\mathcal{C}_{\mathsf{DS}}$. Thus, if party B receives a key package $\mathfrak{D}$ such that $\mathsf{DS.Verify}(dpk_{A,i}, m_d, \sigma_d) = 1$ or $\mathsf{DS.Verify}(dpk_{A,i}, m_{e,k}, \sigma_{e,k}) = 1$, but $(ipk_{A,i}, epk_{A,i,k})$ were not generated honestly by A (and thus would trigger the abort event $abort_{\mathfrak{D}}$), then either m_d, σ_d or $m_{e,k}, \sigma_{e,k}$ is a valid forgery, and breaks the strong unforgeability of the digital signature scheme DS. Thus: $\mathsf{Adv}(break_5) \leq \mathsf{Adv}(break_6) + \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{DS}}(\mathcal{B})$.

Game 7 In this game, we introduce an abort event $abort_{gdh}$ that triggers if the adversary queries $g^{isk_{B,j} \cdot esk_{A,i,k}}$ to a HKDF random oracle. Specifically, we initialise a Gap Diffie-Hellman challenger $\mathcal{C}_{\mathsf{Gap\text{-}DH}}$ that outputs a Diffie-Hellman pair X, Y, which we embed into $ipk_{B,j}$ and $epk_{A,i,k}$. We note that our predicates ensure that $\mathcal{C}$ will not have to answer any query that leaks $isk_{B,j}$, nor $esk_{A,i,k}$.

We now turn to demonstrating how $\mathcal{C}$ may need to perform computations using $isk_{B,j}$, or $esk_{A,i,k}$. In the latter case, this is an ephemeral key that is explicitly only used once, so $\mathcal{C}$ will not need $esk_{A,i,k}$. In the case of $isk_{B,j}$, the challenger will instead pick random keys for the output ms instead of deriving them via $\mathsf{HKDF}(g^{isk_{B,j} \cdot esk} \| g^{esk_{B,j'} \cdot isk} \| g^{esk_{B,j'} \cdot esk})$. The challenge will maintain a list of all sessions in which random keys should have been substituted: the list contains the random session keys as well as the public keys that should have been used to compute each component of the mast secret. $\mathcal{C}$ will ensure that the key values used are consistent with any $g^{isk_{B,j} \cdot esk} \| g^{esk_{B,j'} \cdot isk} \| g^{esk_{B,j'} \cdot esk}$ queries that $\mathcal{A}$ makes to the random oracle. Before answering a random oracle query, $\mathcal{C}$ will go through each entry in the above list of sessions: for each entry, it uses the $\mathcal{C}_{\mathsf{Gap\text{-}DH}}$'s DDH oracle to check if the public keys that should have been used to compute each component of the master secret match the corresponding component $g^{isk \cdot esk}$, $g^{esk \cdot isk}$, $g^{esk \cdot esk}$ of the random oracle query. If all components, when queried in the DDH oracle, return 1, then $\mathcal{C}$ uses the randomly chosen keys from that list as the random oracle response,

otherwise, $\mathcal{C}$ samples a new random value. Note that if $\mathcal{A}$ causes $(ipk_{B,j}, epk_{A,i,k}, g^{isk_{B,j} \cdot esk_{A,i,k}})$ to be queried and the response is 1, then $\mathcal{A}$ has solved the Gap-DH problem and $\mathcal{C}$ submits $g^{isk_{B,j} \cdot esk_{A,i,k}}$ to $\mathcal{C}_{\mathsf{Gap\text{-}DH}}$, which triggers $abort_{gdh}$. Finally, we note that as a result of this change, the value ms computed by $D^{A,i}$ and $D^{B,j}$ is uniformly random and independent of the protocol execution. Thus: $\mathsf{Adv}(break_6) \leq \mathsf{Adv}(break_7) + \mathsf{Adv}^{\mathsf{Gap\text{-}DH}}_{\mathsf{HKDF}}(\mathcal{B})$.

Game 8 In this game, we replace the computation of rch_0, ck_0 in the Olm session between $D^{A,i}$ and $D^{B,j}$ with uniformly random keys $\hat{rch}_0, \hat{ck}_0$. Specifically, when computing rch_0, ck_0, the challenger instead initialises a KDF challenger $\mathcal{C}_{\mathsf{kdf}}$, and queries $g^{esk_{A,i,k} \cdot rsk}$, and replaces ms with the output of said query. We note that by the **Game 7** ms is already a uniformly random and independent value, so this replacement is sound. We note that if the bit b sampled by $\mathcal{C}_{\mathsf{kdf}}$ is 0, then we are in **Game 7**, else we are in **Game 8**. Thus: $\mathsf{Adv}(break_7) \leq \mathsf{Adv}(break_8) + \mathsf{Adv}^{\mathsf{kdf}}_{\mathsf{KDF}}(\mathcal{B})$.

Game 9 In this game, we repeat the process of replacing the computation of rch_l, ck_l in the Olm session between $D^{A,i}$ and $D^{B,j}$ with uniformly random keys $\hat{rch}_l, \hat{ck}_l$ until $D^{B,j}$ produces the T-th Megolm Ratchet. Specifically, when computing rch_l, ck_l, the challenger instead initialises a KDF challenger $\mathcal{C}_{\mathsf{kdf}}$, queries $g^{rsk \cdot rsk}$, and replaces rch_l, ck_l with the output of said query. This replacement is sound since, by **Game 8**, $\hat{rch}_{i-1}$ is already a uniformly random and independent value. We note that if the bit b sampled by $\mathcal{C}_{\mathsf{kdf}}$ is 0, then we are in **Game 8**, else we are in **Game 9**. Thus: $\mathsf{Adv}(break_8) \leq \mathsf{Adv}(break_9) + k \cdot \mathsf{Adv}^{\mathsf{kdf}}_{\mathsf{KDF}}(\mathcal{B})$.

Game 10 In this game, we replace the computation of mk_l in the Olm session between $D^{A,i}$ and $D^{B,j}$ with uniformly random key $\hat{mk}_l$. Specifically, when computing mk_l, the challenger instead initialises a KDF challenger $\mathcal{C}_{\mathsf{kdf}}$, queries `0x01`, and replaces mk_l with the output of said query. This replacement is sound since, by **Game 9**, $\hat{ck}_l$ is already a uniformly random and independent value. Note that if the bit b sampled by $\mathcal{C}_{\mathsf{kdf}}$ is 0, we are in **Game 9**, else we are in **Game 10**. Thus: $\mathsf{Adv}(break_9) \leq \mathsf{Adv}(break_{10}) + \mathsf{Adv}^{\mathsf{kdf}}_{\mathsf{KDF}}(\mathcal{B})$.

Game 11 In this game, we replace the plaintext (the Megolm inbound session $\mathfrak{S}'_{gpk} = (ver, i, R, gpk)$) of the Olm ciphertext keyed by $\hat{mk}_l$, with a new plaintext $\mathfrak{S}'_{gpk} = (ver, i, R^*, gpk)$, where R^* is a uniformly random value sampled from the Megolm ratchet space. Specifically, the challenger initialises an aead conf challenger $\mathcal{C}_{\mathsf{conf}}$, which the challenger queries when $D^{A,i}$ needs to encrypt with $\hat{mk}_l$. If the bit b sampled by $\mathcal{C}_{\mathsf{conf}}$ is 0, we are in **Game 10**, else we are in **Game 11**. We note that any adversary $\mathcal{A}$, capable of distinguishing between the two games can be used to break the conf security of the aead scheme. By **Game 10** $\hat{mk}_l$ is already uniformly random and independent and this replacement is sound. Thus: $\mathsf{Adv}(break_{10}) \leq \mathsf{Adv}(break_{11}) + \mathsf{Adv}^{\mathsf{conf}}_{\mathsf{aead}}(\mathcal{B})$.

Game 12 In this game, we replace all outputs of the Megolm ratchet in epoch T with uniformly random keys.

TABLE 2: Constants used in the Matrix, Olm, Megolm, Key Request and Cross-Signing protocols.

Symbol	Value
`OlmAlg`	`m.olm.v1.curve25519-aes-sha2`
`OlmVer`	`0x03`
`OlmRch0`	`OLM_ROOT`
`OlmRch`	`OLM_RATCHET`
`OlmKeys`	`OLM_KEYS`
`MgAlg`	`m.megolm.v1.aes-sha`
`MgVerIKE`	`0x01`
`MgVerMsg`	`0x03`
`MgVerFwd`	`0x02`
`MgKeys`	`MEGOLM_KEYS`
`CsMstr`	`master`
`CsSelf`	`self_signing`
`CsUser`	`user_signing`
`DvSign`	`device_keys_ed25519`
`DvIdent`	`device_keys_curve25519`
`DvEphm`	`one_time_keys`
`DvFall`	`fallback_keys`
`MsgEnc`	`m.room.encrypted`
`MsgPln`	`m.room.message`
`MsgKey`	`m.room_key`
`MsgReq`	`m.room_key_request`
`MsgFwd`	`m.forwarded_room_key`
`MsgWth`	`m.room_key.witheld`

Specifically, when $\pi^s_{A,i}$ generates the Megolm ratchet R, the challenger now initialises a key indistinguishability FF-PRG challenger $\mathcal{C}_{\mathsf{kind}}$ that outputs a new initial ratchet state R'. In addition, whenever $\pi^s_{A,i}$ encrypts a new message in epoch T, the challenger calls Update to $\mathcal{C}_{\mathsf{kind}}$ to replace the key output k of MgRatchet.Next. We note if the bit b sampled by $\mathcal{C}_{\mathsf{kind}}$ is 0, we are in **Game 11**, else we are in **Game 12**. We note that any adversary $\mathcal{A}$ capable of distinguishing between the two games can be used to break the kind security of the Megolm FF-PRG. By **Game 11**, the Megolm ratchet maintained by $\pi^s_{A,i}$ and its communicating partners is already uniformly random and independent of the protocol execution. Thus: $\mathsf{Adv}(break_{11}) \leq \mathsf{Adv}(break_{12}) + \mathsf{Adv}^{\mathsf{kind}}_{\mathsf{FF\text{-}PRG}}(\mathcal{B})$.

Game 13 In this game, when $\mathcal{A}$ issues $\mathsf{Encrypt}(A, i, s, m_0, m_1)$, the challenger instead initialises a confidentiality authenticated encryption challenger $\mathcal{C}_{\mathsf{conf}}$, and forwards m_0, m_1 to $\mathcal{C}_{\mathsf{conf}}$. We note that by **Game 12**, the key k used to encrypt the message by $\pi^s_{A,i}$ is uniformly random and independent of the protocol flow. When $\mathcal{A}$ terminates and outputs the bit b to the challenger, it simply forwards the guess bit b to $\mathcal{C}_{\mathsf{conf}}$. It is straightforward to see that the advantage of $\mathcal{A}$ guessing the bit b in the DOGM security experiment is now exactly equal to the challenger's advantage in guessing $\mathcal{C}_{\mathsf{conf}}$'s bit b and thus: $\mathsf{Adv}(break_{12}) \leq \mathsf{Adv}^{\mathsf{conf}}_{\mathsf{AuthEnc}}(\mathcal{B})$. □

Appendix C. Extended Discussion

C.1. Message Order

This section describes how the Matrix specification (and its flagship client Element) order messages. In particular, we

discuss the implicit ordering provided by the cryptographic primitives and its interaction with message order displayed to users. We finish by considering how the security result in Section 4 should be interpreted in light of these discrepancies.

C.1.1. In the Matrix specification. Neither the Matrix Client-Server API [1] nor the end-to-end encryption implementation guide [25] give explicit instructions on the order of end-to-end encrypted messages (regarding how they are displayed or ordered internally). As previously discussed, both the Olm and Megolm specifications do provide such an ordering implicitly as part of their key schedule. However, due to Matrix' use of multiple Olm channels, this does not map to a single canonical ordering of Megolm sessions. Thus, messages sent in the composed protocol do not have a canonical ordering. For this reason, our description of Mtx.Dec (see Fig. 9) does not return a message index.[26]

C.1.2. In the Element client. Investigating Element's implementation, we see that the order in which it displays messages is not determined cryptographically (through its index within a Megolm session, for example). Instead, clients use a timestamp provided by the server to order their messages.

C.1.3. Implication for interpreting Theorem 1. The security predicates in Section 4.1 define in what cases Matrix can guarantee confidentiality and authentication. These predicates refer to messages using the stage index (identifying a particular Megolm session) and message index (identifying a particular message within that session). As we note in Section 5, the former does not have a canonical ordering from the perspective of individual clients. Our security proof resolves this by indexing Megolm sessions with the order they were initialised *in the security experiment*. In practice, however, a client decrypting the message m does not know where in the global ordering of events the corresponding session was generated and, thus, we cannot reliably use the security predicates to evaluate when the security guarantees apply to a message displayed in the user interface. In other words, it is not clear how our security result can be interpreted in practice *without trusting the server* to provide a consistent message order that matches the time of session initialisation by the sending client.

Consider the case where Alice is on the third stage ($t = 2$) of their sending session, having sent three outbound Megolm sessions to the other members of the group. However, the homeserver has withheld the second of these sessions from group member Bob, such that Bob views Alice's third session as her second. Since the security experiment knows the global ordering of Megolm sessions, it has correctly set Bob's receiving sessions current stage to $t = 2$. Now, if Alice's sending session is compromised during $t = 2$, the security experiment tracks this and, if the adversary wins the experiment through a forgery from Alice to Bob, the experiment can correctly compare the stage and messages indices to check whether the win was valid or not (this is the third case of MTXAUTH, see Definition 2). Despite Bob's client having the wrong understanding of message order, the security predicate is still able to correctly determine when the security guarantees apply.

How does this issue affect real-world usage? In practice, Bob (and their client) cannot reliably track the correct stage index and are, thus, incapable of completing such a check. But we would never expect it to be able to! The stage indices exist inside the security experiment only; Matrix clients have no awareness of them. We do not expect a receiving client to find out if a particular session was compromised and react accordingly in such situations. Instead, our security result enables us to reason about how the protocol reacts to certain types of compromises *in general*.[27] In this example, the third case of MTXAUTH, combined with our security proof, tells us that we cannot guarantee the authenticity of messages sent in a Megolm session *after the corresponding outbound session has been compromised* (i.e. messages with indices greater than or equal to the index of the compromised outbound session). In other words, assuming neither of the first two cases of MTXAUTH apply, we can guarantee the authenticity of messages sent *before the session was compromised*.

Nonetheless, sessions sent by the attacker using the compromised session will appear in Element's user interface intertwined with sessions sent by Alice using the non-compromised session. This behaviour is avoidable: if clients were able to determine a canonical ordering of Megolm sessions, they would be able to reject new messages from old sessions (or, at the least, display them in the appropriate part of the message history).

C.1.4. Recommendations. We recommend that future iterations of the protocol work to ensure clients can determine a canonical cryptographic ordering of messages. We acknowledge that the distributed nature of Matrix may cause implementation challenges with the requisite design and implementation. Requiring that at most one active Olm channel can be used between any two devices at once is a step towards this. Finally, the protocol should require that clients use this canonical ordering to display messages. For example, all messages sent in stage t must appear after all message sent in stage $t - 1$.

C.2. Message History and State Sharing

Matrix provides a user's new devices with message history by sharing a copy of the inbound Megolm sessions

26. We note that, additionally, protocols relying upon the Sender Keys architecture (such as Megolm) are not able to determine a relative message order between senders. This is because each sender maintains their own independent sequence of Megolm sessions. However, since a lack of consistent message ordering between senders does not affect how we track the compromise of session secrets, this issue is not relevant to the discussion that follows.

27. If such tracking of compromises were to be implemented, the natural identifier to use for such checks would be the session identifier used by Matrix, the group signing key gpk.

the user has access to as well as the original ciphertexts. This requires clients to accept messages sent from historical Megolm sessions (since there is no way for a client to know whether a ciphertext was sent before the next Megolm session was created or not). This issue is compounded by Element's use of server provided timestamps to determine the display order of messages: historical Megolm sessions can be used to send messages that appear to be new. If a canonical message order were available, then (potentially forged) messages sent from historical Megolm sessions could appear alongside the other messages from that session in the message history. This combination of features means that it is practically impossible for Matrix to recover from a compromised session.

We note that this implementation of history sharing requires trust in more parties than is necessary. First, they must trust the user's other devices to distribute the correct inbound Megolm sessions. Second, they must trust the owner of each Megolm session not to modify their message history. Trust in one of these two may be necessary in a protocol that aims for deniability, since there should be no way to prove that a historical message was sent by a particular user or device. It is natural to place this trust in a device under the user's control rather than the original senders. The alternative would require additional trust in every sending device across all historical conversations. Trust in the original senders, thus, proves problematic if the protocol wants to provide a form of post-compromise security.

Importantly, however, trust in the original senders is not necessary to share message history. A simpler solution, where the historical messages are re-encrypted by the user's other devices would only require trust in those devices. Alternative designs could avoid re-encryption by having the sharing device include a commitment to the transcript alongside the session keys. Such a protocol would still provide a form of deniability (with the sharing devices *claiming* a particular history to the user's other trusted devices).

To summarise, as long as history sharing is implemented in a manner that relies on trust in the original message sender not to insert, edit or delete messages, it is not possible for Matrix to recover authentication guarantees after a device compromise.

C.2.1. Recommendations. In order to recover authentication guarantees after a device compromise, it is important that there exists a canonical ordering of messages and that this order is enforced by the client. Additionally, clients should be prevented from accepting new messages sent to historical Megolm sessions and, instead, should freeze the history of the stage $t-1$ as soon as stage t is received. Matrix should consider redesigning the state sharing protocol such that it only requires trust in the user's verified devices, not also in the original owners of the Megolm session. This could be implemented by including a hash of the stage transcript alongside shared Megolm sessions. Alternatively, they could redesign state sharing so that trusted devices directly share messages history.

C.3. Group Membership

Matrix has two core weaknesses with respect to group membership. First, there is no cryptographic control over group membership. Second, clients do not enforce the group membership when receiving messages. Both weaknesses are consistent with an adversarial model that trusts the server to control the group membership.

However, there is a meaningful separation between the two, such that a protocol with both of these weaknesses is weaker than a protocol with just one. Consider a variant of Matrix where clients *do* enforce the group membership when receiving messages. Such a protocol would be able to guarantee that only the group members of a particular stage can send and receive messages within that stage (even though the group membership in that stage is controlled by the adversary). As it stands, Matrix is only able to guarantee the confidentiality of messages between the group members (as well as correctly identifying the origin of any message sent to the group).

C.3.1. Recommendations. We recommend that the Matrix specification requires clients to enforce group membership when receiving messages. Combined with the planned addition of cryptographic control over group membership for clients, Matrix should provide stronger confidentiality and authentication guarantees that better reflect the needs of *group messaging*.

We note, however, that such a design should take into account the issues with message order and history sharing discussed in Sections C.1 and C.2. Without a canonical ordering of messages, it is unclear how clients can determine whether a message was sent by a user when they were (or were not) a member. Additionally, without the user interface both reflecting the cryptographic message order and rejecting new messages from old sessions, removed members may still be able to send messages to a group. We highlight the existing issues with state sharing as an example of such an issue.