Secure Group Messaging in Practice

A cryptographic analysis of real-world multi-device group messaging protocols.

Daniel Jones

Information Security Group
Royal Holloway, University of London

This dissertation is submitted for the degree of

Doctor of Philosophy

27th October 2025

Declaration

These doctoral studies were conducted under the supervision of Prof. Martin R. Albrecht. The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Information Security Group as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

– Daniel Jones, 8$^{\text{th}}$ April 2025

Acknowledgements

First and foremost, I would like to thank my supervisor Martin Albrecht for your incredibly reliable guidance and encouragement over the last four years. Thank you Benjamin Dowling, for joining us on these projects; and for sticking with us as their scope crept ever larger. Thank you both for lending me your expertise; and for always treating my ideas and contributions with the value and respect of a peer, despite my occasional naïvety and relative lack of experience. I hope to mirror your approach with all those I work with in the future.

Thank you Sofía Celi, you were both a pleasure to work with and a helpful guide through the world of cryptography. Thank you to those collaborators, peers and anonymous reviewers who persevered through early drafts of my work. Your feedback was invaluable, and many of your suggested improvements have made their way into this thesis. On this note, thank you Yiannis Tselekounis and Felix Günther for agreeing to examine this thesis. I hope that the comments of former reviewers were not in vain.

Thank you to those staff at Royal Holloway who spent countless hours applying for the grants that have funded my studies. Thank you Claire Hudson for keeping the CDT running smoothly all these years.

This thesis would not exist without the case studies within it. Thank you to those who worked to design, build and deploy these systems; for improving the state of secure messaging, not just in theory, but in practice too.

Thank you all my family and friends for your encouragement and support.

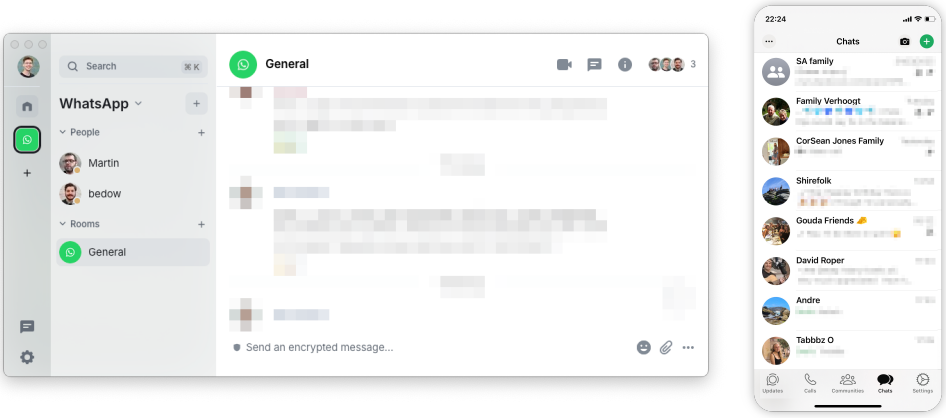


Figure 1. On the left, a project discussion for our analysis of WhatsApp, taking place on our research group's Matrix server. On the right, a selection of WhatsApp group chats from the author's phone.

Abstract

Despite applications such as Matrix, WhatsApp and Signal having provided end-to-end encryption in their group messaging products as early as 2014, the understanding of how these systems work, and the security guarantees they provide, has not been well addressed in the academic literature.

In this thesis, we seek to fill this very gap. We analyse the implementations, documentation and specifications of two secure group messaging applications, Matrix and WhatsApp, resulting in a detailed description of each protocol.

Finding that no existing model effectively captures the relationships between users, their devices, and the groups they are a part of, we introduce the device-oriented group messaging model to do just that. This model aims to capture the common subset of Matrix and WhatsApp that provides multi-device group messaging. It captures the confidentiality and authentication these protocols are able to provide, under varying forms of corruption and state compromise. We, additionally, develop a variant of this model to capture WhatsApp's support for cryptographic device revocation.

During our study of Matrix, we discover vulnerabilities in its design and implementation, before demonstrating how practically-exploitable attacks can be built upon them. We suggest improvements to the protocol and its implementation in order to remediate these issues. We then proceed to express multi-device group messaging in Matrix within our model, and prove that, once the aforementioned vulnerabilities have been fixed, this subset of Matrix provides confidentiality and authentication in certain conditions. We proceed to prove the security of WhatsApp's multi-device group messaging in a similar manner, finding that it too is secure within our model.

In both cases, we develop security predicates that detail in what situations our proof of security applies. We discuss how these predicates (and our results, more generally) can be interpreted in practice. Notably, we find that both protocols are able to satisfy the core requirements of confidentiality and authentication under common usage patterns.

While Matrix and WhatsApp provide limited forward secrecy and post-compromise security guarantees, the same mechanisms that weaken these guarantees also serve to provide important features, such as history sharing, or to aid in recovery after desynchronisation errors. On the other hand, thanks to its support for cryptographic *device revocation*, we find that WhatsApp *is* able to recover after the complete compromise of a device, provided that a user's primary device remains secure.

Finally, we highlight Matrix and WhatsApp's shared lack of *cryptographic membership control*: while both protocols guarantee confidentiality within the members of a group, clients cede control over who these members are to the server.

Our approach combines provable security, following the code-based game-playing approach proposed by Bellare and Rogaway, with detailed protocol descriptions based upon analysis of the implementations they deploy.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Secure group messaging protocols use end-to-end encryption to allow groups of people to communicate securely over an insecure connection. Two such popular deployments are Matrix and WhatsApp.

Matrix has over 125 million users.[1] It is a popular choice within the free and open-source software (FOSS) ecosystem, having been adopted by Mozilla [L23], the KDE project [L1] and the FOSDEM 2022 [L18] conference, to name but a few. It is used in operation (to varying degrees) by governmental organisations in Germany, France [L16], Sweden [L11], and Luxembourg [L4]. In particular, the German ministry of defence [L15] and healthcare system [L9] both use Matrix in the field.

WhatsApp has established itself as the dominant player in the consumer messaging landscape, providing encrypted messaging to over two billion users.[2]

Notably, Matrix and WhatsApp have provided end-to-end encryption in their group messaging products as early as 2016.[3] Yet, almost a decade on, the group messaging protocols they deploy have seen little attention from the academic community. In this thesis, we ask:

1) What functionality and security do secure group messaging applications target in practice, and how do these goals differ from those in the literature?

2) What approach do they take towards realising these goals?

3) What security do they achieve in practice?

[1] As of 25/1/2025, the Element website claims that Matrix has "125M+ users worldwide" [L17].

[2] As of 8/1/2025, their website claims "More than 2 billion people [...] use WhatsApp" [L14].

[3] Matrix introduced end-to-end encryption in beta form in 2016 [L7] and enabled it by default in 2020 [L8]. WhatsApp introduced end-to-end encryption in 2016 [L22].

1.2 Structure

In Chapter 2 we present the necessary background for this thesis. We detail our methodology, define the notation and primitives we use, as well as introduce the domain of multi-device secure group messaging.

The thesis then proceeds with two case studies.

1) In Chapter 3, we provide a formal description of multi-device group messaging in the Matrix standard. We gather information from its various specifications and online documentation, which we proceed to verify through an examination of the source code of Element, Matrix' flagship client and prototypical implementation.

   We follow with several practically-exploitable cryptographic vulnerabilities in the Matrix standard and Element. These, together, invalidate the confidentiality and authentication guarantees claimed by Matrix against a malicious server. This is despite Matrix' cryptographic routines being constructed from well-known and -studied cryptographic building blocks. The vulnerabilities we exploit differ in their nature (insecure by design, protocol confusion, lack of domain separation, implementation bugs) and are distributed broadly across the different subprotocols and libraries that make up the cryptographic core of Matrix and Element.

2) In Chapter 4, we provide a formal description of multi-device group messaging in WhatsApp. We do so despite the limited public documentation that is available. A hurdle we overcome by *reverse-engineering* the WhatsApp web client to determine how WhatsApp implements multi-device group messaging in practice.

In Chapter 5, we review the academic literature to find that no existing formalism for secure group messaging is able to capture the relationships (and *shared state*) between users, their devices and the groups they are a part of. We, thus, initiate the study of multi-device group messaging as a cryptographic primitive, introducing the Device-Oriented Group Messaging model. We define two variants, with the second providing an extension to capture WhatsApp's support for *device revocation*. We, additionally, formalise a number of supporting primitives which we utilise in our security analysis of WhatsApp.

In Chapter 6, we utilise our new formalism to analyse the security of multi-device group messaging in Matrix and WhatsApp. Broadly speaking, their results are similar: we determine that both Matrix and WhatsApp achieve the basic security notions of confidentiality, integrity and authenticity within the DOGM model. We find that both protocols achieve limited forms of protection against temporal compromise (i.e. post-compromise security and forward security). We discuss how to interpret these results, including the security our case studies provide as well as their limitations. Importantly, however, we find that the lack of cryptographic control over group membership in both Matrix and WhatsApp severely limits the *type of authentication* that they are able to provide.

We conclude the thesis in Chapter 7, where we discuss how the results of our analyses in Chapter 6 may be interpreted in real-world usage, and highlight open problems within this thesis, academia and practice.

1.3 Publications

The work in this thesis is based upon the following publications.

Practically-exploitable Cryptographic Vulnerabilities in Matrix
in 44th IEEE Symposium on Security and Privacy (S&P 2023),
by Martin R. Albrecht, Sofía Celi, Benjamin Dowling and Daniel Jones.

The paper is a result of joint work between Martin R. Albrecht, Sofía Celi, Benjamin Dowling and myself. Its technical contributions appear primarily within Chapter 3, whereby the description informed much of Section 3.2 and for which Section 3.3 details the aforementioned vulnerabilities.

Device-Oriented Group Messaging:
A Formal Cryptographic Analysis of Matrix' Core
in 45th IEEE Symposium on Security and Privacy (S&P 2024),
by Martin R. Albrecht, Benjamin Dowling and Daniel Jones.

This paper is the result of joint work between Martin R. Albrecht, Benjamin Dowling and myself. Its technical contributions appear within

- Chapter 3, where Section 3.2 provides a formal description of multi-device group messaging in Matrix,
- Chapter 5, where we develop a formalism for multi-device group messaging, and
- Chapter 6, where Section 6.1 analyses the security of Matrix in this formalism.

A Formal Analysis of Multi-Device Group Messaging in WhatsApp
to appear in 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2025),
by Martin R. Albrecht, Benjamin Dowling and Daniel Jones.

This paper is the result of joint work between Martin R. Albrecht, Benjamin Dowling and myself. Its technical contributions appear within

- Chapter 4, which provides a formal description of multi-device group messaging in WhatsApp,
- Chapter 5, where we extend the initial DOGM formalism to capture device revocation, within which Section 5.4 develops formalisms of some underlying building blocks used by Matrix and WhatsApp, and
- Chapter 6, where we analyse the security of WhatsApp in the DOGM with revocation formalism.

CHAPTER 2

Background

2.1 Method

This thesis starts with two case studies, where we examine the specifications, and analyse the implementations, of Matrix and WhatsApp. The result of this analysis is a detailed description of the relevant cryptographic functionality. Then, utilising the tools provided to us by *provable security*, we aim to discover the security guarantees that these two protocols provide to their users. We now set out this process.

Step 1 – Case studies

We start by studying our subject's public documentation. For Matrix, we were able to collate a variety of resources, including a high-level specification of client behaviour, self-contained specifications of smaller components, and an implementation guide. Public documentation for WhatsApp, on the other hand, was sparse. We primarily made use of WhatsApp's security whitepaper. From these resources, we document our current understanding of the protocol, including a detailed description of its component parts, how they fit together, and any gaps in our understanding.

Next, we turn our attention to the implementation. The reason for this is two-fold. First, it is important to verify that the implementation does indeed match what is described in the documentation. Second, in many cases, the specification does not provide sufficient detail to be analysed in a formal setting. Throughout this process, we refine our description, noting any inconsistencies with the specification and filling in any missing details. For Matrix, we were able to make use of a number of open-source libraries maintained by the Matrix.org Foundation, as well as the source of the flagship client, Element, for higher-level code. WhatsApp, in contrast, has no such code available, leaving us to resort to an inspection and analysis of the WhatsApp client applications themselves.

Step 2 – Capturing functionality with our formalism

Now, we identify and isolate the subset of the protocol that we would like to analyse. In our case, the functionality which is most relevant to multi-device group messaging. For each protocol we wish to analyse, we develop a formalism to capture both its functional and security requirements.

We start the formalisation process by defining a scheme's *syntax*. That is, we define its external interface as a tuple of algorithms, with the interface of each algorithm being specified through its type signature [Rog04]. We then rely on accompanying prose and figures to describe how these algorithms are intended to interact with one another and, thus, how they should be used. We sometimes do so formally, through a *correctness* definition that specifies the expected behaviour of the scheme in a non-adversarial setting.

Step 3 – Defining security for our formalism

Our *security* definitions take the form of security experiments: a probabilistic algorithm that captures a game being played between a *challenger* and *adversary*. Here, the challenger aims to demonstrate the security of the scheme, while the adversary aims to break it. The manner in which the two interact in the game defines the security property we would like to capture, which we specify as a pseudocode description of the challenger's behaviour. The behaviour of the adversary is left unspecified to allow us to reason about the security of the scheme against whole classes of adversary.

In particular, let $\mathsf{Security}^{\Pi}_{\Lambda}$ be a security experiment that aims to capture the security of a protocol Π with parameterisation Λ. The experiment captures the challenger $\mathcal{C}$ playing a security game against the adversary $\mathcal{A}$, outputting 1 if the adversary has won (i.e. they have broken the security of the scheme) and 0 if not (i.e. the security of the scheme has been upheld). As such, the outcome of our security experiment is a binary random variable that we may reason about, and we say that $\mathsf{Pr}(\mathsf{Security}^{\Pi}_{\Lambda}(\mathcal{A}) \mapsto 1)$ is the *win probability* of $\mathcal{A}$ playing the experiment Π (with parameterisation Λ) against our $\mathcal{C}$.

We use the term *advantage* to refer to the ability of an adversary to win a security game with greater probability than an idealised notion of the property we are studying. For example, consider a left-or-right style security game that requires the adversary to determine whether a ciphertext is an encryption of the provided plaintexts m_0 or m_1. In the ideal case, where the ciphertext provides no information about the m_0 or m_1, the adversary would be able to win the security experiment with probability of $1/2$. We name this class of security experiments a *decision*, or *guessing*, security experiment, and define advantage for such experiments as follows.

Definition 2.1 (Adversarial Advantage in Decision Games)**.** Let $\mathsf{Guess}^{\Pi}_{\Lambda}$ be a *decision*, or *guessing*, game that aims to capture the security of a protocol Π with parameterisation Λ, where the adversary must correctly determine the value of a hidden bit b. We define the *decision-advantage* of a particular adversary, $\mathcal{A}$, as

$$\mathsf{Adv}^{\mathsf{Guess}}_{\Pi,\Lambda}(\mathcal{A}) \coloneqq \left|\mathsf{Pr}(\mathsf{Guess}^{\Pi}_{\Lambda}(\mathcal{A}) \mapsto 1) - \mathsf{Pr}(\mathsf{Guess}^{\Pi_{\mathsf{Ideal}}}_{\Lambda}(\mathcal{A}) \mapsto 1)\right|$$

where $\mathsf{Guess}_{\Lambda}^{\Pi_{\mathsf{Ideal}}}$ denotes the game in its idealised form. With an idealised win probability of 1/2, we have

$$\mathsf{Adv}_{\Pi,\Lambda}^{\mathsf{Guess}}(\mathcal{A}) \coloneqq \left|\mathsf{Pr}(\mathsf{Guess}_{\Lambda}^{\Pi}(\mathcal{A}) \mapsto 1) - \frac{1}{2}\right|.$$

Other properties, however, may have idealised notions that can only be solved by an exponential-time brute force search. Take, for example, an unforgeability notion whereby the adversary may win the experiment by producing a valid signature that has not been honestly generated. For an ideal scheme, we would expect that the best strategy would be to search through all possible keys, generate and test candidate signatures. We expect the probability of an adversary winning the game against our idealised notion to be a negligible function of the scheme's security parameters (given polynomial-time). Thus, we define the advantage of an adversary in such experiments as follows.

Definition 2.2 (Adversarial Advantage in Search Games)**.** Let $\mathsf{Find}_{\Lambda}^{\Pi}$ be a *search* game that aims to capture the security of a protocol Π with parameterisation Λ. We define the *search-advantage* of a particular adversary, $\mathcal{A}$, as

$$\mathsf{Adv}_{\Pi,\Lambda}^{\mathsf{Find}}(\mathcal{A}) \coloneqq \left|\mathsf{Pr}(\mathsf{Find}_{\Lambda}^{\Pi}(\mathcal{A}) \mapsto 1) - \mathsf{Pr}(\mathsf{Find}_{\Lambda}^{\Pi_{\mathsf{Ideal}}}(\mathcal{A}) \mapsto 1)\right|$$

where $\mathsf{Find}_{\Lambda}^{\Pi_{\mathsf{Ideal}}}$ denotes the game in its idealised form. With an idealised win probability of zero, we have

$$\mathsf{Adv}_{\Pi,\Lambda}^{\mathsf{Find}}(\mathcal{A}) \coloneqq \mathsf{Pr}(\mathsf{Find}_{\Lambda}^{\Pi}(\mathcal{A}) \mapsto 1).$$

Our security definitions consider a *scheme* to be secure if we can bound the advantage of any probabilistic polynomial-time adversary with a *negligible function* of the security parameters of the scheme. We follow Definition 2.1 of [MF21], with some minor notation adjustments:

Definition 2.3 (Negligible Function)**.** A function, $\epsilon : \mathbb{N} \to \mathbb{R}$, is *negligible* if for any polynomial $\mathsf{poly} : \mathbb{N} \to \mathbb{R}^+$ there exists an integer N such that for all $\lambda \geq N$ we have

$$\epsilon(\lambda) \leq \frac{1}{\mathsf{poly}(\lambda)}.$$

We use Λ to denote the security parameter of the scheme, in combination with the usage restrictions imposed by the experiment.[1]

An example of a *scheme parameter* could be the length of a secret key, e.g. λ_k, while an *experiment parameter* could be the number of messages that can be encrypted with that key, e.g. n_m. In the spirit of concrete security [Rog04, BDJR97], we include both the scheme security parameters and experiment usage parameters in each of our bounds, codifying explicit relations between the

[1] When discussing the win probability or advantage for a security game over a whole class of adversaries, we sometimes remove the adversary $\mathcal{A}$ input from the function inputs. We may, additionally, move the security parameters input Λ into the function inputs to highlight that it is now an independent variable.

protocol, the primitives it is built from and its real-world usage. Note, however, that we do not consider the computational power of the adversary other than those restrictions implied by experiment parameters (and the need for it to be representable as a probabilistic polynomial-time algorithm).

Step 4 – Analysing the security of our case studies

Taking stock, we have isolated the functionality we are interested in analysing, specified its interface through a syntax definition, and defined the security guarantees we wish to capture through a security game. We now turn to analysing and (hopefully) proving the security of the protocol within our formalism.

In doing so, we must describe the protocols we are studying *twice*. Our second description instantiates the protocol we are studying within our newly developed formalism. As such, this second description forms part of our security analysis: it is the simplified subset of the functionality that we wish to analyse. As we translate our description to its new form, we must make decisions on what detail to leave in and what to leave out. While we would, ideally, like to keep as much detail as possible, including too much detail and complexity may make the analysis intractable. We do our best to document these decisions throughout to ensure that any limitations are clear.

The goal of our security analysis is to bound the probability of an adversary winning the security experiment, when instantiated with our protocol, as a function of the security and experiment parameters. We do so using the *game hopping* proof technique [Sho04]. We start with the aforementioned instantiation of our security game, $\mathsf{Security}^{\Pi}_{\Lambda}$, which we denote Game_0. We proceed to iteratively make changes to the challenger's code, resulting in a sequence of games (Game_0, Game_1, …, Game_n). With each change, or *hop*, we usually take away some of the attack surface available to the adversary. We continue to do so until the adversary's winning probability is easy to compute [MF21, Section 3.2.1]. This terminal game should be, in most cases, the idealised notion of the security property being studied, such that the win probability is 1/2 for a decision game and 0 for a search game.

For consecutive games Game_i and Game_{i+1}, we let $\epsilon_{i\to i+1}$ be the difference in win probability such that $\mathsf{Pr}(\mathsf{Game}_i(\Lambda) \mapsto 1) = \mathsf{Pr}(\mathsf{Game}_{i+1}(\Lambda) \mapsto 1) + \epsilon_{i\to i+1}(\Lambda)$. This is known as the *loss* or *compensation probability*. Iteratively applying this definition for each consecutive pair of games, we arrive at an expression for the original game:

$$\mathsf{Pr}(\mathsf{Security}^{\Pi}_{\mathcal{A}}(\Lambda) \mapsto 1) = \mathsf{Pr}(\mathsf{Game}_n(\Lambda) \mapsto 1) + \sum_{i=0}^{n-1} \epsilon_{i\to i+1}(\Lambda).$$

We have now reduced our security analysis to the problem of determining bounds for the individual loss introduced by each game hop. We typically see three types of changes in game hopping proofs [Sho04, Den06].

- **Bridging steps:** These modify the behaviour of the challenger in such a way that maintains equivalent functionality. These steps should not induce any loss.

- **Transitions based on indistinguishability:** These substitute some behaviour of the challenger which, while different, is computationally indistinguishable from its behaviour in the previous game. We bound the loss induced by such transitions by reasoning directly about the quantity $|\mathsf{Pr}(\mathsf{Game}_i(\Lambda) \mapsto 1) - \mathsf{Pr}(\mathsf{Game}_{i+1}(\Lambda) \mapsto 1)|$. In particular, through a security reduction that "interpolates" between the two games, in such a way that an adversary that can distinguish between them can be repurposed to break some underlying problem.

- **Transitions based on failure events:** These introduce an abort in the case that a failure, or *bad*, event. We bound the loss induced by such transitions using the *fundamental lemma of game playing* [BR04, BR06], which we briefly state.[2]

Lemma 2.4 (The Fundamental Lemma of Game Playing)**.** Let $\mathsf{Game}_i(\Lambda)$ and $\mathsf{Game}_{i+1}(\Lambda)$ be two probabilistic security games that are identical except for a single **if**-branch that is present in $\mathsf{Game}_{i+1}(\Lambda)$ but not in $\mathsf{Game}_i(\Lambda)$. Let $event_{i \to i+1}$ denote the event that the corresponding (side-effect free) boolean expression of the **if**-branch evaluates to **true**. Then, for all Λ and $y \in \{0, 1\}$, it holds that

$$|\,\mathsf{Pr}(\mathsf{Game}_i(\Lambda) \mapsto y) - \mathsf{Pr}(\mathsf{Game}_{i+1}(\Lambda) \mapsto y)\,| \leq \mathsf{Pr}(event_{i \to i+1}).$$

Furthermore, it holds that

$$\mathsf{Pr}(\mathsf{Game}_i(\Lambda) \mapsto y) \leq \mathsf{Pr}(\mathsf{Game}_{i+1}(\Lambda) \mapsto y \mid \overline{event_{i \to i+1}}) + \mathsf{Pr}(event_{i \to i+1}).$$

Search or computation-style problems are generally more amenable to use in security reductions for failure events. In contrast, decision problems are generally more amenable to use in security reductions for transitions based on indistinguishability.

Security predicates. As the complexity of a security experiment rises, it is important that we accurately track the cases where we can perform our reductions soundly. For example, the introduction of state corruptions also introduces situations where a reduction relying on key indistinguishability is not possible. As we build our reductions, we collate these together into a *security predicate*, specifying under what circumstances within the security experiment we expect the construction to provide security.

These security predicates capture a mixture of "trivial wins" in the security experiment, as well as breaks in the design of the protocol (intended or otherwise). By "trivial wins" we mean those situations resulting from giving the adversary increased power that allows them to win the experiment directly. An example of this could be the ability to directly compromise the key material that was used to encrypt a challenge ciphertext. We would consider an adversary's ability to decrypt this challenge ciphertext a "trivial win".

In other cases, these security predicates capture situations where the protocol is not able to provide security given a particular set of events, and for which the

[2] We follow Lemma 5.13 in [MF21, Page 223].

resulting attack is less direct. An example of this case could be the compromise of session state *after* it has encrypted a challenge ciphertext. Whether or not this should be considered a "break" of the protocol depends on the security guarantees it targets. If the construction we are studying aims for forward secrecy, for example, this *would* be considered a break of the scheme.

While the protocols we study in this work make use of primitives that may imply that forward secrecy is a goal, they compose these with primitives that have contrasting goals! Rather than making this judgement ourselves, we simply aim to discover the security properties the protocol provides. That is, rather than stating that a particular construction is *secure* (or not), we aim to establish *what* security guarantees it provides and under what conditions. This is, of course, always what a cryptographic security proof does, but we consider it imperative to stress here.

Step 5 – Interpreting our results

Once our security analysis is complete, we have collected

1) security predicates that specify under what circumstances the protocol should provide security,

2) a bound that specifies the relationship between the security provided by the protocol and its real-world usage, and

3) a list of the cryptographic assumptions that it relies on.

This approach could be seen as a (complementary) reversal of *practice-oriented provable security* as spearheaded by Bellare and Rogaway [Bel98, Rog09]: we study existing real-world constructions, and develop formalisms that enable us to reason about and prove the security guarantees these existing constructions provide us. The process we have described is messy, and riddled with opportunities for error. It is clear that many of these opportunities could be avoided through a first-party analysis that takes the approach proposed by Bellare and Rogaway. We stress that the lack of access to WhatsApp's source code, and the potential inaccuracies it may result in, should be considered when interpreting this work.

2.2 Notation

This section provides a reference for the notation we use in this work.

Constants

- **Bit-strings:** Unless otherwise specified, the length of bit-strings are specified in bits (not bytes). We sometimes suffix indices/lengths with 'b' to make this explicit (or 'B' to indicate that they are specified in bytes). For example, '$xs \leftarrow\!\$\ \{0,1\}^{16\mathsf{B}}$' has bit-length 128 such that '$xs[9\mathsf{B} \rightarrow 128\mathsf{b}]$' is equivalent to '$xs[72 \rightarrow 128]$'.

- **Placeholder Constants:** For the purpose of brevity, we make use of placeholders to represent real-world values that were otherwise too unwieldy to include directly within our pseudocode description. We provide tables to reverse this mapping in Appendix A.

Identifiers

We use identifiers to represent the various parties that interact in the protocols. We separate these identifiers into two broad categories.

1. **Protocol Identifiers:** These are the identifiers used by the actual protocol implementations. We generally use the form '$\texttt{x}id$' for such identifiers (where `x` is a letter indicating the type of the identifier).

2. **Analytical Identifiers:** These are the identifiers we utilise in our analysis (both in our prose and, for ease, within security experiments). These generally take the form of counters, for which we use single letter variable names.

Consider Alice. We may allocate her an analytical identifier of A, such that her username `alice@matrix.org` would be denoted by uid_A. She may have two devices, a phone and a laptop, which we could identify with analytical identifiers i and j (and globally indexed as (A, i) and (A, j) to denote that they are Alice's ith and jth devices, respectively). The protocol may have different identifiers for these devices, however, which we denote as $did_{A,i}$ and $did_{A,j}$ (respectively).

Pseudocode

We make some of our pseudocode choices explicit.

- **Sampling:** We use '$x \leftarrow\!\!\$\ \mathcal{D}$' to denote a value being sampled from the distribution $\mathcal{D}$ and assigned to the variable x. When the right-hand side is a set, rather than a distribution, then '$x \leftarrow\!\!\$\ S$' denotes a value being sampled uniformly from the set S and assigned to the variable x.

- **Probabilistic Algorithms:** We use '$x \leftarrow\!\!\$\ \mathsf{fn}(y)$' to denote variable assignment from a probabilistic algorithm 'fn' (called with input 'y') to a variable 'x'. We may write '; r' in the algorithm inputs to unearth the used randomness explicitly, i.e. '$x \leftarrow \mathsf{fn}(y;\ r)$'.

- **Assertions:** The syntax '**require** *boolean-expression*' will evaluate the given expression. Then, if the expression evaluates to **false**, halt execution of the function and return the special failure symbol $\perp$. All state changes occurring during execution of the function containing the assertion are rolled back.

- **Let Expressions:** A *let expression* takes the form '**let** *var* $\in$ *set* **st** *constraints*'. It expresses the process of (a) searching for a value within the set *set* that satisfies the constraints *constraints*, followed by (b) saving the resulting value in the variable *var*. A let expression may, additionally, be extended to handle special cases. First, '**if no matches** : ...' details how to handle the case where no matching value is found. Second,

'**if multiple matches** *subset*: ...' details how to handle the case where there are multiple matching values in the set (saved to the variable named *subset*). Third, '**require unique match** ' asserts that there is exactly one matching value.

- **Deleting Variables:** At times, we may wish to delete a variable. While we could simply set its value to $\perp$, it is sometimes useful to express our intent through the '**del** *var*' expression, which simply sets the value of *var* to $\perp$.

- **Optional Arguments/Return Values:** Procedures may have optional arguments and return values, in which case, they are always filled with a default value. The default value for arguments is represented in the function definition by '*argument-name* $\overset{default}{\longleftarrow}$ *default-value*'. The default for all return values is null, '$\varnothing$'. Since functions always include their optional arguments (or return values), it follows that their type signatures are constant (w.r.t. optional values).

- **Pattern Matching:** Procedures may be defined for particular argument values (or patterns), using the syntax '*argument-name* $\overset{is}{=}$ *pattern-or-value*'. When describing a pattern, the value '$\cdot$' matches any value (including $\varnothing$). If a procedure is not defined for all possible values, any cases left undefined execute no instructions and output '$\perp$' for all return values. If the interpretation of '*pattern-or-value*' is not clear, it should be described in prose as part of the definition. Pattern matching may also be used when unpacking a tuple, in which case the statement should be interpreted as an assertion that is checked before the unpacking operation is executed. In other words,'$x \overset{is}{=} 1, y, z \leftarrow (0, 1, 2)$' is equivalent to '$x_{tmp}, y_{tmp}, z_{tmp} \leftarrow (0, 1, 2)$; **require** $x_{tmp} = 1; x, y, z \leftarrow (0, 1, 2)$'. As with assertions, if a pattern matching operation fails, any state changes that have occurred during execution of the current function are rolled back.

- **Lists:** We represent lists with the notation '$[a, b, c]$'. Sharing most of their semantics with an ordered tuple, lists also support (a) iteration, i.e. '**for** x **in** $[a, b, c]$: ...', (b) indexed access, i.e. '$[a, b, c][1]$' evaluates to b, (c) indexed assignment, i.e. '$[a, b, c][1] \leftarrow d$' results in the list $[a, d, c]$, and (d) slices, i.e. '$[a, b, c][0 \rightarrow 1]$' evaluates to the list $[a, b]$. Slices are inclusive of the endpoint given, i.e. '$xs[s \rightarrow e]$' includes the item at point $xs[e]$. If a slice reaches out-of-bounds, it returns all the elements it can (without error), i.e. '$[a, b, c][0 \rightarrow 10]$' evaluates to $[a, b, c]$. The length of a list can be computed with the len algorithm, i.e. '$\mathsf{len}([a, b, c])$' evaluates to 3. As with tuples, lists can be concatenated, i.e '$[a, b] \parallel [c, d]$' evaluates to $[a, b, c, d]$. We can append an individual item to a list using the $\leftarrow\!\!\!|\!|$ assignment operator, i.e. '$xs \leftarrow [a, b];\ xs \leftarrow\!\!\!|\!|\ c$' results in $xs = [a, b, c]$. We allow enumerating over the elements of a list with the ' **enum in** ' key word, i.e. '**for** (idx, el) **enum in** $[a, b]$: ' will loop over the tuples '$(0, a)$' and '$(1, b)$'. The special index '-1' represents the last item in a list.

- **List Comprehensions:** Lists can be built from other lists using similar syntax (and semantics) to set notation: '$as \leftarrow [\mathsf{fn}(b)$ **for** b **in** bs **if** $\mathsf{cond}(b)]$'.

For example, '$ms \leftarrow [n+1$ **for** n **in** $[0,1,2,\ldots]$ **if** $n \bmod 3 = 0]$' results in $ms = [1,4,7,\ldots]$.

- **Items in Lists:** It can sometimes be useful to make claims about the relative position of two elements in a list. To enable this, we provide the 'a **precedes** b **in** xs' predicate, which is true whenever the element a precedes b in the list xs. Precession implies existence, i.e. '0 **precedes** 1 **in** $[1,2,3]$' is **false**. Such statements are non-inclusive, i.e. '2 **precedes** 2 **in** $[1,2,3]$' evaluates to **false**. We allow pattern matching expressions within a and b, for which 'a **precedes** b **in** xs' returns true if all elements matching a precede all elements matching b in the list xs. This latter case can be regarded as syntactic sugar equivalent to '$\forall\ x, y \in xs : a(x) \wedge b(y) \implies x$ **precedes** y **in** xs'.

- **Functions on Lists:** mru abstracts iteration over a sequence, ordered by most recent use. For lists that utilise this function, we assume that the appropriate state is maintained as they are created and used. sort$(x_1, x_2, \ldots, x_n)$ returns a sorted copy of the list $[x_1, x_2, \ldots, x_n]$.

- **Maps:** A map stores any finite number of key-value pairs, with each key *mapping* to a particular value. Maps are initialised with, and implemented as, an ordered list of tuples: '$m \leftarrow$ Map$\{(x,0),(y,1),(z,2)\}$'. We also support '$key : [value]$' as syntactic sugar, i.e. '$m \leftarrow$ Map$\{x:0, y:1, z:2\}$' is equivalent to the previous example. Note that both the keys and values in a map take the *value* of the identifier given, i.e. '$m[x]$' refers to the slot identified by the value referred to by the identifier 'x' such that for '$n \leftarrow 1$; $m \leftarrow$ Map$\{n : 2\}$' the value of '$m[1]$' is 2. They also support: (a) iteration, i.e. '**for** (k, v) **in** m: ...', in the order of assignment, (b) access, i.e. '$m[y]$' evaluates to 1, and (c) assignment, i.e. '$m[x] \leftarrow 3$' sets the value that m associates with key x to 3. When a value is accessed for a key that does not exist, the result is null: '$m \leftarrow$ Map$\{(0,10),(1,20)\}; x \leftarrow m[2]$' sets the value of x to $\varnothing$. To access a map by insertion order, we use vertical bars such that '$m|0|$' is the first value entered, '$m|2|$' is the second, and so on. These act as ordered tuples or lists, e.g. they may be accessed by index, sliced and the special index '-1' represents the most recently entered item. Replacing a value removes its previous entry and replaces it with a new entry at the end of the list.

- **Objects:** We allow the creation of objects using the '$\langle type, k : v, \ldots\rangle$' notation, which takes as input a string representing the object type followed by a number of key-value pairs and returns an *object*. We model each object as a tuple of values, with each value typed by their given (keyed) slot. Unlike in maps, slots are keyed by an identifier rather than a value (such that the '$.x$' field of an object is unrelated to the value of the identifier 'x'). The type of an object is the type of the resulting tuple (which should be defined in the surrounding prose) and objects can be freely interacted with as if they are an ordered tuple (following the order of the key-value pairs given at creation time). In addition, we afford them the following extra conveniences. Individual properties can be accessed (and written to) using the object name followed by a dot and the respective key, i.e. '$\langle$`example`$, x : 1\rangle.x$' evaluates to `1`. A special

property, '.*type*' returns the type of the object (and is read-only). Objects are implicitly encoded into bit-strings for transmission over the wire, or when generating a signature, for example. Such encodings are intended to approximate, for example, WhatsApp's use of Protocol Buffers to encode messages or Matrix' use of defined JSON structures.

- **Merge:** We support the merging of two objects or two maps using the **merge into** operator, i.e. '**merge** $\mathsf{Map}\{x:1, y:2\}$ **into** $\mathsf{Map}\{y:3, z:4\}$' gives '$\mathsf{Map}\{x:1, y:2, z:4\}$'. Note that values in the first value take overwrite values in the second.

- **Private and Public Keys:** We use the '*sk*' and '*pk*' suffixes to refer to the private and public counterparts of a key pair. These may be prefixed with a letter to indicate their use. For example, we use '(isk, ipk)' to refer to the identity keys of a device and '(xsk, xpk)' to refer to keys used for key exchange. Since WhatsApp re-uses keys for key exchange and signature schemes, we avoid using terms such as *signing key* and *verification key* to specify such keys with dual-use. Within pseudocode, we use the PK algorithm to calculate the public key from a private key.

- **Function Families:** We let '$\mathsf{Func}(n, m)$' denote the set of all functions mapping n-bit strings to m-bit strings.

Security Experiments

- **Oracles:** We signal that a particular algorithm represents an oracle by prepending '$\mathcal{O}$-' to its name. For example, we may write '$\mathcal{O}$-Enc' as the name of an encryption oracle.

- **Experiment Parameters:** We leave the enforcement of usage parameters in an experiment, such as limits on the number of queries, implicit.

- **Shared State:** By default, all algorithms executed by the challenger (i.e. the experiment and its oracles) have access to a shared global state. The adversary only has access to state that is explicitly granted to them through calls to $\mathcal{A}$ or oracles.

- **Abort:** The instruction '**abort**' can be used to end the current experiment early, returning the value 1 to indicate that the adversary has won the experiment.

- **Pattern Matching in Oracles:** We assume that if any check ($\overset{is}{=}$) fails in any of the oracle calls then the oracle stops executing immediately, any state changes made during execution of the current oracle are rolled back, and it returns $\perp$. We suppress type checks as these are not cryptographically enforced.

- **Terminating the Experiment:** The challenger may terminate the experiment by executing the '**terminate with** x' instruction, which ends the experiment immediately with the value 'x'. They may do so at any point in which they have control over the execution, i.e. both within the

main experiment procedure, or while processing an oracle query. This differs from an '**abort** ' instruction only in its intent: we use aborts as a tool within our proofs to handle particular events, while we use early termination within our definition to help capture a particular security notion.

- **Instant Win :** Importantly, the '**terminate with** x' instruction enables the challenger to terminate the experiment immediately upon detecting that a win condition has been fulfilled. Following this approach avoids the possibility of temporarily satisfied win conditions not being captured as a win by the experiment [BCJ+24].

Security Proofs

When building an adversary as part of a security reduction, we highlight the relevant changes to the emulated experiment, e.g. where the challenge from the outer experiment is embedded, marked as '$c \leftarrow \mathsf{AEAD.Enc}(h, m)$'.

When performing a security reduction, in order to bound the change in advantage introduced by the changes between two games, we start by bounding with respect to the particular adversary we have constructed in our reduction, before proceeding to bound this against any adversary limited to the appropriate parameterisation. In most cases, we leave the first of these steps implicit, going straight from the security reduction in which we construct a particular adversary, to a bound with respect to any probabilistic polynomial-time adversary.

2.3 Primitives & Building Blocks

This section provides a reference for the various cryptographic primitives and building blocks utilised by the protocols we study. For each of these, we include a brief intuition, its syntax and the relevant security definitions.

2.3.1 Idealised Hash Functions

Our security analysis and proofs take place in the *random oracle model*. That is, we assume that the hash functions used and, in some cases, the structures built upon them, behave in an idealised manner.

In particular, a security experiment in the random oracle model provides controlled access to an oracle, $\mathcal{O}\text{-}\mathsf{RO}$, that implements a random function of the form $\mathsf{RO} : \{0,1\}^{\mathsf{RO.il}} \to \{0,1\}^{\mathsf{RO.ol}}$ where $\mathsf{RO.il}$ is the input length and $\mathsf{RO.ol}$ is the output length. This oracle is implemented equivalently to the pseudocode in Figure 2.1 (following the definitions of Section 8.1 in [MF21]).

When we make the assumption that a function, say $\mathsf{fn} : \{0,1\}^n \to \{0,1\}^m$, is a random oracle, we replace all executions of fn with a call to a random oracle instantiated with input length $\mathsf{RO.il} = n$ and output length $\mathsf{RO.ol} = m$. We sometimes make multiple such assumptions within a single experiment, in which case we instantiate multiple independent random oracles.

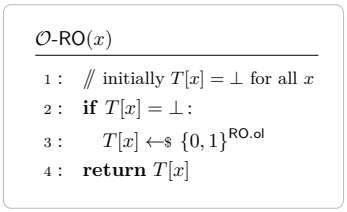


Figure 2.1. A probabilistic algorithm implementing a random oracle, RO, with input length RO.il and output length RO.ol.

2.3.2 Pseudorandom Functions

We follow [KL14, Section 3.5] in defining pseudorandom functions.

Definition 2.5 (Pseudorandom Function)**.** A PRF consists of a single algorithm, PRF, taking as input a key k, of length λ_k, an input x, of length n, and outputting a string y, of length $\ell(n)$:

$$\mathsf{PRF} : \{0,1\}^{\lambda_k} \times \{0,1\}^n \to \{0,1\}^{\ell(n)}$$

We call $\ell(n)$ the PRF's *expansion factor*.

Intuitively, a pseudorandom function captures a family of functions, and the key input selects a particular function within that family. We expect a secure PRF to, given a random key that is secret from the adversary, output pseudorandom bit-strings (even when the adversary has knowledge and control over the message input). We capture this notion as follows.

Definition 2.6 (Security of Pseudorandom Functions)**.** A PRF with key length λ_k, input length n and expansion factor $\ell(n)$ is secure if any probabilistic polynomial-time adversary $\mathcal{A}$, restricted to making up to n_q queries to $\mathcal{O}$-PRF, has a negligible decision-advantage in winning the $\mathsf{PRF}^{\mathsf{PRF}}_{\lambda_k, n_q}(\mathcal{A})$ security experiment detailed in Figure 2.2.

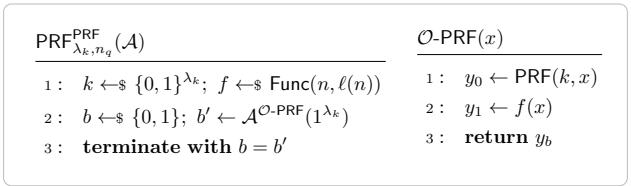


Figure 2.2. The security of PRF with seed length n and expansion factor $\ell(n)$.

The protocols we study in this thesis make frequent use of the HMAC [KBC97] and HKDF [KE10, Kra10] algorithms instantiated with the SHA256 function [SHA02]. Both Matrix and WhatsApp make use of these primitives in such a way that necessitate PRF-style assumptions (in addition to their standard use as a message authentication code and key derivation function, respectively). We discuss these assumptions in the security analysis of each (see Section 6.1.3 for Matrix and Section 6.2.3 for WhatsApp). Throughout, we use HMAC and HKDF to refer to those primitives instantiated with SHA256, i.e. HMAC-SHA256 and HKDF-SHA256 respectively.

2.3.3 Message Authentication Codes

A message authentication code (MAC) allows parties which share a secret to tag messages, guaranteeing that (a) only those parties possessing the shared secret may create valid tags for a message, and (b) a tag will only validate against the original message it was generated for. They may be used, for example, to detect if messages sent over an untrusted network have been modified. We follow the definition and security notions of [Section 4.2][KL14].

Definition 2.7 (Message Authentication Code)**.** A message authentication code (MAC) scheme consists of two algorithms $\mathsf{MAC} = (\mathsf{Gen}, \mathsf{Tag}, \mathsf{Verify})$ with an associated key space $\mathcal{K} = \{0,1\}^\lambda$, message space $\mathcal{M} \subseteq \{0,1\}^*$ and tag space $\mathcal{T} \subseteq \{0,1\}^*$.

1) The key generation algorithm, $\mathsf{MAC.Gen} : \emptyset \to \mathcal{K}$, takes no input before outputting a new key.

2) The tag generation algorithm, $\mathsf{MAC.Tag} : \mathcal{K} \times \mathcal{M} \xrightarrow{\$} \mathcal{T}$, takes as input a key and message before outputting a tag.

3) The tag verification algorithm, $\mathsf{MAC.Verify} : \mathcal{K} \times \mathcal{M} \times \mathcal{T} \to \{0,1\}$, takes as input a key, a message and a tag before outputting a bit indicating whether the tag is valid.

Definition 2.8 (Correctness of MAC Schemes)**.** A MAC scheme is correct if, for all $k \in \mathcal{K}$ and $m \in \mathcal{M}$,

$$\mathsf{MAC.Verify}(k, m, \mathsf{MAC.Tag}(k, m)) = 1.$$

We consider two variants of unforgeabilty for MAC schemes. The first, *existential unforgeability*, asks that only those with access to the secret key may generate a valid tag for a new message (that has not been tagged before). The second, *strong* existential unforgeability, asks that only those with access to the secret key may generate a new tag for a message (regardless of whether it has been tagged before).

Definition 2.9 (Existential Unforgeability of MAC Schemes)**.** A MAC scheme MAC provides existential unforgeability under chosen message attack (EUF-CMA) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible search-advantage of winning the $\mathsf{EUF\text{-}CMA}^{\mathsf{MAC}}_{\Lambda_{\mathsf{EUF\text{-}CMA}}}(\mathcal{A})$ security experiment detailed in Figure 2.3. The experiment is parameterised by $\Lambda_{\mathsf{EUF\text{-}CMA}} = n_q$, for which n_q limits the number of $\mathcal{O}$-Tag queries that the adversary may make before submitting their guess.

Definition 2.10 (Strong Existential Unforgeability of MAC Schemes)**.** A MAC scheme MAC provides strong existential unforgeability under chosen message attack (SUF-CMA) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible search-advantage of winning the $\mathsf{SUF\text{-}CMA}^{\mathsf{MAC}}_{\Lambda_{\mathsf{SUF\text{-}CMA}}}(\mathcal{A})$ security experiment detailed in Figure 2.4. The experiment is parameterised by $\Lambda_{\mathsf{SUF\text{-}CMA}} = n_q$, for which n_q limits the number of $\mathcal{O}$-Tag queries that the adversary may make before submitting their guess.

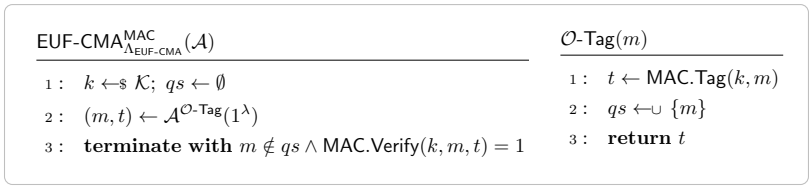


Figure 2.3. The EUF-CMA security experiment for MAC schemes.

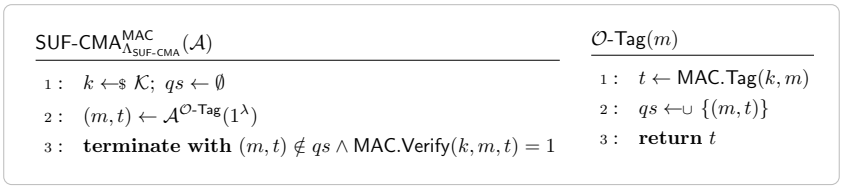


Figure 2.4. The SUF-CMA security experiment for MAC schemes.

The protocols we study use the $\mathsf{HMAC}(k,m)$ (or HMAC-SHA256) Hash-based Message Authentication Code (HMAC) constructed with SHA256 [SHA02], taking as input a key k and message m [KBC97]. The HMAC RFC [KBC97] recommends using at least 128 bits of output for SHA-256 (no less than half the length of the hash output) and no less than 80 bits.

2.3.4 Key Derivation Functions

A key-derivation function takes as input a secret string that, although it may contain sufficient entropy, is not in the correct format to be used as key material, and transforms it into a pseudorandom string of the requisite length [Kra10, KE10]. This is useful in key exchange, for example, where the secret that is output may be *random* but is unsuitable for use in, say, symmetric encryption because it is neither uniformly distributed nor the length required by the encryption algorithm. As such, a key derivation function aims to transform the given string into a string of the require length that is computationally-indistinguishable from uniform, all the while maintaining the entropy contained in the original string. We follow Section 2 of [Kra10] in our definition of key derivation functions (KDFs).

Definition 2.11 (Key Derivation Function)**.** A KDF consists of a single algorithm, KDF, taking as input four values:

1) the salt, $s \in \mathcal{SLT}$ where $\mathcal{SLT} \subseteq \{0,1\}^*$, provides public randomness,

2) the initial keying material, $(k, aux) \in \mathcal{SKM}$, is the secret sample k and auxillary knowledge aux sampled from a source of keying material $\mathcal{SKM}$,

3) the context (or information), $i \in \mathcal{CTX}$ where $\mathcal{CTX} \subseteq \{0,1\}^*$, and

4) the length in bits, $l \in \mathbb{N}$, of the key material to be output

before outputting a key of length l bits. This gives

$$\mathsf{KDF} : \mathcal{SKM} \times \mathbb{N} \times \mathcal{SLT} \times \mathcal{CTX} \to \{0,1\}^l.$$

Note that, in the cases we consider, the KDF function itself is deterministic. Nonetheless, we say that a KDF is *randomised* when it takes a salt as input and *deterministic* when it does not. The source of keying material, $\mathcal{SKM}$, is a two-valued probability distribution outputting a secret sample, k, and auxiliary knowledge, aux, whereby the latter is made available to the attacker.

Definition 2.12 (Security of Key Derivation Functions)**.** A KDF is m-entropy secure, providing mKDF security, if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible decision-advantage of winning the $\mathsf{mKDF}^{\mathsf{KDF}}_{\Lambda_{\mathsf{mKDF}}}(\mathcal{A})$ security experiment detailed in Figure 2.5 with respect to all sources of keying material $\mathcal{SKM}$ that provide m-entropy. The experiment is parameterised by $\Lambda_{\mathsf{mKDF}} = n_q$, for which n_q limits the number of non-challenge queries the adversary may make.

$\mathsf{mKDF}^{\mathsf{KDF}}_{\Lambda_{\mathsf{mKDF}}}(\mathcal{A})$

1 : $b \leftarrow\$ \{0,1\};\ qs \leftarrow \emptyset$
2 : $k, aux \leftarrow\$ \mathcal{SKM};\ s \leftarrow\$ \mathcal{SLT}$
3 : $b' \leftarrow \mathcal{A}^{\mathcal{O}\text{-KDF},\mathcal{O}\text{-Challenge}}(aux, s)$
4 : **terminate with** $b = b'$

$\mathcal{O}\text{-KDF}(i, l)$

1 : **require** $i \notin qs$
2 : $qs \leftarrow\cup \{i\}$
3 : **return** $\mathsf{KDF}(s, k, i, l)$

$\mathcal{O}\text{-Challenge}(i, l)$

1 : **require** $i \notin qs$
2 : $qs \leftarrow\cup \{i\}$
3 : $k_0 \leftarrow \mathsf{KDF}(s, k, i, l)$
4 : $k_1 \leftarrow\$ \{0,1\}^l$
5 : **return** k_b

Figure 2.5. The security of KDF with respect to an m-entropy source of keying material, $\mathcal{SKM}$.

2.3.5 Symmetric Encryption

A symmetric encryption scheme allows parties that share a secret to exchange encrypted messages whose contents remain confidential between themselves. Our syntax loosely follows the definitions of [BDJR97].

Definition 2.13 (Symmetric Encryption Scheme)**.** A symmetric encryption (SE) scheme consists of two algorithms $\mathsf{SE} = (\mathsf{Enc}, \mathsf{Dec})$ with an associated key space $\mathcal{K} = \{0,1\}^\lambda$, message space $\mathcal{M} \subseteq \{0,1\}^*$ and ciphertext space $\mathcal{C} \subseteq \{0,1\}^*$.

1) The encryption algorithm, $\mathsf{SE.Enc} : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$, takes as input a key and plaintext before outputting a ciphertext.

2) The decryption algorithm, $\mathsf{SE.Dec} : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$, takes as input a key and ciphertext before outputting the plaintext.

We expect that ciphertexts reliably decrypt to the original plaintext (given the correct key).

Definition 2.14 (Correctness of Symmetric Encryption)**.** A SE scheme is correct if, for all $k \in \mathcal{K}$ and $m \in \mathcal{M}$,

$$\mathsf{SE.Dec}(k, \mathsf{SE.Enc}(k, m)) = m$$

and the ciphertext output by SE.Enc is equal in length to the plaintext given to it.

We start by considering a left-or-right style security notion. For a fixed secret key, the adversary is given access to an encryption oracle, $\mathcal{O}$-Enc, that takes as input to candidate plaintexts, m_0 and m_1, and outputs the encryption of m_b where b is a hidden bit chosen uniformly at random and known only by the challenger. The adversary wins the game if they correctly guess the hidden bit.

Definition 2.15 (IND-CPA Security of Symmetric Encryption)**.** A SE scheme SE provides *indistinguishability under chosen-plaintext attack* (IND-CPA security) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible decision-advantage of winning the $\mathsf{IND\text{-}CPA}^{\mathsf{SE}}_{n_q,n_{ch}}(\mathcal{A})$ security experiment detailed in Figure 2.6. The experiment is parameterised by $\Lambda_{\mathsf{IND\text{-}CPA}} = (n_q,n_{ch})$, for which n_q limits the total number of queries the adversary may make while n_{ch} limits the number of challenges (calls to Enc for which $m_0 \neq m_1$).

$\mathsf{IND\text{-}CPA}^{\mathsf{SE}}_{\Lambda_{\mathsf{IND\text{-}CPA}}}(\mathcal{A})$

1 : $b \leftarrow\$ \{0,1\}$
2 : $k \leftarrow\$ \mathcal{K}$
3 : $b' \leftarrow \mathcal{A}^{\mathcal{O}\text{-}\mathsf{Enc}}(1^\lambda)$
4 : **terminate with** $b = b'$

$\mathcal{O}\text{-}\mathsf{Enc}(m_0, m_1)$

1 : **require** $\mathsf{len}(m_0) = \mathsf{len}(m_1)$
2 : $c_0 \leftarrow \mathsf{SE.Enc}(k, m_0)$
3 : $c_1 \leftarrow \mathsf{SE.Enc}(k, m_1)$
4 : **return** c_b

Figure 2.6. The IND-CPA security experiment for symmetric encryption.

Chosen-ciphertext security aims to capture contexts where the adversary additionally has access to a decryption oracle. We define this notion as follows.

Definition 2.16 (IND-CCA Security of Symmetric Encryption)**.** A SE scheme SE provides *indistinguishability under chosen-ciphertext attack* (IND-CCA security) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible decision-advantage of winning the $\mathsf{IND\text{-}CCA}^{\mathsf{SE}}_{n_q,n_{ch}}(\mathcal{A})$ security experiment detailed in Figure 2.7. The experiment is parameterised by $\Lambda_{\mathsf{IND\text{-}CCA}} = (n_q,n_{ch})$, for which n_q limits the total number of queries the adversary may make while n_{ch} limits the number of challenge queries, specifically.

$\mathsf{IND\text{-}CCA}^{\mathsf{SE}}_{\Lambda_{\mathsf{IND\text{-}CCA}}}(\mathcal{A})$

1 : $b \leftarrow\$ \{0,1\}$
2 : $k \leftarrow\$ \mathcal{K}$
3 : $ch \leftarrow \emptyset$
4 : $b' \leftarrow \mathcal{A}^{\mathcal{O}\text{-}\mathsf{Enc},\mathcal{O}\text{-}\mathsf{Dec},\mathcal{O}\text{-}\mathsf{Challenge}}(1^\lambda)$
5 : **terminate with** $b = b'$

$\mathcal{O}\text{-}\mathsf{Enc}(m)$

1 : $c \leftarrow \mathsf{SE.Enc}(k, m)$
2 : **return** c

$\mathcal{O}\text{-}\mathsf{Dec}(c)$

1 : **require** $c \notin ch$
2 : $m \leftarrow \mathsf{SE.Dec}(k, c)$
3 : **return** m

$\mathcal{O}\text{-}\mathsf{Challenge}(m_0, m_1)$

1 : **require** $\mathsf{len}(m_0) = \mathsf{len}(m_1)$
2 : $c_0 \leftarrow \mathsf{SE.Enc}(k, m_0)$
3 : $c_1 \leftarrow \mathsf{SE.Enc}(k, m_1)$
4 : $ch \leftarrow\cup \{c_b\}$
5 : **return** c_b

Figure 2.7. The IND-CCA security experiment for symmetric encryption.

The protocols we study primarily use AES [AES01] for symmetric encryption. We write $\mathsf{AES}(k, m)$ to mean AES taking as input a key k and a message block m of length 128 bits and outputting a ciphertext c of length 128 bits.

To encrypt variable length messages, these schemes typically split the plaintext into blocks of a set size, as described by PKCS7 [Hou09], in tandem with the following block cipher modes of operation.

- **Cipher Block Chaining:** $\mathsf{AES\text{-}CBC} = (\mathsf{Enc}, \mathsf{Dec})$ is AES in cipher block chaining (CBC) mode where, for $\mathsf{Enc}(iv, k, m)$ and $\mathsf{Dec}(iv, k, c)$, iv is the nonce, k is an AES encryption key, m is a message and c is a ciphertext [Dwo01].

- **Counter:** $\mathsf{AES\text{-}CTR} = (\mathsf{Enc}, \mathsf{Dec})$ is AES in counter (i.e. CTR) mode where, for $\mathsf{Enc}(iv, k, m)$ and $\mathsf{Dec}(iv, k, c)$, iv is the nonce, k is an AES encryption key, m is a message and c is a ciphertext [Dwo01].

Authenticated Encryption.

An authenticated encryption (AE) scheme aims to protect both the authenticity and privacy of messages that are sent. In particular, parties with access to the shared secret key are able to encrypt messages and decrypt ciphertexts with the guarantees that only those with the same key may (a) access the plaintext of messages, (b) create new valid messages, or (c) modify their contents. We follow [RBBK01] in our definition of AE.

Definition 2.17 (AE Scheme)**.** An AE scheme consists of two algorithms $\mathsf{AE} = (\mathsf{Enc}, \mathsf{Dec})$ with an associated key space $\mathcal{K} = \{0,1\}^\lambda$, nonce space $\mathcal{N} = \{0,1\}^{\lambda_n}$, message space $\mathcal{M} \subseteq \{0,1\}^*$ and ciphertext space $\mathcal{C} \subseteq \{0,1\}^*$.

1) The encryption algorithm, $\mathsf{AE.Enc} : \mathcal{K} \times \mathcal{N} \times \mathcal{M} \to \mathcal{C}$, takes as input a key, nonce, and plaintext before outputting a ciphertext.

2) The decryption algorithm, $\mathsf{AE.Dec} : \mathcal{K} \times \mathcal{N} \times \mathcal{C} \to \mathcal{M}$, takes as input a key, nonce, and ciphertext before outputting the plaintext.

Definition 2.18 (Correctness of AE Schemes)**.** An AE scheme is correct if, for all $k \in \mathcal{K}$, $n \in \mathcal{N}$ and $m \in \mathcal{M}$,

$$\mathsf{AE.Dec}(k, n, \mathsf{AE.Enc}(k, n, m)) = m$$

and the ciphertext output by AE.Enc is equal in length to $\ell(|m|)$, for some efficiently computable *length function* ℓ.

Definition 2.19 (IND$-CPA Security of AE Schemes)**.** An AE scheme AE provides indistinguishability from random under chosen plaintext attack (IND$-CPA) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible decision-advantage of winning the $\mathsf{IND\$\text{-}CPA}^{\mathsf{AE}}_{\Lambda_{\mathsf{IND\$\text{-}CPA}}}(\mathcal{A})$ security experiment detailed in Figure 2.8. The experiment is parameterised by $\Lambda_{\mathsf{IND\$\text{-}CPA}} = n_m$, for which n_m limits the number of encryption queries the adversary may make before submitting their guess of the challenge bit b.

Definition 2.20 (Existential Unforgeability of AE Schemes)**.** An AE scheme AE provides existential unforgeability under chosen message attack (EUF-CMA) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible search-advantage of winning the $\mathsf{EUF\text{-}CMA}^{\mathsf{AE}}_{\Lambda_{\mathsf{EUF\text{-}CMA}}}(\mathcal{A})$ security experiment detailed in Figure 2.9. The experiment is parameterised by $\Lambda_{\mathsf{EUF\text{-}CMA}} = n_m$, for which n_m limits the number of encryption queries the adversary may make before submitting their challenge ciphertext.

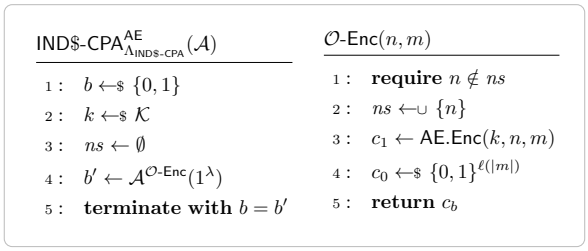


Figure 2.8. The IND$-CPA security experiment for AE schemes.

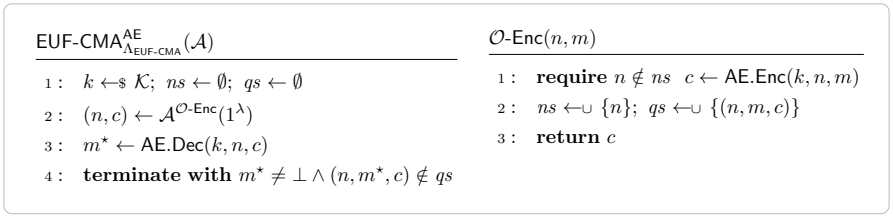


Figure 2.9. The EUF-CMA security experiment for AE schemes.

Note that the definitions above forbid the re-use of nonces, essentially assuming that our adversary is *nonce-respecting*. Since the experiments allow for multiple challenges against the same plaintext, and the encryption and decryption algorithms are deterministic, this restriction is needed to ensure that the notions are satisfiable.

The protocols we study take a different approach to ensuring that the same (n, k, m) combination is never reused. Secure messaging applications typically derive a series of per-message keys, from which a nonce, encryption key and authentication key are all generated, *deterministically*. In this setting, the adversary does not have control over the nonce and, further, this nonce is also opaque to the challenger. While we believe that such constructions should be secure, since both the key and nonce are unique to each message, the security notions above are not appropriate for this setting. We require something different.

In what follows, we define a one-time authenticated encryption (AE) scheme. Such schemes are both deterministic and do not require a nonce, with the caveat that any particular key may only be used to encrypt a single message.

Definition 2.21 (One-Time AE Scheme)**.** A one-time AE scheme consists of two algorithms $\mathsf{AE} = (\mathsf{Enc}, \mathsf{Dec})$ with an associated key space $\mathcal{K} = \{0,1\}^\lambda$ (with a linear-time membership test), message space $\mathcal{M} \subseteq \{0,1\}^*$ and ciphertext space $\mathcal{C} \subseteq \{0,1\}^*$.

1) The encryption algorithm, $\mathsf{AE.Enc} : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$, takes as input a key and plaintext before outputting a ciphertext.

2) The decryption algorithm, $\mathsf{AE.Dec} : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$, takes as input a key and ciphertext before outputting the plaintext.

Definition 2.22 (Correctness of One-Time AE Schemes)**.** A one-time AE scheme is correct if, for all $k \in \mathcal{K}$ and $m \in \mathcal{M}$,

$$\mathsf{Dec}(k, \mathsf{Enc}(k, m)) = m$$

and the ciphertext output by Enc is equal in length to $\ell(|m|)$, for some efficiently computable *length function* ℓ.

Definition 2.23 (IND$-CPA Security of One-Time AE Schemes)**.** A one-time AE scheme $\mathsf{AE} = (\mathsf{Enc}, \mathsf{Dec})$ provides indistinguishability from random under chosen plaintext attack (IND$-CPA) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible decision-advantage of winning the $\mathsf{IND\$\text{-}CPA}_1^{\mathsf{AE}}(\mathcal{A})$ security experiment detailed in Figure 2.10.

$\mathsf{IND\$\text{-}CPA}_1^{\mathsf{AE}}(\mathcal{A})$	$\mathcal{O}\text{-}\mathsf{Enc}(m)$
1 : $b \leftarrow\$ \{0,1\}$	1 : **require** $m^\star = \varnothing$
2 : $k \leftarrow\$ \mathcal{K}$	2 : $m^\star \leftarrow m$
3 : $m^\star \leftarrow \varnothing$	3 : $c_1 \leftarrow \mathsf{AE.Enc}(k, m)$
4 : $b' \leftarrow \mathcal{A}^{\mathcal{O}\text{-}\mathsf{Enc}}(1^\lambda)$	4 : $c_0 \leftarrow\$ \{0,1\}^{\ell\|m\|}$
5 : **terminate with** $b = b'$	5 : **return** c_b

Figure 2.10. The IND$-CPA security experiment for one-time AE schemes.

Definition 2.24 (Existential Unforgeability of One-Time AE Schemes)**.** A one-time AE scheme AE provides existential unforgeability under chosen message attack (EUF-CMA) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible search-advantage of winning the $\mathsf{EUF\text{-}CMA}_1^{\mathsf{AE}}(\mathcal{A})$ security experiment detailed in Figure 2.11.

$\mathsf{EUF\text{-}CMA}_1^{\mathsf{AE.Enc}}(\mathcal{A})$	$\mathcal{O}\text{-}\mathsf{Enc}(m)$
1 : $k \leftarrow\$ \mathcal{K};\ q \leftarrow \varnothing$	1 : **require** $q = \varnothing$
2 : $c \leftarrow \mathcal{A}^{\mathcal{O}\text{-}\mathsf{Enc}}(1^\lambda)$	2 : $c \leftarrow \mathsf{AE.Enc}(k, m)$
3 : $m^\star \leftarrow \mathsf{AE.Dec}(k, c)$	3 : $q \leftarrow c$
4 : **terminate with** $m^\star \neq \perp \wedge c \neq q$	4 : **return** c

Figure 2.11. The EUF-CMA security experiment for one-time AE schemes.

Authenticated Encryption with Associated Data.

It is common to attach public, unencrypted data to such ciphertexts. This *associated data* may help with routing the ciphertext or in determining which key should be used to decrypt it, for example. An authenticated encryption with associated data (AEAD) scheme enables such data to be cryptographically bound to the ciphertext it is attached to, in addition to providing authenticity and privacy for the ciphertext itself. We follow [Rog02] in our definition of AEAD.

Definition 2.25 (AEAD Scheme)**.** An AEAD scheme consists of two algorithms $\mathsf{AEAD} = (\mathsf{Enc}, \mathsf{Dec})$ with an associated key space $\mathcal{K} = \{0,1\}^{\lambda}$, nonce space $\mathcal{N} = \{0,1\}^{\lambda_n}$, header space $\mathcal{H} \subseteq \{0,1\}^*$ (with a linear-time membership test), message space $\mathcal{M} \subseteq \{0,1\}^*$ and ciphertext space $\mathcal{C} \subseteq \{0,1\}^*$.

1) The encryption algorithm, $\mathsf{AEAD.Enc} : \mathcal{K} \times \mathcal{N} \times \mathcal{H} \times \mathcal{M} \to \mathcal{C}$, takes as input a key, nonce, header and plaintext before outputting a ciphertext.

2) The decryption algorithm, $\mathsf{AEAD.Dec} : \mathcal{K} \times \mathcal{N} \times \mathcal{H} \times \mathcal{C} \to \mathcal{M}$, takes as input a key, nonce, header and ciphertext before outputting the plaintext.

Definition 2.26 (Correctness of AEAD Schemes)**.** An AEAD scheme is correct if, for all $k \in \mathcal{K}$, $n \in \mathcal{N}$, $h \in \mathcal{H}$ and $m \in \mathcal{M}$,

$$\mathsf{AEAD.Dec}(k, n, h, \mathsf{AEAD.Enc}(k, n, h, m)) = m$$

and the ciphertext output by $\mathsf{AEAD.Enc}$ is equal in length to $\ell(|m|)$, for some efficiently computable *length function* ℓ.

Definition 2.27 (IND$-CPA Security of AEAD Schemes)**.** An AEAD scheme AEAD provides indistinguishability from random under chosen plaintext attack (IND$-CPA) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible decision-advantage of winning the $\mathsf{IND\$\text{-}CPA}^{\mathsf{AEAD}}_{\Lambda_{\mathsf{IND\$\text{-}CPA}}}(\mathcal{A})$ security experiment detailed in Figure 2.12. The experiment is parameterised by $\Lambda_{\mathsf{IND\$\text{-}CPA}} = n_m$, for which n_m limits the number of encryption queries the adversary may make before submitting their guess of the challenge bit b.

$\mathsf{IND\$\text{-}CPA}^{\mathsf{AEAD}}_{\Lambda_{\mathsf{IND\$\text{-}CPA}}}(\mathcal{A})$

1 : $b \leftarrow\$ \{0,1\}$
2 : $k \leftarrow\$ \mathcal{K}$
3 : $ns \leftarrow \emptyset$
4 : $b' \leftarrow \mathcal{A}^{\mathcal{O}\text{-}\mathsf{Enc}}(1^{\lambda})$
5 : **terminate with** $b = b'$

$\mathcal{O}\text{-}\mathsf{Enc}(n, h, m)$

1 : **require** $n \notin ns$
2 : $ns \leftarrow\cup \{n\}$
3 : $c_1 \leftarrow \mathsf{AEAD.Enc}(k, n, h, m)$
4 : $c_0 \leftarrow\$ \{0,1\}^{\ell(|m|)}$
5 : **return** c_b

Figure 2.12. The $\mathsf{IND\$\text{-}CPA}$ security experiment for AEAD schemes.

Definition 2.28 (Existential Unforgeability of AEAD Schemes)**.** An AEAD scheme AEAD provides existential unforgeability under chosen message attack (EUF-CMA) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible search-advantage of winning the $\mathsf{EUF\text{-}CMA}^{\mathsf{AEAD}}_{\Lambda_{\mathsf{EUF\text{-}CMA}}}(\mathcal{A})$ security experiment detailed in Figure 2.13. The experiment is parameterised by $\Lambda_{\mathsf{EUF\text{-}CMA}} = n_m$, for which n_m limits the number of encryption queries the adversary may make before submitting their challenge ciphertext.

As in Section 2.3.5, we define one-time variants of AEAD schemes in a similar manner to one-time AE.

Definition 2.29 (One-Time AEAD Scheme)**.** A one-time AEAD scheme consists of two algorithms $\mathsf{AEAD} = (\mathsf{Enc}, \mathsf{Dec})$ with an associated key space $\mathcal{K} = \{0,1\}^{\lambda}$, header space $\mathcal{H} \subseteq \{0,1\}^*$ (with a linear-time membership test), message space $\mathcal{M} \subseteq \{0,1\}^*$ and ciphertext space $\mathcal{C} \subseteq \{0,1\}^*$.

$\mathsf{EUF\text{-}CMA}^{\mathsf{AEAD}}_{\Lambda_{\mathsf{EUF\text{-}CMA}}}(\mathcal{A})$

1 : $k \leftarrow\!\$\ \mathcal{K};\ qs \leftarrow \emptyset;\ ns \leftarrow \emptyset$
2 : $(n, h, c) \leftarrow \mathcal{A}^{\mathcal{O}\text{-}\mathsf{Enc}}(1^\lambda)$
3 : $m^\star \leftarrow \mathsf{AEAD.Dec}(k, n, h, c)$
4 : **terminate with** $m^\star \neq \perp \wedge (n, h, m^\star, c) \notin qs$

$\mathcal{O}\text{-}\mathsf{Enc}(n, h, m)$

1 : **require** $n \notin ns$
2 : $c \leftarrow \mathsf{AEAD.Enc}(k, n, h, m)$
3 : $qs \leftarrow\!\cup\ \{(n, h, m, c)\};\ ns \leftarrow\!\cup\ \{n\}$
4 : **return** c

Figure 2.13. The EUF-CMA security experiment for AEAD schemes.

1) The encryption algorithm, $\mathsf{AEAD.Enc} : \mathcal{K} \times \mathcal{H} \times \mathcal{M} \to \mathcal{C}$, takes as input a key, header and plaintext before outputting a ciphertext.

2) The decryption algorithm, $\mathsf{AEAD.Dec} : \mathcal{K} \times \mathcal{H} \times \mathcal{C} \to \mathcal{M}$, takes as input a key, header and ciphertext before outputting the plaintext.

Definition 2.30 (Correctness of One-Time AEAD Schemes)**.** A one-time AEAD scheme is correct if, for all $k \in \mathcal{K}$, $h \in \mathcal{H}$ and $m \in \mathcal{M}$,

$$\mathsf{Dec}(k, h, \mathsf{Enc}(k, h, m)) = m$$

and the ciphertext output by Enc is equal in length to $\ell(|m|)$, for some efficiently computable *length function* ℓ.

Definition 2.31 (IND$-CPA Security of One-Time AEAD Schemes)**.** A one-time AEAD scheme $\mathsf{AEAD} = (\mathsf{Enc}, \mathsf{Dec})$ provides indistinguishability from random under chosen plaintext attack (IND$-CPA) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible decision-advantage of winning the $\mathsf{IND\$\text{-}CPA}^{\mathsf{AEAD}}_1(\mathcal{A})$ security experiment detailed in Figure 2.14.

$\mathsf{IND\$\text{-}CPA}^{\mathsf{AEAD}}_1(\mathcal{A})$

1 : $b \leftarrow\!\$\ \{0, 1\}$
2 : $k \leftarrow\!\$\ \mathcal{K}$
3 : $m^\star \leftarrow \varnothing$
4 : $b' \leftarrow \mathcal{A}^{\mathcal{O}\text{-}\mathsf{Enc}}(1^\lambda)$
5 : **terminate with** $b = b'$

$\mathcal{O}\text{-}\mathsf{Enc}(h,\ m)$

1 : **require** $m^\star = \varnothing$
2 : $m^\star \leftarrow m$
3 : $c_1 \leftarrow \mathsf{AEAD.Enc}(k, h, m)$
4 : $c_0 \leftarrow\!\$\ \{0, 1\}^{\ell(|m|)}$
5 : **return** c_b

Figure 2.14. The IND$-CPA security experiment for one-time AEAD schemes.

Definition 2.32 (Existential Unforgeability of One-Time AEAD Schemes)**.** A one-time AEAD scheme AEAD provides existential unforgeability under chosen message attack (EUF-CMA) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible search-advantage of winning the $\mathsf{EUF\text{-}CMA}^{\mathsf{AEAD}}_1(\mathcal{A})$ security experiment detailed in Figure 2.15.

The protocols we study use one-time AEAD schemes constructed from AES-CBC (for encryption) and HMAC-SHA256 (for authentication).

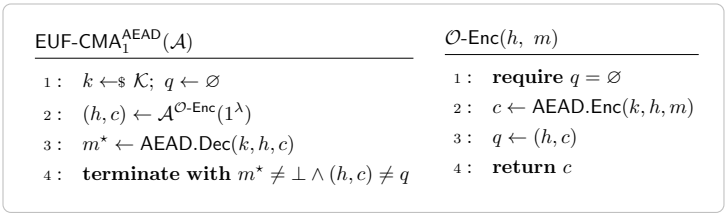


Figure 2.15. The EUF-CMA security experiment for one-time AEAD schemes.

2.3.6 Signature Schemes

Definition 2.33 (Signature Scheme)**.** A signature scheme DS consists of three probabilistic polynomial-time (PPT) algorithms (Gen, Sign, Verify) such that:

1) The key generation algorithm is a randomised algorithm that takes as input a security parameter 1^λ and outputs a pair (sk, pk), the *secret key* and *public key*, respectively. We write $(sk, pk) \leftarrow\$ \mathsf{DS.Gen}(1^\lambda)$.

2) The signing algorithm takes as input a secret key sk, a message m and outputs a signature σ. We write this as $\sigma \leftarrow\$ \mathsf{DS.Sign}(sk, m)$. The signing algorithm may be randomised or deterministic.

3) The verification algorithm takes as input a public key vk, a signature σ and a message m and outputs a bit b, with $b = 1$ meaning the signature is valid and $b = 0$ meaning the signature is invalid. $\mathsf{DS.Verify}$ is a deterministic algorithm. We write $b \leftarrow \mathsf{DS.Verify}(vk, \sigma, m)$.

We require that except with negligible probability over $(sk, pk) \leftarrow\$ \mathsf{DS.Gen}(1^\lambda)$, it holds that $\mathsf{DS.Verify}(pk, \mathsf{DS.Sign}(sk, m), m) = 1$ for all m.

Definition 2.34 (Existential Unforgeability of Signature Schemes)**.** A signature scheme DS provides existential unforgeability under chosen message attack (EUF-CMA) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible search-advantage of winning the $\mathsf{EUF\text{-}CMA}^{\mathsf{DS}}_{\Lambda_{\mathsf{EUF\text{-}CMA}}}(\mathcal{A})$ security experiment detailed in Figure 2.16. The experiment is parameterised by $\Lambda_{\mathsf{EUF\text{-}CMA}} = n_q$, for which n_q limits the number of signature queries the adversary may make before submitting their forgery.

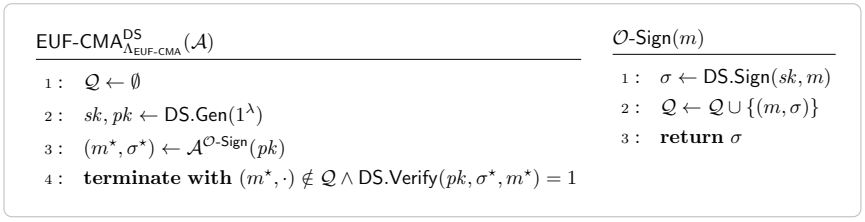


Figure 2.16. The EUF-CMA security experiment for signature schemes.

Definition 2.35 (Strong Existential Unforgeability of Signature Schemes)**.** A signature scheme DS provides strong existential unforgeability under chosen

message attack (SUF-CMA) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible decision-advantage of winning the $\mathsf{SUF\text{-}CMA}^{\mathsf{DS}}_{\Lambda_{\mathsf{SUF\text{-}CMA}}}(\mathcal{A})$ security experiment detailed in Figure 2.17. The experiment is parameterised by $\Lambda_{\mathsf{SUF\text{-}CMA}} = n_q$, for which n_q limits the number of signature queries the adversary may make before submitting their forgery.

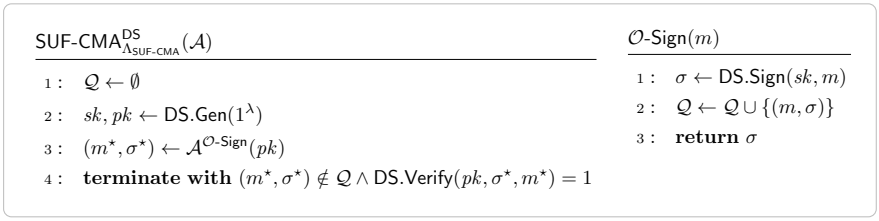


Figure 2.17. The SUF-CMA security experiment for signature schemes.

Matrix uses the Ed25519 = (Gen, Sign, Verify) digital signature scheme, utilising the Edwards 25519 curve [Ber06]. WhatsApp uses the Curve25519 variant of XEdDSA [Per16], which we represent through XEd = (Gen, Sign, Verify) where XEd.Gen is equivalent to XDH.Gen.

2.3.7 Key Exchange

Key exchange, or key agreement, allow two or more parties to determine a shared secret key over an insecure network [BR94, Sho99, BCK98, CK01]. Typically, in authenticated variations, the two parties generate cryptographic identities and are expected to ensure that the resulting key is secret to those parties with the expected cryptographic identities [BR94]. Such constructions generally require cryptographic identities to remain secure in settings where the adversary is active in their attack.

The protocols we study use variations of Diffie-Hellman key exchange. In particular, both Matrix and WhatsApp utilise a Curve25519-based Diffie-Hellman (DH) key exchange introduced by Bernstein [Ber06]. We refer to this scheme as XDH and denote it with two algorithms: XDH.Gen, which generates a Curve25519 key pair, and XDH, which performs DH key exchange using the executing party's private key and the communicating partner's public key. Both Matrix and WhatsApp use XDH in their construction of secure pairwise channels.

We briefly review the Diffie-Hellman problems [DH76] (following Section 3.2 of [OP01]).

Definition 2.36 (The Computational Diffie-Hellman Problem)**.** Consider a group $\mathcal{G}$ of prime order q. Given a triple of elements (g, g^a, g^b), where g is a generator for $\mathcal{G}$ (such that $\mathcal{G} = \{ g^0, g^1, \ldots, g^{q-1} \}$) and where a and b are sampled uniformly at random from $\{ 0, 1, \ldots, q-1 \}$ (i.e. $a \leftarrow\$ \mathbb{Z}_q$; $b \leftarrow\$ \mathbb{Z}_q$), find the element $C = g^{ab}$.

Definition 2.37 (The Decisional Diffie-Hellman Problem)**.** Consider a group $\mathcal{G}$ of prime order q. Given a quadruple of elements (g, g^a, g^b, g^c), where g is a generator for $\mathcal{G}$, $a \leftarrow\$ \mathbb{Z}_q$, $b \leftarrow\$ \mathbb{Z}_q$, and c is either chosen uniformly at random from $\mathbb{Z}_q$ or calculated as $a \cdot b \bmod q$, determine whether $c = a \cdot b \bmod q$ or not.

Gap DH Security of Curve25519. The gap problems aim to capture the *gap* in difficulty between a computational problem and its decision variant [OP01]. Intuitively, they ask: given an oracle that provides solutions to the decision problem, can you solve the computational problem?

We will be most interested in the Gap Diffie-Hellman (Gap DH) problem, which captures the gap between the Decisional Diffie-Hellman and Computational Diffie-Hellman problems [DH76]. We follow the definition from Section 3.2 of [OP01].

Definition 2.38 (The Gap Diffie-Hellman Problem)**.** Consider a group $\mathcal{G}$ of prime order q. Given a triple (g, g^a, g^b), where g is a generator for $\mathcal{G}$, $a \leftarrow\!\$\ \mathbb{Z}_q$, $b \leftarrow\!\$\ \mathbb{Z}_q$, and access to a Decisional Diffie-Hellman oracle $\mathcal{O}\text{-}\mathsf{DDH}(g, g^x, g^y, g^z) := (z \stackrel{?}{=} x \cdot y \mod q)$ that may be queried up to n_q times, find the element $C = g^{ab}$.

Our security proofs for both Matrix and WhatsApp[3] rely on the Gap DH problem being computationally intractable in the Curve25519 group. In short, this is required by our modelling of the key derivation function as a random oracle when it is applied to the outputs of a Diffie-Hellman key exchange (within the pairwise key exchange).

2.3.8 Hash Chains & Structures

A *hash chain* is a structure whereby a hash function is applied to a piece of initial randomness iteratively to generate a sequence of pseudo-random values. We follow the definition of [Pag09].

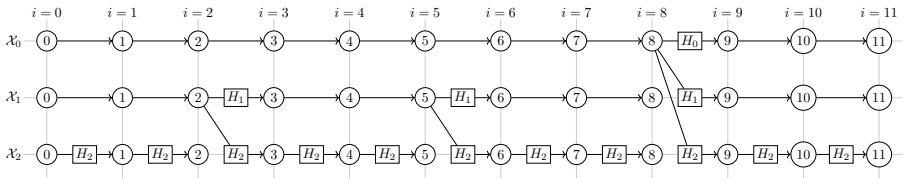
Definition 2.39. A *hash chain* $\mathcal{X}$ over the function $H : \{0,1\}^* \to \{0,1\}^b$ is a sequence of random variables $(X_0, X_1, \ldots)$ with

$$X_0 \leftarrow\!\$\ \{0,1\}^b$$
$$X_{i+1} \leftarrow H(X_i).$$

Intuitively, a hierarchical hash chain represents a hierarchy of hash chains where each layer is periodically reseeded from the layer above it. The Fortuna PRNG [DSSW14] is an example of a widely-used construction that could be rephrased as a hierarchical hash chain. A hierarchical hash chain is specified by its depth d (the number of layers or dimensions) and width m (the interval at which layers reseed from those above them). We adapt the definition of [Pag09] for our setting as follows.

Definition 2.40. An $(m, d)-$hierarchical hash chain is a sequence of random variables $\mathcal{X} = (X_0, X_1, \ldots, X_{m^d-1})$ defined by d interdependent hash chains $\mathcal{X}_k = (X_{0,k}, X_{1,k}, \ldots, X_{m,k})$ for $k \in \{0, 1, \ldots, d-1\}$. The i-th element of the

[3] In the case of WhatsApp, this assumption is implicit. We rely on the proof of MS-IND security for Signal's pairwise channels of [CCD+20] which, in turn, relies on Curve25519 providing Gap Diffie-Hellman security.



- Circle nodes represent elements in the given sequence and contain the element's index within that sequence.
- Square nodes denote processing through a hash function.

Figure 2.18. A visualisation showing the first 12 elements of a $(3, 3)-$hierarchical hash chain, represented as its three constituent sequences $\mathcal{X}_0$, $\mathcal{X}_1$ and $\mathcal{X}_2$ from which each element of $\mathcal{X}$ is built (e.g. $X_6 = X_{6,0} \parallel X_{6,1} \parallel X_{6,2}$).

combined chain $\mathcal{X}$ is a tuple consisting of the i-th element from each of its subsequences:

$$X_i = (X_{i,0}, X_{i,1}, \ldots X_{i,d-1})$$

The structure is initialised with given values $X_0 = (X_{0,0}, X_{0,1}, \ldots X_{0,d-1})$. Then, each inner hash chain $\mathcal{X}_k$ progresses as follows:

$$X_{i+1,k} \leftarrow \begin{cases} H_k(X_{i,0}) & \text{if } i \bmod m^{(d-1)-0} = 0 \\ H_k(X_{i,1}) & \text{if } i \bmod m^{(d-1)-1} = 0 \\ \quad \vdots & \\ H_k(X_{i,k}) & \text{if } i \bmod m^{(d-1)-k} = 0 \\ X_{i,k} & \text{else} \end{cases}$$

where each $H_k(x) : \{0,1\}^* \to \{0,1\}^b$ is a distinct function.

See Figure 2.18 for a visualisation of a $(3, 3)-$hierarchical hash chain.

Fast-Forwarding with Hierarchical Hash-Chains

Both Matrix and WhatsApp use a hierarchical hash-chain to construct a symmetric ratchet that allows *fast-forwarding* to future messages in logarithmic time, allowing a client to quickly catch up with a conversation. We make use of an existing formalisation for fast-forwarding ratchets introduced in [DJK22]. We briefly summarise their definition from Section 3.2 of the paper, and adapt it to our setting. Notably, the constructions we analyse are symmetric and do not require use of a bulletin board.

Definition 2.41. A *fast-forwardable pseudorandom generator* consists of the deterministic algorithms $\mathsf{FFPRG} = (\mathsf{Init}, \mathsf{Update}, \mathsf{Leap})$, where:

1) The $(st_1, R_1) \leftarrow \mathsf{Init}(k)$ algorithm takes initial key material $k \in \{0,1\}^{\mathcal{K}}$ and produces an initial state st_1 and output R_1.

2) The $(st_{i+1}, R_{i+1}) \leftarrow \mathsf{Update}(st_i)$ algorithm takes a state st_i as input, and produces an updated state and the next output.

3) The $(st_j, R_j) \leftarrow \mathsf{Leap}(st_i, j)$ algorithm takes a state st_i and the target epoch $j > i$, to leap to the j-th state and output the updated state and the requested output R_j.

Intuitively, a fast-forwarding ratchet scheme is *correct* if it the outputs of Update and Leap are guaranteed to match for the same epoch. Precisely, [DJK22] defines correctness as follows:

Definition 2.42 (Correctness of an FFPRG)**.** A *fast-forwardable pseudorandom generator*, FFPRG, is correct if every PPT adversary $\mathcal{A}$ has a negligible search-advantage in winning the correctness game depicted in Figure 2.19.

$\mathsf{Correct}^{\mathsf{FFPRG}}_{\Lambda}(\mathcal{A})$

1 : $win \leftarrow$ **false**
2 : $n \leftarrow 1$
3 : $k \leftarrow\$ \{0,1\}^{\mathcal{K}}$
4 : $(st_1, R_1) \leftarrow \mathsf{FFPRG.Init}(k)$
5 : $\mathcal{A}^{\mathsf{Update},\mathsf{Leap}}(st_1, R_1)$
6 : **return** win

Update()

1 : $n \leftarrow n + 1$
2 : (st_n, R_n)
3 : $\quad \leftarrow \mathsf{FFPRG.Update}(st_{n-1})$
4 : **return** (st_n, R_n)

Leap(i, j)

1 : **if** $i \geq j \ \vee j > n$:
2 : $\quad$ **return** $\perp$
3 : $(st'_j, R'_j) \leftarrow$
4 : $\quad \mathsf{FFPRG.Leap}(st_i, j,)$
5 : **if** $st'_j \neq st_j \ \vee R'_j \neq R_j$:
6 : $\quad win \leftarrow$ **true**

Figure 2.19. The correctness game for FFPRGs.

For an fast-forwardable pseudorandom generator (FF-PRG) scheme to be non-trivial, it should be able to fast-forward i iterations in time and communication complexity sub-linear in i.

Intuitively, we expect that a secure FF-PRG outputs pseudorandom keys that remain pseudorandom even to those with access to future private states. Security is captured through a key indistinguishability game that additionally allows the ratchet's internal state to be compromised, while requiring that the indistinguishability of all keys generated before that point is maintained.

Definition 2.43 (Key Indistinguishability of an FFPRG)**.** A *fast-forwardable pseudorandom generator*, FFPRG, is secure if every PPT adversary $\mathcal{A}$ has a negligible decision-advantage in winning the key indistinguishability game depicted in Figure 2.20 parameterised by the number queries n_q.

$\mathsf{KIND}^{\mathsf{FFPRG}}_{\Lambda}(\mathcal{A})$

1 : $b \leftarrow\$ \{0,1\}$; $safe? \leftarrow$ **true**
2 : $n \leftarrow 1$; $k \leftarrow\$ \{0,1\}^{\mathcal{K}}$
3 : $(st, R) \leftarrow \mathsf{FFPRG.Init}(k)$
4 : **if** $b = 1$: $R \leftarrow\$ \{0,1\}^{\mathcal{K}}$
5 : $b' \leftarrow \mathcal{A}^{\mathsf{Update},\mathsf{Corrupt}}(R)$
6 : **return** $b = b'$

Update()

1 : $n \leftarrow n + 1$
2 : $(st, R) \leftarrow \mathsf{FFPRG.Update}(st)$
3 : **if** $b = 1 \ \wedge safe?$: $R \leftarrow\$ \{0,1\}^{\mathcal{K}}$
4 : **return** R

Corrupt()

1 : $safe? \leftarrow$ **false**
2 : **return** st

Figure 2.20. The key-indistinguishability security game for FFPRGs.

2.4 Secure Messaging

Before we embark upon our first case study, we briefly survey the landscape of secure messaging protocols. These protocols provide a means for two or more parties to securely exchange messages over an untrusted network, in a broadly asynchronous manner. They generally aim to guarantee the confidentiality, integrity and authenticity of messages.

- **Authenticity:** Was the message sent by the user it claims to have been sent by?
- **Integrity:** Can we guarantee that the message received is exactly the same, bit-for-bit, as the message that was sent?
- **Confidentiality:** Can we guarantee that the contents of a message remain private only to the sender and the intended recipients?

These first three properties are typically achieved through a combination of symmetric encryption (for confidentiality), a MAC (for integrity and implicit authentication) and/or digital signatures (for integrity and explicit authentication).

Depending on the application, secure messaging protocols may also target additional security guarantees.

- **Forward Secrecy:** In the face of compromise of our current state, can we guarantee the secrecy of previous ciphertexts? Forward secrecy ensures that past messages remain confidential, even if an adversary gains access to current key material [UDB+15]. This property is typically achieved through key evolution. The Silent Circle messaging application achieved key evolution through a hash chain, for example [VGP12, VL16, Bel14]; a practice that has continued in many protocols since.
- **Post-Compromise Security:** Can we recover security after the current session state has been compromised? Post-compromise security (PCS) allows for the recovery of secure communication at some point after compromise, providing certain conditions are met. The notion was formalised in [CCG16], and is closely related to the notion of *perfect future secrecy* [Gün90] popularised in the secure messaging context by the OTR protocol [BGB04]. It is typically achieved through a *continuous key exchange* that is executed by the communicating parties as they exchange messages. As discussed in [CCG16], the extent to which this approach achieves post-compromise security depends on the key hierarchy used and the extent of compromise. An alternative conception is explored in [BBL+23], which considers a variety of state-compromises as well as forward secrecy.
- **Deniability:** Deniability aims to allow users to deny having sent a particular message, *within a cryptographic context*. That is, it aims to prevent the authentication provided by the secure messaging scheme from later being used to prove that a particular message was sent by a participant [BGB04, CDNO97]. This can be achieved through a variety

of mechanisms. In its first iteration, OTR made use of per-message MAC keys that were later revealed (combined with an otherwise malleable encryption scheme). Alternatively, TextSecure with 3DH [Mar13b] (and Signal with X3DH [Mar16b]) take the approach of providing deniability for the initial key exchange only. This allows these protocols to require less synchrony, enabling features such as immediate out-of-order decryption (see below), with the trade-off of requiring whole conversation transcripts to be denied at once. See [UG15] for a detailed discussion deniability and its various qualities.

- **Metadata Hiding:** Even if the content of the messages is encrypted, metadata can still reveal sensitive information about the communication patterns and relationships between users. Metadata hiding aims to protect the metadata associated with messages, such as participant identities, timestamps, and message sizes.

- **Participant Consistency:** Can the protocol guarantee that all participants have the same view of the group membership? This notion is referred to as *group integrity* in [AST98], and, more recently, as *participant consistency* in [UDB+15].

- **Membership Control:** Do the participants have control over who is in the conversation? This property is also referred to as *additive closeness* [RMS18]. In the direct messaging setting, this is implicit: the "participants of a conversation" are tautologically defined by the participants conversing. In contrast, a group may exist as an independent entity whose membership changes over time. In this latter case, it is natural to consider control over group membership to be a requirement for authentication. The `zkgroups` system, designed for use in Signal, provides such functionality with the addition of hiding the knowledge of the group participants from the server [CPZ20]. Messaging Layer Security (MLS) can provide control over group membership, if configured appropriately [BBR+23]. Indeed, this component of the MLS standard has since been analysed in isolation using the F* language (see [WPBB23]). Additionally, recent work has extended this design to to enable cryptographically controlled membership management by a subset of administrative group members [BCV23].

- **Insider Security:** To what extent can a protocol maintain the aforementioned security properties, while one or more conversation participants are malicious [AJM22, Ung21]?

In addition to security guarantees, secure messaging protocols may fulfil a variety of different *functional requirements*.[4]

- **(A)synchronicity:** When does the protocol allow for one or more parties to be offline? To what extent does the protocol maintain its functionality and security guarantees? The Signal family of protocols allow for asynchronous session setup by using Triple Diffie-Hellman for the initial key

[4] Of course, the distinction between a functional requirement and a security guarantee is not always clear.

exchange, with the server distributing pre-generated single-use public keys on behalf of clients [Mar16b]. Once the session has started, the Double Ratchet [Mar16a] allows for continuous key exchange that is *opportunistic*, with clients sending new contributions alongside messages and performing the new key exchange whenever there are sufficient contributions from both sides.

- **Transcript Consistency:** Can the protocol guarantee that all participants will (eventually) compute the same transcript? This property can be decomposed into (a) speaker transcript consistency, which asks if all participants agree on the transcript of a single participant, and (b) global transcript consistency, which asks if all participants agree on the complete conversation transcript across all participants [UDB+15].

- **Immediate Decryption:** If a client receives a message out-of-order, is it able to decrypt and process the message immediately while maintaining the ability to decrypt any skipped messages [PP22, ACD19]? This is naturally implemented in protocols that derive per-message key material by caching the unused message keys, while continuing to ratchet the key schedule forward [PP22] (see Section 2.6 of [Mar16a], for example).

- **Fast-Forwarding:** If a client is offline for a long period of time, they may wish to display the most recent messages to the user without decrypting every single message that has been sent while they were offline. Fast-forwarding allows a client to quickly skip ahead in a time sub-linear in the number of messages they are skipping [DJK22]. As we will see, both Matrix and WhatsApp support fast-forwarding in some contexts, which they implement by using a hierarchical hash chain[5] as a symmetric ratchet.

- **Group Messaging:** Does the protocol support groups of people communicating within a single conversation? To what extent does the protocol maintain its functionality and security guarantees as the group scales?

- **Multiple Devices:** Users may wish to interact with the protocol through multiple devices. This feature is widely implemented in practice; for a review of the varying approaches see [DGGL21].

- **History Sharing:** It may sometimes be desirable to provide new parties with a copy of the conversation history as they join. This is particularly useful in a multi-device context, where a single user expects to have access to a consistent conversation history across all of their devices.

- **Desynchronisation & Session Handling:** How does the protocol handle temporary desynchronisation between participants? For example, it may be possible that two participants initiate a conversation at the same time, resulting in two separate sessions. Signal utilises the Sesame session management protocol [Mar17] to handle such situations. Allowing multiple parallel sessions does come with security trade-offs, as highlighting

[5] Section 3 of [Pag09] contains a number of useful definitions of data structures built from hash functions, including hierarchical hash chains.

by recent work on cloning attacks [CFKN20] and a formal analysis of the protocol [CJN23]. There, additionally, exists a recent line of work developing solutions to this problem for MLS [AMT23, AAN+24].

While this is, surely, an incomplete discussion of secure messaging, we direct interested readers towards

1) *"SoK: Secure Messaging"* [UDB+15] for a systemisation of the secure messaging literature as of 2015 (including discussion of a wide variety of security properties not considered here),

2) *"SoK: Multi-Device Secure Instant Messaging"* [DGGL21] for multi-device secure messaging as of 2021 (surveying the techniques used to implement multi-device secure messaging), and

3) *"SoK: Game-Based Security Models for Group Key Exchange"* [PRSS21a, PRSS21b] for group key exchange as of 2021 (discussing many of the intricacies of formalising key exchange for secure group messaging).

Device Management

As we will soon see, much of the cryptographic machinery within the secure messaging applications we study is dedicated to the management of a user's devices. Keybase represents a notable piece of related work in this area.

Keybase utilises a chain of signed statements, known as a *sigchain*, that describe changes to a user's account over time [Key22]. It is initialised with the public key of their first device, allow them to add and remove devices over time (among other operations). Replaying the chain allows a client to determine the current set of valid devices that represent a user. A causal relationship is created by including a hash of the previous end of the chain in each signed statement. To prevent the malicious rollback of a user's devices, Keybase includes a reference to the current generation of each user's sigchain in their key transparency tree. This approach has since been adopted by Zoom [BBC+23]. The ELEKTRA system applies recent developments in privacy-preserving transparency trees to this multi-device context, in addition to formalising the notion of a sigchain [LCG+23].

Similar problems have also been studied in adjacent areas, such as certificate revocation [Koc98], proactive security in signature schemes [BPR22, TT01], revocation within anonymous credential schemes [CL01, CL02] and recent proposals for revocation in FIDO2 [HLW23].

CHAPTER 3

Case Study: Matrix

Our first case study investigates the secure group messaging features of Matrix. We start with a high-level overview of the protocol and its architecture before isolating the parts relevant to the topic at hand, multi-device group messaging. We provide a detailed description combining the specification and documentation with that of the reference implementation. After detailing a variety of cryptographic vulnerabilities discovered during our investigation, we finish with a discussion of their impact, the disclosure process and the overall design of Matrix.

3.1 Introduction

Matrix [Mata] is an open standard for interoperable, federated and real-time communication over the Internet. It consists of a number of specifications which, together, define a federated secure group messaging protocol that enables clients, with accounts on different Matrix servers, to exchange messages. Since this setting inherently involves interacting through untrusted third party servers, the specification enables end-to-end encryption by default [L8].[1] While the underlying Matrix protocol can and is[2] used for a wide variety of applications where groups of devices need to exchange messages in real-time, this same protocol is used in a relatively direct manner to provide secure group messaging. Indeed, this seems to be its primary purpose; the most popular implementations of a Matrix server and client are *Synapse* [L5] and *Element* [L17] respectively.

Usage. While Matrix' federated nature makes it difficult to assess how widely it is used, several notable organisations and institutions have either adopted it or announced plans to do so.

For instance, the French government announced plans to create their own instant messaging app – *Tchap* – based on Matrix which was released [L2, L16] in 2019; the German ministry of defence launched *BwMessenger* [L15] – for use in internal, official (and classified) communication – based on Matrix in 2020 with a view to move [L26] over other parts of the German government; the German healthcare system announced [L9] its plans to adopt Matrix in 2021.

Matrix is a popular choice within the FOSS ecosystem, with Mozilla [L23], KDE [L1] and the FOSDEM 2022 [L18] conference all using it. A number of existing applications have developed Matrix integrations including the email client Mozilla Thunderbird [L6] and the enterprise messaging platform Rocket.Chat [L25]. Overall, Element's website reports that Matrix has over 125 million users [L17].

Our focus. We focus on the assortment of custom cryptographic protocols with which Matrix realises multi-device secure group messaging in the setting of end-to-end encryption. As such, we do not consider Matrix' use of Transport Layer Security (TLS) for transport security between the client and server, as well as between servers for the purpose of federation. Similarly, we consider the network of federated servers to be a single untrusted server for the purposes of modelling. We will further restrict our scope as we turn to our analysis.

3.1.1 Related Work

Security analysis. While the security guarantees of Matrix and these popular implementations have received attention from the information security practitioners community – e.g. CVE-2022-31052 [L36], CVE-2022-23597 [L35],

[1] In addition to standard security considerations such as breaches [L24] or lack of trust in a single-server setting.

[2] Take the Third Room [L3] project, for example, which utilises the Matrix protocol as a decentralised communication layer from which three-dimensional virtual worlds can be built.

CVE-2021-41281 [L34], CVE-2021-39163 [L31], CVE-2021-39164 [L32], CVE-2021-32659 [L29], CVE-2021-32622 [L28] and CVE-2021-29471 [L27] – its bespoke cryptographic protocol had, prior to this work, not received an in-depth treatment from the cryptographic (academic or practitioner) community.

An audit of the Olm and Megolm protocols (along with their implementations) was performed by NCC Group in 2016 [MB16]; this audit found a number of security issues that have since been fixed or recorded as limitations [Mat22a, Mat22b]. Since then, several further cryptographic vulnerabilities have been reported, e.g. in CVE-2021-34813 [L30], CVE-2021-40824 [L33], while Wong reported a vulnerability in out-of-band verification [Won21, Chapter 11].

Additionally, an audit [L10] of the Vodozemac implementation was conducted by Least Authority in 2022 [AAD+22]. In contrast, our analysis utilises the `libolm` library that, at the time this work was completed, was the reference implementation provided by the Matrix Foundation, though it has since been deprecated [L12] in favour of Vodozemac.

Shared heritage. As we will soon see, the Olm protocol is built upon variants of Triple Diffie-Hellman key exchange [Mar13b, Mar16b] and the Double Ratchet algorithm [Mar16a, Mar13a], while Megolm shares its architecture with the *Sender Keys* variant of the Signal protocol [Mar14]. For this reason, existing analysis of these protocols will be relevant; various aspects of Signal have received multiple formal analyses over the last seven years [FMB+16, CCD+20, ACD19], while Sender Keys has seen informal analysis [RMS18]. More recently, WhatsApp's implementation of Sender Keys has seen a provable security analysis in [BCG23].

State synchronisation. Matrix deploys a custom state synchronisation protocol that allows federating servers to synchronise the event graph between one another in a distributed fashion [Matd]. This protocol has seen substantial study in the literature on distributed systems, see [JGH19, JBGH20, JBHH21].

3.1.2 Contributions

This chapter includes two contributions.

Contribution 3.1. Section 3.2 gives the first formal description of multi-device secure group messaging in Matrix, covering group messaging itself, device management, state sharing and their composition. The description synthesises information from the various specifications and documentation available, and has been verified against the flagship implementation.

Contribution 3.2. Section 3.3 details five practically-exploitable (and one theoretical) cryptographic vulnerabilities spread across[3] the Matrix standard, its reference implementations and the flagship client Element.

[3] The vulnerabilities we describe make varying use of implementation-specific behaviour. In each case, however, we pinpoint issues with the specification that led to such issues.

These vulnerabilities resulted in ten advisories being added to the Common Vulnerabilities and Exposures (CVE) database: CVE-2022-39246 [L37], CVE-2022-39248 [L38], CVE-2022-39249 [L39], CVE-2022-39250 [L40], CVE-2022-39251 [L41], CVE-2022-39252 [L42], CVE-2022-39254 [L43], CVE-2022-39255 [L44], CVE-2022-39257 [L45], and CVE-2022-39264 [L46].

3.1.3 Scope & Limitations

Coverage. The Matrix standard, and ecosystem of extensions in the form *Matrix Spec Changes*, is vast and constantly evolving [L20]. We focus our case study on the components that are most relevant to multi-device secure group messaging. In particular, our description covers *Olm*, *Megolm*, the *Cross-Signing Framework* and *Key Request* protocol in detail, along with their composition. We, additionally and where relevant, describe parts of the *Verification Framework* for out-of-band verification, Megolm session backups and the *Secure Secret Sharing and Storage* component for storing and sharing user-level secrets.

We synthesise information from the Matrix Client-Server API [Mata], the specifications of Olm [Mat22b, Mate] and Megolm [Mat22a], the `libolm` [Mat17] reference implementations of Olm and Megolm, the implementation guide [Mat23], source code of the `matrix-react-sdk` [Matc] and `matrix-js-sdk` [Matb] reference libraries provided by the Matrix foundation and that of the flagship client, Element [New22].[4]

The research for this case study was performed between June 2021 and December 2022 and was, to the best of our knowledge, an accurate description of the standard during this time.

Implementations. Given the federated and open source nature of the Matrix ecosystem, a variety of clients and alternative implementations of the Matrix standard are in use. It follows that there will be variance across implementations. As such, we have chosen to focus on libraries maintained by the Matrix.org foundation and, when needing to lift our analysis to the user interface, the Element client (since it is the flagship client and makes full use of the aforementioned libraries).

In particular, this means that the vulnerabilities and attacks presented in Section 3.3 are *specific to Element*. While the libraries `libolm`, `matrix-js-sdk` and `matrix-react-sdk` served, at the time of writing, as the basis of a number of clients in the Matrix ecosystem, we did not investigate to what extent these other clients are vulnerable to our attacks or variants thereof. For the avoidance of doubt, any implementation specific behaviour reported throughout this section refers to that of `libolm`, `matrix-js-sdk`, `matrix-react-sdk` and/or Element, even when we write “Matrix”.

[4] Our analysis is based on, and our proof-of-concept attacks tested against, Element Web at commit `#479d4bf` [New22] with `matrix-react-sdk` at commit `#59b9d1e` [Matc] and `matrix-js-sdk` at commit `#4721aa1` [Matb].

3.2 Multi-Device Group Messaging in Matrix

In this section, we study the assortment of custom cryptographic protocols with which Matrix realises multi-device group messaging. To do so, we collate information across their various specifications and implementations to construct a description of the multi-device secure group messaging functionality in Matrix. Our formal description, in particular, covers *Olm*, *Megolm*, the *Cross-Signing Framework* and *Key Request* protocols as well as their composition.

We begin with a high-level description of secure messaging in Matrix, introducing the key concepts and terminology as well as its functionality and the components that provide it.

Entities. 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, with each user having an account on a particular homeserver, and each user having a number of associated devices. A device represents a particular login of a user on a client (e.g. a phone and a laptop).

We represent users with identifiers $uid \in \mathcal{U}$ of the form `@username:homeserver.tld`, with the `username` portion chosen by the user (and allocated by the homeserver). Devices have their own identifiers, which we represent as $did \in \mathcal{D}$, which are chosen by the client (and allocated by the homeserver). The specification requires that device identifiers are unique within a user. As discussed, for the purpose of end-to-end encryption, we consider the network of federated homeservers as a single untrusted actor.

Messaging. 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. Conceptually, a room is a logical entity in and of itself. Each room has a number of events associated with it: state events and message events. State events modify the room configuration and/or metadata. A state event could be the room creation event, or membership change, for example. An example of a message event could be an instant message or reaction. While Matrix' decentralised architecture means that these events form a graph, homeservers are expected to present a linear sequence of events for their clients. Room membership is managed as a list of users, for which users are added through invites (requiring acceptance), and implement an access control policy. State events are generally left unprotected, while message events secure their contents using the Megolm protocol. Thus, without authentication, group membership and access control is fully mediated by the homeserver.

Cryptographically-speaking, room membership is implemented as a collection of devices, with each client maintaining a list of devices that represent each user they communicate with. They then use the *Megolm* protocol [Mat22a] to protect the contents of conversations between these devices. Megolm provides a secure unidirectional channel between one sending device and many receiving devices. After channel establishment, senders (and receivers) symmetrically

ratchet (via the bespoke *Megolm ratchet* [Mat22a]) the shared secret state forward after each message sent (resp. received) by the unidirectional channel, aiming to achieve forward secrecy. We note, however, that the specification allows implementations to keep old copies of the ratchet on the receiving side [Mat22a, MB16] – something which Element does – and that this invalidates forward secrecy guarantees. In addition, senders can periodically generate a new (and independent) Megolm secret state, and send it to the receiving devices in the room, thus aiming to achieve some form of post-compromise security. Such rotations are triggered when a device leaves the room, for example. These unidirectional channels are composed together, with each room member maintaining and distributing their own, to form a single conversation.

Pairwise channels. Devices may also communicate with one another outside of a room for the purposes of signalling and non-user visible interactions (known as *to-device* messages). One such example, relevant to our setting, is channel establishment for the aforementioned Megolm channels [Mat22a]. Device-to-device communications are secured using pairwise *Olm* channels [Mat22b]. The Olm protocol is an implementation of a modified Triple Diffie-Hellman (3DH) key exchange [Mar13b][5] and the Double Ratchet algorithm [Mar16a, Mata]. Matrix clients, additionally, implement a session handling layer on top of Olm, allowing multiple active Olm sessions between any particular pair of devices.

User and device management. Users are identified, cryptographically, by an Ed25519 [BDL+12] key pair which we refer to as their master cross-signing key. By default, user cross-signing keys are generated and stored on the device from which they create their account. Devices are, similarly, identified by an Ed25519 key pair.

When logging in to a new device, the device cross-signing keys are generated and users are prompted to (optionally) verify it. If verification is successful, user and device identities are linked together using Ed25519 signatures. The *verification framework* (see in Section 3.2.4) provides the (a) Short Authenticated String (SAS), and (b) QR code protocols that enable users to verify their own devices, or those of other users, using an out-of-band channel. These protocols allow a user to verify that the devices they are communicating with are controlled by the people they expect them to be.

Initially, Matrix required users to go through this process for every single pair of devices they communicate with (i.e. every one of Alice's devices must verify every one of Bob's devices). *Matrix Spec Change 1756* [Mat21a] introduced *cross-signing*, a feature which enables each pair of users to verify each other only once. Here, once two users have verified one another, they each trust the other to verify their own devices. Assuming that each user self-verifies each of their own devices, this enables pairs of users to verify each other's identities, then rely on the other user to verify each of their own devices. See Section 3.2.1.

[5] 3DH is a precursor to the Extended Triple Diffie-Hellman (X3DH) key exchange protocol [Mar16b].

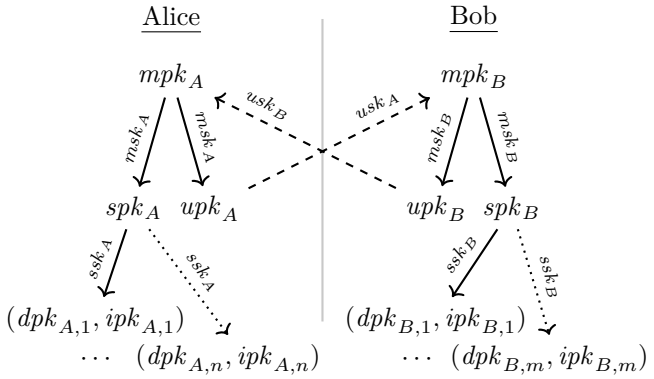
Secret sharing, backup and recovery. By default, a user's secret cross-signing keys are generated and stored from the device with which they create their account. However, it is important for a user to be able to recover their cross-signing identity if they lose access to this device (or simply log out).

The cross-signing module uses the *Secure Secret Storage and Sharing* (SSSS) module to store and backup users' secret keys. SSSS enables users to (a) backup *user-level* secrets to their homeserver (which they encrypt using a recovery passphrase), and (b) share those secrets with their verified devices (over the Olm protocol).

Matrix offers two more methods, aside from SSSS, that enable devices to backup and share Megolm sessions *specifically*, i.e. they allow backup recovery and sharing of session-level secrets. The *Key Request* protocol provides a means for devices to request and share copies of inbound Megolm sessions with each other. These allow a user (or the original message sender) to share message history with new devices by sharing the keys with which they may decrypt ciphertexts. This is achieved through a protocol overlaid on top of Olm channels. *Server-side Megolm backups* enables devices to backup copies of inbound Megolm sessions on the server. This allows a user's new device to gain access to old messages the user has access to, even if no other devices are online (that could otherwise distribute the sessions using the Key Request protocol). The recovery key is generated by one of the user's devices and is a *user-level secret*, itself stored and shared using the SSSS module.

3.2.1 Device Setup and Management

We now detail the *cross-signing protocol* that Matrix uses to create and maintain trust relationships between users and devices, which we describe with a tuple of five algorithms, $\mathsf{CrossSign} = (\mathsf{Init}, \mathsf{SignUser}, \mathsf{SignDevice}, \mathsf{VerifyDevice})$:



- Dashed arrows denote a cross-signing signature (from user verification).
- Dotted arrows denote a self-signing signature (from self-verification).

Figure 3.1 . The long-term key hierarchy for two users, Alice and Bob, and each of their devices [Mata]. Since the cross-signing session exists on the first device where cross-signing is enabled, the first device identity signed by each user does not require out-of-band verification. Diagram based on [Mata, Cross-signing].

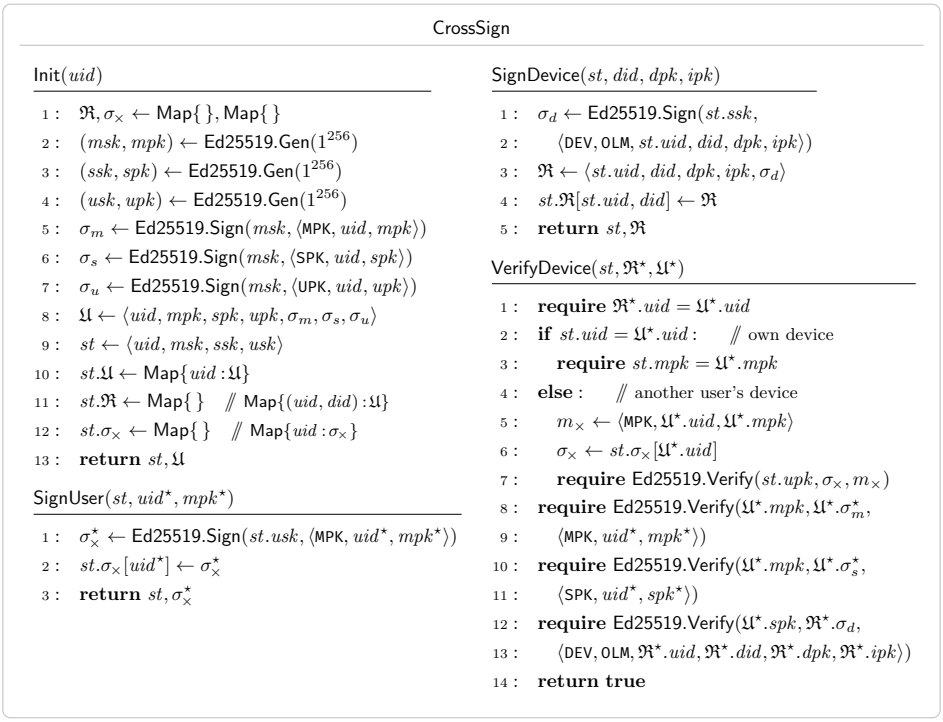


Figure 3.2. Pseudocode describing cross-signing in Matrix.

- $(st_{cs}, \mathfrak{U}) \leftarrow\$$ CrossSign.Init(uid) takes as input a security parameter and user identifier uid. It initialises a cross-signing identity for the user, returning a secret cross-signing state, st_{cs}, for the user and a message, $\mathfrak{U}$, containing their public keys as well as the signatures that link them together.

- $(st_{cs}, \sigma_\times^\star) \leftarrow$ CrossSign.SignUser(st_{cs}, $uid^\star$, $mpk^\star$) takes as input a user's secret cross-signing state st_{cs} and the identity $did^\star$ of a different user that they wish to link with the given master cross-signing key $mpk^\star$. This algorithm is executed after the uid and $uid^\star$ have completed out-of-band verification. It returns an updated cross-signing state st_{cs} for user uid as well as a cross-signing signature $\sigma_\times$ attesting that uid believes that $mpk^\star$ is under the control of $uid^\star$.

- $(st_{cs}, \mathfrak{R}) \leftarrow$ CrossSign.SignDevice(st_{cs}, did, dpk, ipk) takes as input the user's cross-signing state st_{cs}, and the device identifier did, device key dpk and Olm identity key ipk that the user wishes to link with their account. 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}, for the user and a signed message $\mathfrak{R}$ attesting that did is one of the user's devices.

While the creation of new links between users and devices requires the user's cross-signing secrets, the verification of such links is possible from any of their devices. As such, the device verification algorithm requires a minimal copy

of the cross-signing state that only includes the public keys and signatures required to establish a root-of-trust.[6]

- *success*? $\leftarrow$ CrossSign.VerifyDevice($st_{cs}, \mathfrak{R}^\star, \mathfrak{U}^\star$) takes as input the cross-signing state st_{cs} of a user (or one of their devices), as well as the device record $\mathfrak{R}^\star$ and public user identity $\mathfrak{U}^\star$ claimed by the device to be verified. The algorithm proceeds to verify the chain of signatures from their own user's master public key to that of the device to be verified. Since the device record $\mathfrak{R}^\star$ serves to claim an association with a particular user identity, their master public key and a particular device identity, the algorithm returns **true** if these claims passed verification and $\perp$ if not.

We provide explicit pseudocode for these algorithms in Figure 3.2, which we accompany with the following description.

User setup. Each user sets up an account with a particular homeserver, which allocates them a user identifier, uid_A. Next, the user generates their *cross-signing keys* (as described by CrossSign.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 3.1).

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 separate the creation and verification of cross-signing signatures from the out-of-band verification process which triggers their creation (see below for details of the SAS protocol). CrossSign.SignUser describes the process of a user cross-signing another user's identity (as the result of completing out-of-band verification).

In practice, however, each user generates and stores their cross-signing secrets inside one of their devices (in most cases this will be the device they used to create their account). Figure 3.3 details how the initialisation of a new device and cross-signing identity are intertwined in practice, and we proceed to describe initial device setup below.

Device setup. When a new client logs in with their account credentials, the homeserver allocates a device identifier (which we denote by $did_{A,i}$). The client

[6] While the Element client we examined does distribute each user's cross-signing secrets across all of their verified devices, there is, nonetheless, a distinction between a user's secrets and those of their device.

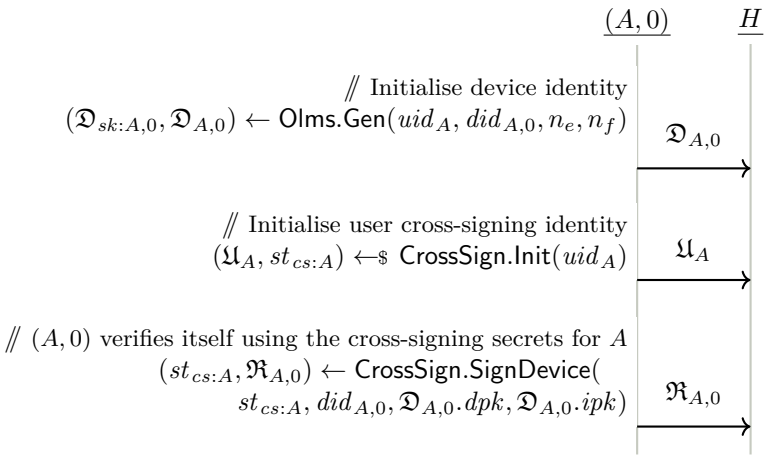


Figure 3.3. A typical example of how a client may setup the first device for a user, in this case Alice, and initialise their cross-signing identity through it.

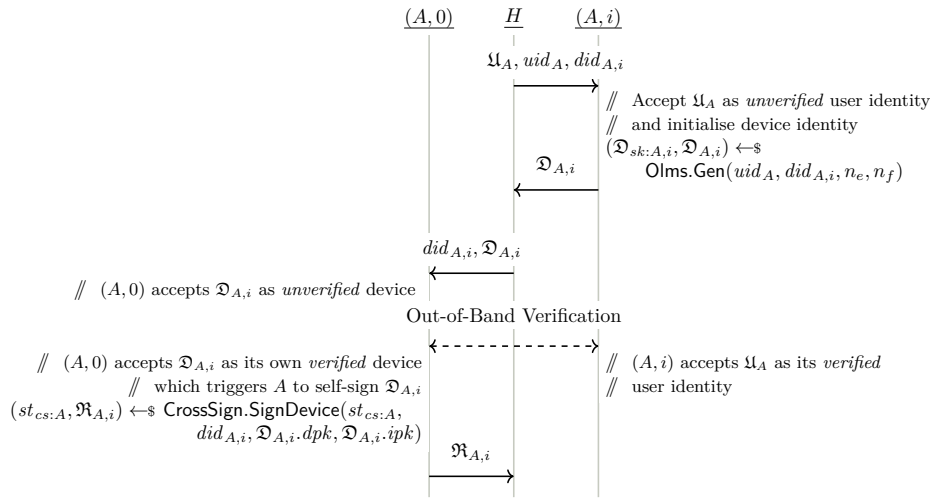


Figure 3.4. A typical example of how one user, Alice, may onboard a new device using the self-verification process.

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) the device fingerprint/signing key ($dsk_{A,i}$, $dpk_{A,i}$), and

2) the Olm key bundle ($isk_{A,i}$, $ipk_{A,i}$, $esk_{A,i}$, $epk_{A,i}$, $fsk_{A,i}$, $fpk_{A,i}$).

The device signing key, $dpk_{A,i}$, and Olm identity key, $ipk_{A,i}$, both serve as cryptographic identifiers for the device (depending on the context). The device key is an Ed25519 key that is, generally, used within the cross-signing protocol (for *signatures*), while the identity key is an X25519 key used by the Olm protocol (as an implicit authenticator for *key exchange*). The rest of the key

pairs in the Olm key bundle serve as medium-term and one-time keys for the Triple Diffie-Hellman key exchange in Olm. In particular, $(esk_{A,i}, epk_{A,i})$ 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 (which we will soon describe in Section 3.2.2). The tuple $(fsk_{A,i}, fpk_{A,i})$ are bundles of one or more *fallback key pairs* [Mat21b]. The public parts of each of these keypairs are distributed to other devices through the homeserver (as a bundle self-signed with $dsk_{A,i}$). We leave the details of how these keys are initialised to our description of the Olm protocol and its session handling layer (see Olms.Gen in Figure 3.8 from Section 3.2.2).

Once a new device has setup their device identity, an existing device (with possession of the user's secret cross-signing keys) may now verify the new device out-of-band. This process is known as *self-verification*. Once verification is complete, the verifying device signs a bundle 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 (which we describe in Section 3.2.4). We describe the result of this process (i.e. how clients behave under the assumption that verification succeeds) with the CrossSign.SignDevice algorithm (in Figure 3.2).

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. For a summary of the various keys and their relationships within the cross-signing protocol, see Figure 3.1 which displays two users, their individual key hierarchies and the relation between them.

Checking device verification. The CrossSign.VerifyDevice$(st_{cs}, \mathfrak{R}^\star, \mathfrak{U}^\star)$ algorithm describes a client checking whether the given device identity, contained in $\mathfrak{R}^\star$, is a verified device controlled by the given user identity, $\mathfrak{U}^\star$. It uses a reduced version of the cross-signing state that should contain a *root-of-trust* in the form of the client's cross-signing key hierarchy. The client proceeds to check each signature in the chain, from the master cross-signing key, to the user-signing key, to the other communicating partner's master cross-signing, their self-signing key and, finally, the device record (refer to the key hierarchy in Figure 3.1, for clarity). In our description, the algorithm outputs a boolean indicating whether the device represented by $\mathfrak{R}^\star$ is a verified device of the user represented by $\mathfrak{U}^\star$.

3.2.2 Pairwise Channels

Matrix uses the bespoke Olm protocol to create a secure channel between two devices. It aims to provide confidentiality and authenticity, as well as varying degrees of forward secrecy (FS), post-compromise security (PCS) and deniability [Mat22b, Mate].[7]

[7] We expect Olm's initial key exchange, as described in this section, to provide deniability. However, since *Matrix* signs the ephemeral keys (see Olms in Figures 3.18 and 3.19), the Olm

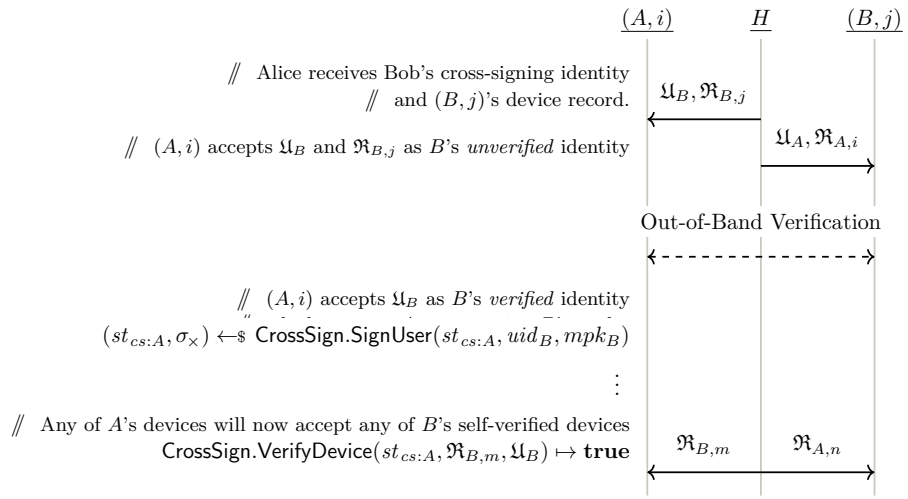


Figure 3.5. A typical example of the verification process between two users, Alice and Bob. To see an example of the out-of-band verification process, we describe the SAS protocol in Section 3.2.4. While we only show the actions performed by Alice's client, Bob's client behaves identically (with roles reversed).

Following the same broad design as the Signal protocol [Mar13b, Mar16b, Mar16a], Olm uses a modified version of triple Diffie-Hellman (3DH) key exchange for session initialisation, which we refer to as Olm Triple-Diffie Hellman (O3DH) [Mat22b, Mate, Mata]. Olm uses O3DH to set up the first epoch of the Double Ratchet algorithm [Mar16a], which takes over the session from this point onwards. 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.

We describe Olm through a tuple of three algorithms, $\mathsf{Olm} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, with the following syntax.

- $(isk, esk, fsk), (ipk, epk, fpk) \leftarrow\!\!\$\ \mathsf{Olm.Gen}(n_e, n_f)$ describes the initial device setup. It generates a fresh key bundle for a new device, consisting of an identity key pair as well as a number of ephemeral keys (specified by n_e) and fallback keys (specified by n_f).

- $(st_{olm}, c) \leftarrow \mathsf{Olm.Enc}(st_{olm}, m, isk_{A,i}, ipk_{B,j}, epk_{B,j,k})$ describes the process of sending a message. It takes as input a private session state, plaintext message, the sender's secret identity key, the intended recipient's public identity and ephemeral keys before outputting an updated state and ciphertext.

- $(st_{olm}, m) \leftarrow \mathsf{Olm.Dec}(st_{olm}, c, isk_{A,i}, esk_{A,i})$ describes the process of receiving a ciphertext. It takes as input a private session state, ciphertext, the recipient's secret identity key, the claimed sender's public identity and ephemeral keys before outputting an updated state and ciphertext.

specification claims that deniability is reduced in exchange for stronger FS guarantees [Mate, "Signing one-time keys"].

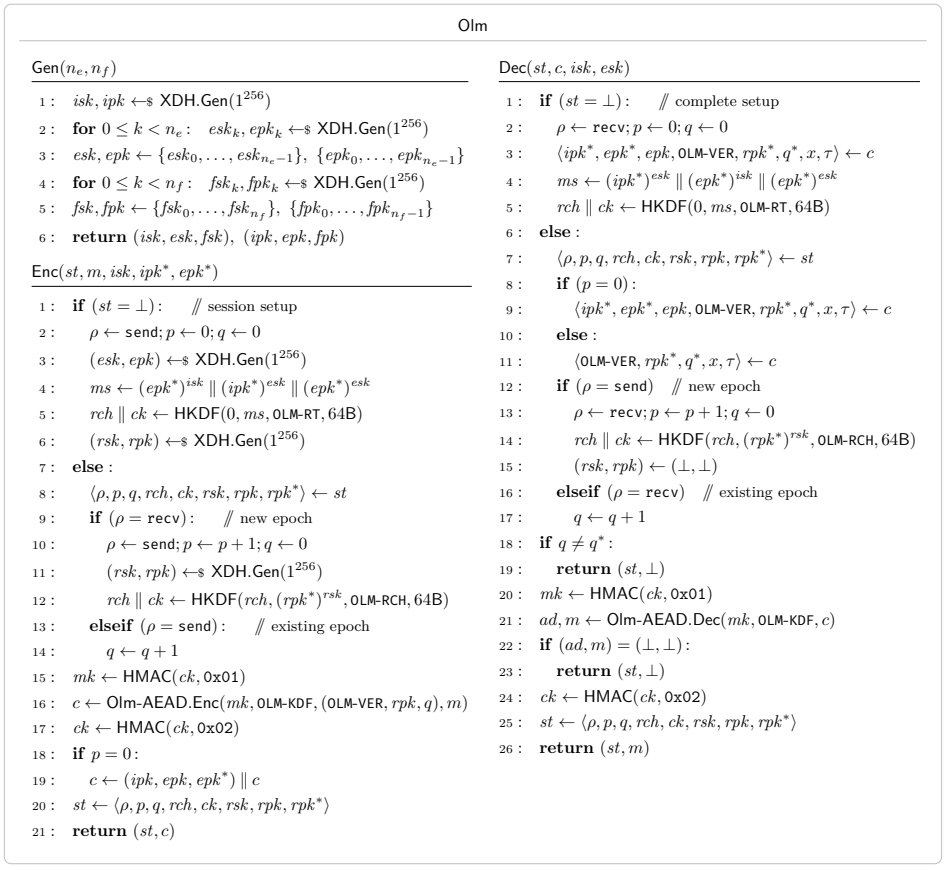


Figure 3.6. Pseudocode describing the Olm protocol.

Figure 3.6 provides pseudocode descriptions for each of these algorithms, while Figure 3.7 demonstrates how they interact with one another.

Key generation. For initial device setup, for the Olm protocol, consists of generating three sets of keys. Consider Alice's ith device, $did_{A,i}$, which generates

1) a long-term identity key pair (isk, ipk),

2) n_e ephemeral key pairs $\{(esk_{A,i,k}, epk_{A,i,k}) : 0 \le k < n_e\}$, and

3) n_f fallback key pairs $\{(fsk_{A,i,k}, fpk_{A,i,k}) : 0 \le k < n_f\}$.

The public parts of each key pair are grouped together, as the *Olm key bundle*, and distributed through the homeserver.

Long-term keys stay constant for the lifetime of the device, while the ephemeral and fallback keys require regular rotation. In particular, each ephemeral key pair should only be used once. When there are no unused ephemeral keys available, a fallback key can be used to complete a key exchange with (possibly) reduced forward secrecy guarantees. They may be used multiple times but should be replaced as soon as possible. While honest homeservers are

expected to aid in enforcing this (by distributing unused ephemeral key pairs to communicating partners), clients must enforce this themselves and must not rely on the homeserver to do this for them.

We describe the process of initial key generation with the Olm.Gen algorithm. While we do not provide a pseudocode description of the ephemeral and fallback key rotation, our description of the session management layer (see below) shows how clients prevent ephemeral key reuse.

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

Consider Alice and Bob, interacting through devices $did_{A,i}$ and $did_{B,j}$ respectively. To initialise a session with Bob's device, Alice's device first fetches the relevant Olm key bundle from the homeserver, containing $ipk_{B,j}$ 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 then executes Olm.Enc, providing the first message they would like to send, their private identity key and the public keys of Bob's device. 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 (A, i)'s ephemeral contribution to the key exchange. (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 (A, i) to compute the shared secret without any interaction from (B, j).

Now, (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. (A, i) then uses $ck_{0,0}$ to derive key material for the first message. Messages sent when (B, j) has not yet responded are *pre-key messages*. These additionally contain, as unauthenticated data wrapping the Olm ciphertext, the public key pairs (B, j) needs to derive the shared secret for the first epoch: (a) $ipk_{A,i}$ (b) $epk_{A,i}$ (c) $epk_{B,j,k}$. Other than this, pre-key messages are encrypted as normal Olm messages. As such, (A, i) generates a new XDH key pair and includes the public part as a contribution towards the root key for the next epoch.

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

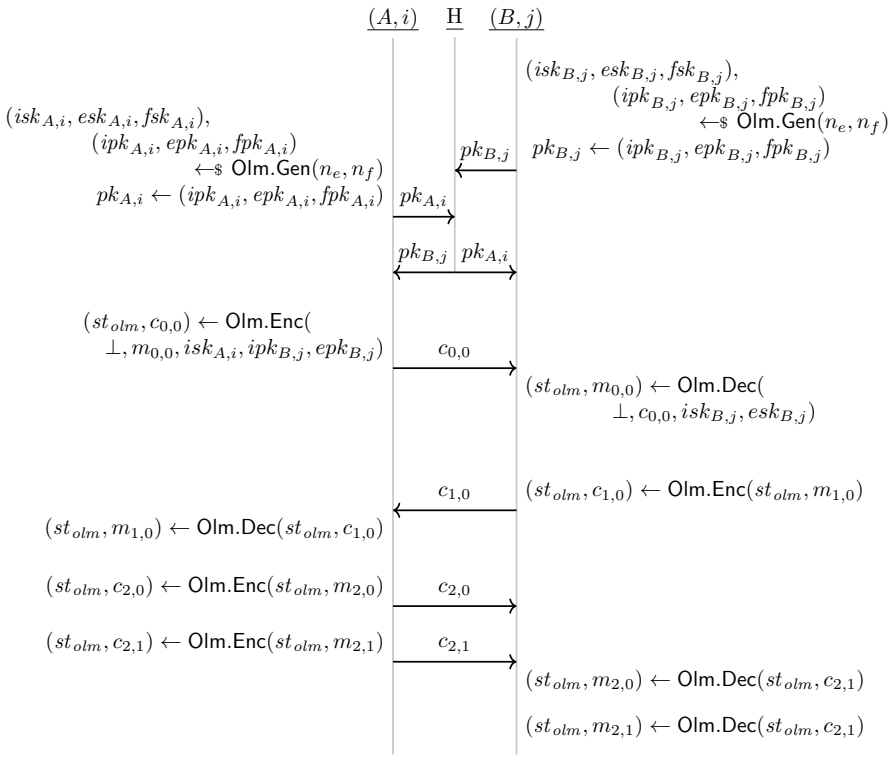


Figure 3.7. An example execution of the Olm protocol between two devices demonstrating initial device setup, session setup and three epochs of messaging. In the broader Matrix system, the Olm key bundle is distributed alongside its cross-signing keys (some of which are signed, see Figure 3.8).

If (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 (A, i)) and a freshly generated ratchet key rsk_p. Otherwise, if (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 (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 $m_{p,q}$, (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-SHA-256. Each message includes the current Olm protocol version, the ratchet key generated by (B, j) during encryption, and the current chain index q as authenticated data as part of the ciphertext $c_{p,q}$. Finally, (B, j)'s copy of the chain key is ratcheted forward, ready to encrypt the next message $m_{p,q+1}$.[8]

[8] 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.

When (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 (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 Olm-AEAD.Dec for decryption. Finally, (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.

Olms

Gen(uid, did, n_e, n_f)

$dsk, dpk \leftarrow\$$ Ed25519.Gen(1^{256})
$(isk, esk, fsk),\ (ipk, epk, fpk) \leftarrow\$$ Olm.Gen(n_e, n_f)
$\sigma_d \leftarrow$ Ed25519.Sign($dsk, \langle did, \texttt{DPK}, dpk, \texttt{IPK}, ipk\rangle$)
$\sigma_e \leftarrow [\,$Ed25519.Sign($dsk, \langle \texttt{EPK}, epk_k\rangle$) : $epk_k \in epk\,]$
$\sigma_f \leftarrow [\,$Ed25519.Sign($dsk, \langle \texttt{FPK}, fpk_k\rangle$) : $fpk_k \in fpk\,]$
$\mathfrak{D}_{sk} \leftarrow \langle dsk, isk, esk, fsk\rangle$
$\mathfrak{D} \leftarrow \langle dpk, ipk, \sigma_d, epk, \sigma_e, fpk, \sigma_f\rangle$
return $(\mathfrak{D}_{sk}, \mathfrak{D})$

Enc($st, dpk^\star, ipk^\star, m$)

let $\mathfrak{D}^\star \in st.\mathfrak{D}$ **st** $\mathfrak{D}^\star.ipk = ipk^\star \wedge \mathfrak{D}^\star.dpk = dpk^\star$
if $ipk^\star \notin st.st_{olm}$:
  $type \leftarrow 0$ // pre-key message
  require Ed25519.Verify($dpk^\star, \mathfrak{D}^\star.\sigma_d$,
    $\langle \mathfrak{D}^\star.did, \texttt{DPK}, dpk^\star, \texttt{IPK}, ipk^\star\rangle$)
  if len($\mathfrak{D}^\star.epk$) ≥ 1 :
    $efpk \leftarrow \mathfrak{D}^\star.epk[0]$
    require Ed25519.Verify($dpk^\star, \mathfrak{D}^\star.\sigma_e[0], \langle \texttt{EPK}, efpk\rangle$)
  else :
    require len($\mathfrak{D}^\star.fpk$) ≥ 1
    $efpk \leftarrow \mathfrak{D}^\star.fpk[0]$
    require Ed25519.Verify($dpk^\star, \mathfrak{D}^\star.\sigma_f[0], \langle \texttt{FPK}, efpk\rangle$)
  $st_{olm} \leftarrow$ Olm.Enc($\perp, m, st.isk, ipk^\star, efpk$)
  if $efpk$ **in** $\mathfrak{D}^\star.epk$: **del** $efpk$ **in** $st.\mathfrak{D}$
  $st.st_{olm}[ipk^\star, epk^\star, \varnothing] \leftarrow st_{olm}$
else :
  $type \leftarrow 1$ // normal message
  $(epk^\star, epk), st_{olm} \leftarrow$ mru($st.st_{olm}[ipk^\star]$)
  $st_{olm}, c \leftarrow$ Olm.Enc(st_{olm}, m_w)
  $st.st_{olm}[ipk^\star][epk^\star, epk] \leftarrow st_{olm}$
$C \leftarrow \langle \texttt{CTXT}, \texttt{OLM}, st.ipk, type, c\rangle$
return (st, C)

Dec($st, C \overset{is}{=} \langle \texttt{CTXT}, \texttt{OLM}, ipk^\star, type, c\rangle$)

if $type = 0$: // pre-key message
  require $c.ipk = ipk^\star$
  if $(ipk^\star, c.epk, c.epk^\star) \in st_{olm}$:
    $st_{olm} \leftarrow st.st_{olm}[ipk^\star, epk, epk^\star]$
    $st_{olm}, m \leftarrow$ Olm.Dec(st_{olm}, c)
  else :
    require $c.q = 0$
    let $efsk \in st.esk \cup st.fsk$
      st PK(esk) $= c.epk^\star$
      require unique
    $st_{olm}, m \leftarrow$ Olm.Dec($\perp, c, st.isk, efsk$)
    del $efsk$ **in** $st.esk$ **if** $efsk \in st.esk$
elseif $type = 1$: // normal message
  for $(epk^\star, epk_k, st_{olm})$ **in** $st.st_{olm}[ipk^\star]$
    $st_{olm}, m \leftarrow$ Olm.Dec(st_{olm}, c)
    if $m \neq \perp$ **break**
require $m \neq \perp$
$st.st_{olm}[ipk^\star, epk, epk^\star] \leftarrow st_{olm}$
return $(st, ipk^\star, m)$

Figure 3.8. The Olms algorithms form a wrapper around the Olm protocol, customised for Matrix' inclusion of signed ephemeral key pairs and session management.

Session management. Matrix implements a session management layer on top of the Olm protocol, allowing any pair of devices to have multiple active Olm

sessions at a given time.[9] For the purposes of this work, we name this layer *Olms* and describe it as a tuple of three algorithms, Olms = (Gen, Enc, Dec). We detail its behaviour with pseudocode in Figure 3.8.

When initialising their device identity, Matrix clients construct a bundle of public keys signed with their device signing key (as previously introduced in Section 3.2.1). In doing so, they create a cryptographic link between the device key, the identity key and each of their ephemeral and fallback keys. We name this bundle of keys the *device key package* and denote it with $\mathfrak{D}$. See Olms.Gen for a description of how these signatures are created.

Matrix clients, additionally, maintain a bank of active Olm sessions addressed by the keys used to initialise each session (i.e. those used in the O3DH key exchange). Namely, they address sessions by (a) the other party's identity key, (b) the other party's ephemeral key contribution, and (c) their own ephemeral key contribution (with their own identity key being implicit). This can be seen in the maintenance of the st_{olm} variable in the Olms session state. These algorithms require access to a pre-initialised session store, consisting of an empty map, in the 'st_{olm}' slot of their state variable (something provided by the Matrix client and reflected in Figures 3.18 and 3.19).

When sending messages, clients will check if there already exists one or more active Olm sessions with the intended recipient. If so, they encrypt the message using the most recently used session. If not, they initialise a new session after verifying the signatures between the relevant keys of the recipients key bundle (being sure to make a record of having used an ephemeral key). When receiving messages, clients will check if it is a pre-key message or not. If it is a pre-key message, they first check to see if a matching session exists in their session store and attempt decryption with it (such a situation may occur if the session initiant sent multiple messages in the first epoch). If it is not a pre-key message, the message no longer has the requisite metadata to fetch the correct session directly. As such, clients will perform trial decryption for each session in the store matching this identity key.

3.2.3 Group Messaging

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 (see Figure 3.11a).

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.

We capture Megolm using four algorithms, Megolm = (Init, Recv, Enc, Dec), with the following syntax:

[9] This is similar to the Sesame protocol used by Signal [Mar17].

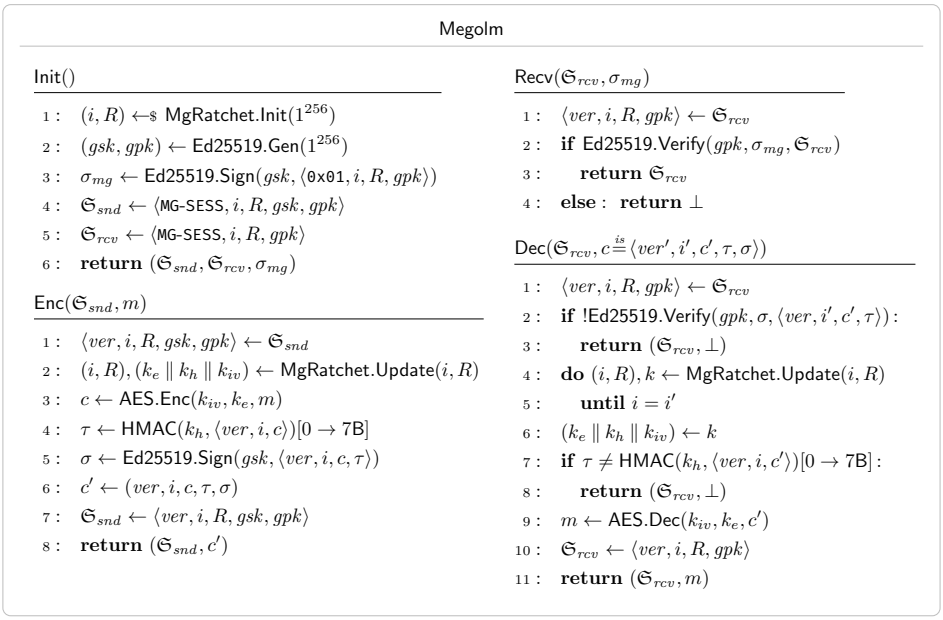


Figure 3.9. Pseudocode describing the Megolm protocol.

- $\mathfrak{S}_{snd}, \mathfrak{S}_{rcv}, \sigma_{mg} \leftarrow\$ \mathsf{Megolm.Init}()$ generates a new Megolm session consisting of a group signing key pair, a shared symmetric key and a message counter. The algorithm returns the outbound session, an inbound session and a signature over the inbound session (created with the group signing key).

- $\mathfrak{S}_{rcv} \leftarrow \mathsf{Megolm.Recv}(\mathfrak{S}_{rcv}, \sigma_{mg})$ takes as input an inbound Megolm session $\mathfrak{S}_{rcv}$ and its signature σ_{mg}. It verifies the signature using the group verification key gpk contained within $\mathfrak{S}_{rcv}$. It returns $\mathfrak{S}_{rcv}$ if the signature is valid and $\perp$ if it is not.

- $\mathfrak{S}_{snd}, c \leftarrow \mathsf{Megolm.Enc}(\mathfrak{S}_{snd}, m)$ takes as input an outbound Megolm session $\mathfrak{S}_{snd}$ 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 $\mathfrak{S}_{snd}$ and the resulting ciphertext c. If the encryption fails, it returns an updated outbound session $\mathfrak{S}_{snd}$ and $\perp$.

- $\mathfrak{S}_{rcv}, m \leftarrow \mathsf{Megolm.Dec}(\mathfrak{S}_{rcv}, c)$ takes as input an inbound Megolm session $\mathfrak{S}_{rcv}$ and ciphertext c produced by $\mathsf{Megolm.Enc}$. The algorithm checks the signature, then decrypts the message. It returns an updated inbound session $\mathfrak{S}_{rcv}$ and plaintext m if the decryption succeeds. If not, it returns an updated inbound session $\mathfrak{S}_{rcv}$ and $\perp$.

We describe each algorithm in the sections that follow (refer to Figure 3.9 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*, $\mathfrak{S}_{snd} = (ver, i, R, gsk, gpk)$ are kept by the sending device and used to encrypt messages to the room. *Inbound sessions*, $\mathfrak{S}_{rcv} = (ver, i, R, gpk)$, 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 Megolm.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[10]. When a device receives an inbound Megolm session, they use the Megolm.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 $\perp$. Clients store accepted sessions locally, indexed by the public part of the session's Megolm signing key, gpk, in combination with the identity key of the device that it believes created it, ipk.[11] We give an example of such a session store in Figure 3.11b.

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. The receiving device then uses their copy of R to verify the MAC and decrypt the message. Figure 3.11c shows how clients use the Megolm.Enc and Megolm.Dec algorithms, in combination with the Megolm session store, to send and receive messages.

The Megolm specification recommends that sessions keep old copies of the ratchet state but, since this is optional behaviour, we avoid including it in the our description of Megolm. Instead, this behaviour can be seen in our pseudocode description of the composed Matrix protocol (see Figures 3.18 and 3.19). In other words, we reflect the *application's* choice to use Megolm in such a way that undermines its forward secrecy guarantees. This choice is replicated in the reference implementations, where recipient Megolm sessions are kept at the oldest version the device has access to.

We give a sequence diagram of an example execution of the Megolm protocol in Figure 3.10.

[10] $\mathfrak{S}_{rcv}$ is output by Megolm.Init in *session export* format. When combined with its signature, $\mathfrak{S}_{rcv} \parallel \sigma_{mg}$, it is said to be in *session sharing* format [Mat22a].

[11] This is not necessarily the case for sessions received through Matrix's history sharing functionality (see Section 3.2.3).

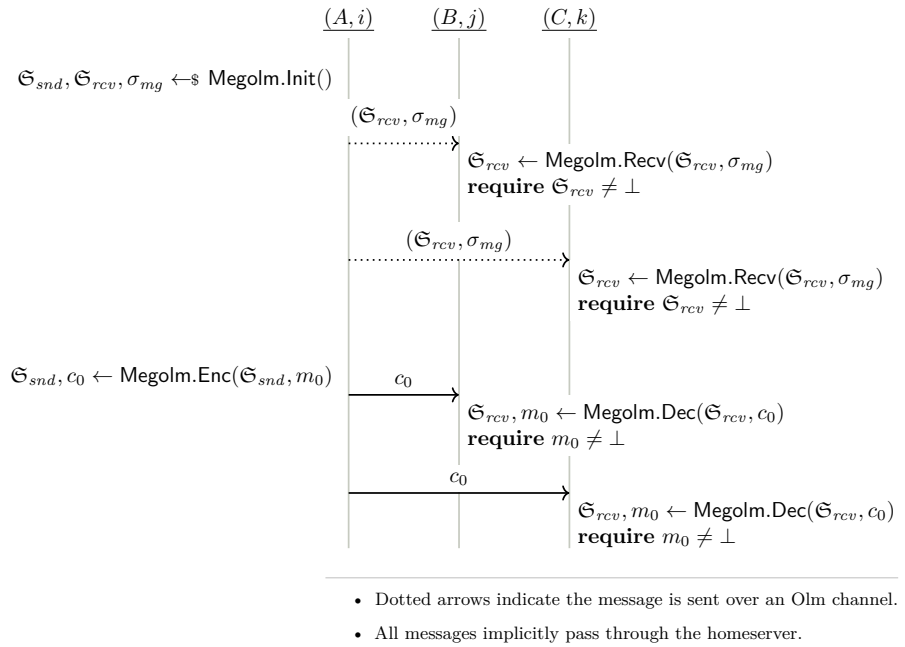


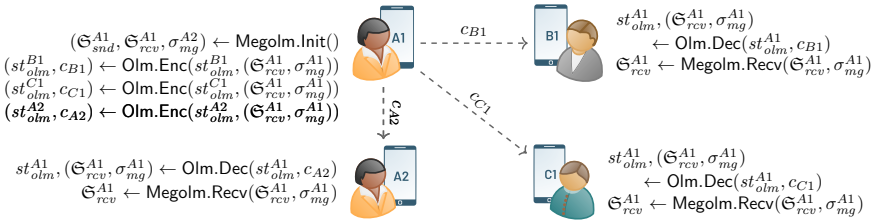
Figure 3.10. An example execution of the Megolm protocol. It shows Alice's device, $did_{A,i}$, initialising a new session and distributing it to Bob, $did_{B,j}$, and Claire, $did_{C,k}$, before sending the message m_0. Note that, whilst session setup requires an individual ciphertext for each recipient, future messages only require one for the whole group.

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 [Mat22a][12].

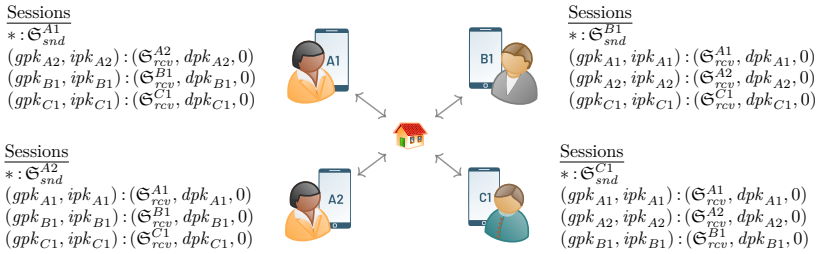
To rotate a Megolm session, the sending device simply executes Megolm.Init (effectively resetting the message index to 0, then generating a new ratchet state and group signing keypair). Next, they distribute the new inbound session $\mathfrak{S}'_{rcv}$ over Olm to the current set of group members. The public part of the Megolm signing key, *gpk*, is used to differentiate between (sub)sessions (as seen in Figure 3.11b). 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.

Megolm ratchet. Since Matrix targets large groups of participants distributed across multiple servers, it must remain performant in groups where thousands of participants regularly exchange messages. In this context, it is not unusual for participants to be inactive for a long period of time, before rejoining

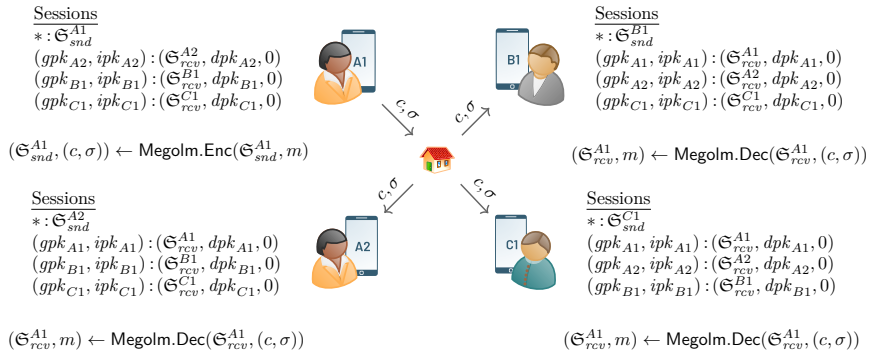
[12] The PCS of an Olm channel relies on particular usage patterns and it is not necessarily the case that Megolm session rotation follows such patterns. The security model in [BCG23] captures this relationship.



(a). Alice's first device initialises a new outbound Megolm session, and distributes the inbound state to the rest of the group (including those other devices under her control).



(b). Once all member devices have sent their first message, the Megolm session store of each member device should be populated with an inbound session for every other member device, as well as its own outbound session (indicated by '$*$').



(c). Alice's first device sends a message to the group using its sending session, $\mathfrak{S}^{A1}_{snd}$, which they proceed to receive using their copy of the appropriate inbound session, $\mathfrak{S}^{A1}_{rcv}$.

Figure 3.11. A series of figures demonstrating how a logical group is constructed through the composition of many Megolm sessions.

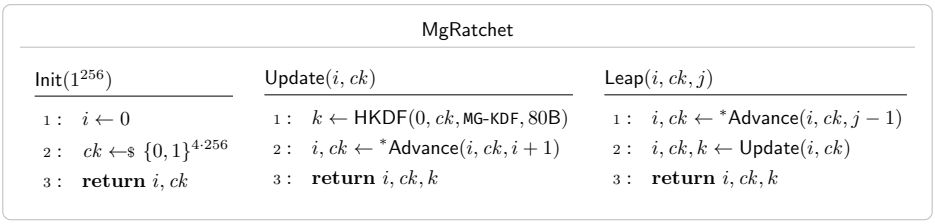


Figure 3.12. Megolm's fast-forwarding symmetric ratchet, MgRatchet.

when the discussion is relevant. In such large groups, it is possible that hundreds of thousands of messages have been exchanged during this period of inactivity. With a traditional symmetric ratchet, consisting of a hash chain, the participant's client must consecutively generate the ratchet state for each message that was missed. Megolm's ratchet allows clients to fast-forward to a state of their choosing, in logarithmic time.

The Megolm ratchet consists of a 1024 bit string, represented by ck, split into four 256 bit sections, $ck[1]$ to $ck[4]$, such that $ck = ck[1] \parallel ck[2] \parallel ck[3] \parallel ck[4]$. The ratchet is initially seeded with 1024 bits of random data (sourced from the system pseudorandom number generator). It is then constructed as a hierarchical hash chain (see Definition 2.40 in Section 2.3.8).

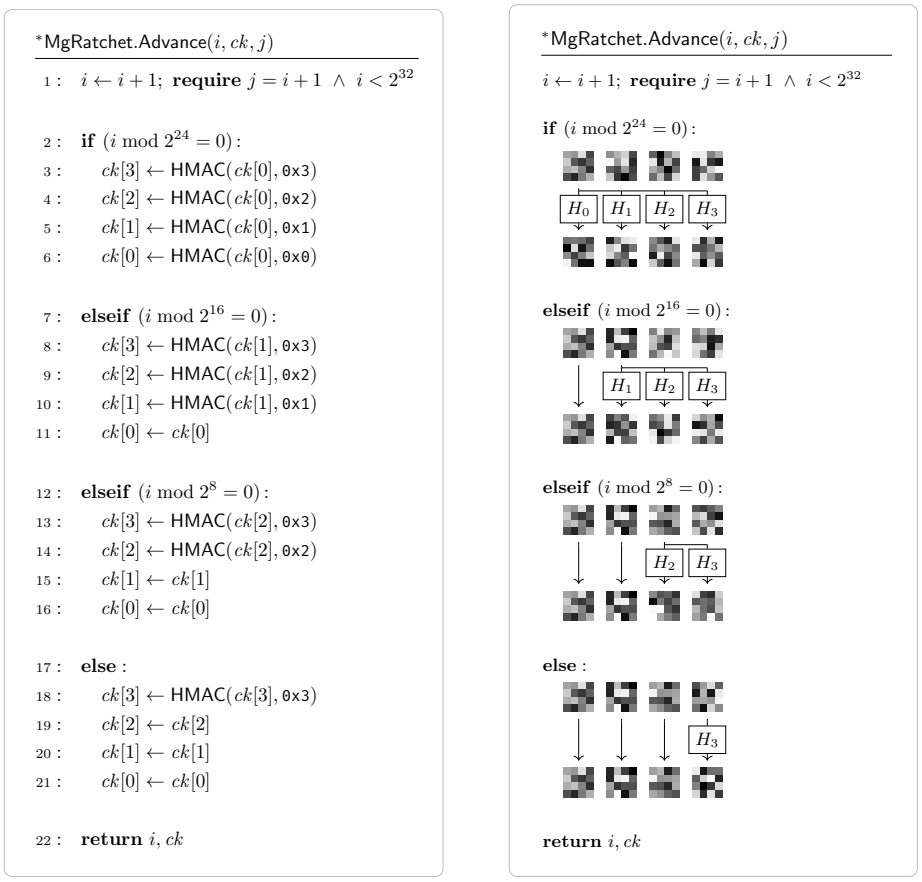
Intuitively, every 2^{24} iterations the first section advances (by applying HMAC-SHA256 to itself) after reseeding the other three sections, every 2^{16} iterations the second section advances after reseeding the third and fourth sections, every 2^8 iterations the third section advances after reseeding the fourth section and the fourth section advances itself in all other cases. The *MgRatchet.Advance algorithm (described in Figure 3.13) details how the ratchet progresses forward. The description and accompanying diagram in Figure 3.13 presents the simplified case of advancing the ratchet by a single step, in order to better communicate its structure.

This structure allows the Megolm ratchet to jump between any two indices in logarithmic time. Fast-forwarding works by ratcheting $ck[0]$ as far as possible without surpassing the target index. At this point, $ck[1]$ to $ck[3]$ are all reseeded from $ck[0]$. We proceed to ratchet $ck[1]$ as far as possible without surpassing the target index (while reseeding $ck[2]$ to $ck[3]$). This repeats for $ck[2]$, and $ck[3]$ in turn, until we reach the target index.[13]

We describe the Megolm ratchet as a tuple of three algorithms, MgRatchet = (Init, Update, Leap), which we detail in Figure 3.12.

- $(i, ck) \leftarrow\$$ MgRatchet.Init(1^{256}) initialises the ratchet, outputting a counter (with the value 0) and internal ratchet state ck.
- $(i+1, ck, k) \leftarrow$ MgRatchet.Update(i, ck) steps the ratchet forward by a single iteration. It takes as input the current counter and internal ratchet state, and proceeds to calculate the next piece of key material and ratchet

[13] In other words, we decompose the target index into a polynomial over 2^8 such that the coefficients tell us which point each section needs to be advanced to.



(a). Pseudocode. **(b).** Visualisation.

- Each four by four greyscale grid represents an HMAC-SHA256 output, and maps back to one of the chain key's constituent parts.
- The boxes labelled H_j each represent a call to HMAC-SHA256$(\cdot, \texttt{0x}j)$.

Figure 3.13. An algorithm demonstrating how the Megolm ratchet advances a single step forward.

the internal state forward, before outputting an updated counter, updated internal ratchet state and key material.

- $(j, ck, k) \leftarrow \mathsf{MgRatchet.Leap}(i, ck, j)$ progresses the ratchet an arbitrary number of steps forward. It takes as input the current counter, internal ratchet state and the target index. It forwards the internal ratchet state to j, derives the key material, then ratchets the internal state one step forward. To finish, it outputs the updated counter, internal ratchet state and key material.

Session management. Matrix clients manage a bank of inbound Megolm sessions, which we represent in our description by the 'st_{mg}' slot of the device and session state. The store is addressed through a combination of the Olm

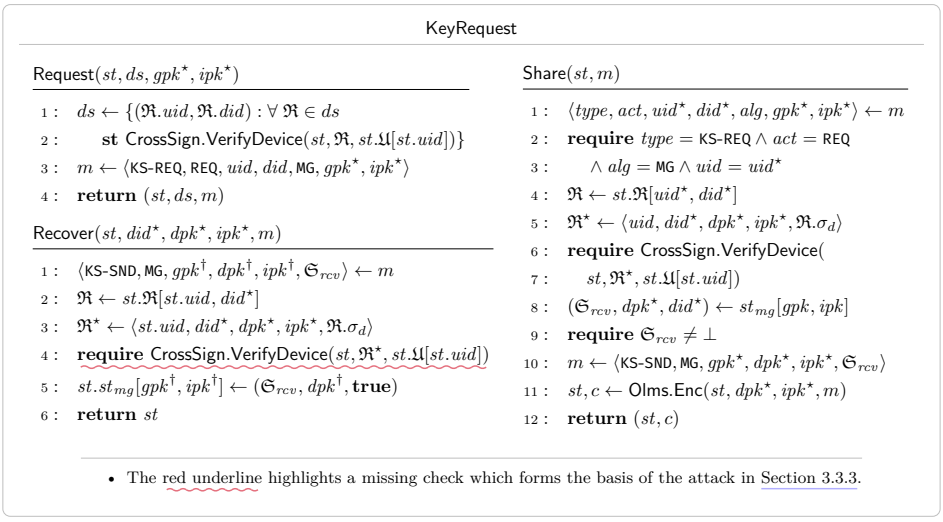


Figure 3.14. Pseudocode describing how keys are shared in Matrix.

identity key of the session owner, and the Megolm verification key of the particular session. Alongside the inbound session state, each entry additionally contains (a) the device key of the session owner, and (b) a flag indicating whether the session was acquired directly from its owner (rather than the key request or backup protocols). We detail this session management within the pseudocode of Figures 3.18 and 3.19. Figure 3.11b provides an informal example of Megolm session stores.

History sharing. 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 keys) 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 other member devices.

We describe the protocol as a tuple of three algorithms, KeyRequest = (Request, Share, Recover).

- $(st, ds, m) \leftarrow$ KeyRequest.Request$(st, ds, gpk^\star, ipk^\star)$ generates a set of key request messages. It takes as input an existing state, st, a list of device records to request from, ds, the group verification key of the Megolm session to request $gpk^\star$ and the Olm identity key of its owner $ipk^\star$. The existing state should contain the executing device's user identifier uid, device identifier did, cross-signing state (for the purposes of verifying devices) and a Megolm session store (for the purposes of sharing and receiving sessions). It checks whether each device in ds has been self-verified by the executing user. For each device that passes this check, a

plaintext `m.room_key_request` message requesting the session identified by gpk. It returns the updated state, filtered list of devices ds and the request m.

- $(st, m) \leftarrow \mathsf{KeyRequest.Share}(st, m)$ takes as input the session state and a key request message m. After ensuring the requesting device is a verified device from the same user, and it holds the requested session, it encrypts an Olm message containing the requested key for the verified cryptographic identity $ipk^\star$ of the requesting device $dpk^\star$.

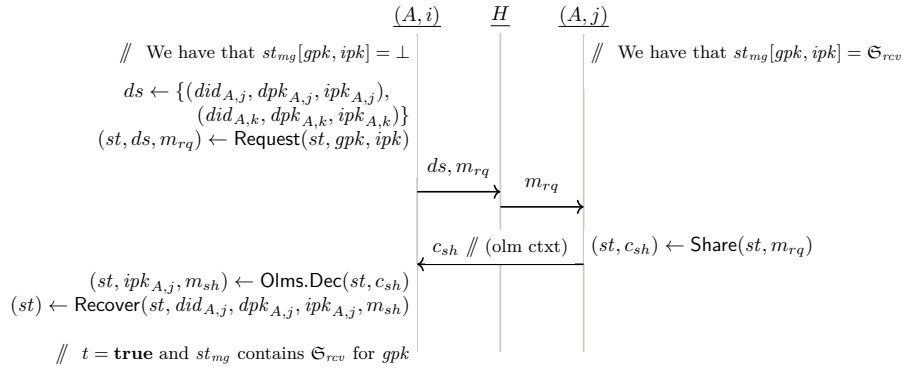
- $(st) \leftarrow \mathsf{KeyRequest.Recover}(st, did^\star, dpk^\star, ipk^\star, m)$ processes an incoming `m.forwarded_room_key` message and saves the given inbound Megolm session. It takes as input the session state, the sending device's identifier $did^\star$, the cryptographic identity of the device which sent the Olm message $(dpk^\star, ipk^\star)$, and the decrypted Olm message m containing the key share. The recipient *should* proceed to ensure the sender is a verified device from the same user. This was not the case during our initial analysis, forming the basis of the attack in Section 3.3.3. Nonetheless, if the verification passes, the client will save the session it has received into the Megolm session store and returns an updated state (st). If the verification fails, it returns $\perp$ to indicate failure.

Its construction utilises the cross-signing and Olm protocols. The cross-signing protocol determines which devices can be trusted as senders (when requesting keys) or recipients (when fielding requests). It uses the secure pairwise channels provided by Olm to share key material between protocol participants.

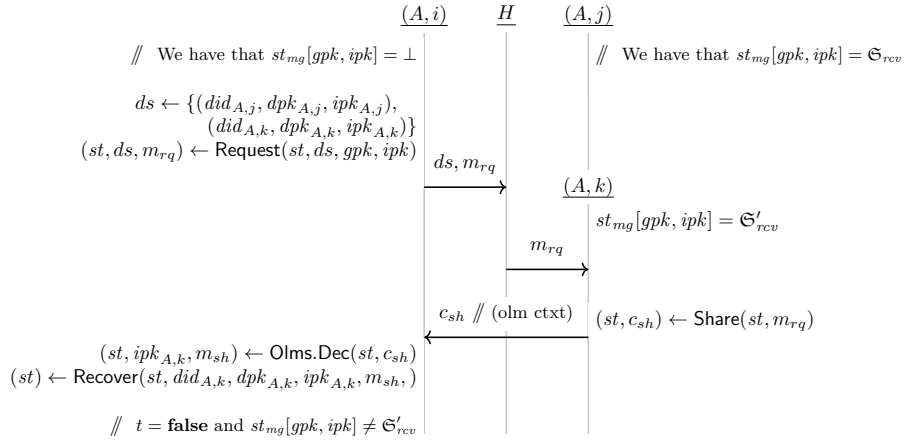
We now give a brief description of the Key Request protocol (detailed in Figure 3.14).

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 message of type `m.room_key_request` (represented in our pseudocode as `KS-REQ`) 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. KeyRequest.Request in Figure 3.14 describes the process of generating a key request message in detail.

When a device receives a key request message, they first determine whether to share the key with the requesting device. The standard specifies that Megolm sessions may only be shared between devices of the same user. Additionally, devices with cross-signing enabled may only share sessions with other verified devices of the same user. If these checks pass, the sharing device packages their copy of the inbound Megolm session in *session export format*, without the signature σ_{mg}, alongside the claimed identity of the session owner, inside a message of type `m.forwarded_room_key` (represented in our pseudocode as `KS-SND`– see Figure 3.24 for its format), and sent over Olm [Mata] (see Figure 3.20). Note that this is the only message in the Key Request protocol that is sent over Olm. KeyRequest.Share in Figure 3.14 details the process of processing and responding to a key request message.



(a). Alice's device (A, i) requests an inbound Megolm session from her other devices. Whilst (A, j) is verified, (A, k) is not, resulting in Request removing it from the list of devices ds. The homeserver forwards the (plaintext) key request message to (A, j). (A, j) uses the Share to algorithm to check whether it should share the session with (A, i). The checks pass, so (A, j) shares its copy of inbound Megolm message (encrypted with Olm using the verified device identity provided by Share). When (A, i) receives the encrypted message, its decrypt the message and passes it to the Recover algorithm to (potentially) update the Megolm session store with the requested session. Recover returns and updated state (st) to indicate success.



(b). Alice's device (A, i) requests an inbound Megolm session from her other devices. Since (A, k) is unverified, Request only requests the session from the verified device (A, j). However, in this example, the homeserver ignores (A, i)'s request to distribute the key request to (A, j) only and proceeds to distribute it to Alice's unverified device (A, k). Upon receipt of the request, (A, k) uses the Share to algorithm to share the session with (A, i). After decrypting the key sharing message, (A, i) passes it to the Recover algorithm to (potentially) update the Megolm session store with the requested session. Recover returns $\perp$ to indicate that the key share was rejected (since its sender *is not verified* as one of Alice's devices).

Figure 3.15. Two example executions of the Key Request protocol. Here, (A, i), (A, j) and (A, k) are all Alice's devices. However, while (A, i) and (A, j) are both verified, (A, k) is not.

Upon receiving a `m.forwarded_room_key` 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. KeyRequest.Recover in Figure 3.14 describes the process of receiving a `m.forwarded_room_key` message in detail.

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

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

1. **Asymmetric Megolm Key Backups:** For setup, the client generates an X25519 [Ber06] recovery key pair. The secret part of the recovery key pair is either generated from a user-supplied passphrase, or encrypted and backed up to the server using SSSS (described in Section 3.2.5). The public key is signed by the user's master cross-signing master key *msk*, then uploaded to the homeserver as part of the backup configuration.

   To backup a Megolm session, each device fetches the backup configuration from the homeserver. Clients trust the configuration if they possess the private part of the key, or if the public key has been signed by *msk*. Next, they generate a shared secret by performing DH key exchange with the recovery key and an ephemeral X25519 key pair. This shared secret is fed into HKDF-SHA256 to generate an IV and key material for authenticated encryption using AES-CBC followed by HMAC-SHA256, used to encrypt the inbound Megolm session. The resulting ciphertext is uploaded to the homeserver (with the public part of the ephemeral key).

   An asymmetric encryption scheme such as this does not authenticate the party that has created the backup [Mat21c]. This opens clients to a impersonation attack (detailed in Section 3.3.5).

2. **Symmetric Megolm Backups:** MSC 3270 [Mat21c] introduces an alternative scheme for encrypting server-side backups that does not have such a shortcoming. The scheme uses a shared secret to encrypt backups, since only devices in possession of the secret can create and decrypt such backups. Thus, clients do not necessarily mark sessions they receive through symmetric server-side backups as untrusted.

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

   To backup a Megolm session, each device fetches the backup configuration from the homeserver. Clients trust the configuration if they possess the key, or if the key has been signed by *msk*. To encrypt the session, the shared secret is fed into HKDF-SHA256 to generate key material for authenticated encryption using AES-CTR and HMAC-SHA256 (using a randomly generated IV).

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

3.2.4 Authenticating Cryptographic Identities

The Matrix standard defines an out-of-band verification framework allowing users to verify their own devices, as well as those of the people they interact with [Mata]. This functionality enables users to ensure that the cryptographic identity they are communicating with correctly maps to the intended user.

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

Our descriptions use calls to the VF.User and VF.Device algorithms to signify the execution of user-to-user verification and self-verification (respectively) using one of the out-of-band verification protocols. Each takes as input the protocol state *st* and either the user identifier and master public key (in the case of user-to-user verification) or the user identifier, device identifier and device signing key (in the case of self-verification).

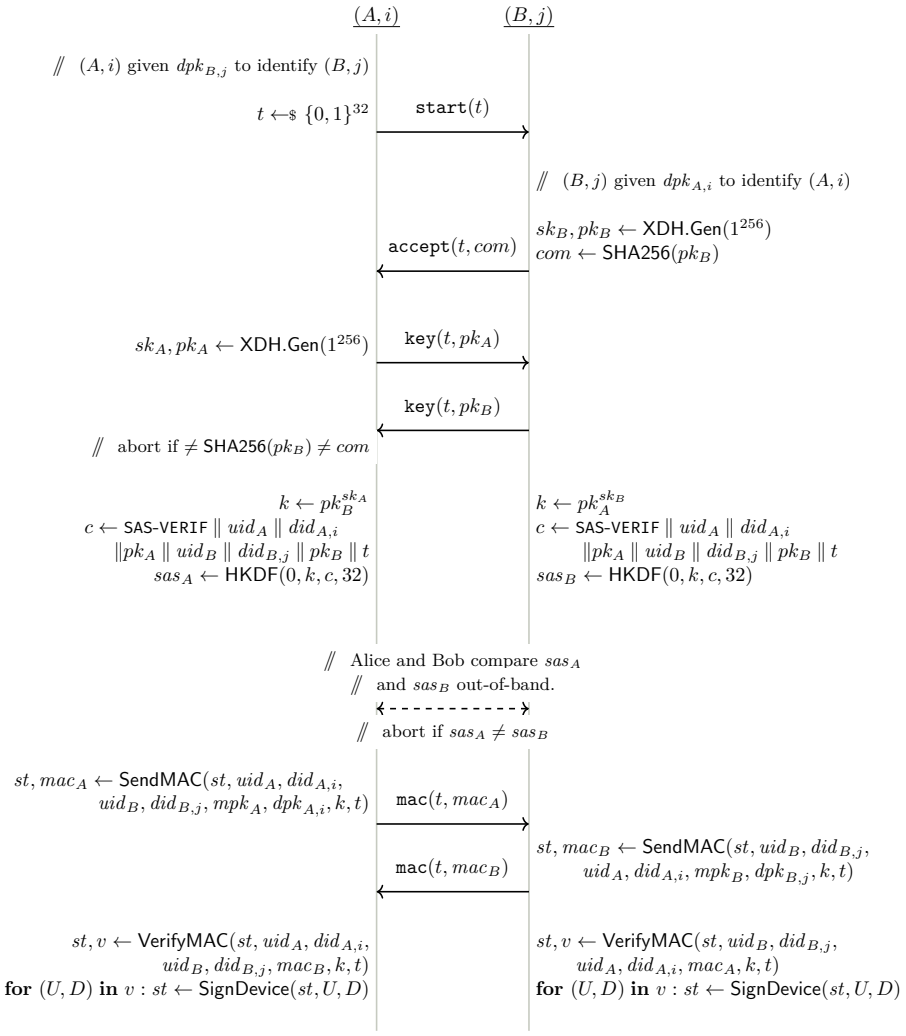
The Short Authenticated String protocol. We briefly describe Matrix' SAS protocol, which builds upon ZRTP's key agreement handshake [ZAJC11] (which, itself, builds upon the work of Vaudenay in [Vau05]). It uses an ephemeral X25519 key exchange to compute a shared secret. They subsequently derive a short authenticated string using the shared secret and details of the connection, constructed such that any attempts to modify the connection between the two parties should result in differing strings. To detect this, the parties then compare their short authentication strings through an out-of-band channel. Having established a shared secret, the two devices establish an authenticated channel over the (untrusted) network, enabling them to share their cryptographic identities.

Once the shared secret has been generated, each party compiles a list of the keys they wish to have signed into an `m.key.verification.mac` message (Figure 3.21). The shared secret is used to compute a message authentication code (MAC) for each key, calculated over its public part and details from the SAS protocol execution. A second MAC is computed over a list of key identifiers, corresponding to the list of keys for which MACs have been included. These MACs are added to the message, and ensure that only parties in possession of the shared secret can request keys for signing. Figure 3.21 describes the format of such messages used in this protocol.

How this list of keys is constructed depends on the context in which the SAS protocol is executing:

1. **User Verification:** Here, the protocol is executed between two devices, one representing each user (and holding the user's secret cross-signing keys). Each device includes their master cross-signing key *mpk* in a `m.key.verification.mac` message, which the other device will sign using their user signing key *usk*.

2. **Self Verification:** Here, the protocol is executed between the verifying device (which holds the cross-signing secret keys) and the new device, both



- Message types have had the prefix '`m.key.verification.`' removed.
- All messages (other than the out-of-band comparison) pass through the homeserver.

Figure 3.16. A sequence diagram describing the SAS protocol. The contents of `m.key.verification.mac` messages are described in Figure 3.21, while pseudocode descriptions of SAS = (SendMAC, VerifyMAC, SignDevice) are in Figure 3.17.

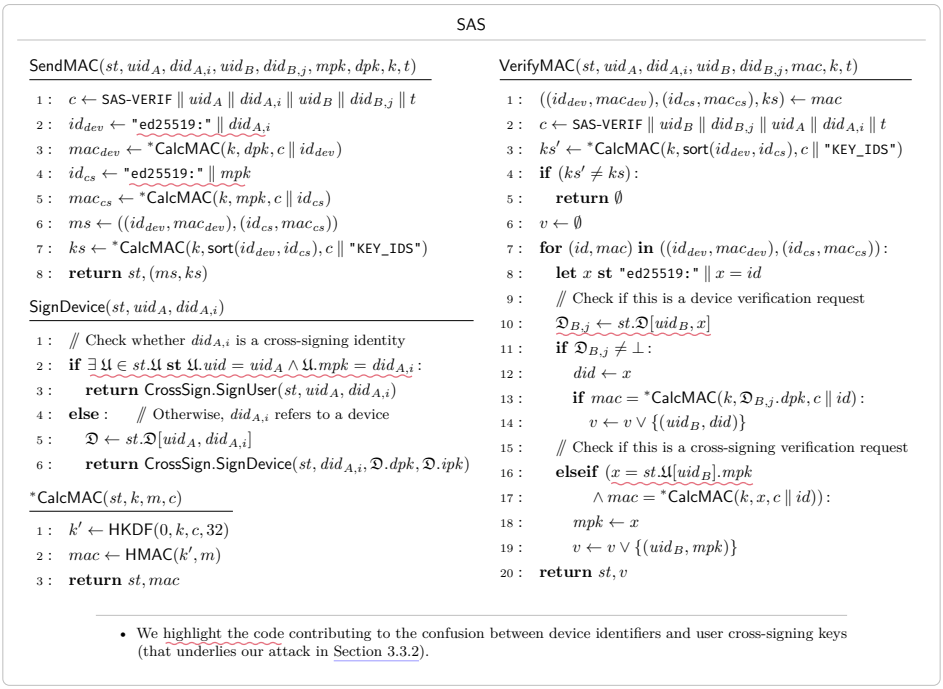


Figure 3.17. Algorithms describing Matrix' SAS protocol. Figure 3.16 demonstrates how these algorithms function together as a complete protocol.

representing the same user. The device being verified uses a `m.key.verification.mac` message to send their device identity key *dpk* to the verifying device. The verifying device then uses the device self-signing key *ssk* to sign the new device's identity key *dpk* and Olm identity key *ipk*.

A detailed description of the protocol can be found in Figures 3.16 and 3.17.

3.2.5 Secret Sharing, Backup and Recovery

The *Secure Secret Storage and Sharing (SSSS)* module enables devices to share secrets with the user's other trusted devices [Mata]. It provides two sets of functionality:

1) Backup secret key material to the server, encrypted symmetrically using a key generated from a master passphrase.

2) Distribute these secrets to other devices via a request-response protocol (similar to the Key Request protocol described in Section 3.2.3).

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

Secret sharing. An example of where this process is triggered is directly after self-verification. Here, the newly verified device will request copies of the user's

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

Secret backups. Given a user-provided master passphrase, the SSSS scheme will back up their secret user cross-signing keys along with the keys used to decrypt Megolm backups to the server. We do not consider the SSSS backup protocol in this work. See Section 10.13 (*Secrets*) in the Matrix specification for more details [Mata].

3.2.6 The Matrix Multi-Device Group Messaging Protocol

We now describe how the Matrix specification combines the protocols already described into a multi-device group messaging protocol. We express this subset of the standard as seven algorithms, Matrix = (Gen, Reg, Init, Recv, Add, Rem, Enc, Dec, KeyRequest). These algorithms (which we detail in Figures 3.18 and 3.19) capture the interactions of a single group, its users and their devices.

User and device setup. The Matrix.Gen algorithm captures the initialisation of a user and their cryptographic identity, as described in Section 3.2.1, and outputs the user-level secret state. The Matrix.Reg algorithm 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.

Group initialisation. Matrix.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 Matrix.Add algorithm must then be executed by each device, for every other device, to initialise the group membership.[14]

Adding devices. To add a device, the current members' devices execute Matrix.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 Matrix.Rem algorithm is run by each device in the group. They generate a fresh Megolm session using Megolm.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

[14] It is possible for different devices in the group to have different views of the group membership.

Matrix (1)

Gen(uid)

```
1 : // Set up new user's cross-signing identity
2 : st_u ← CrossSign.Init(uid); return st_u
```

Reg(st_u, did)

```
1 : // Setup new device and register with user
2 : // (simulates protocol between user and device).
3 : // Generate device identity and keys.
4 : 𝔇_sk, 𝔇 ← Olms.Gen(st_u.uid, did, 10, 1)
5 : // (Optionally) perform self-verification.
6 : st_u, ℜ ← VF.Device(st_u, did, 𝔇.dpk, 𝔇.ipk)
7 : // Initialise device state with
8 : st_d ← ⟨st_u.uid, did   // user & device identifiers,
9 :   𝔘.mpk, 𝔘.upk, 𝔘.spk,   // root of trust, and
10 :  𝔇.dpk, 𝔇.ipk, 𝔇.epk, 𝔇.fpk,   // device keys
11 :  𝔇_sk.dsk, 𝔇_sk.isk, 𝔇_sk.esk, 𝔇_sk.fsk⟩
12 : return st_u, st_d, (𝔇, ℜ)
```

Init(st_d, gid)

```
1 : // Initialise session state (from device state st_d for
2 : // group gid) with root-of-trust and device secrets.
3 : st ← ⟨gid, st_d.uid, st_d.did,
4 :   st_d.mpk, st_d.upk, st_d.spk
5 :   st_d.dpk, st_d.ipk, st_d.epk, st_d.fpk,
6 :   st_d.dsk, st_d.isk, st_d.esk, st_d.fsk⟩
7 : st.CU ← {uid}   // Group members and devices
8 : st.CD ← {(uid, did, st_d.dpk, st_d.ipk)}
9 : st.𝔘 ← Map{ }   // User key packages
10 : st.𝔇 ← Map{ }   // Device key packages
11 : st.ℜ ← Map{ }   // Device records
12 : st.σ_× ← Map{ }   // User-to-user cross-signatures
13 : st.st_olm ← Map{ }   // Olm states
14 : st.st_mg ← Map{ }   // Inbound Megolm sessions
15 : // Initialize outbound Megolm session
16 : 𝔖_snd, 𝔖_rcv, σ_mg ←$ Megolm.Init()
17 : st.𝔖_snd ← 𝔖_snd
18 : st.st_mg[gpk, st_d.ipk] ← (𝔖_rcv, st_d.dpk], false)
19 : return (st, gid)
```

Add(st, $uid^\star$, $did^\star$, $dpk^\star$, $ipk^\star$)

```
1 : // Add device did* to group.
2 : // If user is new to group, check their identity.
3 : if uid* ∉ st.CU :
4 :   mpk* ← st.𝔘[uid*]
5 :   // (Optionally) trigger out-of-band verification
6 :   st ← VF.User[st, uid*, mpk*]
7 :   // (if successful, cross-signature added to st.σ_×).
8 : // Verify given device identity.
9 : require CrossSign.VerifyDevice(
10 :   st, st.ℜ[uid*, did*], st.𝔘[uid*])
11 : // If checks pass, add to member list.
12 : st.CD ← st.CD ∪ {(uid*, did*, dpk*, ipk*)}
13 : st.CU ← st.CU ∪ {uid*}
14 : // Then share outbound session.
15 : ⟨ver, i, R, gsk, gpk⟩ ← st.𝔖_snd
16 : 𝔖_rcv ← ⟨ver, i, R, gpk⟩
17 : σ_mg ← Ed25519.Sign(gsk, 𝔖_rcv)
18 : m ← ⟨MG-KEY, MG, gid, 𝔖_rcv, σ_mg⟩
19 : st, c ← Olms.Enc(st, dpk*, ipk*, m)
20 : return (st, c)
```

Rem(st, $uid^\star$, $did^\star$, $dpk^\star$, $ipk^\star$)

```
1 : // Remove given device from group.
2 : st.CD ← st.CD \ {(did*, dpk*, ipk*)}
3 : // Remove a user if all of their devices
4 : // have been removed.
5 : if ∄(uid*, did', ·, ·) ∈ st.CD :
6 :   st.CU ← st.CU \ {uid*}
7 : // Generate new outbound session
8 : 𝔖_snd, 𝔖_rcv, σ_mg ←$ Megolm.Init()
9 : st.st_mg[gpk, 𝔇.ipk] ← (𝔖_rcv, 𝔇.dpk, false)
10 : st.𝔖_snd ← 𝔖_snd
11 : // Share new session with remaining members
12 : cs ← {}
13 : for (uid', did', dpk', ipk') in st.CD
14 :   (st, c) ← Matrix.Add(st, ipk')
15 :   cs ← cs ∪ {c}
16 : return (st, cs)
```

Figure 3.18. Pseudocode describing secure group messaging in Matrix.

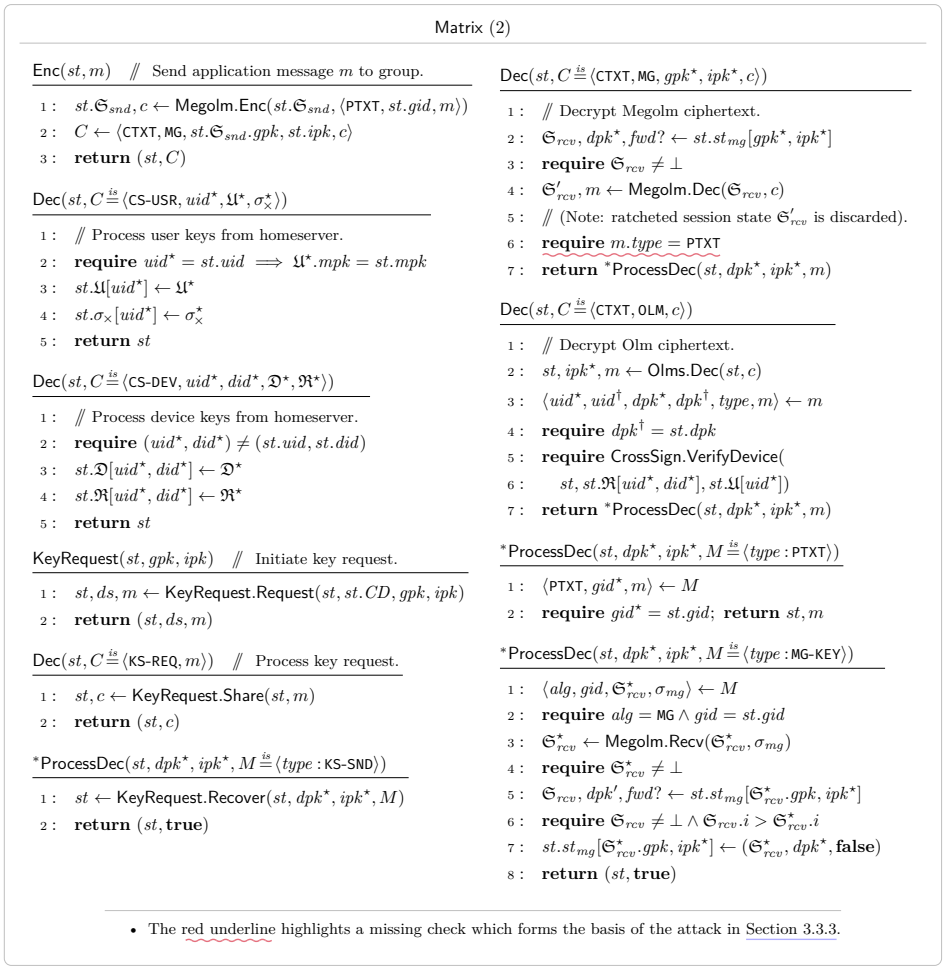


Figure 3.19. Pseudocode describing secure group messaging in Matrix.

one operation, reducing the overall cost. Our description in Matrix.Add and Matrix.Rem 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 Matrix.Add (and Matrix.Rem, resp.) for each of the user's devices.

State sharing. When a device receives a ciphertext that it is unable to decrypt, they may initiate the Key Request protocol using the Matrix.KeyRequest algorithm. The device executes Matrix.KeyRequest(st_{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 KeyRequest.Request.

3.3 Vulnerabilities in Matrix and Element

We proceed to detail five practically-exploitable (and one theoretical) cryptographic vulnerabilities spread across the Matrix standard, its reference implementations and the flagship client Element.[15]

Our attacks target a variety of settings, but focus on cases where encrypted messaging and verification are enabled, i.e. in the presence of the strongest protections offered by the protocol. When relying on implementation specific behaviour, we target the Matrix standard as implemented by the `matrix-react-sdk` and `matrix-js-sdk` libraries.[16] These libraries provide the basis for the aforementioned flagship client, Element [L17].

We briefly describe each attack.

1. **Server-controlled Group Membership:** In Section 3.3.1 we describe how a lack of cryptographic control of group membership enables malicious homeservers to control the list of users in a room.

2. **Undermining Out-of-Band Verification:** In Section 3.3.2 we report a vulnerability in out-of-band verification in Matrix which, thanks to a lack of domain separation between device identifiers and the encoding of user master keys, enables a mallory-in-the-middle (MITM) attack.

3. **(Partial) Authentication Break in Group Messaging:** Whilst Matrix clients restrict who they share keys with for the purpose of history sharing, no such verification was implemented when receiving keys. Our attack exploits this lack of verification to send attacker controlled group messaging sessions to a target device, claiming they belong to a session of the device they wish to impersonate. In Section 3.3.3 we detail this vulnerability and how it may be exploited. Messages decrypted using keys shared via history sharing functionality are marked in the user interface. Thus, we describe this attack as providing *partial* authentication break.

4. **Authentication Break in Group Messaging:** In Section 3.3.4 we report a second impersonation attack which builds upon the first. Here, we exploit a protocol confusion vulnerability, whereby message types expected to originate from a pairwise channel will be accepted when sent over a group messaging channel. This enables an adversary to perform a group key exchange over an existing group messaging channel. We use this to build upon the previous attack, allowing for impersonation without any trust issues.

5. **Confidentiality Break in Group Messaging:** When a user verifies a new device (through the self-verification process), this new device will request from the verifying device a copy of the key used to encrypt server-side backups. The new device will then proceed to backup inbound

[15] The vulnerabilities we describe make varying use of implementation-specific behaviour. In each case, however, we pinpoint issues with the specification that led to such issues.

[16] Our analysis is based on, and our proof-of-concept attacks tested against, Element Web at commit `#479d4bf` [New22] with `matrix-react-sdk` at commit `#59b9d1e` [Matc] and `matrix-js-sdk` at commit `#4721aa1` [Matb].

Megolm sessions to the server, encrypted with the provided backup key, so that they may recover them in the future. In Section 3.3.5, we exploit the same protocol confusion as above to perform this sharing of backup keys over a group messaging channel. This can be combined with our previous impersonation attack to allow an adversary to choose the backup key that new devices will use.

6 **IND-CCA Insecure Backup Scheme:** AES-CTR is used as the basis of an authenticated encryption scheme in a number of places across Matrix, *without* authenticating the initialisation vector. Section 3.3.6 describes how this may be used to break the IND-CCA security of the scheme. To the best of our knowledge, this attack is of theoretical interest and cannot be exploited in practice.

To demonstrate their effectiveness, we developed proof-of-concept attacks for vulnerabilities two to five. For vulnerability two, we developed a variant of the Synapse homeserver application that will MITM all encrypted channels (between different users) without detection. For vulnerabilities three to five, we wrote a combined attack that enabled arbitrary impersonation and access to the plaintext contents of all messages.

For each attack, we describe (a) the vulnerabilities that led to it, (b) the attack itself, (c) its scope, and (d) any planned or implemented remediations. We discuss the impact of these vulnerabilities, the disclosure process and the design decisions that led to them in Section 3.4.

Remark 3.1. *While much of this section is written in the present tense, i.e. as if the vulnerabilities are still present, we disclosed our attacks to the Matrix developers between 20th May 2022 and 6th July 2022. Remediations for many of these issues were distributed by Matrix as part of a coordinated disclosure process, culminating in a public disclosure on 28th September 2022.*

```
{
  "messages": {
    "<receiver_user_id>": {
      "<receiver_device_id>": {
        "algorithm": "m.olm.v1.curve25519-aes-sha2",
        "sender_key": sender_ipk,
        "ciphertext": {
          "<receiver_ipk>": {
            "type": olm_msg_type,
            "body": ctxt
          }
  ...
}
```

(a). Message sent by the device.

```
{
  "type": "m.room.encrypted",
  "sender": sender_user_id
  "content": {
    "algorithm": "m.olm.v1.curve25519-aes-sha2",
    "sender_key": sender_ipk,
    "ciphertext": {
      "<receiver_ipk>": {
        "type": olm_msg_type,
       "body": ctxt
      }
  ...
}
```

(b). Message as forwarded by homeserver.

Figure 3.20. Figure 3.20a shows the format of an Olm encrypted to-device message as it is sent to the homeserver (the message type, `m.room.encrypted`, is encoded in the URL). The homeserver will split up to-device messages, collate them by device, then redistribute them as a list of messages in the format seen in Figure 3.20b. The `sender` field is added by the homeserver (amongst other fields not considered in this document). The `sender_key` field is used by the receiving device to locate the correct Olm session with which to decrypt the message.

3.3.1 Server-controlled Group Membership

In this section we consider the control the homeserver has over group membership. These attacks compromise confidentiality, and do so trivially since no cryptographic protection exists by design.

Forging invitations. The Matrix standard allows roles and permissions to be assigned to users in a room. Amongst other things, these roles and permissions control which users are allowed to manage the room membership. Unfortunately, room management messages are neither encrypted, checked for integrity nor cryptographically authenticated. A malicious homeserver can forge room management messages to appear as if they are from users with permission to change room membership, simulating the process of a new user being invited then joining the room. Thus, the homeserver has control of the member list also for encrypted rooms.

The Element client does exhibit some behaviour to help mitigate such attacks. That is, the user interface displays an up-to-date list of members and, when a user is added to a room, this is also displayed as an event in the timeline. Thus, such attacks are, in principle, detectable by users. Nonetheless, we stress that such a detection requires careful manual membership list inspection from users and that this event appears to participants as a legitimate group membership event. In other words, the user interface presents the membership change as if an existing member invited the new member.

Remediation

The developers consider the fact that the homeserver controls room membership as a risk they accept as part of their threat model. Whilst they do not plan to include any countermeasures for this attack at the time of disclosure, they are developing a solution to target this stronger threat model [Fay]. A brief summary of their design follows. When inviting a user to join the room, the inviting user must include the master cross-signing key of the new user in a signed message. In doing this, the transcript of invites form a tree of signatures, rooted in the room's creation event. This solution is currently in the design phase.[17]

3.3.2 Undermining Out-of-Band Verification

In this section, we describe a vulnerability that undermines the security of the SAS protocol for out-of-band verification (described in Section 3.2.4). This vulnerability (colliding encodings between user cross-signing identities and server-controlled device identifiers) enables an attack whereby the server can trick parties executing the SAS protocol into signing server-controlled cross-signing identities (rather than their own). This can be used to perform an active MITM attack between any pair of users by simply advertising a server controlled master cross-signing identity for each user and, later, if they perform out-of-band verification, exploiting this vulnerability.

[17] In [CPZ20] a solution to a similar problem is detailed (with the additional requirement that the membership list is kept private).

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

(a). The format of an `m.key.verification.mac` message for a user with cross-signing setup.

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

(b). The format of `m.key.verification.mac` messages created by the target of our attack.

Figure 3.21. A demonstration of how messages in the attack described in Section 3.3.2 are constructed. Here, the homeserver has generated mpk' itself, proceeding to assign it to the target as its device identifier, and to all other clients as its cross-signing master key. Whilst the two entries in the `mac` dictionary could be distinguished by the differing second argument given to *CalcMAC, SAS.VerifyMAC interprets the first entry as a device, and then passes it to SAS.SignDevice which interprets it as a cross-signing identity (see Figure 3.17).

Vulnerability

Recall that, in the context of cross-signing, out-of-band verification is used in two cases. First, where two users verify each other (user-to-user verification); and, second, where a user verifies a new device (self-verification). However, the SAS.SendMAC algorithm (see Figure 3.17), which generates `m.key.verification.mac` messages (see Figure 3.21), does not distinguish between these two cases. It always sends both keys.

Cross-signing keys as devices. Within `matrix-js-sdk` and Synapse, cross-signing identities are sometimes treated as devices. The same is true in the SAS protocol, as noted by the specification [Mata].[18] Thus, in some cases the string x in '`ed25519:<x>`' is interpreted as a device identifier, and in others it is interpreted as a cross-signing master verification key.

Since the homeserver allocates device identifiers, it is able to generate a string that is both a valid device identifier and a valid cross-signing master verification key. In this attack, the homeserver generates a cross-signing identity for the user they would like to impersonate, then sets this as the device identifier for the user's first device. Figure 3.21 demonstrates the format of such a message.

Processing of `m.key.verification.mac` messages. Referring to Figure 3.21, $mpk_{\mathcal{A}}$ is both a valid device identifier and cross-signing master verification key. However, the MAC that has been calculated includes the device identity key of the device it maps to. It can only pass the MAC verification as a device, not as a cross-signing master key. When processing `SAS-MAC` messages, however,

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

the `matrix-js-sdk` handles the aforementioned ambiguity inconsistently in a manner that enables MAC verification to pass.

- When verifying the MACs, SAS.VerifyMAC (see Figure 3.17) first checks if the string is a device identifier, then checks if it maps to a cross-signing identity. The entry for $mpk_{\mathcal{A}}$ passes verification as if it were a request for a device to be verified.

- When fulfilling the signing request, SAS.SignDevice (see Figure 3.17) first checks if the string maps to a cross-signing identity, then checks whether it is a device identifier. If the processing device has an entry for $mpk_{\mathcal{A}}$ in its cross-signing directory, it will sign the user and cross-signing identity then upload it to the homeserver.

Attack

Consider a setting with two users: Alice, uid_A, and Bob, uid_B. Alice is represented by the device $did_{A,1}$ and Bob is represented by the device $did_{B,1}$. A malicious homeserver may proceed as follows:

1) When Alice registers their account with the homeserver, the homeserver generates a parallel cross-signing identity with verification keys $(mpk_{\mathcal{A}}, spk_{\mathcal{A}}, upk_{\mathcal{A}})$.

2) When Alice logs in for the first time, the homeserver sets the device identifier as the parallel cross-signing identity: $did_{A,1} \leftarrow mpk_{\mathcal{A}}$.

3) When Bob logs in for the first time, the homeserver sets the device identifier $did_{B,1}$ as normal.

4) Alice and Bob each setup their own cross-signing identities with verification keys (mpk_A, spk_A, upk_A) and (mpk_B, spk_B, upk_B) respectively. They upload these to the homeserver.

5) The homeserver proceeds to present two versions of the cross-signing state:

   a) When Alice requests their own cross-signing information, they are presented with the version they uploaded (mpk_A, spk_A, upk_A).

   b) When Bob requests Alice's cross-signing information, they are presented with the version generated by the malicious homeserver $(mpk_{\mathcal{A}}, spk_{\mathcal{A}}, upk_{\mathcal{A}})$.

6) Alice and Bob perform an out-of-band verification using the SAS protocol. At the end, they exchange `m.key.verification.mac` messages containing their cryptographic identity (for signing). Figure 3.21 shows the structure of the message Alice sends. $did_{B,1}$ processes it as follows:

   a) SAS.VerifyMAC interprets the entry for $mpk_{\mathcal{A}}$ as a request for device verification. It fetches the expected device identity key $dpk_{A,1}$, then calculates a matching MAC. The device identity key pulled from the homeserver is legitimate, and matches the one used by Alice's device to generate the MAC. Thus, the message passes verification.

b) SAS.SignDevice interprets the entry for $mpk_{\mathcal{A}}$ as a request for cross-signing verification. This is because the homeserver has led Bob's client to believe that Alice's cross-signing identity is $mpk_{\mathcal{A}}$.

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

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

Limitations

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

Remediation

At the time of disclosure, the Matrix developers claim to have fixed the inconsistent processing that led to this attack. In particular, they claim that Element now consistently processes identifiers in the same order, such that colliding cross-signing user identities and device identifiers are consistently interpreted throughout the codebase. We recommend that this order is formalised in the Matrix specification (with the security implications of a mistake explained).

However, the proposed fix does not solve the underlying vulnerability: a lack of domain separation between device identifiers and cross-signing keys. In the long-term, the Matrix developers plan to separate the format for device identifiers and Ed25519 keys.

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

3.3.3 (Partial) Authentication Break in Group Messaging

In this section, we describe an attack that allows a malicious device and homeserver to impersonate any device (in group messaging sessions). While the resulting messages are displayed with a warning: "*The authenticity of this encrypted message can't be guaranteed on this device*", this is shown alongside all messages decrypted using sessions obtained via the Key Request protocol. In other words, this attack achieves the same level of trust as keys legitimately shared via the Key Request protocol. Nonetheless, given these warnings, we consider this attack to only *partially* break authentication.

Vulnerability

Unlike sessions received via Megolm session distribution messages, those received via the Key Request protocol do not have a cryptographic link with the session owner. Instead, they are cryptographically linked with the forwarding party (via the Olm channel they are sent over). This works, in theory, as long as the recipient trusts the device that shared it.[19]

The key sharing restrictions in the Matrix standard, previously discussed in Section 3.2.3, require that the sharing device is another verified device from the same user. Clients using `matrix-js-sdk` only check this when *requesting* a session, not when *receiving* it.[20] This can be seen in our pseudocode description of the Key Request protocol in Figure 3.14.

Attack

We now describe how this issue can be exploited to perform an impersonation attack. See Figure 3.22 for an accompanying diagram.

Consider a setting with three users: Alice, uid_A, Bob, uid_B, and an adversarial user, $uid_{\mathcal{A}}$. Each user has a single logged-in device: $did_{A,i}$, $did_{B,j}$, and $did_{\mathcal{A},k}$ (respectively). We let gid identify a group consisting of Alice and Bob.

In order to impersonate Bob to Alice in this group, the adversary may do the following.

1) Generate a new Megolm session $\mathfrak{S}_{snd} = (i, R, gsk, gpk)$ with inbound counterpart $\mathfrak{S}_{rcv} = (i, R, gpk)$ and signature σ_{mg}.

2) Share this session using the Key Request protocol with the claim that Bob is the session owner. In particular, the adversary constructs `m.forwarded_room_key` message, ensuring to set the fields identifying the original session owner to Bob's identity.

3) Send Alice the `m.forwarded_room_key` message over an Olm channel.

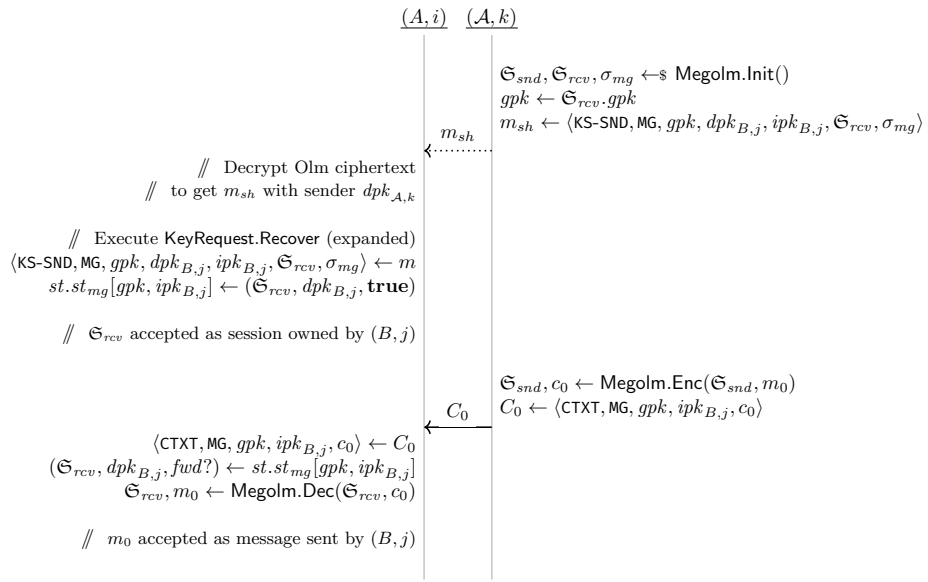
When Alice's device receives this message, it will accept the forwarded key and store it associated with the identity key $ipk_{B,j}$ and device key $dpk_{B,j}$ claimed in the message.

At the end of this process, the adversary knows the $\mathfrak{S}_{rcv}$ of a Megolm session, i.e. knows the outbound session secrets for an inbound session, that Alice's device believes to be owned by Bob's device. This allows the adversary to send messages to Alice that will be displayed in the user interface as having come from Bob.

When determining the user that sent a message, each device starts by looking at the '`sender`' field of the ciphertext wrapper. The device then ensures that

[19] Since keys can be shared multiple times, the recipient needs to trust every device in the resulting chain.

[20] This can be seen in the function `MegolmDecryption.onRoomKeyEvent` in `matrix-js-sdk` (commit `#4721aa1`) which checks the `.senderKey` field in an event object to ensure the message was encrypted. However, it will accept `m.forwarded_room_key` messages from any user or device.



- Dashed arrows indicate the message is sent over an Olm channel.
- All messages implicitly pass through the homeserver.

Figure 3.22. The impersonation attack described in Section 3.3.3.

the cryptographic identity used to send the message matches the value in this field. Since this field is set by the homeserver, this attack requires the aid of a colluding homeserver.

Remediation

At the time of public disclosure, the Matrix developers claim to have included checks to ensure that clients will only accept Megolm sessions shared as a response to previously issued requests. Since clients correctly request keys from trusted devices only, this aims to ensure that they only accept them from trusted devices, as well.[21]

3.3.4 Authentication Break in Group Messaging

Recall that Matrix relies on Olm to provide secure pairwise channels between pairs of devices. This is utilised as a signalling layer, through which protocols such as Megolm and the Key Request protocol operate. Importantly, this allows these protocols to inherit the authentication guarantees of the underlying Olm channels (and, in turn, the authentication guarantees provided by cross-signing).

In this section, we describe a vulnerability in the processing of decrypted messages that allows signalling messages, intended to only be exchanged over

[21] Matrix optionally allows sharing room history with new members. This is implemented by sharing previous inbound Megolm sessions with the new member. The fix described will not apply to such messages, though the resulting sessions will be marked as untrusted in the code and user interface.

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

(a). Message sent by the device.

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

(b). Message as forwarded by homeserver.

Figure 3.23. The figure on the left (a) shows the format of a Megolm encrypted to-device message as it is sent to the homeserver. The homeserver splits up this structure, collating messages by recipient device, before redistributing them as a list of messages in the format seen on the right (b). Synapse does not preserve the `room_id` field of messages when converting between the two formats shown here, and thus our attack requires a colluding homeserver to enable the protocol confusion. The `sender` field is added by the homeserver and, similarly, requires cooperation from the homeserver to set it to the attacker's desired value.

Olm, to be exchanged over Megolm channels as well. This is problematic since, as we have seen in the previous section, the authentication guarantees of Megolm channels have been lowered in certain circumstances in the name of reliability.

In particular, our attack will use an existing Megolm channel (shared by an adversary through the attack described in the previous section) to perform an initial distribution of a new Megolm session. This session, not having been shared via the Key Request protocol, is no less trusted than one distributed via Olm. Thus, an adversary employing the attack described in this section is able to achieve a complete authentication break in group messaging sessions.

Vulnerability

We now describe the vulnerabilities that enable this attack.

Protocol confusion. The Matrix and Megolm specifications require that Megolm distribution messages are sent over encrypted Olm channels. Whilst the specification does not include this requirement for key sharing messages in the Key Request protocol, we believe this to be the intended behaviour.

However, the handler for these incoming messages only requires that they have been encrypted; it does not check which algorithm they were encrypted with. It is therefore possible to encrypt Megolm distribution messages (of type `m.room_key`) using Megolm rather than Olm, provided they are distributed through to-device messaging. This issue is highlighted in our pseudocode description of Matrix, see Figure 3.18. Since Megolm sessions are not intended to be distributed through the to-device mechanism, their formatting requires some massaging to ensure clients process them appropriately. We describe the necessary modifications in Figure 3.26 (which require a colluding homeserver).

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

(a). Format of Megolm session distribution messages (i.e. those with '*type* = `MG-KEY`' in our description).

```
{
  "type": "m.forwarded_room_key",
  "content": {
    "algorithm": "m.megolm.v1.aes-sha2",
    "room_id": room_id,
    "session_id": gpk,
    "session_key": ver || i || R || gpk,
    "chain_index": i,
    "sender_key": claimed_ipk,
    "sender_claimed_ed25519_key":
      claimed_dpk,
    "forwarding_curve25519_key_chain":
      [ipk_1, ipk_2, ..., ipk_n],
  }
}
```

(b). Format of messages used to share Megolm sessions as part of the Key Request protocol (i.e. those with '*type* = `KS-SND`' in our description).

Figure 3.24. A comparison between the message format of Megolm session distribution messages and those of the Key Request protocol, both of which should be distributed over Olm channels.

Inheriting sender identity. When Megolm messages are decrypted, they inherit the sender identity[22] associated with the Megolm session used for decryption. Similarly, when Megolm sessions are received through a Megolm distribution message (of type `m.room_key`), they inherit the sender identity of the encrypted channel they were sent over. In the expected case, this will be the sender identity of the Olm channel it was sent over. When sending such messages over a Megolm session, however, it will inherit the sender identity that the Megolm session inherited (which, in turn, is inherited from the encrypted channel through which it was distributed).

Attack

The attacker uses the forwarded Megolm session from the attack in Section 3.3.3 to deliver a second Megolm session to the target device that is indistinguishable from a legitimate session. The inheritance strategy described above maintains the cryptographic link between the Olm channel first used to send a Megolm session, and any subsequent Megolm sessions sent over it.

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

This attack starts at the end of the attack describe in Section 3.3.3, where the adversary's device has forwarded the Megolm session $\mathfrak{S}_{rcv}$ to Alice's device in order to impersonate Bob's device. The adversary proceeds as follows:

1) Generate a new Megolm session $\mathfrak{S}^{\star}_{snd} = (i^{\star}, R^{\star}, gsk^{\star}, gpk^{\star})$ with inbound counterpart $\mathfrak{S}^{\star}_{rcv} = (i^{\star}, R^{\star}, gpk^{\star})$ and signature $\sigma^{\star}_{mg}$.

[22] Specifically, we use *sender identity* to refer to the combination of a user's cross-signing keys and a set of long-term device keys (dpk, ipk).

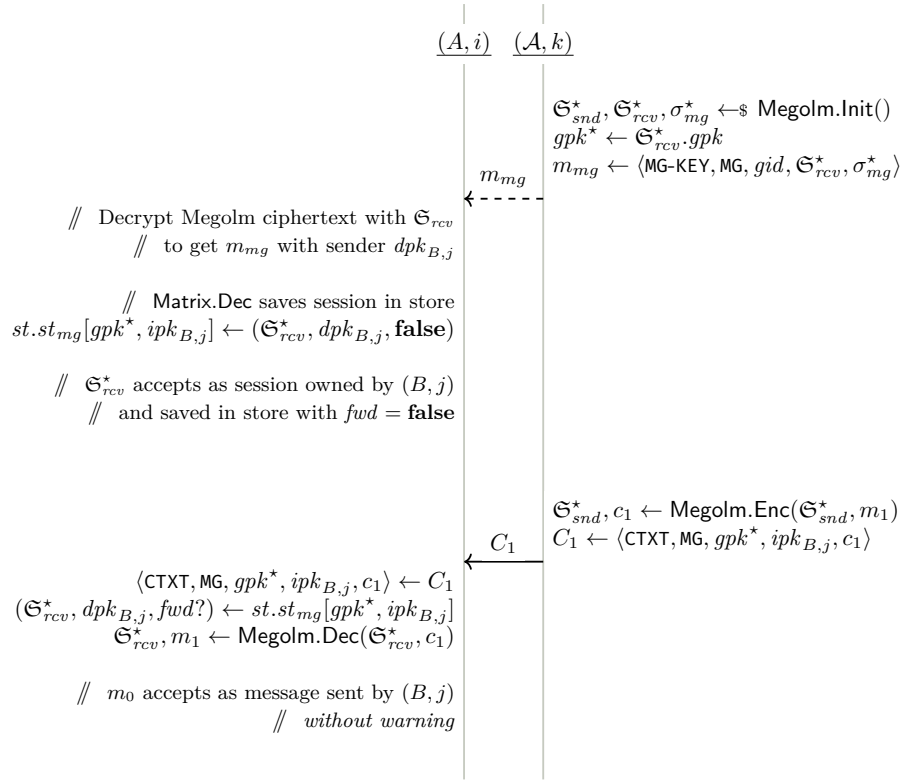


Figure 3.25. The impersonation attack described Section 3.3.4. This diagram continues on from the sequence diagram in Figure 3.22, which itself describes the attack in Section 3.3.3.

2) Construct a Megolm distribution message to share the inbound session $\mathfrak{S}^\star_{rcv}$ and signature $\sigma^\star_{mg}$.

3) Encrypt this message using the previously constructed outbound Megolm session, $\mathfrak{S}_{snd}$, from Section 3.3.3.

4) Construct a to-device message container (as in Figure 3.26b) to distribute the ciphertext from the previous step to Alice's device.

We illustrate this attack in Figure 3.25. When Alice's device receives the second Megolm session $\mathfrak{S}^\star_{rcv}$, the new session will be saved as if it were sent via an Olm channel with Bob's sender identity.

The adversary now has control over $gsk^\star$ from an outbound Megolm session that Alice's device believes is owned by Bob's device. In contrast with the attack in Section 3.3.3, there is no evidence presented to the user that this Megolm session was forwarded.

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

```
{
  "type": "m.room_key",
  "room_id": room_id,
  "content": {
    "algorithm": "m.megolm.v1.aes-sha2",
    "room_id": room_id,
    "session_id": gpk_prm,
    "session_key":
      ver || i_prm || R_prm || gpk_prm ||
      sign(gsk_prm, ver || i_prm || R_prm || gpk_prm),
    "chain_index": i_prm
  }
}
```

(a). Plaintext message.

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

(b). To-device container.

Figure 3.26. This message allows an adversary to impersonate Bob to Alice. It works by placing a Megolm session into Alice's device using an existing Megolm session that is already associated with Bobs sender identity. The plaintext in (a) is encrypted using the Megolm session from the previous attack, $\mathfrak{S}_{rcv}$. The resulting ciphertext is placed in a to-device message container, which can be seen in (b). This allows the second Megolm session to be sent using an `m.room_key` message, without being marked as a forwarded session.

Remediation

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

3.3.5 Confidentiality Break in Group Messaging

We now describe an attack that relies on the same combination of vulnerabilities as the authentication break just described. The attack enables a malicious homeserver to set the secret key used by target devices when encrypting backups of inbound Megolm sessions. This enables the homeserver, which stores these encrypted backups, to decrypt them and, in turn, decrypt the ciphertexts the session provides access to. Thus, a malicious server may compromise the confidentiality of messages in group messaging sessions.

As we describe in Section 3.2.5, when a new device completes self-verification, they request a copy of the user's Megolm backup key from the device that verified them. In our attack, the homeserver impersonates the verifying device (using the Olm/Megolm protocol confusion described in the previous sections) and responds with an attacker-controlled backup key. From this point onwards, the target device will backup the inbound Megolm sessions to the homeserver, encrypted with a homeserver-controlled key. Thus, this attack extends the authentication attack of Section 3.3.3 to additionally break confidentiality.

The SSSS secret sharing protocol uses Olm channels to distribute user-level secrets between devices. In particular, it is expected that `m.secret.send` messages are encrypted with the Olm protocol; however, as in Section 3.3.4, it

is possible to send a to-device `m.secret.send` message over a Megolm instead. When the receiving client decrypts the message, it will inherit the Olm identity key *ipk* associated with the Megolm session.[23]

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

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

1) The homeserver generates a symmetric key $k_{\mathcal{A}} \leftarrow\$ \{0,1\}^{256}$. It constructs a backup configuration signifying[24] the use of symmetric server-side key backups[25] with key $k_{\mathcal{A}}$. The homeserver will present this backup configuration to Alice's second device, $did_{A,2}$.

2) At this point in time, Alice's second device will not trust the key and will not enable backups.

3) Alice completes out-of-band verification between her two devices, $did_{A,1}$ and $did_{A,2}$.

4) Alice's second device sends an `m.secret.request` message to her first, requesting the key for server-side Megolm backups. This is triggered automatically after a successful self-verification.

5) The homeserver does not distribute this request message to Alice's first device.

6) The homeserver and colluding device, $did_{\mathcal{A},1}$, perform the attack in Section 3.3.3 against Alice's second device, $did_{A,2}$, giving them control over a Megolm session that it believes originally came from her verifying device, $did_{A,1}$.

7) The homeserver uses this Megolm session to respond to the request from Alice's second device (from step 4), sending an `m.secret.send` message containing the attacker-controlled $k_{\mathcal{A}}$.

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

[24] The details of server-side Megolm backup configuration, known as `auth_data`, are described in Section 10.12.3.(*Server-side key backups*) 2 of the Matrix Client-Server specification [Mata].

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

8) Alice's second device receives and decrypts the message, storing $k_{\mathcal{A}}$ as the secret key for server-side Megolm backups.

From this point onwards, Alice's second device will now trust the homeserver's backup configuration, since its private Megolm backup key matches the key identifier in the configuration. Alice's second device proceeds to enable server-side Megolm backups, uploading inbound Megolm sessions to the homeserver (encrypted using the homeserver-controlled key $k_{\mathcal{A}}$).

Remediation

Since the attack described in this section relies on the same underlying vulnerability as that of Section 3.3.4, the same remediation applies. That is, at the time of public disclosure, clients will ensure that encrypted to-device messages use the Olm protocol only.

3.3.6 IND-CCA Insecure Backup Scheme

In this section, we demonstrate that the encryption scheme used in Matrix' symmetric *Megolm key backups* and *Secure Secret Storage and Sharing* protocols does not provide IND-CCA security.

Vulnerability

Matrix uses an encrypt-then-MAC encryption scheme, formed as a composition of AES-CTR and HMAC-SHA256. However, the initialisation vector used for encryption is not included as input to the authentication tag.[26]

Attack

Consider an adversary against the IND-CCA security experiment (see Definition 2.16), when instantiated with the aforementioned composition of AES-CTR and HMAC-SHA256. The attack works as follows.

1) The adversary requests a challenge with two 128 bit messages, m_0 and m_1, and denotes the response as c_0:

$$c_0 \coloneqq iv_0 \parallel \mathsf{AES}(k_0, iv_0) \oplus m_b \parallel \mathsf{HMAC}\big(k_1, \mathsf{AES}(k_0, iv_0) \oplus m_b\big).$$

2) The adversary requests an encryption of the all-zero string from the encryption oracle and receives

$$c_1 \coloneqq iv_1 \parallel \mathsf{AES}(k_0, iv_1) \oplus 0 \parallel \mathsf{HMAC}\big(k_1, \mathsf{AES}(k_0, iv_1) \oplus 0\big)$$

for some initialisation vector iv_1.

[26] A similar issue exists for attachments which are shared out-of-band in encrypted form [Mata, Sending encrypted attachments]. Here the hash shared over Megolm (which takes the role of the authentication tag) does not include the initialisation vector. Since the initialisation vector itself is also shared over Megolm and thus implicitly authenticated, we did not see a way to exploit this behaviour.

3) The adversary then constructs a new ciphertext

$$c_2 \coloneqq iv_0 \parallel \mathsf{AES}(k_0, iv_1) \oplus 0 \parallel \mathsf{HMAC}\big(k_1, \mathsf{AES}(k_0, iv_1) \oplus 0\big)$$

which it passes to the decryption oracle.

The authentication tag of our constructed ciphertext, c_2, will verify correctly since, without the initialisation vector, c_1 and c_2 match. Thus, the call to the decryption oracle in step 3 will succeed and the adversary will receive a plaintext, m, with contents

$$m_2 \coloneqq \mathsf{AES}(k_0, iv_1) \oplus \mathsf{AES}(k_0, iv_0) \oplus 0.$$

The adversary may now use their knowledge of $\mathsf{AES}(k_0, iv_1)$ from step 2 to recover $\mathsf{AES}(k_0, iv_0)$ from m_2, with $\mathsf{AES}(k_0, iv_0) \leftarrow m_2 \oplus \mathsf{AES}(k_0, iv_1)$. In turn, the adversary may use their knowledge of $\mathsf{AES}(k_0, iv_0)$ to recover m_b from c_0.

Limitations

To perform such an attack, the adversary needs access to encryption and decryption oracles. An adversary with a colluding server may construct an encryption oracle using the Key Request protocol by sharing Megolm sessions with their target and observing the resulting encrypted backup. Similarly, the adversary may construct a decryption oracle by placing candidate ciphertexts on the server, and requesting the relevant session from our target over the Key Request protocol.

These oracles are sufficiently limited, however, that such an attack is not practical. This is thanks, primarily, to Matrix' use of key separation in the scheme it uses to encrypt Megolm backups. In particular, Matrix clients derive a per-session symmetric key for each Megolm session they backup. As such, the aforementioned decryption oracle is limited in scope to ciphertexts that encrypt the same session the adversary already has access to. Additionally, the resulting plaintexts must be parsed, processed and validated before they can be shared with the adversary (through the Key Request protocol). It is likely that modified ciphertexts will be invalid JSON and, thus, sharing will fail part way through this process.

Remediation

The Matrix developers plan to include the initialisation vector alongside the ciphertext when calculating the authentication tag. Similarly, clients will include the initialisation vector of encrypted attachments inside Megolm ciphertexts alongside their SHA-256 and encryption key. These fixes were not yet deployed at the time of disclosure.

3.4 Discussion

In this chapter we have studied the secure group messaging features of Matrix. We have given a high-level overview of the protocol and its architecture, alongside a formal description of the components that work together to provide

multi-device group messaging. We have done so with reference to the varying specifications, documentation, and implementations. As part of this process we discovered a number of vulnerabilities, which we develop into practical attacks against real-world clients.

The attacks we present together invalidate the authentication and confidentiality of the messages exchanged between users. In particular, the version of Matrix we studied in this chapter, as implemented in the reference libraries and flagship client Element, provided neither confidentiality nor authentication against malicious homeservers.

Disclosure. We disclosed our attacks to the Matrix developers over the period of 20$^{\text{th}}$ May 2022 to 6$^{\text{th}}$ July 2022. They acknowledge these as vulnerabilities except for one of our attacks on confidentiality (discussed in Section 3.3.1) which they initially considered as an accepted risk (but aimed to mitigate regardless).[27] We coordinated a public vulnerability disclosure for the 28 September 2022, to coincide with the first set of countermeasures. These aimed to provide immediate fixes (to varying degrees) for the attacks in Sections 3.3.1 to 3.3.5. At the time of public disclosure, the Matrix specification and Element were no longer vulnerable to the attack against out-of-band verification (in Section 3.3.2)[28], the partial authentication break (in Section 3.3.3)[29], the authentication break (in Section 3.3.4)[30] and the confidentiality break (from Section 3.3.5).[31] A second set of countermeasures is currently in the design phase, which aim to provide complete fixes for every vulnerability in this work.

In particular, the attack concerning homeserver control of room membership (from Section 3.3.1) was not fixed by the time of public disclosure. In the long-term, the Matrix developers plan to develop fixes for these attacks. They intend to distribute a fix for the IND-CCA break (from Section 3.3.6) at a later date. Since the IND-CCA break appears not to be practically exploitable, this should not affect users.

We have listed the intended remediations alongside each attack as they have been communicated to us by the Matrix developers. Note, however, that we have not checked nor have we verified these claims.

Analysis. On the one hand, some of our attacks highlight a rich attack surface by chaining different vulnerabilities across multiple protocols in order to achieve their goals. In particular, we compose (a) the partial authentication break in

[27] Post public disclosure, the co-founder and project lead for Matrix and the CEO and CTO at Element, went on record stating: "On the other hand, many in the cryptography community consider this a serious misdesign. Eitherway, it's avoidable behaviour and we're ramping up work now to address it by signing room memberships so the clients control membership rather than the server." [L19]

[28] The Matrix developers assigned CVE-2022-39250 [L40] to this vulnerability.

[29] The Matrix developers assigned CVE-2022-39246 [L37], CVE-2022-39249 [L39] and CVE-2022-39257 [L45] to this vulnerability. In their review of the ecosystem they also discovered further clients vulnerable to variants of our attack and assigned CVE-2022-39252 [L42], CVE-2022-39254 [L43] and CVE-2022-39264 [L46].

[30] The Matrix developers assigned CVE-2022-39248 [L38], CVE-2022-39251 [L41] and CVE-2022-39255 [L44] to this vulnerability.

[31] This vulnerability is covered by the CVEs for the previous item.

Section 3.3.3 which exploits missing verifications, (b) a stronger authentication break in Section 3.3.4 which exploits a protocol confusion aided by the design choice to check cryptographic properties at display rather than receiving time, and (c) a MITM attack that breaks confidentiality by convincing a target to use an adversary controlled key as backup in Section 3.3.5.

On the other hand, our attacks are well distributed across the different parts of the Matrix standard and implementation. In particular, we show (a) that Matrix offers no confidentiality guarantees against a malicious homeserver due to a design flaw allowing the homeserver to trivially add new users and devices to a room in Section 3.3.1, (b) that an identifier confusion in a separate protocol allows to break authentication and thus confidentiality even for the lowest level Olm channels in Section 3.3.2, and (c) that the key backup scheme in yet another subsystem does not achieve formal IND-CCA security in Section 3.3.6.

Besides the observed implementation and specification errors, these vulnerabilities highlight a lack of a unified and formal approach to security guarantees in Matrix. Rather, the specification and implementations seem to have grown organically with new sub-protocols adding new functionalities and thus inadvertently subverting the security guarantees of the core protocol. This suggests that, besides fixing the specific vulnerabilities reported here, Matrix will need to receive a formal security analysis to establish confidence in the design. Further, it highlights a need for a formalism that is able to capture such interactions and the interplay between the various sub-protocols that implement them. The development of such a formalism will be the focus of Chapter 5, and the security analysis that utilises it in Chapter 6.

Temporal compromise. Our attacks focus on authentication and confidentiality, i.e. the two most fundamental security properties provided by cryptography, without consideration of client secret compromise. The designs of Olm and Megolm indicate an interest in providing stronger notions of security, such as forward secrecy (i.e. before a client secret compromise), post-compromise security (i.e. eventually after a client secret compromise) and deniability. We choose not to investigate attacks into properties of this nature. This is due, in part, to a somewhat inconsistent approach to providing these properties across the various sub-protocols.

Take, forward secrecy, for example. While both Megolm and Olm make use of a symmetric ratchet, Matrix clients save the original copy of each inbound Megolm sessions that is shared with them. This is then shared with other devices of that user through the history sharing feature (namely, the Key Request protocol and server-side Megolm backups). Similarly, while Olm utilises an asymmetric ratchet (through its use of the Double Ratchet) and Megolm specifies that applications should regularly rotate their sessions, the allowance of multiple active Olm and Megolm sessions increases the recovery time substantially.

Group membership. Indeed, it seems as if these mechanisms primarily serve a somewhat different purpose: the cryptographic enforcement of membership changes. That is, when a device is added to a group, existing members may prefer for the message history to remain secure from the new device. This

is provided by Matrix through the symmetric ratchets in Olm and Megolm. Similarly, when a device is removed from a group, remaining members would expect that new messages are protected from the removed member. This is implemented in Matrix by having the existing members generate a new session.

However, as previously discussed, the server is given complete control over the list of users in a group. This lack of cryptographic control over group membership presents a number of issues, on top of the attack detailed in Section 3.3.1.

Group messaging in Matrix is designed in such a way that there is a disconnect between groups, or rooms, as they are defined by the specification and the cryptography used to implement secure communication within them. Each unidirectional Megolm channel, maintained by a particular client, is nominally separate from the others. Indeed, it is not clear how Megolm could support many of the consistency guarantees discussed in Section 2.4 (e.g. participant and transcript consistency). For example, the protocol provides no guarantee that clients have a shared view of the group membership.

Varying threat models. Our study, and the vulnerabilities we describe, assume a setting where Matrix aims to provide the strongest guarantees, i.e. where every device and user have performed out-of-band verification. Matrix, however, targets a range of threat models. For example, users are able to disable end-to-end encryption in rooms if they prefer.

Further, in the idealised scenario we study, the client interface will display a warning next to messages that are unencrypted, or cannot be cryptographically linked to the claimed sender. From the perspective of an attacker, this is the most challenging and thus interesting setting. However, when a user has not been verified, the Element client will no longer display such warnings. The Matrix specification does not enforce that messages in encrypted rooms are indeed encrypted. This renders impersonation attacks trivial: the attacker, in collusion with the homeserver, simply sends an unencrypted message with a forged sender.

As such, even when the issues described in this work are fixed, clients operating in this non-ideal setting do not offer any cryptographic authentication guarantees. In other words, outside of these idealised conditions “all bets are off” and e.g. impersonation becomes trivial. While Element already supports the option of refusing to send messages to unverified devices it does not reject messages from such devices. Thus, unless the client-side option is provided to reject all communication from unverified devices or rooms with such devices within them, Matrix will not provide a secure chat environment regardless of cryptographic guarantees provided for verified devices.

CHAPTER 4

Case Study: WhatsApp

Our second case study investigates the secure group messaging features provided by WhatsApp. We start with a high-level overview of the application and its architecture. We then isolate the parts that are relevant to the topic at hand, multi-device group messaging. We provide a detailed description that combines the public documentation and the behaviour of the implementation, which we obtain by reverse-engineering the WhatsApp web client. We finish with a discussion of WhatsApp's design and implementation.

4.1 Introduction

WhatsApp provides end-to-end encrypted messaging to over two billion users. However, due to a lack of public documentation and source code, the specific security guarantees it provides are unclear. Seeking to rectify this situation, we combine the limited public documentation with information we gather through *reverse-engineering* its implementation to provide a formal description of the subset of WhatsApp that provides *multi-device group messaging*.

Our focus. WhatsApp uses a variety of cryptographic protocols to provide secure messaging to their users. These mechanisms provide direct messaging, group messaging, state sharing, backup and recovery, to name but a few.

In this work, we focus on the protocols with which WhatsApp realises multi-device secure group messaging. That is, we study the protocols used for group messaging, device management and history sharing.

We consider the WhatsApp server to be adversarial (in this context) and our description is framed with this in mind. As such, we do not consider any separation between the insecure network and the server. We give no consideration to the security of channels between the client and server, for example.

4.1.1 Related Work

Shared heritage. End-to-end encryption in WhatsApp is based on the Signal two-party protocol and, for groups, the Sender Keys multiparty extension [Wha23a, BCG23]. Thus, existing analysis of these protocols will be relevant. The Signal two-party protocol has seen varying analyses of its initial key exchange [FJ24], the Double Ratchet algorithm it uses [ACD19, BFG+22] as well as their composition [CCD+20, FMB+16].

Group messaging. Sender Keys is less well-studied, however. In 2017, a number of weaknesses in deployed group messaging protocols, including WhatsApp, were identified [RMS17]. More recently, a formalism and security analysis of the Sender Keys protocol, targeting WhatsApp specifically, was performed in [BCG23].

In the academic literature to date, the ground truth for answering the question of how these building blocks are composed precisely is established by the WhatsApp security whitepaper [Wha23a] or unofficial third-party protocol implementations, cf. [RMS17, BCG22, BCG23].

Direct analysis. WhatsApp's key transparency implementation [Wha23b] builds upon [L13] the design of PARAKEET [MKKS+23]. WhatsApp's application state synchronisation feature utilises [Wha23b] the LtHash homomorphic hashing algorithm [LKMW19, BM97]. Recently, the security of WhatsApp's end-to-end encrypted backups have been analysed [DFG+23].

4.1.2 Contributions

This chapter includes the following key contribution.

Contribution 4.1. We reverse-engineer the WhatsApp web client to verify the description reported in the WhatsApp security whitepaper [Wha23a].[1] This culminates in a formal description of multi-device group messaging in WhatsApp that covers group messaging, pairwise channels, session management, device management and history sharing.

Both Signal's original description of the Sender Keys protocol [Mar14] and WhatsApp's security whitepaper [Wha23a] implies that the number of underlying two-party channels between any two parties is one. This is not the case, however. The `libsignal` library allows multiple active Signal channels between a single pair of devices [Mar17, CJN23, CFKN20]. We confirm, in this work, that this translates into WhatsApp's implementation. Further still, all of these channels may be used to distribute Sender Keys sessions.[2] As explored in [CJN23, CFKN20], adversaries with the ability to initialise new pairwise sessions can undermine the security guarantees of an existing channel post-compromise. As discussed in [ADJ24], this has additional and compounding effects on the PCS guarantees of the Sender Keys protocol.

4.1.3 Scope & Limitations

This description is based upon a manual inspection of the client's implementation, a process which is necessarily imperfect without access to its source code. Our work is based primarily on the WhatsApp web client, archived on 3rd May 2023, and version 6 of the WhatsApp security whitepaper [Wha23a].

Remark 4.1. *We sent our work to WhatsApp for comments and received feedback, including that our description of the protocol is correct, by WhatsApp engineers.*

We sometimes simplify WhatsApp's functionality. We aim to document all such simplifications as they arise in the text. Additionally, our description is not complete: we focus on functionality that is most relevant to multi-device group messaging (and the security analysis in later chapters).

In particular, our description does not cover distribution of user identities. We refer to WhatsApp's documentation on out-of-band verification [Wha23a, Page 24] and key transparency [Wha23b] whitepaper for WhatsApp's claimed functionality in this area.

4.2 Multi-Device Group Messaging in WhatsApp

In this section, we present a description of WhatsApp's multi-device group messaging functionality, combining the public security whitepaper with our own

[1] We also performed some spot checks using decompiled source code of the WhatsApp Android application.

[2] This is also the case in Matrix [ACDJ23, ADJ24].

reverse-engineering of WhatsApp's implementation. In doing so, we identify and define the sub-protocols that, together, provide multi-device group messaging in WhatsApp. The description in this section is an abstracted description of the functionality it aims to describe. For example, we may simplify data structures, algorithms or functionality.

4.2.1 Device Setup and Management

We start by describing the facilities that WhatsApp provides for users to setup and manage their devices. In WhatsApp, each user has a primary device (typically a phone) and a number of companion devices (e.g., a laptop and a tablet). The primary device creates and maintains the links between itself and its companion devices. This is done through a variety of structures that, together, we call a user's *multi-device state* when describing WhatsApp. It consists of a signed list of devices (containing both the primary device and each of their companion devices), as well as a *device signature* and *account signature* for each companion device. This structure is public and, modulo any cryptographic controls, distributed and controlled by the server. Thus, the cryptographic identity of the primary device forms the user's root of trust and represents them, *cryptographically speaking*. Changing the primary device requires resetting the user's cryptographic identity.

We collect this functionality into a sub-protocol, $\mathsf{MD} = (\mathsf{Setup}, \mathsf{Link}, \mathsf{Unlink}, \mathsf{Refresh}, \mathsf{Devices})$. We briefly describe these algorithms and how they work together.

- $isk, ipk, md \leftarrow\!\!\$\ \mathsf{MD.Setup}()$ generates a fresh cryptographic identity for a user's identity device, (isk, ipk), and initialises public state, md.

- $md \leftarrow \mathsf{MD.Link}(\rho, isk, md, ipk^{\star})$ takes as input the executing party's role, ρ, private identity key, isk, the current public device state, md, and the public identity key of the other party, $ipk^{\star}$, before outputting an updated multi-device state, md. It describes the linking process between a primary and companion device. It is first executed by the primary device, in the '$\rho = \texttt{primary}$' case, which outputs an updated multi-device state. This state is then passed to the companion device, which executes the algorithm in the '$\rho = \texttt{companion}$' case, before outputting an updated multi-device state.

- $md \leftarrow \mathsf{MD.Unlink}(\rho, isk, md, ipk^{\star})$ takes as input the executing party's role, ρ, private identity key, isk, public multi-device state, md, and the public identity key of the companion being removed, $ipk^{\star}$, before outputting the updated multi-device state.

- $ipks^{\checkmark} \leftarrow \mathsf{MD.Devices}(ipk_p, \gamma, md)$ takes as input the identity key of the primary device, ρ, the minimum device list generation to accept, γ, and the public multi-device state, md, before outputting the set of identity keys representing the verified devices of the given primary device, $ipks^{\checkmark}$ (for the given minimum device list generation and public device state).

- $md \leftarrow \mathsf{MD.Refresh}(\rho, isk, md)$ takes as input the role of the executing party, ρ, the private identity key of the primary device, isk, and the

public multi-device state before generating a new public multi-device state (without changing the device composition) output as md.

We now proceed to describe each algorithm in the sections that follow (refer to Figure 4.1 for a formal description).

Primary device setup. When a user first sets up their account, they do so by setting up their primary device (as we describe in MD.Setup in Figure 4.1). The primary device's cryptographic identity consists of a Curve25519 key pair, the *identity keys* ipk_p and isk_p, generated at device setup (see line 2). This key pair is used for a variety of purposes across WhatsApp, such as managing a user's devices (as we describe here) and initialising pairwise channels (as described in Section 4.2.2). The public key, ipk_p, is uploaded to the server to register a server-controlled association between the logged-in user and their primary device. The device proceeds to generate an initial version of the user's multi-device state , the contents of which we describe below. The multi-device state is signed with the private identity key, isk_p, before being uploaded to the server (see line 3). While WhatsApp does take steps to secure the mapping from a user to their primary identity key, this is outside the scope of our analysis[3] and, as such, is left out of our description.

Registering companion devices. To link a new companion device with their account, a registration sub-protocol is executed between the companion and primary devices. This is initiated in the user interface, where the companion device presents a QR code that is scanned with the primary device. This code communicates the companion device's identity key as well as a linking secret. The linking secret helps realise a SAS-style protocol [Vau05] between the two devices, ensuring the authenticity of later messages in the registration sub-protocol that are routed through the server. We describe this process in Section 4.2.5.

The companion device starts by generating their own identity keys, isk_c and ipk_c (see line 1 of WA.NewCompanionDevice in Figure 4.13). The companion and primary device are linked together through two signatures, the account signature and device signature. The primary device additionally maintains a signed list of valid devices, known as the device list. The primary device allocates each device a local identifier that indexes them within the device list. The primary device is given an index of 0, the first companion device an index of 1, and so on. The device list is constructed with the following fields: (a) non-cryptographic user identifier, (b) timestamp, (c) the current maximum valid index, and (d) a list of the valid device indices within the range $\{0, 1, \ldots, max\}$.[4] Devices with indices that are within this range but missing from the list of valid device indices have been revoked. Our description replaces user identifiers, device identifiers and device indices with the appropriate public

[3] The security experiment in Section 5.2.2 allows the challenger to provide a trusted mapping between user identities and their primary identity key ipk_p without interference from the adversary.

[4] WhatsApp supports accounts for which end-to-end encryption support is "hosted". This is for businesses managing customer facing accounts. The device list and linking metadata also includes flags indicating this, however this is not something that we consider.

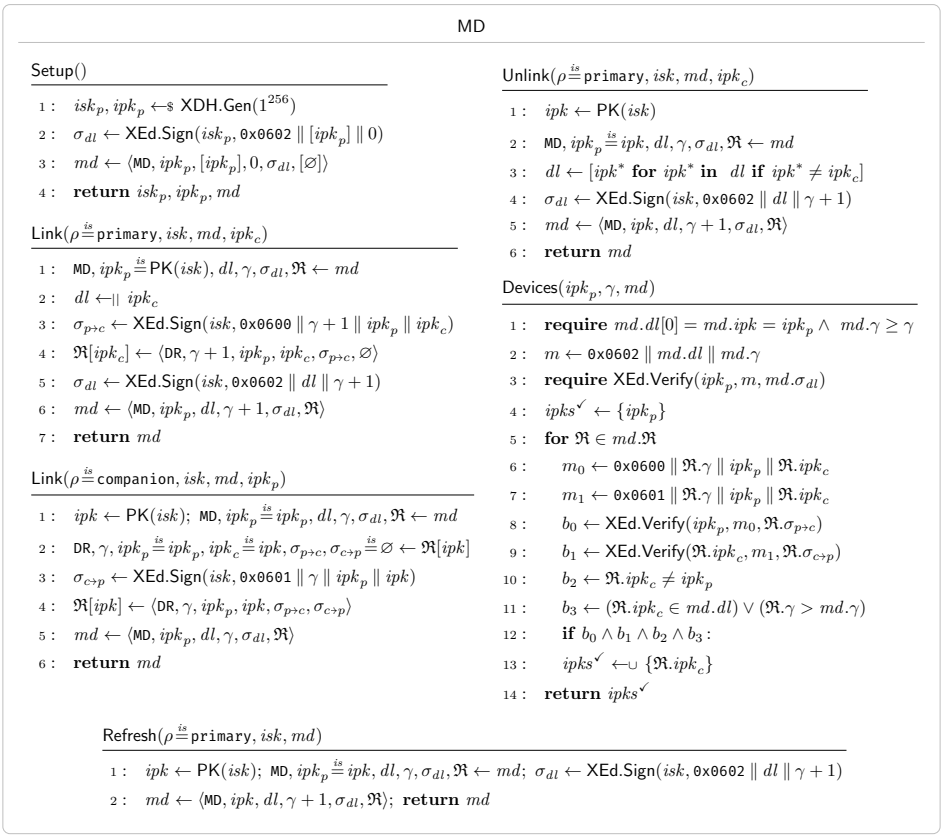


Figure 4.1 . Pseudocode describing how WhatsApp manages user devices; the multi-device sub-protocol MD.

key, under the assumption that WhatsApp maintains these mappings correctly. We replace timestamps with a counter: the device list *generation*. As such, our description captures the device list as containing the primary device's identity key, a list of valid companion device identity keys, the current generation and its signature (see lines 1 and 5 of MD.Link($\rho \overset{is}{=}$ primary, ...) in Figure 4.1).

When a new companion device is registered, they are allocated the index $max+1$ and an updated device list is generated and signed (see lines 1 and 2 of MD.Link($\rho \overset{is}{=}$ primary, ...)). The primary device creates and signs an *account signature*: a XEd25519 signature computed over the 0x0600 prefix, the "linking metadata" and the companion device's identity key (line 3). Similarly, the companion device creates and signs the *device signature*, an XEd25519 signature computed over the 0x0601 prefix, the linking metadata and primary device's identity key (see lines 2 and 3 of MD.Link($\rho \overset{is}{=}$ companion, ...)). This mutual signing ensures that companion devices cannot be forcibly adopted by a malicious primary device (and vice versa). The *linking metadata* is a structure consisting of the device's non-cryptographic identifier, an index into the device list for this device, and a timestamp. As above, we replace the linking metadata with the companion device's identity key and the current device list generation.

The account and device signature are combined with other, non-cryptographic metadata, to form its *device record*.

Revoking companion devices. To revoke a companion device, the primary device generates a new device list without the target companion device (see MD.Unlink in Figure 4.1). In order for other devices to enact this revocation, it is important that they are made aware of this change.

Refreshing the device list. Device lists expire after 35 days to ensure any device revocations are enacted even in the case where the server is blocking distribution of the updated device list. To ensure there is always a valid list, the primary device will periodically update the signed device list (see MD.Refresh in Figure 4.1). If all device lists have expired, clients will only communicate with the primary device for that account. This design ensures that a device removal will be enforced within a maximum of 35 days, allowing eventual recovery from compromised *companion devices*.[5]

Verifying devices. WhatsApp clients keep track of the list of verified devices for each user they communicate with (including themselves). Under normal operation, the server will distribute changes to a user's multi-device configuration, such as the updated device list and the accompanying linking metadata, to all of their own devices and the devices they communicate with. Changes to device lists are primarily synchronised through notifications pushed by the server, but may also be requested by clients when they detect that their list is out-of-date. This is triggered in a variety of scenarios, such as when the client's local copy of the device list has expired, they receive a message from a device that is not in their copy of the device list,[6] or they receive in-chat device consistency (ICDC) information that does not match their local state (see below). Clients then rely on this list when authenticating messages, or determining who they should send messages to.

We now briefly describe the two mechanisms through which clients receive updates to a user's multi-device state.

1) A copy of the device list, accompanied by all relevant device records for that generation. Ostensibly, these device list structures have been generated and uploaded by the relevant user's primary device. The server is expected to send a notification to clients when a new copy of a communicating partner's device list becomes available (with the structure embedded within the notification). Alternatively, this may be received as the result of an explicit synchronisation request by the client. When receiving a new copy of the device list, clients must decide how to merge it with the existing information they hold.

[5] Looking ahead, our model does not capture this automatic expiry of device lists. Whilst our model is accurate within the 35 day window before a device list expires, WhatsApp is stronger than what we capture in the model in this respect. We consider this as a reasonable trade-off, in comparison to the alternative of including global time in our model.

[6] Whenever a device initiates a new pairwise Signal channel (see Section 4.2.2) through a pre-key message, they attach their device record. Receiving a device record referencing a newer device list than you have, or a device not currently present, can trigger the synchronisation process.

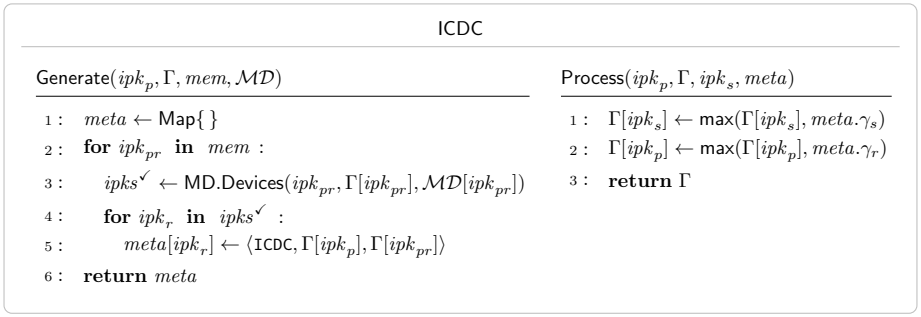


Figure 4.2. Pseudocode describing how WhatsApp's ICDC information is computed.

2) A copy of a single device record that accompanies pre-key ciphertexts form a Signal pairwise channel. As mentioned above, this is also known as a *device identity package*. These are sent any time a device initialises a new pairwise channel with another device, so that the recipient may verify the sender.[7] In this case, clients may accept device records that are from future generations.

In both of these cases, the server has ultimate control over which device list and device records the receiving client has access to during decryption. Thus, we model the verification process as taking as input an adversarially controlled *md* (consisting of the device list, its generation, signature and a list of device records) and outputting a list of verified device identity keys for the given user. It is up to this algorithm to determine which devices are valid for the given user (identified by their primary device's identity key) and for the given generation (since companion devices may be revoked or expire).

The algorithm MD.Devices in Figure 4.1 details the process. A device is considered verified if there exists verifying account and device signatures (see lines 6 to 9) with an included timestamp that is greater than or equal to that of the latest device list the verifying device has seen for that user (see lines 1 and 11). Additionally, the device list signature must verify (see lines 2 to 3) and, if the two timestamps are equal, the device must be present in this version of the device list (see line 11). Note that, since the primary device's identity key also identifies the user, verification of primary devices is trivial insofar as we are able to map a device's cryptographic identity to a user's cryptographic identity (see lines 4 and 10).[8]

In-Chat Device Consistency. To help communicating partners detect changes to their multi-device state, clients include ICDC information whenever they send pairwise Signal messages. Note that such information is *not* included in group

[7] Note that the whitepaper does not mention the inclusion of device records alongside pre-key messages.

[8] Verifying that the user's cryptographic identity maps to the person they expect to communicate with is not trivial. Section 4.2.4 discusses WhatsApp's approach to this problem: users verify each other's primary device identities either directly (through out-of-band verification) or through the key transparency functionality. This is outside the scope of our formal analysis in Section 6.2.7.

messaging messages (described in Section 4.2.3). This information is included within the ciphertext, helping to detect cases where a malicious server may withhold multi-device state updates from clients: if the server allows *application messages* to be sent, they allow clients to detect changes to multi-device state.

We describe this behaviour through two algorithms, ICDC = (Generate, Process).

- $meta \leftarrow \mathsf{ICDC.Generate}(ipk_p, \Gamma, mem, \mathcal{MD})$ takes as input the primary device of the executing party, ipk_p, the store of minimum device list generations, Γ, a list of group members, mem, and the executing party's public multi-device state md then prepares the per-recipient metadata, output as $meta$.

- $\Gamma \leftarrow \mathsf{ICDC.Process}(ipk_p, \Gamma, ipk_s, meta)$ takes as input the executing party's primary identity key, ipk_p, the store of minimum device list generations, Γ, the primary identity key of the sender, ipk_s, and the ICDC metadata accompanying a message. It calculates the new minimum device list generation for both the executing party and the sending party, before outputting an updated store, Γ.

WhatsApp's whitepaper describes the ICDC metadata as including: (a) The timestamp of the sender's most recent signed device list. (b) A flag indicating whether the sender has any companion devices linked. (c) The timestamp of the recipient's most recent signed device list. (d) A flag indicating whether the recipient has any companion devices linked. Differing slightly from the whitepaper, the WhatsApp web client we investigated included (a) the timestamp of the sender's most recent signed device list, (b) a list of the sender's current key indices, (c) the sender's key hash[9], (d) the timestamp of the recipient's most recent signed device list, (e) a list of the recipient's current key indices, and (f) the recipient's key hash.

The inclusion of ICDC information about *the recipient*[10] user allows the sender's other devices to check whether they have a consistent view of the recipient's multi-device state (since the sender's devices are also recipients of such messages, albeit not *the recipient* in this context).

Further, the code we investigated only makes use of the device list timestamp when processing and reacting to ICDC updates. Thus, our description and security analysis model the ICDC as containing the most recent device list generations of both the sender and recipient (see line 5 of ICDC.Generate in Figure 4.2). Upon receiving ICDC information in a message, the recipient will check that it is consistent with their view of the sender's multi-device state. If not, the whitepaper claims that clients will accelerate the device list expiration to either 48 hours (or keep the time that is already remaining, if it is less than 48 hours). We could not find evidence of such a delay in the client in our investigation. Instead, clients seem to immediately invalidate the relevant

[9] The key hash is a hash of all the primary and public identity keys associated with an account. The binary representation of each public key are concatenated together and piped through the SHA256 function.

[10] When a message is being sent to another user, but this ciphertext is distributing information to a companion device of the sender, then "recipient" refers to the primary device identity of the other user (not the sender) [Wha23a].

device records.[11] In our description and security analysis, we capture this by having the client set the minimum device list generation for the relevant user to that which is in the ICDC information, if it is greater than the client's current value (see ICDC.Process in Figure 4.2). Next time the list of verified devices is checked, they will use this new minimum device list generation (as input into MD.Devices).

4.2.2 Pairwise Channels

To secure pairwise channels, WhatsApp uses X3DH [Mar16b] for the initial key exchange and the Double Ratchet [Mar16a] from there onwards. The device identity key, (isk, ipk), is used as a contribution towards the X3DH key exchange in addition to signing the medium-term signed pre-key. Note that the signature over the pre-key is computed over its raw serialised public key, and without explicit domain separation (other than a preceding byte encoding that it is an XDH key). Unlike Matrix, WhatsApp does not sign the one-time keys used in X3DH.[12]

WhatsApp clients maintain multiple active pairwise channels between themselves and other devices. Whilst this is not documented by WhatsApp, our analysis confirms that its implementation largely matches the session management design of Signal as documented in the Sesame specification [Mar17] and recently analysed in the symbolic setting [CJN23]. Maintaining multiple underlying sessions can help with issues such as desynchronisation, e.g. if two devices initialise a session at the same time, but undermines PCS guarantees [CFKN20, CJN23].

We capture WhatsApp's pairwise device-to-device communications, including their use of multiple channels, with the scheme $\mathsf{DM} = (\mathsf{Init}, \mathsf{Enc}, \mathsf{Dec})$ in Figure 4.3. The external interface of such a scheme aims to allow pairs of devices to securely exchange messages, all the while managing a number of underlying Signal channels internally. It initialises Signal channels, selects the appropriate channel to use, and rotates medium- and short-term keys as needed. We briefly describe the interface of these algorithms.

- $pst, skb \leftarrow\$ \mathsf{DM.Init}(isk, n_e)$ takes as input a private identity key, isk, and the number of ephemeral keys to generate, n_e. It proceeds by generating a medium-term key pair, (ssk, spk), and n_e single-use keys, $\{esk_i, epk_i\}_{0 \leq i < n_e}$. It initialises its private state, pst, consisting of the private parts of these keys, which it outputs alongside the key bundle, skb, consisting of the public parts of the medium-term and single-use key pairs.

- $pst, c \leftarrow \mathsf{DM.Enc}(pst, skb_r, m)$ takes as input the private state, pst, the key bundle of the recipient, skb_r, and the message to be sent, m. It outputs an updated private state, pst, and a ciphertext, c.

[11] Since we choose not to model time in Section 6.2.7, this inconsistency does not affect our analysis.

[12] This contrasts with Matrix' Olm, which signs the one-time keys using the identity key. The Olm specification claims that this trades off reduced deniability for improved forward security [Mate, "Signing one-time keys"].

DM

Init(isk, n_e)

1 : // Generate and sign medium-term pre-key
2 : $spk, ssk \leftarrow\$$ Signal.MedTermKeyGen()
3 : $\sigma_{spk} \leftarrow$ XEd.Sign(isk, 0x05 $\|$ spk)
4 : // Generate ephemeral/one-time keys
5 : **for** $i = 0, 1, 2, \ldots, n_e - 1$:
6 : $epk_i, esk_i \leftarrow\$$ Signal.EphemKeyGen()
7 : $esks \leftarrow \{esk_i : 0 \leq i < n_e\}$
8 : $epks \leftarrow \{epk_i : 0 \leq i < n_e\}$
9 : // Key bundle for distribution by server
10 : $skb \leftarrow \langle$SKB, PK(isk), $spk, \sigma_{spk}, epks\rangle$
11 : // Initialise our private DM protocol state
12 : $ssts \leftarrow$ Map{} // Signal session states by ipk
13 : $pst \leftarrow \langle$DM, $isk, ssk, esks, ssts\rangle$
14 : **return** pst, skb

Dec(pst, skb_s, c_P)

1 : **if** $c_P.c_{kex}.type =$ pkmsg :
2 : $pst, m \leftarrow$ *DM.DecPreKeyMsg(pst, skb_s, c_P)
3 : **else** :
4 : $pst, m \leftarrow$ *DM.DecNormalMsg(pst, skb_s, c_P)
5 : **return** pst, m

Enc(pst, skb_r, m)

1 : // Unpack our protocol state and recipient's key bundle
2 : DM, $isk, ssk, esks, ssts \leftarrow pst$
3 : $ipk_r, spk_r, \sigma_{spk,r}, [epk_r] \leftarrow skb_r$
4 : // (expect server to allocate a single ephemeral key epk_r)
5 : // Initiate new Signal session if we have no pre-existing one
6 : **if** ($ssts[ipk_r] = \varnothing$) :
7 : **require** XEd.Verify(ipk_r, 0x05 $\|$ $spk_r, \sigma_{spk,r}$)
8 : $sst, \cdot \leftarrow\$$ Signal.Activate(isk, ssk, init, ipk_r)
9 : $sst, c_{kex} \leftarrow\$$ Signal.Run(
10 : $isk, ssk, sst, (spk_r, \sigma_{spk,r}, epk_r)$)
11 : **else** : // Otherwise use the active matching session
12 : $sst \leftarrow ssts[ipk_r][0]$
13 : $sst, c_{kex} \leftarrow\$$ Signal.Run($isk, ssk, sst, \varnothing$)
14 : **require** $sst.status[sst.stage] =$ accept
15 : $c_{msg} \leftarrow$ WA-AEAD.Enc($sst.k[sst.stage], c_{kex}, m$)
16 : $ssts[ipk_r] \leftarrow$ *DM.UpdateSession($ssts[ipk_r], \varnothing, sst$)
17 : $pst \leftarrow \langle$DM, $isk, ssk, esks, ssts\rangle$
18 : **return** $pst, (c_{kex}, c_{msg})$

Figure 4.3. Pseudocode describing how WhatsApp uses the Signal two-party protocol to build secure channels between pairs of devices, its direct messaging sub-protocol DM; see Figure 4.4 for descriptions of *DM.DecNormalMsg and *DM.DecPreKeyMsg.

- $pst, m \leftarrow$ DM.Dec(pst, skb_s, c_P) takes as input the private state, pst, the key bundle of the sender, skb_s, and a ciphertext, m. It outputs an updated private state, pst, and the decrypted message, m.

WhatsApp uses the (non-cryptographic) device identifier, alongside the initiator's one-time key, to store and locate Signal sessions. In contrast, our description addresses sessions using the device identity key in place of the device identifier. It follows that our analysis relies on WhatsApp correctly maintaining this mapping, something that is required by Sesame [Mar17] but not in [Wha23a].

DM.Init captures a device's initial setup. It takes the device's secret long-term identity key and the number of ephemeral keys to generate as input, then creates medium- and short-term key pairs before setting up the protocol state (see lines 1 to 6). It outputs the private protocol state pst (see lines 8 and 9) and a bundle of public keys for distribution by the server, skb, which includes the long-term identity key, signed pre-key with its signature, and a list of ephemeral pre-keys (see line 7). WhatsApp allows clients to upload fresh sets of ephemeral key pairs, allowing key rotation to continue indefinitely. Our modelling, and the pseudocode in Figure 4.3, does not capture this ability. We discuss this decision further in Section 6.2.5.

DM.Enc encrypts the given message, m, before returning a ciphertext, c_P, and updated protocol state. The device unpacks the provided recipient's key bundle

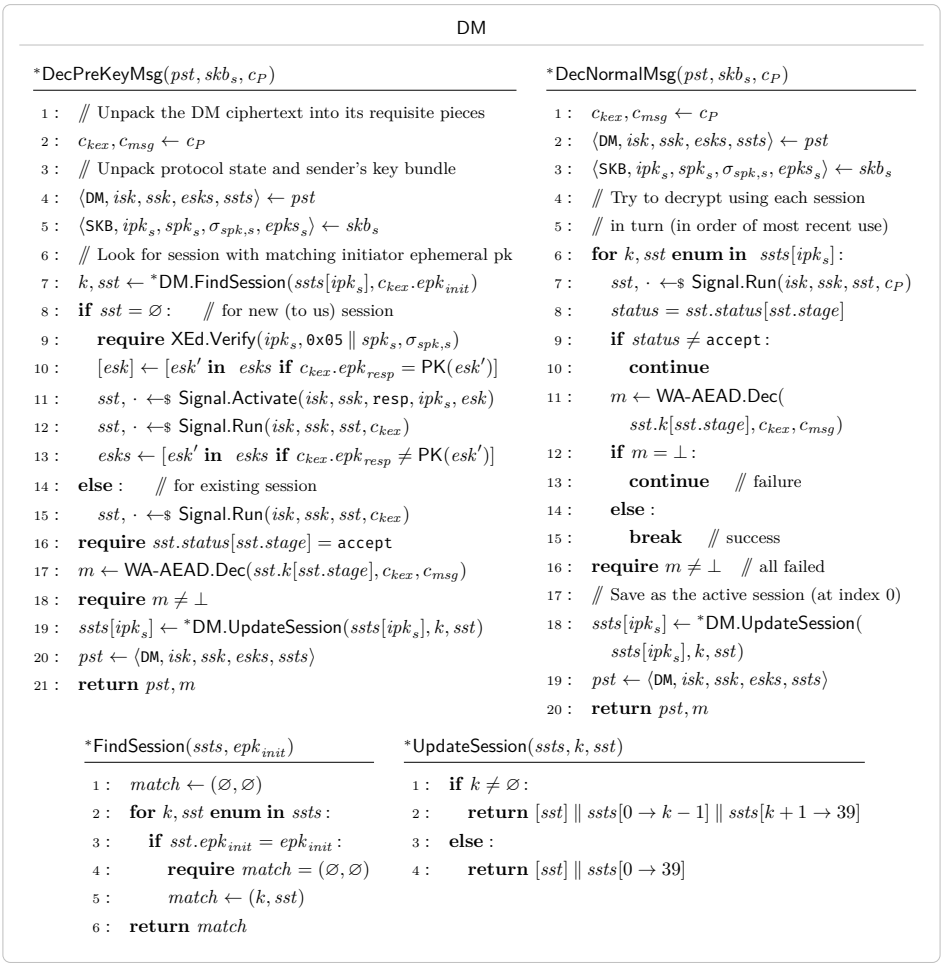


Figure 4.4. Helper functions completing our description of DM (cf. Figure 4.3).

and locates any existing sessions with the recipient. If no such session exists, the signature on the recipient's medium-term key is verified before initialising a new Signal session using the recipients identity key, medium-term key and one of their ephemeral keys (see lines 3 to 6). If one or more matching sessions do exist, they utilise the most recently used session. In both cases, they use the fresh key output by the underlying Signal protocol as input into an AEAD scheme before saving any changes to the Signal sessions state (lines 10 to 12). It is important that the encryption algorithm outputs both the encrypted message, c_{msg}, and the key exchange ciphertext, c_{kex}, which allows the underlying Signal channel to perform the continuous key exchange.

DM.Dec decrypts the given ciphertext , c_P, before returning an updated protocol state, pst, alongside the decrypted message m. If decryption fails due to invalid input or mismatched identities, it outputs an updated protocol state, pst, and an empty message, $\varnothing$. Decryption of pre-key and normal Signal ciphertexts are handled separately, in *DM.DecPreKeyMsg and *DM.DecNormalMsg respectively. In the case of a pre-key message, clients search for an existing session with

a matching initialising ephemeral key (see line 4 of *DM.DecPreKeyMsg). If such a session does not exist, the client initialises a new session after ensuring that the chosen responding ephemeral key is in fact one of their own (and has not previously been used). They proceed to execute the protocol, deriving the appropriate per-message key, using either the newly initialised session (see line 9) or the pre-existing session that was previously found (see lines 10 and 11). The key is used to authenticate and decrypt the application message contained in c_{msg}. In the case of normal messages, the client fetches the sessions for the claimed sending device and applies trial decryption[13] to each session in the order of most recent use (see lines 4 to 11 of *DM.DecNormalMsg). In all cases, the session store for the communicating device is updated with the new session state, storing up to 40 sessions ordered by most recent use (see *DM.UpdateSession).

Note that this work does not aim to analyse or verify WhatsApp's implementation of Signal pairwise channels; instead, we focus on how the session management layer impacts the security guarantees of the composed protocol. Thus, we describe and analyse DM's constituent algorithms without including pseudocode for the Signal scheme. In our analysis, we utilise a variant of the multi-stage key exchange (MSKE) model [FG14]. Originally introduced to aid in the analysis of Google's QUIC protocol [FG14], a variant of this model was developed in the analysis of Signal's pairwise channels [CCD+20]. We make use of this existing result in our work. We model the underlying Signal pairwise channels as a MSKE, with Signal = (KeyGen, MedTermKeyGen, Activate, Run), and assume that WhatsApp's implementation of Signal pairwise channels provides multi-stage key indistinguishability (MS-IND). We provide a brief description of the Signal variant of the MSKE model's syntax later in the thesis, see Definition 6.7 in Section 6.2.5.

Since WhatsApp uses the identity key for purposes outside of the Signal scheme, this formalisation cannot be fully separated from its use in WhatsApp. For this reason DM does not make use of Signal's initial key generation function Signal.KeyGen. Additional minor changes are required for our security analysis in Section 6.2.5 because our work targets message security rather than key security. Thus, we must include WhatsApp's AEAD scheme in our analysis, which we denote as WA-AEAD and detail in Figure 4.5.

Direct Messaging. Our modelling and security analysis does not cover the direct messaging component of WhatsApp. Briefly, WhatsApp uses the secure pairwise channels (described in Section 4.2.2) to exchange direct messages between users. This is even the case in the multi-device setting. Here, clients use the multi-device components (described in Section 4.2.1) to determine the list of devices representing their communicating partner, then distribute the same application message over multiple independent Signal two-party channels.

WA-AEAD

$\mathsf{Enc}(mk, meta, m)$

1 : $ek \leftarrow mk[0 \rightarrow 31\text{B}];\ hk \leftarrow mk[32 \rightarrow 63\text{B}]$
2 : $iv \leftarrow mk[64 \rightarrow 79\text{B}]$
3 : $c \leftarrow ipk_s \parallel ipk_r \parallel \mathsf{AES\text{-}CBC.Enc}(ek, iv, m)$
4 : $c_h \leftarrow \langle \texttt{SIG-HMAC}, meta, c \rangle$
5 : $\tau \leftarrow \mathsf{HMAC}(hk, c_h)[0 \rightarrow 7\text{B}]$
6 : **return** $c \parallel \tau$

$\mathsf{Dec}(mk, meta, c)$

1 : $ek \leftarrow mk[0 \rightarrow 31\text{B}];\ hk \leftarrow mk[32 \rightarrow 63\text{B}]$
2 : $iv \leftarrow mk[64 \rightarrow 79\text{B}]$
3 : $c \parallel \tau \leftarrow c;\ c_h \leftarrow \langle \texttt{SIG-HMAC}, meta, c \rangle$
4 : **require** $\tau = \mathsf{HMAC}(hk, c_h)[0 \rightarrow 7\text{B}]$
5 : $ipk_s \parallel ipk_r \parallel c \leftarrow c$
6 : $m \leftarrow \mathsf{AES\text{-}CBC.Dec}(ek, iv, c)$
7 : **return** m

Figure 4.5. Pseudocode describing the AEAD scheme used by WhatsApp when sending messages over pairwise channels, WA-AEAD.

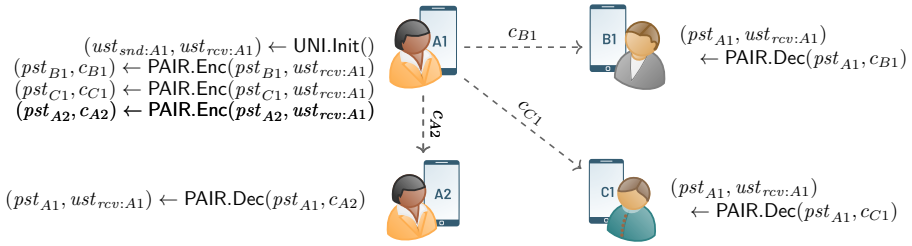


Figure 4.6. Example of one of Alice's devices initialising a new outbound Sender Keys stage and distributing the inbound session state to the group.

4.2.3 Group Messaging

Group messaging is achieved through the *Sender Keys* multiparty extension to the Signal two-party protocol. Introduced in [Mar14], and previously analysed within [BCG23], Sender Keys utilises existing secure channels between pairs of participants to distribute per-sender sessions between members of the group. When a device first sends a message to the group, they generate a new session and distribute the keys necessary to authenticate and decrypt messages to the other devices in the group. These sessions consist of a signing key (for the purposes of authentication) and a symmetric secret (for the purposes of confidentiality), the latter of which is ratcheted forward to derive unique key material for each message. Each device participating in the group maintains their own session which they use to send messages to the group. We split our examination of Sender Keys into two. First, we consider a single session stage that provides a secure unidirectional channel between one sending device and many recipient devices, captured through the UNI scheme. Such channels are expected to provide authentication, confidentiality and forward secrecy for a linear sequence of messages. Second, we consider how these individual sessions are distributed through an untrusted network, using the aforementioned pairwise channels, as well as the rotation of individual sessions as the list of intended recipients changes. We capture this functionality in the SK scheme.

[13] For the interested reader, the use of trial decryption is not unique to WhatsApp. Signal seems to do something similar (see `decrypt_message_with_record` in libsignal, for example), as does Matrix (see Figure 3.8).

UNI

Init()

1 : $usid \leftarrow\$ \{1, 2, \ldots, 2^{31}\}$; $z \leftarrow 0$
2 : $ck \leftarrow\$ \{0,1\}^{256}$; $(gsk, gpk) \leftarrow\$ \mathsf{XDH.Gen}(1^{256})$
3 : $ust_{snd} \leftarrow \langle \texttt{UNI-OUT}, usid, z, ck, gsk \rangle$
4 : $ust_{rcv} \leftarrow \langle \texttt{UNI-IN}, usid, z, ck, gpk \rangle$
5 : **return** (ust_{snd}, ust_{rcv})

Enc(ust_{snd}, m)

1 : $\texttt{UNI-OUT}, usid, z, ck, gsk \leftarrow ust_{snd}$
2 : $z \leftarrow z + 1$
3 : $mk \leftarrow \mathsf{HMAC}(ck, \texttt{0x01})$
4 : $ck \leftarrow \mathsf{HMAC}(ck, \texttt{0x02})$
5 : $k \leftarrow \mathsf{HKDF}(\varnothing, mk, \texttt{WhisperGroup}, 50\mathsf{B})$
6 : $iv, ek \leftarrow k[0\mathsf{B} \rightarrow 15\mathsf{B}], k[16\mathsf{B} \rightarrow 47\mathsf{B}]$
7 : $c \leftarrow \mathsf{AES\text{-}CBC.Enc}(ek, iv, m)$
8 : $c_U \leftarrow \langle \texttt{UNI-CTXT}, usid, z, c \rangle$
9 : $\sigma_U \leftarrow \mathsf{XEd.Sign}(gsk, c_U)$
10 : $ust_{snd} \leftarrow \langle \texttt{UNI-OUT}, usid, z, ck, gsk \rangle$
11 : **return** $ust_{snd}, (c_U, \sigma_U)$

GenInbound(ust_{snd})

1 : $\texttt{UNI-OUT}, usid, z, ck, gsk \leftarrow ust_{snd}$
2 : $ust_{rcv} \leftarrow \langle \texttt{UNI-IN}, usid, z, ck, \mathsf{PK}(gsk) \rangle$
3 : **return** ust_{rcv}

Dec($ust_{rcv}, (c_U, \sigma_U)$)

1 : $\texttt{UNI-IN}, usid, z, ck, gpk \leftarrow ust_{rcv}$
2 : **require** $usid = c_U.usid \ \wedge z = c_U.z$
3 : **require** $\mathsf{XEd.Verify}(gpk, c_U, \sigma_U)$
4 : $mk \leftarrow \mathsf{HMAC}(ck, \texttt{0x01})$
5 : $ck \leftarrow \mathsf{HMAC}(ck, \texttt{0x02})$
6 : $k \leftarrow \mathsf{HKDF}(\varnothing, mk, \texttt{WhisperGroup}, 50\mathsf{B})$
7 : $iv, ek \leftarrow k[0\mathsf{B} \rightarrow 15\mathsf{B}], k[16\mathsf{B} \rightarrow 47\mathsf{B}]$
8 : $m \leftarrow \mathsf{AES\text{-}CBC.Dec}(ek, iv, c_U.c)$
9 : **require** $m \neq \perp$
10 : $z \leftarrow z + 1$
11 : $ust_{rcv} \leftarrow \langle \texttt{UNI-IN}, usid, z, ck, gpk \rangle$
12 : **return** (ust_{rcv}, m)

Figure 4.7. Pseudocode describing a single stage of group messaging in WhatsApp, a unidirectional channel which we denote UNI.

Consider the UNI scheme described in Figure 4.7. It consists of four algorithms, UNI = (Init, Enc, Dec, GenInbound), that describe a forward secure channel from one sender to many recipients. We briefly describe the interface of these algorithms before describing their behaviour in detail below.

- $ust_{snd}, ust_{rcv} \leftarrow\$ \mathsf{UNI.Init}()$ generates a new unidirectional Sender Keys session and outputs the outbound session, ust_{snd}, and inbound session, ust_{rcv}.

- $ust_{snd}, c \leftarrow \mathsf{UNI.Enc}(ust_{snd}, m)$ takes as input an outbound session, ust_{snd}, and plaintext message, m, before outputting an updated outbound session, ust_{snd}, and the resulting ciphertext, c.

- $ust_{rcv}, m \leftarrow \mathsf{UNI.Dec}(ust_{rcv}, (c_U, \sigma_U))$ takes as input an inbound session, ust_{rcv}, and signed ciphertext, (c_U, σ_U), before outputting an updated inbound session, ust_{rcv}, and the resulting message, m.

- $ust_{rcv} \leftarrow \mathsf{UNI.GenInbound}(ust_{snd})$ takes as input an outbound session, ust_{snd}, and calculates its inbound counterpart, ust_{rcv}.

The sender initialises a new session with UNI.Init. The device generates the session identifier and sets the initial message index to zero (see line 1). They then generate the initial symmetric key material, the chain key ck, and a fresh signing key pair, gsk and gpk (see line 2). The device constructs an outbound session state, consisting of the session identifier, message index, chain key and signing key (see line 3). The device additionally constructs an inbound

sessions state, consisting of the session identifier, message index, chain key and verification key (see line 4). The algorithm outputs an outbound session, kept by the sender, and an inbound session, distributed to the intended recipients.

The sender can encrypt new messages by calling UNI.Enc with the outbound session state and plaintext. To encrypt a message, the message index is incremented (see line 2) before deriving the per-message key material and ratcheting the chain key forward (see lines 3 and 4).[14] The per-message key material, mk, is stretched using HKDF to derive an initialisation vector and encryption key for the AES-CBC encryption (lines 5 to 7). The resulting ciphertext, session identifier and message index are signed with the signing key, gsk (line 9).[15] The outbound session state is updated (line 10) and the signed ciphertext is distributed to the recipients.

The UNI.Dec algorithm describes how messages are decrypted. Clients start by ensuring that the session identifier of the ciphertext and local session state match (line 2). We additionally include a check that enforces in-order message decryption. This is inaccurate, since WhatsApp will perform out-of-order decryption for these sessions. It does so by caching the per-message key material for up to 2000 skipped messages. The device proceeds to verify the signature using gpk from their local state. If it passes, they proceed to derive the per-message key material before ratcheting the chain key forward (lines 4 to 5). The key material is stretched using HKDF to derive the initialisation vector and encryption key for AES-CBC (line 6 to 8). Finally, the inbound session state is updated, saving the new chain key and updating the message index to reflect that of the next message to be decrypted.

We additionally include the UNI.GenInbound algorithm, which describes how clients derive an inbound session from their own copy of the outbound session. This is used by the sending session when they need to distribute the key material to new group members.

We now describe how WhatsApp clients distribute and manage the sessions for unidirectional channels, through the $\mathsf{SK} = (\mathsf{Init}, \mathsf{Add}, \mathsf{Rem}, \mathsf{Enc}, \mathsf{Dec})$ scheme which we detail in Figure 4.8.

- $skst \leftarrow\!\$\ \mathsf{SK.Init}(\rho, ipk, mem)$ initialises a new Sender Keys session with the role, ρ, the executing device's public identity key, ipk, and a list of participants, mem.
- $skst, pst, \overrightarrow{c_P}, (c_U, \sigma_U) \leftarrow \mathsf{SK.Enc}(skst, pst, \mathcal{SKB}, meta, m)$ sends a message to the group. It takes as input a sending session state, $skst$, pairwise session state, pst, a store of key bundles, $\mathcal{SKB}$, the ICDC metadata, $meta$,

[14] We note a minor mistake in the whitepaper regarding key derivation. They claim that the message key is an 80 byte value with 16 for the IV, 32 for AES key and 32 for a MAC key. But they only show it being derived as HMAC-SHA256. In practice, the *message key* is an HMAC-SHA256 value (derived from the chain key) which they then apply HKDF-SHA256 to, for which the output length depends on the context. In particular, group messages do not use a MAC key so they only extract 50 bytes, while pairwise messages need a MAC key so they extract 80 bytes.It is not clear why 50 bytes are extracted while 48 bytes are used.

[15] Note that, while Sender Keys ciphertexts contain a protocol version field that is inside the signature, we do not include this field in our description or formal analysis.

and the message to be sent, m. It outputs an updated sending session state, *skst*, and pairwise session state, *pst*, as well as a list of pairwise ciphertexts to (possibly) distribute the Sender Keys session, $\overrightarrow{c_P}$, and the signed Sender Key ciphertext, (c_U, σ_U).

- $skst, pst, meta, m \leftarrow \mathsf{SK.Dec}(skst, pst, skb_s, c_P, (c_U, \sigma_U))$ receives a message from the sending session. It takes as input a recipient session state, *skst*, pairwise session state, *pst*, the sender's key bundle, skb_s, an optional key distribution ciphertext, c_P, and the signed Sender Keys ciphertext, (c_U, σ_U). It outputs an updated recipient session state, *skst*, and pairwise session state, *pst*, as well as the ICDC metadata, *meta*, and the plaintext message, m.

- $skst \leftarrow \mathsf{SK.Add}(skst, ipk^\star)$ adds a new device to a sending session. It takes as input the sending session state, *skst*, and the new device's public identity key, *ipk*, then outputs an updated sending session state, *skst*.

- $skst \leftarrow \mathsf{SK.Rem}(skst, ipk^\star)$ removes a device as a recipient from a sending session. It takes as input the sending session state, *skst*, and the device's public identity key, *ipk*, then outputs an updated sending session state, *skst*.

As described above, each member of the group generates and maintains their own unidirectional channel, for which they use the outbound session to send messages to the group. The inbound sessions, which allow the other members to decrypt and verify sent messages, are then distributed over pairwise channels using the existing pairwise channels (see Section 4.2.2). Membership changes are implemented *lazily*. That is, when a recipient is added or removed, the sender will take note of the change then implement the necessary key rotation when the next message is sent. When a user or device is added to the group, they are sent a copy of the inbound session with the current chain key, allowing them to decrypt all future messages but none that have been sent previously (see SK.Add and lines 8 to 12 of SK.Enc).[16] When a user or device is removed from the group, the session owner must generate a new session and distribute it to the remaining members (see SK.Rem and lines 2 to 7 of SK.Enc). This ensures that devices associated with the removed member cannot decrypt future messages using their copy of the inbound session.

We would expect clients to enforce group membership by deleting inbound sessions originating from a user that has been removed from the group. We were not able to find evidence of this behaviour during our investigation. It is possible that group membership is enforced non-cryptographically.[17] This could be implemented, for example, by authenticating the sender and checking their identity against the server-provided member list before accepting the message.[18]

[16] The level of *forward security* provided in this architecture depends on the properties of the underlying unidirectional channel. In the case of Sender Keys as it is used by WhatsApp and Signal, this provides forward secrecy but not forward authenticity [BCG22, BCG23, ACDJ23, ADJ24].

[17] Since our inspection of the implementation was not exhaustive, we are not confident in claiming that this functionality does not exist.

[18] Since WhatsApp clients trust the server to provide the group member list, something which is reflected in our security model, this issue does not appear in our analysis.

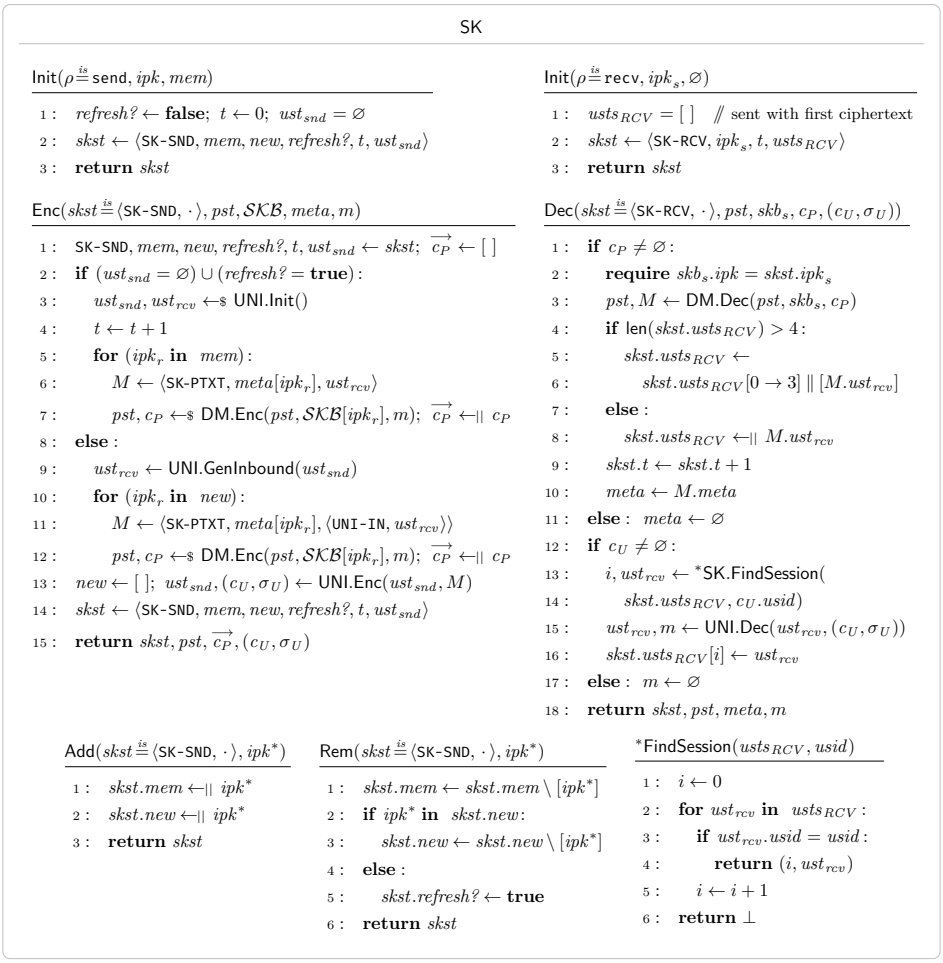


Figure 4.8. Pseudocode describing WhatsApp's variant of the Sender Keys protocol, capturing the rotation of unidirectional channels to manage recipients.

Our description of Sender Keys maintains a focus on unidirectional channels even as we lift to the session level and the over-arching group messaging protocol. As such, each instantiation of the protocol has a predefined role of either sender or receiver. Composed together, with one SK session per group member, we can see how the protocol forms a logical "group chat". Note, however, that Sender Keys sessions are not cryptographically bound to a "logical group". This means, Bob can take an inbound session from Alice in group G and distribute it as an inbound session for himself in group H.

As can be seen in Figure 4.8, WhatsApp's implementation keeps the five most recent Sender Keys sessions they have received from a particular device for a particular group. This is likely to ensure that any out-of-order, delayed or missed ciphertexts from a previous session can be decrypted. Note that senders do not keep old copies of their outgoing sessions. Nonetheless, this decision results in a slower recovery of security after compromise of Sender Keys session state. This issue can be effectively and efficiently mediated by including the

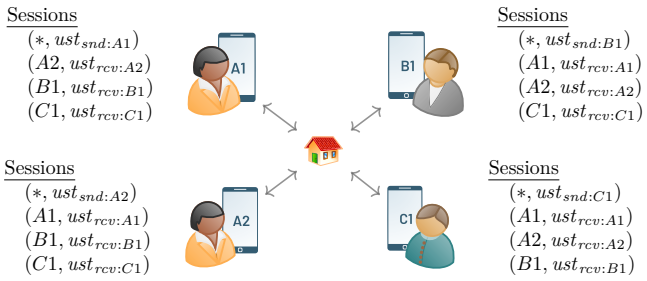


Figure 4.9. An example of a WhatsApp client's session store after each member has initialised an outbound Sender Keys stage and distributed the respective inbound UNI state to the group. This illustration simplifies the structure and contents of the store (see Figure 4.8 for such details). We use '$*$' to indicate that this is the active sending session for this participant.

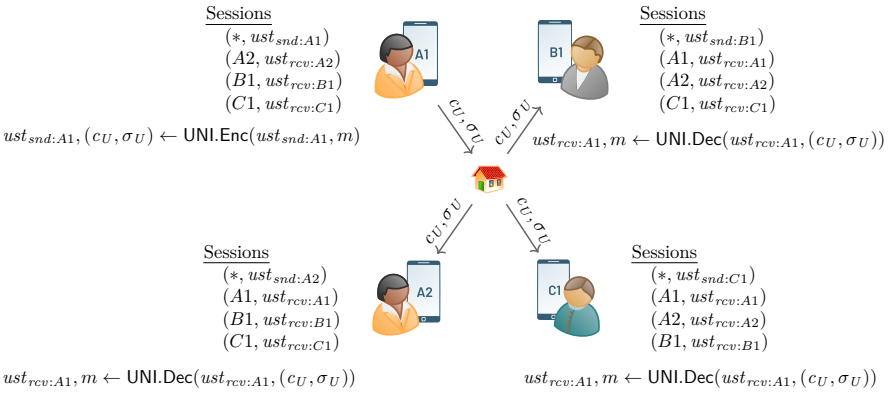


Figure 4.10. An example of a WhatsApp client using an outbound UNI session to send a message m to the group.

number of messages sent in the last session when initiating a new session. The recipient may then derive the message key for each of the missing messages from the previous session, allowing it to safely destroy the chain key. A similar set of issues exists for Signal pairwise channels, for which a similar improvement has previously been suggested [DGP22].

Group management. In WhatsApp, group membership is managed through control messages protected by transport security between client and server but without end-to-end guarantees. Thus, group membership is controlled by the server [RMS17, BCG22, BCG23].

While group membership is determined at the user-level, this is implemented in the cryptography by adding and removing devices from the group. Our description in Section 4.2.5 reflects the former, while our security analysis in Section 6.2.7 necessarily reflects the latter. Clients trust the server to provide a list of users that are members of the group, but verify the list of companion devices for each of those users. Additionally, each client may have a different view of the group membership; WhatsApp provides no guarantees in this respect.

ATTACH

$\mathsf{Enc}(m, t)$

1 : $r \leftarrow\$ \{0,1\}^{32\mathsf{B}}$
2 : $k \leftarrow \mathsf{HKDF}(\varnothing, r, t, 112\mathsf{B})$
3 : $aiv \leftarrow k[0\mathsf{B} \to 15\mathsf{B}];\ aek \leftarrow k[16\mathsf{B} \to 47\mathsf{B}]$
4 : $ahk \leftarrow k[48\mathsf{B} \to 79\mathsf{B}];\ ark \leftarrow k[80\mathsf{B} \to 111\mathsf{B}]$
5 : $c \leftarrow \mathsf{AES\text{-}CBC.Enc}(aek, aiv, m)$
6 : $\tau \leftarrow \mathsf{HMAC}(ahk, c)[0\mathsf{B} \to 9\mathsf{B}]$
7 : $c' \leftarrow c[16\mathsf{B} \to \ldots]$ // remove iv
8 : $h \leftarrow \mathsf{SHA256}(c' \,\|\, \tau)$
9 : $m_{ptr} \leftarrow \langle \mathsf{ATTACH}, t, r, h \rangle$
10 : **return** m_{ptr}, c', τ

$\mathsf{Dec}(m_{ptr}, c', \tau)$

1 : $(t, r, h) \leftarrow \langle \mathsf{ATTACH}, m_{ptr} \rangle$
2 : **if** $(h = \mathsf{SHA256}(c' \,\|\, \tau))$
3 : $k \leftarrow \mathsf{HKDF}(\varnothing, r, t, 112\mathsf{B})$
4 : $aiv \leftarrow k[0\mathsf{B} \to 15\mathsf{B}];\ aek \leftarrow k[16\mathsf{B} \to 47\mathsf{B}]$
5 : $ahk \leftarrow k[48\mathsf{B} \to 79\mathsf{B}];\ ark \leftarrow k[80\mathsf{B} \to 111\mathsf{B}]$
6 : **if** $(\tau = \mathsf{HMAC}(ahk, aiv \,\|\, c')[0\mathsf{B} \to 9\mathsf{B}])$
7 : $m \leftarrow \mathsf{AES\text{-}CBC.Dec}(aek, aiv, c')$
8 : **return** m
9 : **return** $\perp$

Figure 4.11. Pseudocode describing how attachments are secured in WhatsApp.

As such, we avoid capturing logical groups in our security analysis. It is for these reasons that the model we choose to apply captures messaging sessions at the level of unidirectional channels, and group membership at the level of devices rather than users while expecting that clients maintain eventually consistent views of a user's device composition.

Add-on Sender Keys. WhatsApp supports *Community Announcement Groups*. These are group chats where members can have one of two roles. Administrators are able to post messages, while normal group members may only react to existing messages. Under this constrained interaction model, WhatsApp's design aims to keep the PCS guarantees that we get from Sender Keys, whilst requiring that only administrators rotate their sender key when someone leaves the group (rather than all the group's members). To achieve this, WhatsApp introduces a second sender key, the *Add-on Sender Key*. These work similarly to the sender keys normally used in group chats, however they are restricted to particular interactions (such as reactions to existing messages). We do not consider add-on sender keys in this work.

Attachments. WhatsApp allows users to attach media (and other files) as encrypted blobs distributed alongside a message. We describe this functionality with the $\mathsf{ATTACH} = (\mathsf{Enc}, \mathsf{Dec})$ algorithms.

- $m_{ptr}, c', \tau \leftarrow\$ \mathsf{ATTACH.Enc}(m, t)$ encrypts an attachment, m, of the given type, t, and outputs a pointer, m_{ptr}, the attachment ciphertext, c', and an authentication tag, τ.

- $m \leftarrow \mathsf{ATTACH.Dec}(m_{ptr}, c', \tau)$ decrypts an attachment, c', of the given type, t, using the pointer, m_{ptr}, and outputs the resulting contents, m.

The sender generates new key material (see lines 1 to 4) that is used to encrypt the attachment contents using a combination of AES-CBC and HMAC-SHA256 (see lines 5 to 6). Different types of attachments use a different information string for key derivation (line 2). For example, images use the string '`WhatsApp Image Keys`'. Note that the initialisation vector is removed from

the ciphertext for distribution, but included when computing the authentication tag (line 6). The encrypted blob is accompanied by a pointer structure that includes (a) the type of the attachment, (b) the 32 byte key material required to authenticate and decrypt the ciphertext, (c) a hash of the ciphertext and authentication tag with the initialisation vector removed, and (d) a pointer to the attachment's ciphertext and tag on the server (lines 1, 8 and 9). We do not include the attachment's location on the server in the pseudocode description. This attachment pointer, m_{ptr}, is transmitted as the contents of a normal encrypted message (over a Signal pairwise channel for direct messages or a Sender Keys session for groups).

The decryption process mirrors the encryption process. First, the hash within the pointer is used to fetch the encrypted attachment from the store. Note that ATTACH.Dec does not describe this process, taking the encrypted attachment as input instead. Next, the client ensures that the blob given matches the hash given in the pointer structure (line 2). The provided random key material is stretched using HKDF to derive the initialisation vector and encryption key for AES-CBC as well as the key for the authentication tag (lines 3 to 5). The client proceeds to check the authentication tag against the ciphertext (with the initialisation vector prepended) and, if this is successful, decrypts the ciphertext and returns the resulting attachment (lines 6 to 8). Note that the inclusion of the hash within the message pointer cryptographically links a particular attachment ciphertext with the message, avoiding a (potentially compromised) sender changing the attachment contents in the future. Additionally, the inclusion of the attachment type during key derivation cryptographically protects against type confusion.

To send an attachment, a WhatsApp client calls into ATTACH.Enc, uploads the encrypted attachment blob to the server and sends the attachment pointer over a secure channel to the intended recipient(s). In a direct message, this message will be sent over a Signal pairwise channel, while in group chats the message will be sent over a sender keys channel. When the recipient(s) receive a two-party or sender keys message containing an attachment pointer, they retrieve the encrypted attachment blob from the server, then call ATTACH.Dec with both the attachment pointer and blob to retrieve its plaintext, m. A return value of $\perp$ indicates an error. We do not model the security of ATTACH explicitly. However, since this scheme is used by WhatsApp for history sharing, we do so implicitly. Additionally, attachments in WhatsApp offer a number of features, such as chunking and previews, that we do not explore in this document. See *Transmitting Media and Other Attachments* in the WhatsApp security whitepaper [Wha23a].

History sharing. History sharing occurs from the primary device to new companion devices. The primary device encrypts historic messages and sends them using the attachment mechanism described above. This happens once upon linking a new companion device, and can also be triggered on-demand. History sharing is one-way (from the primary device to verified companion devices of the same user) and shares the transcript directly, not key material.

We describe WhatsApp's history sharing through the HS.Share and HS.Receive algorithms.

- $pst, c_P, c_{hist}, \tau_{hist} \leftarrow\$ \mathsf{HS.Share}(\rho, pst, skb, T)$ describes a primary device, with role '$\rho = \texttt{primary}$' and pairwise session state pst, sharing the given message transcript, T, with the intended recipient, identified by the key bundle skb. It outputs an updated pairwise session state, pst, a pairwise ciphertext, c_P, an encrypted attachment, c_{hist}, and authentication tag, τ_{hist}.

- $pst, T \leftarrow \mathsf{HS.Receive}(\rho, pst, skb, c_P, c_{hist}, \tau_{hist})$ describes a companion device, with role '$\rho = \texttt{companion}$' and pairwise session state pst, receiving a history share, from the device identified by the key bundle skb, consisting of a pairwise ciphertext, c_P, an encrypted attachment, c_{hist}, and authentication tag, τ_{hist}. If successful, it outputs an updated pairwise session state, pst, and the transcript, T.

Once a primary device finishes linking a new companion, it executes the HS.Share algorithm. This algorithm utilises the aforementioned attachments mechanism to create an encrypted attachment containing the appropriate message transcript (see line 1). The resulting encrypted blob is uploaded to the server, while the attachment pointer is shared with the companion device over a pairwise channel (see line 2). When a companion device receives such a message, they execute the HS.Receive algorithm. To start, the pairwise ciphertext is decrypted using the claimed device identity (line 1). The resulting attachment pointer and encrypted attachment blob are provided to the attachment decryption algorithm which, in turn, outputs the transcript upon success (line 2). Note that WhatsApp uses the '`WhatsApp History Keys`' attachment type to differentiate history sharing attachments from others.

The HS algorithms in this section do not capture the verification and enforcement that clients perform to determine which devices to share or accept history from. We build upon the informal description in this section with explicit pseudocode in Section 4.2.5.

HS

$\mathsf{Share}(\rho \overset{is}{=} \texttt{primary}, pst, skb, T)$

1 : $m_{ptr}, c_{hist}, \tau_{hist} \leftarrow\$$
2 : $\quad \mathsf{ATTACH.Enc}(T, \texttt{WhatsApp History Keys})$
3 : $pst, c_P \leftarrow\$ \mathsf{DM.Enc}(pst, skb, m_{ptr})$
4 : **return** $pst, c_P, c_{hist}, \tau_{hist}$

$\mathsf{Receive}(\rho \overset{is}{=} \texttt{companion}, pst, skb, c_P, c_{hist}, \tau_{hist})$

1 : $pst, m_{ptr} \leftarrow \mathsf{DM.Dec}(pst, skb, c_P)$
2 : **require** $m_{ptr}.t = \texttt{WhatsApp History Keys}$
3 : $T \leftarrow \mathsf{ATTACH.Dec}(m_{ptr}, c_{hist}, \tau_{hist})$
4 : **return** pst, T

Figure 4.12. Pseudocode describing how history is shared in WhatsApp.

4.2.4 Authenticating Cryptographic Identities

WhatsApp provides two methods of authenticating the cryptographic identity of other users (and their devices).

Out-of-band Verification using QR Codes. User-to-user verification is an optional secondary step that involves either QR code verification or comparison of security fingerprints. When scanning QR codes, they directly encode a list of all

primary and companion device identity keys in the QR code. When comparing security fingerprints, they generate a 60-bit number that is a function of all these keys.

A similar procedure is used for device-to-device verification and is performed as part of the companion device linking process (such that it is not possible to add a companion device which has not been verified out-of-band).

Key Transparency. WhatsApp uses a key transparency log to assure users, through independent auditors, that the server is honestly distributing their cryptographic identities. As discussed, we do not cover such functionality in this work. For more details, see WhatsApp's blog post on key transparency [L13], whitepaper [Wha23b], open-source implementation `akd` [L21] and the academic works it is built upon [MKKS+23, CDGM19, MBB+15].

4.2.5 The WhatsApp Multi-Device Group Messaging Protocol

We now describe how WhatsApp combines these components to construct a multi-device group messaging protocol. We do so, primarily, through the algorithms in Figure 4.13. These describe the WA protocol, a subset of WhatsApp capturing how clients perform user and device cryptographic identity management, group messaging and history sharing.

User and device management. When a user sets up a new account, the device they are using creates a cryptographic identity for itself which becomes the user's *primary device*. We describe this process in the WA.NewPrimaryDevice algorithm. WhatsApp will maintain a mapping between the user's account, their phone number and the primary device's cryptographic identity. As discussed in Section 4.2.4, WhatsApp provide a number of methods to verify such mappings. Functionally, the primary device initialises an instance of the multi-device sub-protocol, which we represent with a call to MD.Setup (see line 1). They additionally initialise the pairwise channels sub-protocol using the identity key from the multi-device protocol (line 2). This outputs a public key bundle containing the device's identity key that identifies the user both in the context of device management and pairwise channels. The key bundle additionally includes the medium-term keys, ephemeral keys and signatures that allow other devices to initialise Signal two-party channels. A private device state, *wst*, is initialised to store the secret state for the multi-device and pairwise channel sub-protocols (lines 3 and 4). The device state is kept private while the key bundle is distributed through the server. The primary device may now engage and participate in new messaging sessions.

When the user logs into a new device, such as the WhatsApp web client, this *companion device* will generate its own cryptographic identity. We describe this process in the WA.NewCompanionDevice algorithm, which proceeds similarly to WA.NewPrimaryDevice. Companion devices must be linked with a primary device before they can participate in messaging sessions. Thus, the companion device additionally generates a 32 byte *linking secret* that, along with its public identity key, will be encoded as a QR code to be scanned by the primary device

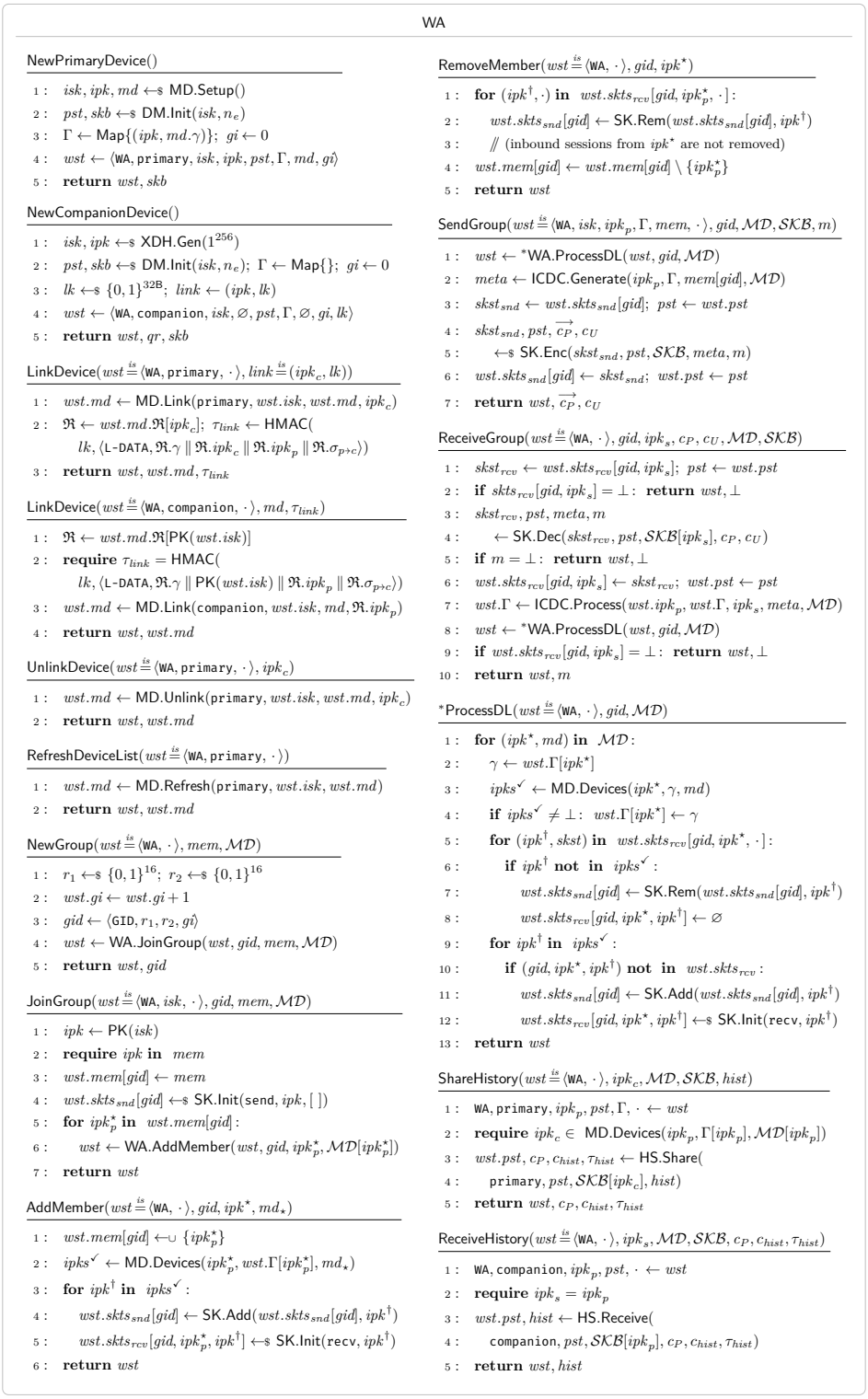


Figure 4.13. Pseudocode describing multi-device group messaging in WhatsApp.

(lines 3 and 4).[19] Upon scanning the QR code, the primary device creates the *account signature* and an updated device list. This is described by the primary device case of the WA.LinkDevice algorithm. Once the multi-device sub-protocol has performed the linking action (see line 1), the primary device additionally computes a *linking HMAC* (line 2). Here, the linking secret is used to key an HMAC digest containing the linking metadata, the primary device's identity key and the account signature. This is sent alongside the device record through the untrusted server to the companion device. The companion device case of WA.LinkDevice describes how the companion device completes the linking process. Before adding their own *device signature*, the companion device verifies the providing linking HMAC to ensure that they are linking with the primary device they intend to (see lines 1 and 2). If this check succeeds, the companion device proceeds to calculate the device signature and save it to their local state (line 3) before publishing it through the server (line 4). Similarly, a user may remove a companion device from the account by producing an updated device list with the companion device's identity removed. We describe this process in the WA.UnlinkDevice algorithm.

Note that, our description captures how the cryptographic link is made *without* capturing how the creation of that link is secured. In other words, the WA.NewCompanionDevice algorithm simulates the linking protocol.

Each client synchronises their list of a user's devices by downloading the user's signed device list from the server. This is triggered when (a) the client is interacting with the user, but does not already have a copy of their device list, (b) the device list has expired, (c) they have received ICDC information that indicates a new device list for that user is available, or (d) they have received a message from a companion device that is not present in the device list. In our description, we remove this synchronisation mechanism and replace it by providing the device list as a direct input to each algorithm that uses it. These algorithms then verify the device list before using it, in a manner equivalent to WhatsApp's verification routine during synchronisation. This is principally achieved through a call to the MD.Devices algorithm.

Group management. We describe the creation of a new group with the WA.NewGroup algorithm. The group creator specifies an initial list of users, before generating a group identifier that consists of two random 16-bit numbers and a local counter (see lines 1 to 3). Since, from this point onwards, WA.NewGroup follows the same procedure that new members perform when they join an existing group, we capture this process through a call to the WA.JoinGroup algorithm (line 4).

When a device joins a group, they take as input the group identifier, a list of users that are currently members and any relevant multi-device state for those users (see WA.JoinGroup). Together, the list of users and their public multi-device state allows the joining device to determine the current group membership as a list of device identity keys. After initialising its own sending

[19] WhatsApp has since introduced an alternative to QR codes for device pairing utilising confirmation codes [Wha24]. The resulting protocol is substantively different and is not covered here.

Sender Key session with a call to SK.Init (line 4), the device iteratively adds each user in the member list with a call to WA.AddMember (lines 5 and 6).

WhatsApp clients maintain a sender key session store, represented in our description by $skts_{snd}$ and $skts_{rcv}$ (which store the client's outbound and inbound sender key sessions respectively). Inbound sessions are addressed by the *sender key name* which consists of the group identifier as well as the session owner's user and device identifier. As in Section 4.2.1, our description replaces the user identifier with the primary device's identity key and the device identifier with the sending device's identity key under the assumption that such mappings are correctly maintained by the implementation.

The WA.AddMember algorithm captures the process of adding a new device to the group. Given the group identifier, identity key of the new member and their multi-device state, the device adds their user identity key to the member list (see line 1) before computing the current set of verified devices for that user with a call to MD.Devices (line 2). The client registers each device as a recipient in its sending Sender Keys session (lines 3 and 4). It also initialises a recipient Sender Keys session for each device (line 5). The process of adding a new member to the group requires the server to trigger the execution of WA.JoinGroup for each of the new member's devices, as well as notify every existing device in the group with by triggering WA.AddMember.

Similarly, removing a user from a group requires the server to trigger the WA.RemoveMember algorithm to be executed by every device remaining in the group. Given the group identifier and primary identity key of the removed user, the client removes any recipient device associated with this identity key from the client's sending Sender Keys session. As discussed in Section 4.2.3, we were unable to find evidence that the Sender Keys sessions originating from the removed device are removed from the local state (reflected in line 3 of WA.RemoveMember).

Group messaging. The WA.SendGroup and WA.ReceiveGroup algorithms capture how WhatsApp clients send and receive application messages, respectively. Internally, these algorithms use the SK scheme to handle message encryption and decryption, as well as session initialisation and rotation (see line 4 of WA.SendGroup and line 3 of WA.ReceiveGroup).

Both sending (and receiving) messages may require the creation (or processing) of pairwise ciphertexts in addition to Sender Keys ciphertexts (since the former may be used to manage the sessions of the latter). WhatsApp clients maintain a session store for Signal pairwise sessions, the management of which we leave to the DM scheme previously described. Clients will search for the appropriate session to decrypt ciphertexts with using the identity claimed in a plaintext wrapper around the ciphertext (for both pairwise Signal messages and group Sender Keys messages). In practice, these will be a user and device identifier in the format expected for XMPP messages [SA11a, SA11b]. Consistent with previous modelling decisions, we replace such identifiers with a claimed identity key ipk_s. During decryption, clients will execute MD.Devices to check the trust they have in the sender, as well as to ensure that the claimed identity in the plaintext wrapper matches the cryptographic identity used to initialise the session.

Multi-device updates. WhatsApp allows the server to push multi-device state updates to clients. We capture this by including a multi-device state input to the WA.SendGroup and WA.ReceiveGroup algorithms. When sending messages, clients will process any updates before they generate proceed with sending a message (see the call to *WA.ProcessDL on line 1). When receiving messages, clients process such updates after (see the call to *WA.ProcessDL on line 7). In both cases, ICDC information is generated and processed (lines 2 and 6 respectively) as metadata within pairwise ciphertexts (passed as the *meta* parameter to the Sender Keys scheme in lines 4 and 3 respectively).

The *WA.ProcessDL algorithm describes how WhatsApp clients process and react to changes to the multi-device state of their communicating partners. As described in Section 4.2.1, each client stores the timestamp of the most recent device list they have observed for each user they communicate with. We capture these values in the Γ dictionary which maps a user's identity key to the minimum device list generation they will accept for that user. The client uses the multi-device sub-protocol to determine the list of verified devices, given the minimum device list generation stored for that user and multi-device state (see lines 2 and 3). If this succeeds, clients will update their minimum device list generation (line 4). Having determined the list of verified devices for this user, clients locate revoked devices to remove them from their Sender Keys sessions (see lines 5 to 8) and locate new devices to add them (lines 9 to 12).

History sharing. The WA.ShareHistory and WA.ReceiveHistory algorithms provide a minimal description of how WhatsApp clients handle history sharing. It does not describe how clients maintain a message transcript, nor does it capture how received transcripts are processed. Additionally, it does not describe under what circumstances history sharing is triggered. Rather, our description focuses on capturing the cryptography used to secure its transfer. When history sharing is triggered, clients ensure that they are the primary device (see line 1) and that the recipient identity is a verified companion device (line 2), before passing the request to the HS.Share algorithm. When a client receives a history sharing ciphertext, they ensure that they are a companion device (see line 1) and that the sender is their primary device (line 2), before passing the request to the HS.Receive algorithm.

4.3 Discussion

In this chapter we have studied WhatsApp's secure group messaging functionality. We have given both a high-level overview of the protocol and its architecture, as well as formal descriptions of the components that work together to provide multi-device group messaging. Namely, we detail the device management, pairwise channels, session management, group messaging, and history sharing components as well as their composition. To do so, we fill in any gaps within, and verify the contents of, the WhatsApp security whitepaper by reverse engineering the official web client.

Accuracy. Our findings confirm that the WhatsApp security whitepaper is largely accurate: the functionality it describes matches the functionality of the client we investigated (for the subset of WhatsApp that we study). The majority

of the deviations we discovered were minor, such as an incorrectly specified output length of the HKDF function during encryption in Sender Keys (see UNI.Enc in Figure 4.7). There are, nonetheless, deviations that do meaningfully impact security. We now focus on the impact of session management.

Temporal compromise. The whitepaper does not include details of the session management layers it uses for pairwise channels and group messaging. As we have seen, WhatsApp allows for up to fourty active pairwise channels between any single pair of devices. In such a setup, it seems that it is not possible to restore security after a long-term identity key is compromised.[20] Even in the case where a single session state is compromised, without the identity and pre-keys remaining secure, security cannot be restored until the compromised session has been removed from the session store. This would require, at the least, fourty new sessions to be created.

Something similar is true for group messaging, whereby each client stores the last five Sender Keys sessions it has received from each communicating partner. In isolation, this requires at least five session rotations before security is restored. Since Sender Keys sessions are distributed over pairwise channels, this further amplifies the slow recovery of group messaging after a compromise of the pairwise channels.

In both cases, we posit that these design decisions were made to avoid messages from being lost when clients have become out-of-sync. While this may be a valid trade-off, to increase the reliability of the service for example, we stress that the details of such trade-offs are not clear to the public.

Group membership. Critically, group membership is not cryptographically authenticated in WhatsApp, as already established in prior works [RMS17, BCG22, BCG23]. Clients display group membership and thus participants in group chats *could* potentially review their groups regularly to mitigate the effect of this. However, as previously discussed, this puts an unduly burden on those who (have to) rely on WhatsApp, especially in a setting of up to 1024 members per group.[21] We, here, do not "report" this behaviour as a vulnerability simply because this behaviour has been reported in the literature before. For the avoidance of doubt, we do consider this a critical vulnerability undermining otherwise strong cryptographic guarantees.

A compounding issue, one that WhatsApp shares with Matrix, is that there is seemingly no attempt to provide *participant consistency* for group messaging. That is, WhatsApp provides no guarantee that the clients in a group have a consistent view of that group's membership.

Device consistency. In contrast, WhatsApp's use of ICDC information aims to ensure that communicating users *do* have a consistent view of one another's device composition. This contrasts with Matrix, which provides no such functionality.

[20] In contrast, a configuration that strictly enforces a single session being initiated between any two identity keys would be able to restore security.

[21] We note that the same issue was reported as a vulnerability in [ACDJ23], in response to which the Matrix developers committed to fixing the issue, at the time of writing this work is ongoing.

Sharing messages, not keys. We briefly note a difference in approach between WhatsApp and Matrix. As we saw in the previous chapter, Matrix clients share access to historical messages by sharing the appropriate session keys. In contrast, WhatsApp clients share the contents of historical messages directly as an encrypted attachment. Intuitively, these two approaches require similar levels of trust in the sharing party at the time of sharing, but WhatsApp's approach seemingly requires less trust in the broader network. Namely, in the case of Matrix, if the adversary has compromised a historical session key, they may distribute ciphertexts encrypted with the session and a target, receiving a Megolm session, would accept them. In contrast, such injection of historical messages is only possible in WhatsApp if the primary device of the recipient has been compromised.

Authenticating cryptographic identities. Our description does not extend to the distribution of user identities and we refer to WhatsApp's documentation on out-of-band verification [Wha23a, Page 24] and key transparency [Wha23b] whitepaper for WhatsApp's claimed assurances in this area.

CHAPTER 5

Formalising Multi-Device Group Messaging

We start by identifying a common subset of functionality and security guarantees targeted by the two protocols in our case study. Looking to formally model and analyse the security of these two protocols, we start by reviewing existing security models in the literature. Finding that no existing models capture the relationships between users, their devices, and the groups they are a part of, we proceed to develop the Device-Oriented Group Messaging (DOGM) model to do just that. Along the way, we identify and formalise a few common components within such systems.

5.1 Introduction

In this chapter, we develop a formalism to capture multi-device group messaging as implemented by our two case studies. After a brief review of existing models, we proceed to develop our formalism, the device-oriented group messaging model. As we do, we endeavour to discuss the design decisions and trade-offs we made along the way.

5.1.1 Goals

Our goal is to develop a model that captures the security of messages sent within group messaging conversations of Matrix and WhatsApp. In doing so, such a model should capture how the dynamics of membership changes, device management and state sharing cumulatively affect the security of those messages.

Further, we would like to understand how temporal compromise affects the security of past, present and future messages. For example, how does the compromise of the current session affect the security of past or future messages? How do state sharing protocols, such as the history sharing features of Matrix and WhatsApp, affect the confidentiality or authentication of messages? As we saw in Section 3.3, Matrix' history sharing feature allowed the injection of historical messages and, further still, makes no meaningful distinction between historical and new messages. Alternatively, is it possible for a user to recover from a complete compromise of one of their devices?

In contrast, we do not aim for this model to cover out-of-band verification (e.g. the verification framework in Matrix) nor do we intend it to cover the distribution of user identities. We do, however, wish to capture the distribution of device identities through an untrusted network, such that sessions must verify those identities and their links to pre-distributed user identities. We avoid consideration of insider attacks outside of those resulting from the aforementioned state compromises.

5.1.2 Related Work

Group key exchange. The study of formal models for group key exchange followed on closely from those initial works in the two-party [BR94, BCK98, Sho99, CK01] setting.

To the best of our knowledge, the first formal model for authenticated group key exchange (which we refer to as BCPQ01) was introduced in [BCPQ01]. In this model, each user possesses a long-term key (which may be a symmetric secret or asymmetric key pair). The adversary may orchestrate the execution of key exchange sessions between these users. To do so, the adversary initialises a new instance for each user they wish to participate in the key exchange, and then may proceed to schedule the protocol execution between these instances. The adversary is given complete control of the network, and may also compromise the long-term keys of users or the computed session keys of individual instances (provided the session has accepted). Partnering is defined by a session identifier that is constructed from the communication between protocol instances. An extension, which we denote BCP01,

captures dynamic group membership [BCP01]. Notably, the model enforces a consistent view of group membership through synchronous processing of operations that change the group membership. This model was further extended in [BCP02] to consider compromise of the internal session state (in addition to the session key of accepted instances and long-term user keys as in prior work). This remained an active line of work over the following decade, see [KY03, KY07, Man06, Man07, BM08, BBM09, Man09, ACMP10].

See [PRSS21a] for a recent systemisation of knowledge on game-based security models for group key exchange. This work culminates in a proposed formal model, which we refer to as PRSS21 (and is detailed in full by [PRSS21b]).

Asynchronous ratchet trees and continuous group key agreement. The study of asynchronous ratchet trees was initiated in [CCG+18], in which the authors apply tree-based Diffie-Hellman group key exchange to the setting of continuous key exchange. Such structures were initially used as the basis of the Messaging Layer Security (MLS) standard [BBR+23].

Leader-based and non-contributory group key exchange. The protocols and formalisms discussed thus far have focused on *contributory group key exchanges*. That is, each participant contributes equally to the generation and freshness of the shared key [AST98, Definition 3.2]. During its development, MLS moved to a non-contributory construction based on TreeKEM [BBR18], which has seen a number of models and analyses during its development [ACDT20, ACJM20, KPPW+21].

Sender Keys is another example of a widely deployed approach to non-contributory group key exchange. Recently, it has been studied in the context of WhatsApp [BCG23]. A non-contributory key exchange protocol is also used by Zoom for encrypted meetings, for which a model is proposed in [DJKM23].

More recently, a similar approach has been proposed in the design of efficient post-quantum variants of the protocols used by MLS [BVJ+23]. Within this work, they develop a formalism for multi-recipient variants of a key encapsulation mechanism (KEM). An approach that is further studied in [AHK+23]. An alternative approach to the one taken in this work could be to frame the key distribution of Megolm and Sender Keys as a multi-recipient key encapsulation mechanism.

Group messaging. [ACDT21] lift the study of group key exchange to the messaging setting. This builds upon the aforementioned line of work on continuous group key agreement [ACDT20].

Multi-device messaging. To the best of our knowledge, there exists no prior models that formalise secure messaging in the multi-device setting. As discussed in Section 2.4, Keybase represents a notable construction of multi-device (group) messaging that is deployed in practice. The analysis of the ELEKTRA system, which builds upon the design of Keybase, formalises the sigchain [LCG+23]. This formalism does not extend to messaging, however.

Compromise recovery. The notion of PCS was formalised in [CCG16]. The work focused on the compromise of long-term key material with a focus on single sessions (or ratchet chains). Formalisms capturing the PCS guarantees of Signal pairwise channels are developed for the analyses of [CCD+20, ACD19]. The resulting security guarantees apply to adversaries that at some point become passive and do not consistently interfere with protocol execution after a compromise. Building on prior works such as [CDV21], the work in [BCC+23] considers PCS against active attackers (in a single session setting). It considers both in-band, adding information to the ciphertexts, and out-of-band communication, requiring a second channel. In either case, recovery requires the adversary to allow at least one message through without tampering. In 2020, it was demonstrated that the PCS guarantees of Signal pairwise channels are undermined in practice by clients allow multiple sessions between parties [CFKN20]. This work was followed by a formal analysis of this setting to derive the resulting PCS guarantees [CJN23]. Similarly, a framework for capturing a protocol's ability to recover from varying levels of state compromise, from session state to long-term identity keys, is introduced in [BBL+23]. In the group setting, [CHK21] study the ability of groups to recover security after individual members are compromised and, in particular, how this affects the security of multiple groups with overlapping members.

5.1.3 Contributions

This chapter contains the following contributions.

Contribution 5.1. Section 5.2 introduces the Device-Oriented Group Messaging (DOGM) model for multi-device group messaging which captures messaging for dynamic groups, device management and state sharing together.

Contribution 5.2. We propose an extension to the DOGM model to capture device revocation. This enables our extended model to capture how the revocation of a compromised device propagates to the PCS guarantees of the underlying messaging channels and, in turn, the security of user messages.

We use these formalisms to analyse the security of Matrix and WhatsApp in the following chapter (see Sections 6.1 and 6.2 respectively).

We additionally develop a number of formalisms for common components of multi-device group messaging we have seen in our case studies.

Contribution 5.3. Section 5.4.1 introduces a model to capture the device management component of secure messaging applications, whereby a single primary cryptography identity can be represented by a changing group of companion device identities.

Contribution 5.4. Section 5.4.2 introduces a model to capture the session management layer of two-party secure messaging protocols. To the best of our knowledge, this is the first such model in the computational setting.

Contribution 5.5. Section 5.4.3 introduces a model to capture a single epoch of unidirectional group messaging, such as a single Sender Keys or Megolm session.

We utilise these component formalisms in our security analysis of WhatsApp in Section 6.2.

5.2 Device-Oriented Group Messaging

Before we introduce the model formally, we start by discussing the motivation behind it and how this guides our approach.

First, we highlight that the protocols we saw in Chapters 3 and 4 work through a mixture of user-level and device-level operations. Take group membership, for example. In Matrix and WhatsApp, the user interfaces expose operations to change the group membership in terms of the users in the group. That is, people may choose to invite, add and remove other users, but not individual devices. As a corollary, it is not possible to add a subset of a user's devices to a conversation. In contrast, the messaging sessions in Matrix (i.e. Megolm) and WhatsApp (i.e. Sender Keys) work primarily between devices. When a new user is added to a group, the clients of existing members first calculate the current list of verified devices for the newly added user, and then distribute the appropriate key material to each device individually. Since the construction of Megolm and Sender Keys treat all devices equally, a newly added device of an existing user is provided with the same key material as the device of a newly added user. The history sharing features of both Matrix and WhatsApp can be seen as a method to "smooth over" this technicality.

One could imagine an alternative design for multi-device group messaging for which users remain the primary unit of action. In the case of WhatsApp, this could take the form of Sender Key sessions between users, whereby a user's devices would share the necessary key material and/or plaintexts themselves, using a separate protocol. In such a construction, the Sender Key sessions would have no notion of a device. Indeed, previous versions of WhatsApp used such a design: each user had a primary device which acted as proxy to the messaging protocol on behalf of companion devices. Communication between a user's devices was orthogonal to (and used a different protocol to) communication between users. Furthermore, [CDDF20] have previously proposed a user-oriented design to implement multi-device functionality for applications such as Signal.

Note that the formalisms we discuss in the previous section, such as the BCPQ01 model [BCPQ01], its various developments, and the PRSS21 model [PRSS21a] permit capturing a single layer above sessions: either users or devices, but not both. However, since the protocols within Matrix and WhatsApp make mixed use across these two layers, we require a formalism that can capture this interplay.

In this chapter, we aim to formalise *device-oriented group messaging* protocols; a term we use to describe protocols for messaging between groups of devices while maintaining an awareness of the users these devices represent. The DOGM model aims to bridge the gap between device-oriented protocols and

user-oriented protocols by capturing which device belongs to which user (and vice versa) at any particular point in time. This enables our security result to capture message attribution at the user level, and to determine different levels of trust between devices. Looking forward, capturing these relationships will be vital in our ability to capture *history sharing*.

5.2.1 Syntax

We now define the syntax of device-oriented group messaging protocols.

Definition 5.1 (Device-Oriented Group Messaging Protocol)**.** A Device-Oriented Group Messaging (DOGM) protocol is a tuple of algorithms $\mathsf{DOGM} = (\mathsf{Gen}, \mathsf{Reg}, \mathsf{Init}, \mathsf{Add}, \mathsf{Rem}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{StateShare})$ for which $\mathsf{StateShare}$ represents the optional state-sharing functionality.

The first two algorithms, $(\mathsf{Gen}, \mathsf{Reg})$, concern the creation and management of long-term cryptographic identities.

- The *user creation* algorithm, Gen, models the generation of a user's long-term cryptographic identity.

  $$upau, usau \leftarrow\!\$\ \mathsf{Gen}(uid)$$

  It takes as input a user identifier, $uid \in \mathcal{U}$, before outputting a public and secret authenticator pair, $upau \in \mathcal{PAU}$ and $usau \in \mathcal{SAU}$.

- The *device registration* algorithm, Reg, models the creation of a new device, the initialisation of their cryptographic identity and their linking with a user's cryptographic identity (by simulating the results of a linking sub-protocol).

  $$upau, usau, dpau, dsau, c \leftarrow\!\$\ \mathsf{Reg}(uid, did, usau, upau)$$

  It takes as input a user identifier, $uid \in \mathcal{U}$, the chosen device identifier, $did \in \mathcal{D}$ and the user's secret and public authenticators, $upau \in \mathcal{PAU}$ and $usau \in \mathcal{SAU}$, before returning the updated user authenticators, $upau \in \mathcal{PAU}$ and $usau \in \mathcal{SAU}$, new device authenticators, $dpau \in \mathcal{PAU}$ and $dsau \in \mathcal{SAU}$, alongside an optional ciphertext, $c \in \mathcal{C}$ (or a failure state $\perp$).

The next five algorithms, $(\mathsf{Init}, \mathsf{Add}, \mathsf{Rem}, \mathsf{Enc}, \mathsf{Dec})$, concern the creation and use of individual messaging sessions.

- The *session initialisation* algorithm, Init, models the creation of a new unidirectional messaging session, in the role of either sender or recipient, associated with a particular group.

  $$dsau, st, gid \leftarrow\!\$\ \mathsf{Init}(uid, did, \rho, dsau, gid)$$

  This algorithm should be run once for each device that participates in a session, before Add, Rem, Enc or Dec. It takes as input the user identifier, $uid \in \mathcal{U}$, device identifier, $did \in \mathcal{D}$, role of the session $\rho \in \{\mathtt{send}, \mathtt{recv}\}$, a secret authenticator, $dsau \in \mathcal{SAU}$ and (optional) group identifier, $gid \in \mathcal{G}$, before returning an updated secret authenticator, $dsau \in \mathcal{SAU}$, session state, $st \in \mathcal{ST}$, and group identifier (or a failure state $\perp$).

- The *membership management* algorithms, Add and Rem, model the addition or removal of a device from a session (respectively). To add (or remove) the device from a particular messaging session, the appropriate algorithm is expected to be executed by every participating session in the group.

$$dsau, st, c' \leftarrow\$ \ \mathsf{Add}(dsau, st, uid, did, c)$$
$$dsau, st, c' \leftarrow\$ \ \mathsf{Rem}(dsau, st, uid, did, c)$$

  They take as input a secret authenticator, $dsau \in \mathcal{SAU}$, session state, $st \in \mathcal{ST}$, user identifier, $uid \in \mathcal{U}$, device identifier, $did \in \mathcal{D}$, and (optional) ciphertext, $c \in \mathcal{C}$, before returning an updated secret authenticator, $dsau \in \mathcal{SAU}$, session state, $st \in \mathcal{ST}$, and (optional) ciphertext, $c \in \mathcal{C}$ (or a failure state $\perp$).

- The *messaging* algorithms, Enc and Dec, model sending and receiving messages using an outbound (or inbound) session (respectively).

$$dsau, st, c \leftarrow\$ \ \mathsf{Enc}(dsau, st, m)$$
$$dsau, st, m \leftarrow \mathsf{Dec}(dsau, st, c)$$

  The encryption algorithm takes as input a secret authenticator, $dsau \in \mathcal{SAU}$, session state, $st \in \mathcal{ST}$, and the message to be sent, $m \in \mathcal{M}$, before returning an updated secret authenticator, $dsau \in \mathcal{SAU}$, session state, $st \in \mathcal{ST}$, and ciphertext, $c \in \mathcal{C}$. The decryption algorithm takes as input a secret authenticator, $dsau \in \mathcal{SAU}$, session state, $st \in \mathcal{ST}$, and ciphertext, $c \in \mathcal{C}$, before returning an updated secret authenticator, $dsau \in \mathcal{SAU}$, session state, $st \in \mathcal{ST}$, and the decrypted message, $m \in \mathcal{M}$ (or a failure state $\perp$). While these algorithms may be used to exchange non-application messages, the decryption algorithm must only return a non-null result in the plaintext slot if the given ciphertext included an application message that was accepted by the session.

The final algorithm, StateShare, captures state sharing between sessions.

- The *state sharing functionality*, StateShare, models the sharing of secret state between devices. While, nominally, a sub-protocol in its own right, it is represented by a single algorithm that encapsulates the requisite logic for each participant in the protocol.

$$dsau, st, c \leftarrow\$ \ \mathsf{StateShare}(dsau, st, uid, did, t, z, c)$$

  It takes as input a secret authenticator, $dsau \in \mathcal{SAU}$, secret state, $st \in \mathcal{ST}$, and (optionally) a user identifier, $uid \in \mathcal{U}$, device identifier, $did \in \mathcal{D}$, stage index, $t \in \mathbb{N}$, (optional) message index, $z \in \mathbb{N}$, and ciphertext, $c \in \mathcal{C}$. It outputs an updated secret authenticator, $dsau \in \mathcal{SAU}$, session state, $st \in \mathcal{ST}$, and (potentially) a ciphertext, $c \in \mathcal{C}$, the special failure symbol $c = \perp$, or a special success symbol $c = \top$.

These algorithms are defined with respect to

- a set of user identifiers, $\mathcal{U} \subseteq \{0,1\}^*$,

- a set of device identifiers, $\mathcal{D} \subseteq \{0,1\}^*$,
- a set of group identifiers, $\mathcal{G} \subseteq \{0,1\}^*$,
- a set of plaintext messages, $\mathcal{M} \subseteq \{0,1\}^*$,
- a set of ciphertexts, $\mathcal{C} \subseteq \{0,1\}^*$,
- a set of secret authentication values, $\mathcal{SAU} \subseteq \{0,1\}^*$,
- a set of public authentication values, $\mathcal{PAU} \subseteq \{0,1\}^*$, and
- a set of possible session states, $\mathcal{ST}$.

The session state, $st \in \mathcal{ST}$, must contain the following fields:

$.uid \in \mathcal{U}$ – the user identifier for this party.

$.did \in \mathcal{D}$ – the device identifier for this party.

$.gid \in \mathcal{G}$ – the group identifier for this session.

$.\rho \in \{\texttt{send}, \texttt{recv}\}$ – the role of this session.

$.t \in \mathbb{N}$ – the current stage of this session.

$.z \in \mathbb{N}$ – the current message index of the current stage.

$.\alpha \in \{\perp, \texttt{active}, \texttt{reject}\}$ – the current session status.

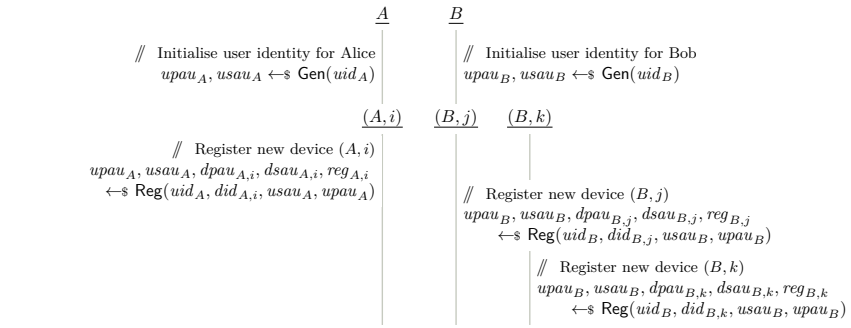
$.CU \in \mathbb{N} \to \mathcal{U}$ – the list of communication partners, where $CU[0]$ is the sending user.

$.CD \in \mathbb{N} \to \mathcal{D}$ – the list of devices associated with communication partners, where $CD[0]$ is the sending device.

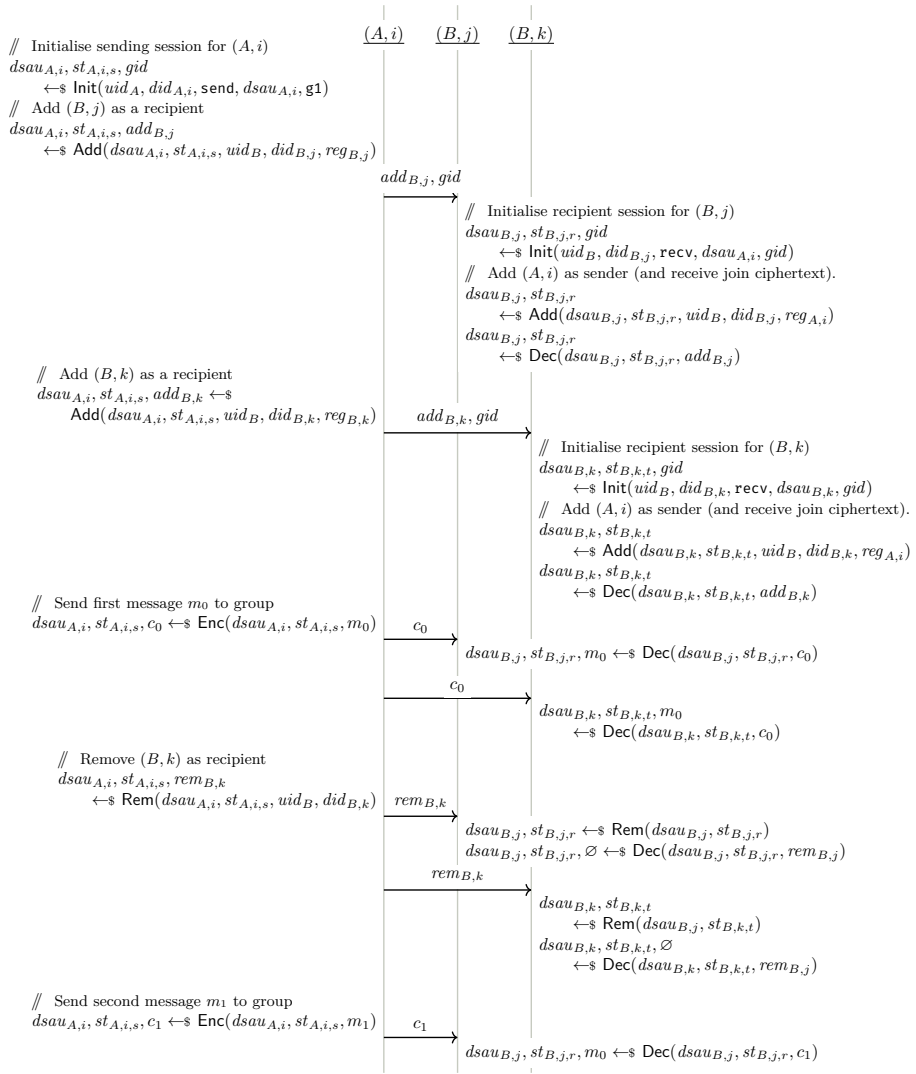
$.T \in \mathbb{N} \times \mathbb{N} \to \mathcal{C} \cup \{\perp\}$ – the message transcript for which $T[t,z]$ contains the z-th message sent in the t-th stage sent or received by the session. We use $|T[t]|$ as shorthand for the first value z such that $T[t,z] = \perp$, i.e. the number of messages accepted in the t-th stage.

The state may contain any additional fields needed by the protocol. These can be accessed both directly by name, or collectively through the '$.st$' field.

We, additionally, define separate types for authenticators of users and devices. In other words, we let $u\mathcal{SAU}$ and $u\mathcal{PAU}$ be the subsets of $\mathcal{SAU}$ and $\mathcal{PAU}$ that represent user authenticators (respectively) while $d\mathcal{SAU}$ and $d\mathcal{PAU}$ are the subsets of $\mathcal{SAU}$ and $\mathcal{PAU}$ that represent device authenticators (respectively). This gives, $\mathcal{SAU} = u\mathcal{SAU} \cup d\mathcal{SAU}$ and $\mathcal{SAU} = u\mathcal{PAU} \cup d\mathcal{PAU}$. These serve largely to aid in our exposition by indicating the intended type of a particular value.



(a). Examples of user setup and device registration. Here, $reg_{A,i}$, $reg_{B,j}$ and $reg_{B,k}$ are ciphertexts containing a public record of the link between a user and device (to be distributed by the server as required).



(b). Example of a (unidirectional) group messaging session. Here, $add_{A,i}$, $add_{B,j}$ and $add_{B,k}$ are ciphertexts created by the sending party to distribute key material to new recipient members.

Figure 5.1. Examples of how the DOGM algorithms may be combined during protocol execution. These examples are not prescriptive, however: different instantiations may require different scheduling and/or message routing.

Usage. The Gen algorithm initialises the long-term cryptographic identity of a user. New devices can be linked to this identity through the Reg protocol. In principle, device registration is a cooperative sub-protocol between the user (or a device representing them) and the new device. In the case of both WhatsApp and Matrix, this sub-protocol is executed over a secure channel that is setup through an out-of-band procedure. As discussed, we choose not to model this procedure. Rather, the Reg algorithm simulates this sub-protocol as a single algorithm that takes as input the user's secret and public authenticators, which it is able to update, as well as to generate the secret and public authenticators for the new device with the registration ciphertext. The resulting device identities can then be re-used across many group messaging sessions, both within the same group as well as across multiple groups, all of which may run simultaneously.

The Init, Add, Rem, Enc and Dec algorithms manage group messaging sessions and capture a series of unidirectional messaging epochs between one sender and many recipients. A logical group, in the sense that users interact with a "group", can be captured as a composition of these unidirectional channels, one maintained by each participating device. The formalism, mirroring the structure and guarantees of the protocols in our case study, does not require that these sessions construct a consistent logical group. While we include syntactic support for group identifiers in order to support the notion of a logical group, the formalism (and security experiment) do not make use of it.

In order to correctly implement a logical group, clients rely on the server to manage the group membership correctly and honestly. In other words, whenever a user is added to the group, the server must trigger the Add algorithm for every relevant session, and repeat this process for every device that is associated with the user. It follows that there is little in the way of consistency guarantees between sessions: Alice's sending session may have a different view of the group membership than Bob's sending session. This differs from the approach taken in [BCG23], where the security experiment ensures that sessions have a synchronised viewpoint of group membership (in a similar manner to that of the aforementioned BCP01 and CGKA formalisms).

The StateShare algorithms capture how sessions can gossip and share secrets. Specifically, we intend for this algorithm to capture the history sharing functionality of Matrix and WhatsApp.

Figure 5.1 depicts how the DOGM algorithms can be used to (a) manage long-term cryptographic identities, and (b) capture messaging sessions.

5.2.2 Security

As discussed, we aim to determine the security guarantees of messages sent within groups. We briefly intuit what we mean by this, which we follow with a formal definition in the form of a security experiment.

Security goals. To start, the protocols in our case studies provide no guarantees regarding shared views of group membership. Our security goals reflect this. Intuitively, we wish to capture the ability of the protocol to protect the contents of messages from everyone but the intended recipient users (as determined from

the perspective of the sending session). Similarly, we wish to capture the ability of the protocol to ensure that any message attributed to a particular session was in fact sent by that session, and that the session is an honest session controlled by the user it claims to represent.

Thus, we consider a message to be authentic if the message contents cannot be modified by anyone other than the sending session, it is correctly attributed to the honest session that sent it, and that session is recognised as being owned by the correct user. Under certain constraints, devices which are not a message's sender may be able to modify its attribution, linked session or contents without breaking the authenticity of the message from the perspective of the model. In WhatsApp, for example, a device that is entrusted to share history may be allowed to "break" the authentication guarantees of the message as an *expected limitation* of the protocol, but only to devices that belong to the same user.

Similarly, a message is confidential if only devices that were part of the group (from the perspective of the message's sending device) can decrypt it. Under certain constraints, other devices may also be able to decrypt messages without breaking the secrecy of the message. For example, if a device is linked under the same user as a device which was part of the session when a message was sent, we would expect them to be able to decrypt such a message through the history sharing feature.

We are primarily interested in the security of ciphertexts used for group messaging. As such, this security model does not *directly* consider the security of ciphertexts used in the process of securing the group messaging ciphertexts. For example, in Matrix, we aim to capture the authentication and confidentiality of the Megolm ciphertexts. We only consider the security of Olm ciphertexts in as much as they provide security for the Megolm protocol. This is captured by the requirement for Dec to only return a non-null plaintext message upon acceptance of an application message. In other words, when Dec is provided with a ciphertext containing only control or signalling information, e.g. a key distribution ciphertext, it will return a null value in the plaintext slot (since the signalling information triggered modifications to the internal state only).

Summary. The experiment starts by initialising a pre-determined number of users. The adversary may then corrupt a selection of users by requesting their long-term secrets. Once this initial stage is complete, the adversary is given complete control of the experiment, where they may trigger the creation of new devices (registered to the user of their choice), create new messaging sessions and trigger the sending and receiving of messages of their choice. Throughout, they may corrupt the long-term device secrets or session state.

We capture message authenticity through a forgery game: if the adversary is able to trick a session into accepting a message that they should not be able to, the adversary wins the experiment. Similarly, we capture message confidentiality using a distinguishing game: when requesting a session to encrypt a plaintext, the adversary can submit two challenge plaintexts. The challenger decides, based on a coin flip at the start of the experiment, whether to return encryptions of the first or second plaintext. These are the *challenge ciphertexts*.

At the end of the experiment, the adversary returns a guess as to whether the challenger was returning the first or second plaintexts during the experiment.

We allow the adversary to corrupt secrets throughout the experiment. As such, we expect them to be able to win the game under some circumstances. For example, if the adversary directly compromises a particular sending session's state, we would expect them to be able to construct messages that matching recipient sessions will accept (until/if PCS is achieved). We track these circumstances using two predicates, DOGM.CONF and DOGM.AUTH, for confidentiality and authenticity, respectively. These are protocol specific and, thus, any security statements need to be made with respect to a particular pair of predicates.

Execution Environment. Consider an instance of the DOGM experiment, $\mathsf{IND\text{-}CCA}^{\mathsf{DOGM}}_{\Lambda}(\mathcal{A})$, played between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. The challenger maintains a set of n_u parties $uid_1, \ldots, uid_{n_u} \in \mathcal{U}$ that represent users interacting with each other via the DOGM protocol. A maximum of n_d devices can be created for each party uid_A, identified by $did_{A,1}, \ldots, did_{A,n_d} \in \mathcal{D}$. Each device can initiate up to n_i sessions of the 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 uid's device did and the collection of per-session variables that it maintains, i.e. the protocol state as specified in Definition 5.1.

The experiment begins with the challenger, $\mathcal{C}$, initialising their book keeping state in addition to randomly sampling a bit $b \leftarrow \{0, 1\}$ which will be used to select challenge ciphertexts. It proceeds to execute $\mathsf{DOGM.Gen}(uid)$ n_u times to generate a public key pair (pk_A, sk_A) for each party $uid \in \{P_1, \ldots, P_{n_u}\}$ and deliver all public-keys pk_A to the adversary, $\mathcal{A}$. The adversary may now issue $\mathcal{O}$-CorruptUser queries (which we describe below). After, we enter the main stage of the experiment, in which the adversary may interact with the challenger via the queries listed below (with the exception of $\mathcal{O}$-CorruptUser). The challenger collects the challenge ciphertexts in a set, $\mathbf{C}$, throughout the experiment.

Eventually, the adversary terminates and outputs a guess b' of the challenger bit b. The adversary wins the DOGM security experiment if $b' = b$, and the confidentiality predicate DOGM.CONF is satisfied. If the confidentiality predicate is *not* satisfied, the challenger ignores the adversary's guess and the experiment terminates with a random choice between a win or loss.[1]

The adversary may also win the DOGM security experiment by breaking the authentication of the protocol during the experiment (before they return control to the challenger). Such instances are detected within the $\mathcal{O}$-Decrypt and $\mathcal{O}$-StateShare oracles. In the $\mathcal{O}$-Decrypt oracle, the challenger checks for successful replay attacks and forgeries and, if one is detected and the experiment satisfies the authenticity predicate DOGM.AUTH, terminates the experiment immediately with an output of 1. The challenger performs similar checks within the $\mathcal{O}$-StateShare oracle, albeit to detect the injection of a state sharing message.

[1] We do so in order to ensure that the adversary is not able to force a loss of the experiment and, in doing so, achieve a non-negligible advantage. In other words, we take the penalisation approach to misbehaving adversaries [BCJ+24].

Adversarial Model. We now define how the adversary in the DOGM security experiment may interact with the challenger; in turn, specifying the types of interaction between an attacker and protocol instances. We give the adversary complete control of the communication network; able to modify, inject, delete or delay messages.

We, additionally, allow them to *compromise* secrets at three levels: (a) adaptive compromise of the current session state, (b) adaptive compromise of a device's long-term key material, and (c) non-adaptive compromise of a user's long-term key material. The first models state-compromising attacks, such as temporary device access or the accidental reveal of session backups. The latter two capture key misuse in addition to stronger forms of state-compromising attacks. This enables the model to capture a nuanced understanding of PCS and FS.

In the first stage of the experiment, the adversary is given access to a single oracle.

- The *user corruption* oracle, $\mathcal{O}$-CorruptUser, gives the adversary access to the secret authenticator of the user uid_A. Providing the user exists, i.e. $0 \leq A < n_u - 1$, the challenger returns $usau_A$ to the adversary.

$$\mathcal{O}\text{-CorruptUser}(A) \rightarrow \{usau_A, \perp\}$$

This captures static compromise of the user's long-term key material. Once complete, the adversary returns control to the challenger, which initiates the second stage of the experiment. In this stage, the adversary is given access to the following oracles.

- The *device creation* oracle, $\mathcal{O}$-Create, allows the adversary to trigger the creation of new devices.

$$\mathcal{O}\text{-Create}(A, i) \xrightarrow{\$} \{dpau_{A,i}, \perp\}$$

- The *session initialisation* oracle, $\mathcal{O}$-Init, allows the adversary to trigger the initialisation of a new session for the device $did_{A,i}$ representing the user uid_A.

$$\mathcal{O}\text{-Init}(A, i, \rho, gid) \xrightarrow{\$} \{(s, gid), (\perp)\}$$

- The *member addition* oracle, $\mathcal{O}$-AddMember, allows the adversary to direct a session, $\pi^s_{A,i}$, to add a device, $did_{B,j}$, representing the party uid_B, to its group messaging session.

$$\mathcal{O}\text{-AddMember}(A, i, s, B, j, c) \xrightarrow{\$} \{c, \perp\}$$

- The *member removal* oracle, $\mathcal{O}$-RemoveMember, allows the adversary to direct a session, $\pi^s_{A,i}$, to remove a device, $did_{B,j}$, representing the party uid_B, from its group messaging session.

$$\mathcal{O}\text{-RemoveMember}(A, i, s, B, j, c) \xrightarrow{\$} \{c, \perp\}$$

Figure 5.2. The security experiment defining the security of DOGM protocols. We leave the enforcement of experiment parameters, such as limits on the number of sessions, implicit.

- The *encryption* oracle, $\mathcal{O}$-Encrypt, allows the adversary to direct a session, $\pi^s_{A,i}$, to encrypt one of two messages, m_0 or m_1, depending on the challenge bit b.

$$\mathcal{O}\text{-Encrypt}(A, i, s, m_0, m_1) \xrightarrow{\$} \{c, \perp\}$$

- The *decryption* oracle, $\mathcal{O}$-Decrypt, allows the adversary to direct a session $\pi^s_{A,i}$ to process a message and receive its output.

$$\mathcal{O}\text{-Decrypt}(A, i, s, c) \rightarrow \{m', \perp\}$$

- The *state sharing* oracle, $\mathcal{O}$-StateShare, allows the adversary to orchestrate state sharing between two sessions, $\pi^s_{A,i}$ and $\pi^r_{B,j}$, with state relating to stage t and (optionally) message z. Here, $\pi^s_{A,i}$ is the executing session and $\pi^r_{B,j}$ the session they are interacting with. The ciphertexts c and c' are used for communication between the two sessions.

$$\mathcal{O}\text{-StateShare}(A, i, s, B, j, t, z, c) \xrightarrow{\$} \{c', \perp\}$$

- The *device corruption* oracle, $\mathcal{O}$-CorruptDevice, gives the adversary access to the current value of the secret device authenticator value, $dsau_{A,i}$. $\mathcal{C}$ returns $dsau_{A,i}$ to $\mathcal{A}$.

$$\mathcal{O}\text{-CorruptDevice}(A, i) \rightarrow \{dsau_{A,i}, \perp\}$$

- The *user corruption* oracle, $\mathcal{O}$-CorruptUser, gives the adversary access to the current session state of the session $\pi^s_{A,i}$.

$$\mathcal{O}\text{-Compromise}(A, i, s) \rightarrow \{\pi^s_{A,i}, \perp\}$$

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 $\mathcal{O}$-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 $c \leftarrow$ DOGM.Enc(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. The same is (analogously) true for WhatsApp.

The DOGM provides two predicates, DOGM.AUTH and DOGM.CONF, that we use to capture trivial wins in the experiment. The DOGM.AUTH predicate is checked before the game accepts an authentication win (as part of a DOGM.Dec call). The DOGM.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 the adversary's behaviour does not follow the rules set out by the authentication predicate, they are not able to win the experiment through an authentication break. If the adversary's behaviour does not follow the rules set out by the

confidentiality predicate, we disallow the adversary from guessing the challenge bit.

This ensures the adversary cannot gain an advantage through expected protocol behaviour that results from the additional power the experiment provides the adversary. 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.

Both predicates must be defined such that, once the predicate has been set to false within an experiment, it cannot be set to true. In other words, it should not be possible for the adversary to be able to "undo" the record of their misbehaviour. For example, if a protocol enables the recovery of confidentiality some period after compromise, the predicate should capture whether a challenge ciphertext was ever requested during a compromised period (it should not set itself back to true once confidentiality has been recovered). The security experiment encourages the specification of such predicates by providing the predicate with an append-only log of the events in the experiment, and having those predicates determine the security of individual messages as addressed within this log.

Together, this leads us to the following definition.

Definition 5.2 (Security of Device-Oriented Group Messaging Protocols)**.** A DOGM protocol, DOGM, is secure with respect to predicates CONF and AUTH if any probabilistic polynomial-time adversary $\mathcal{A}$, limited by the experiment parameterisation Λ, has a negligible decision-advantage in winning the $\mathsf{IND\text{-}CCA}_{\Lambda}^{\mathsf{DOGM}}$ security experiment detailed in Figure 5.2 under the parameterisation $\Lambda = (n_u, n_d, n_i, n_s, n_m)$ where

- n_u is the number of users created at the start of the security experiment,
- n_d is the number of devices that may be registered for each user,
- n_i is the maximum number of sessions each device may participate in,
- n_s is the maximum number of stages or epochs in each session,
- n_m is the maximum number of messages exchanged within each stage.

5.3 Device-Oriented Group Messaging with Device Revocation

In this section, we detail our extension to the DOGM model that captures the device revocation functionality of WhatsApp and, in doing so, we aim to capture the stronger compromise recovery such mechanisms enable. We ask that a DOGM protocol with revocation must additionally implement a Rev

algorithm as well as implement some additional structure in their session state. The Rev algorithm captures the process of a user revoking one of their linked devices, while the additional state exposes the latest version of the multi-device state that the given session has observed for each communicating partner. We then augment the security experiment with the requisite machinery for the challenger to track both (a) the current set of linked devices for a particular user, and (b) the set of linked devices any particular session should calculate for each communicating partner, given the messages they have received.

Revocation. We augment the DOGM model developed in the previous section to allow users to revoke their linked devices via Rev. This allows a user to indicate to their communication partners that they have lost access to a device, and thus their partners should no longer encrypt to that device. Recall that WhatsApp implements device revocation by sharing an updated device list through the server. They additionally notify their communicating partners of such changes using metadata embedded within the Signal pairwise channels (see Section 4.2.1). To do so, we augment public authenticator values to contain a generation, which may be accessed through the '.γ' field.

Tracking changes. The challenger tracks the current generation of a user's multi-device state within the security experiment. When a user is first created through a call to Gen, we initialise their generation to zero. We proceed to increment a user's generation every time they register or revoke a device, i.e. whenever a call to Reg or Rev succeeds. The challenger stores the current generation of each user in the dictionary Γ. We, additionally, expect each session to track the current generation of each communicating partner within their local state, accessible as $\pi.\Gamma$. We leave it to the protocol to propagate changes to users' generations and, similarly, to correctly declare accurate values within $\pi.\Gamma$. We expect each session to correctly enforce the user generation that they are aware of. For example, if Alice revokes device *did* leading to generation γ, we expect any session with the same generation stored for Alice, i.e. with $\pi.\Gamma[A] = \gamma$, not to authenticate messages from *did* as originating from *uid*.

In other words, rather than forcing synchronised views of a user's multi-device state, we take the approach of tracking which version a particular session should have seen (given their communication) and require that the session correctly enforces this to the best of their ability. Take the example in Figure 5.1. Recall that Alice, through the device (A, i), and Bob, through the devices (B, j) and (B, k), are communicating in a messaging session in which (A, i) is the sender and (B, j) and (B, k) are the recipients. If Bob proceeds to revoke device (B, k), when should we expect this to be enforced by the other devices? We expect that sessions accurately enforce the most recent version they have seen. In this case, as soon as (A, i, s) or (B, j, r) receives a ciphertext indicating the change in the Bob's multi-device state, we expect them to update the value of '.$\Gamma[B]$' appropriately. From this point onwards, the challenger expects the respective session to accurately enforce these changes.

5.3.1 Syntax

We now define the syntax of device-oriented group messaging protocols with revocation.

Definition 5.3 (Device-Oriented Group Messaging with Revocation)**.** A Device-Oriented Group Messaging with Revocation (DOGMR) protocol extends the functionality of a DOGM protocol (see Definition 5.1) with a device revocation algorithm, Rev, and an additional field in its session state, '.Γ', that holds the latest version of its communicating partners multi-device state that it is aware of.

The Gen, Reg, Init, Add, Rem, Enc, Dec and StateShare algorithms follow the same syntax as their counterparts in the DOGM definition. We define Rev as follows.

- The *revocation* algorithm, Rev, enables a user to revoke one of their linked devices.

  $$upau, usau, dpau, dsau, c \leftarrow\$ \ \mathsf{Rev}(uid, did, upau, usau, dpau, dsau)$$

  It takes as input a user identifier, $uid \in \mathcal{U}$, the identifier of the device to be removed, $did \in \mathcal{D}$, a user public and secret authenticator, $upau \in \mathcal{PAU}$ and $usau \in \mathcal{SAU}$, and a device public and secret authenticator, $dsau \in \mathcal{SAU}$ and $dpau \in \mathcal{SAU}$, then returns updated public and secret authenticator values for the user, $upau \in \mathcal{PAU}$ and $usau \in \mathcal{SAU}$, updated public and secret authenticator values for the device, $dpau \in \mathcal{PAU}$ and $dsau \in \mathcal{SAU}$, as well as an optional ciphertext, $c \in \mathcal{C}$.

We define these algorithms with respect to

- a set of user identifiers, $\mathcal{U} \subseteq \{0,1\}^*$,
- a set of device identifiers, $\mathcal{D} \subseteq \{0,1\}^*$,
- a set of group identifiers, $\mathcal{G} \subseteq \{0,1\}^*$,
- a set of plaintext messages, $\mathcal{M} \subseteq \{0,1\}^*$,
- a set of ciphertexts, $\mathcal{C} \subseteq \{0,1\}^*$,
- a set of secret authentication values, $\mathcal{SAU} \subseteq \{0,1\}^* \times \mathbb{N}$,
- a set of public authentication values, $\mathcal{PAU} \subseteq \{0,1\}^* \times \mathbb{N}$, and
- a set of possible session states, $\mathcal{ST}$.

The space of secret and public authenticator values are augmented to include a generation field, '.γ'. Similarly, we require that each session state $\pi \in \mathcal{ST}$ includes a '.Γ' field that maps from user identifier to the latest known authenticator generation for that user, $\Gamma = \mathsf{Map}\{uid : \gamma\} \in \mathcal{U} \times \mathbb{N}$.

5.3.2 Security

We define security similarly to that of DOGM protocols without revocation (as in Section 5.2.2). Notably, we provide the adversary with the additional ability to trigger a device to be revoked.

Figure 5.3. The security experiment defining the security of DOGM protocols with revocation (DOGMAR). We highlight changes from the original DOGM security experiment (see Figure 5.2).

- The *device revocation* oracle, $\mathcal{O}$-Revoke, allows the adversary to direct party A to revoke device (A, i). This, in turn, triggers the challenger to execute the revocation algorithm, DOGM.Rev, on behalf of party A and return the resulting ciphertext, if output, to the adversary.

$$\mathcal{O}\text{-Revoke}(A, i) \mapsto \{c, \perp\}$$

Once a corrupted device has been revoked, we expect honest sessions to start (a) rejecting messages sent by the device, and (b) sending ciphertexts that the revoked device cannot decrypt. Thus, the addition of the device revocation oracle provides two new lines of attack for the adversary. For a device that has been corrupted and revoked the adversary may attempt to (a) cause an honest session to accept a message from the revoked device, or (b) cause an honest session to send it an inbound session allowing decryption of the challenge ciphertext.

In some cases, such breaks are the result of the intended behaviour of a session. For example, in the case where a revoked device remains in the intended recipient list of a sending session, or if a recipient session accepts a message from a revoked device. The challenger detects these breaks through explicit checks in the relevant oracle. Other cases are less direct, however. For example, if the revocation of a device does not trigger the appropriate key rotation from the sending session. We capture these breaks through the existing mechanisms for detecting authentication and confidentiality breaks, and rely on the security predicates to encode the appropriate behaviour for the construction.

Since device management is mediated through challenger-provided oracles, it has complete and correct knowledge of the linked devices for any particular (honest) user at any point in time. The challenger tracks the current status of each linked device by adding the appropriate entry to the experiment log.

1) As in the DOGM experiment, the registration of a new device, say (A, i), results in a log entry of the form '$(\texttt{create}, A, i, upau_A, usau_A, dpau_{A,i}, dsau_{A,i}, c)$' being added to the experiment log.

2) When a device, say (A, i), is revoked, the challenger adds a log entry of the form '$(\texttt{revoke}, A, i, \gamma)$' to indicate that the previously linked device representing user A was revoked in generation γ of the user's authenticator. The value of γ is taken from the updated private user authenticator output by the challenger's execution of DOGM.Rev.

3) The challenger disallows the registration of previously revoked devices.

Thus, the challenger can determine the current status of a linked device by querying the experiment log. Of course, while the challenger has complete and correct knowledge of the linked devices for a user, individual sessions may not. The revocation predicate, $^*\text{Revoked}?(\mathbf{L}, A, i, s, B, j) \rightarrow \{\mathbf{true}, \mathbf{false}\}$ returns **true** when the session $\pi^s_{A,i}$ should view the device (B, j) as having been revoked and **false** when not. The predicate checks the experiment log to determine whether the device (B, j) has been revoked and, if so, at what user authenticator generation. It then checks whether the session, $\pi^s_{A,i}$, has a sufficiently up-to-date awareness of the user authenticator for B, i.e. whether $\pi^s_{A,i}.\Gamma[B] \geq \gamma$.

Finally, it needs to check for compromise of user B's secret authenticator. If the adversary has gained access to $usau_B$ through a call to $\mathcal{O}$-CorruptUser(B), then we expect neither the challenger nor honest sessions to be able to accurately track the linked devices of that user. Through this predicate, the challenger is able to track when an authentication or confidentiality break is legitimate. See Figure 5.3 for our implementation of the *Revoked? predicate.

We proceed to define security as follows.

Definition 5.4 (Security of Device-Oriented Group Messaging with Revocation)**.** A DOGM protocol with revocation, DOGMAR, is secure with respect to predicates CONF and AUTH if any probabilistic polynomial-time adversary $\mathcal{A}$, limited by the experiment parameterisation Λ, has a negligible decision-advantage in winning the IND-CCA$^{\text{DOGMAR}}_{\Lambda}$ security experiment detailed in Figure 5.3 under the parameterisation $\Lambda = (n_u, n_d, n_i, n_s, n_m)$ where

- n_u is the number of users created at the start of the security experiment,
- n_d is the number of devices that may be registered for each user,
- n_i is the maximum number of sessions each device may participate in,
- n_s is the maximum number of stages or epochs in each session,
- n_m is the maximum number of messages exchanged within each stage.

5.4 Components of Multi-Device Group Messaging

In this section, we formalise a few components of multi-device group messaging identified during our case studies.

5.4.1 Device Management with Public Key Orbits

In both Matrix and WhatsApp, we have seen the construction and maintenance of a key hierarchy for each user. Here, each user maintains a group of devices which they delegate to represent them within the protocol. In the case of WhatsApp, such devices may also be revoked.

We formalise *public key orbits* which bind together one *primary* key pair (pk_p, sk_p) with several *companion* key pairs (pk_c, sk_c). This captures WhatsApp's device management sub-protocol, MD, where one primary device authenticates possibly several companion devices which also attest their membership to the group of devices orchestrated by the primary device.

Related work. This formalism is intended to capture the device management features of Matrix and WhatsApp. As such, it covers similar ground to that of Keybase's sigchain [Key22] and the related constructions used by Zoom [BBC+23] and ELEKTRA [LCG+23]. The latter presents a formalism for sigchains. Our formalism differs in a couple of ways. First, public key orbits encode a hierarchy of keys, with the primary key being an authority over which devices may be linked (or unlinked). The sigchains used by the aforementioned protocols are

more flexible. Secondly, WhatsApp's reuse of the identity key across multiple sub-protocols necessitates the provision of a restricted signing oracle (and to prove security of the protocol in the face of such oracles).

We define the syntax of a public key orbit as follows.

Definition 5.5 (Public Key Orbit)**.** A PO scheme is a five-tuple of algorithms, (Setup, Attract, Repel, Refresh, Orbit), which we define with respect to a digital signature scheme DS = (Gen, Sign, Verify).

- The *setup* algorithm, Setup, captures the creation of a new key hierarchy. It takes in a security parameter and outputs as digital signature (private, public) key pair (sk_p, pk_p), an initial state orb where $orb.\gamma = 0$ and a predicate Reject? accepting a message m and outputting a bit $\{0, 1\}$. This is a probabilistic algorithm.

$$sk_p, pk_p, orb, \mathsf{Reject?} \leftarrow\$\ \mathsf{Setup}(1^\lambda)$$

- The *attraction* algorithm, Attract, links a new key to an existing key hierarchy (i.e. it attracts the key into its orbit). It takes in a signing key sk, a verification key pk and a state orb and outputs a new state orb' or $\perp$. The signing key here is either sk_p or a signing key output by DS.Gen. This is a probabilistic algorithm.

$$orb' \leftarrow\$\ \mathsf{Attract}(sk, pk, orb)$$

- The *repelling* algorithm, Repel, unlinks a key from a key hierarchy (i.e. it repels the key from its orbit). It takes in the key hierarchy signing key sk_p, the verification key to unlink pk, and a state orb, before outputting a new state orb' or $\perp$. This is a probabilistic algorithm.

$$orb' \leftarrow\$\ \mathsf{Repel}(sk_p, pk, orb)$$

- The *refresh* algorithm, Refresh, captures the process of reasserting the set of currently linked devices, without making any changes. It takes in the signing key sk_p, a state orb and outputs a new state orb' or $\perp$. This is a probabilistic algorithm.

$$orb' \leftarrow\$\ \mathsf{Refresh}(sk_p, orb)$$

- The *orbit calculation* algorithm, Orbit, takes in the key hierarchy verification key pk_p, a state orb and a reference generation i. The algorithm calculates the set of linked keys representing pk_p in the generation i, which it returns as a set of verification keys $\mathcal{P}$ (or $\perp$, to indicate failure). This is a deterministic algorithm.

$$\mathcal{P} \leftarrow\$\ \mathsf{Orbit}(pk_p, orb, i)$$

Remark 5.6. *Note that this definition does not permit companion keys to repel themselves from a primary key, i.e.* (sk_c, pk_c) *cannot repel itself from* (sk_p, pk_p)*.*

We define a series of correctness properties.

Definition 5.7 (Correctness of Public Key Orbits)**.** Let $\mathsf{PO} = (\mathsf{Setup}, \mathsf{Attract}, \mathsf{Repel}, \mathsf{Refresh}, \mathsf{Orbit})$; and $z = \mathsf{poly}(\lambda)$ be the maximum value of any $orb.\gamma$ observed in any calls to functions of PO. Let $(sk_c, pk_c) \leftarrow \mathsf{DS.Gen}(1^\lambda)$. Let orb_i imply that $orb_i.\gamma = i$. We say PO is correct, if $\forall\, (sk_p, pk_p), orb, \mathsf{Reject?} \leftarrow \mathsf{Setup}(1^\lambda)$, we have

1. **Operations with sk_p increase generation.** For $\mathsf{op} \in \{\mathsf{Attract}, \mathsf{Repel}, \mathsf{Refresh}\}$ we have $orb'.\gamma > orb.\gamma$ for $orb' \leftarrow \mathsf{op}(sk_p, \ldots, orb)$.

2. **Index consistency.** Let orb be the output of any series of calls to $\mathsf{Attract}, \mathsf{Repel}, \mathsf{Refresh}$ following a call to Setup. Then for $\mathcal{P} \leftarrow \mathsf{Orbit}(pk_p, orb, k)$ we have $\mathcal{P} \neq \bot$ if $orb \neq \bot$ and $k = orb.\gamma$.

3. **Attracting without repelling includes.** Let $0 \leq i < z$ be some index such that $\mathsf{Attract}(sk_p, pk_c, orb_i)$ was called; $i < j < z$ be the smallest index such that $\mathsf{Repel}(sk_p, pk_c, orb_j)$ was called, or $j = z$ if no such call occurred; and i' be the smallest index i' such that $\mathsf{Attract}(sk_c, pk_p, orb_{i'})$ was called. It holds that $\forall\ k$, $\mathsf{max}(i, i') < k \leq j$

$$pk_c \in \mathsf{Orbit}(pk_p, orb_k, k).$$

4. **Repelling excludes.** Let i be the smallest index such that $\mathsf{Repel}(sk_p, pk_c, orb_i)$ or $i = 0$ if no such call happened. Let $j > i$ be the smallest index such that both $\mathsf{Attract}(sk_p, pk_c, orb)$ and $\mathsf{Attract}(sk_c, pk_p, orb')$ have been called for some $orb.\gamma \geq j$. It holds that

$$pk_c \notin \mathsf{Orbit}(pk_p, orb_k, k)\ \forall i < k \leq j.$$

5. **Refresh does not change list.** If $orb' \leftarrow \mathsf{Refresh}(sk_p, orb)$ was called then for $\mathcal{P} \leftarrow \mathsf{Orbit}(sk_p, orb, orb.\gamma)$ and $\mathcal{P}' \leftarrow \mathsf{Orbit}(pk_p, orb', orb'.\gamma)$ we have $\mathcal{P} = \mathcal{P}'$.

Intuitively, we define the security of a public key orbit protocol as the inability of an attacker to produce a PO state orb that (1) verifies and (2) outputs a device list containing devices not produced by honest calls to $\mathsf{Attract}$ or were followed by an Repel call.

We insist on this guarantee even in the presence of a signing oracle that will sign any message except those specified by $\mathsf{Reject?}$.[2] Finally, we demand that even if the primary signing key sk_p is compromised, the holder cannot "forcefully adopt" some pk_c. We capture compromise of the primary signing key through the $\mathcal{O}$-$\mathsf{Compromise}$ oracle.

We proceed to formalise this security notion as follows. Note that we call this notion "weak", denoted $w\mathsf{PO}$, to highlight that stronger notions are possible and might be desirable. In particular, our definition allows the adversary to drop devices from the orbit without winning the game, i.e. we rule this out as a *trivial win*. This notion captures WhatsApp's multi-device security guarantees.

[2] This will allow us to model the lack of domain separation at the level of signing keys in WhatsApp.

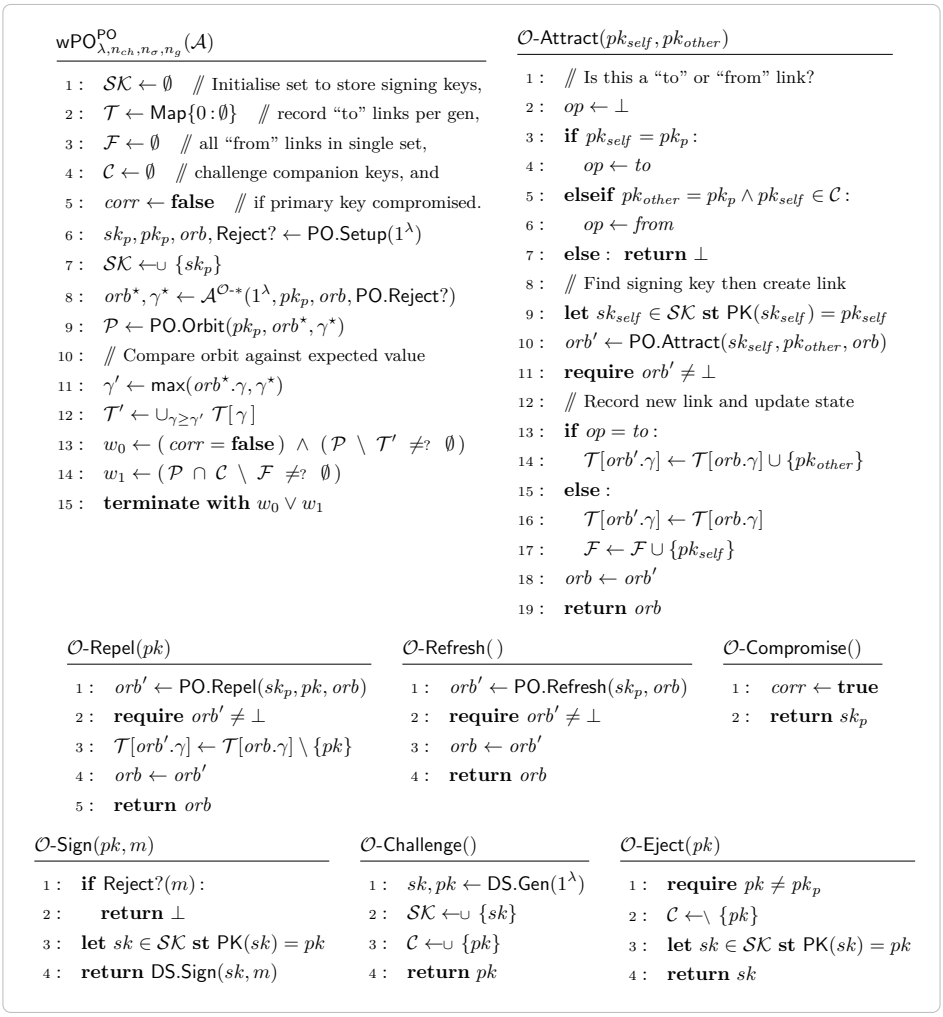


Figure 5.4. Security experiment capturing Weak Public Key Orbit security.

Definition 5.8 (Weak Public Key Orbit Security)**.** Let $\mathsf{PO} = (\mathsf{Setup}, \mathsf{Attract}, \mathsf{Repel}, \mathsf{Refresh}, \mathsf{Orbit})$. The search-advantage of an adversary $\mathcal{A}$ breaking $w\mathsf{PO}$ security is defined as $\mathsf{Adv}^{\mathsf{wPO}}_{\mathsf{PO},\mathcal{A}}(\Lambda_{w\mathsf{PO}}) \coloneqq \Pr\left[\mathsf{wPO}^{\mathsf{PO}}_{\Lambda_{w\mathsf{PO}}}(\mathcal{A})\right]$ where $\mathsf{wPO}^{\mathsf{PO}}_{\Lambda_{w\mathsf{PO}}}$ is defined in Figure 5.4 under parameters $\Lambda_{w\mathsf{PO}} = (\lambda, n_{ch}, n_\sigma, n_g)$.

5.4.2 Pairwise Channels with Session Management

Both Matrix and WhatsApp make use of a collection of two-party channels for communication between pairs of devices. For example, WhatsApp allows up to 40 simultaneously active two-party sessions between any two pairs of devices. In the case of Matrix, these channels are (intended to be) used solely for the purpose of control messages (i.e. they do not transport application messages). In contrast, WhatsApp uses these for messaging in certain circumstances. In this section, we introduce a formalism to capture pairwise channels with session management.

Related Work. WhatsApp's use of multiple two-party channels in parallel mirrors Signal's use of the Sesame session management protocol, the implications of which have previously been explored in [CFKN20, CJN23]. The formalism we present here follows the general approach of [CJN23], albeit in the computational setting. Additionally, the resulting formalism is not too dissimilar from the MSKE variant introduced in [CCD+20], since the security experiment also captures multiple parallel sessions executing in parallel. Indeed, the core differentiator between the formalism we present here and the MSKE formalism is that we move the session management responsibilities from the challenger to the protocol itself. We will later use an existing analysis of Signal two-party channels within this formalism for our security analysis of WhatsApp's pairwise channels. As such, we detail the MSKE formalism of [CCD+20] and discuss how the two formalisms differ in Section 6.2.5.

We start by defining the syntax of such protocols.

Definition 5.9 (Pairwise Channels)**.** A PAIR scheme for secure pairwise channels is a three-tuple of algorithms, (Init, Enc, Dec) and a three-tuple of *state extractor* algorithms, (IDENTITY, SHARED, SESSION).

1) The *initialisation* algorithm, PAIR.Init, takes in a security parameter and outputs linked private and public authenticators, and participant information to be distributed publicly.

$$sk, pk, info \leftarrow\$ \ \mathsf{PAIR.Init}(1^\lambda)$$

2) The *sending* algorithm, PAIR.Enc, takes in the private state of the sender, the public identity of the recipient, their public information and a plaintext message and outputs an updated private sender state, an identifier for the session that was used, an identifier for the message and ciphertext.

$$sk_i, sid, z, c \leftarrow\$ \ \mathsf{PAIR.Enc}(sk_i, pk_j, info_j, m)$$

3) The *receiving* algorithm, PAIR.Dec, takes in the private state of the recipient, the public identity of the sender, their public information, an identifier for the session that was used, an identifier for the message and a ciphertext and outputs an updated private recipient state and the resulting plaintext.

$$sk_i, m \leftarrow \mathsf{PAIR.Dec}(sk_i, pk_j, info_j, sid, z, c)$$

The *state extractor* algorithms specify the varying types of secrets kept within the private state and how it may be accessed.

1) PAIR.IDENTITY takes in a private state and outputs the long-term identity secrets it contains: $ident\text{-}sk \leftarrow \mathsf{PAIR.IDENTITY}(sk)$.

2) PAIR.SHARED takes in a private state and outputs any secret state that is shared across multiple sessions: $share\text{-}sk \leftarrow \mathsf{PAIR.SHARED}(sk)$.

3) PAIR.SESSION takes in a private state and session identifier and outputs the current state of the specified session: $sess\text{-}sk \leftarrow \mathsf{PAIR.SESSION}(sk, sid)$.

Throughout the above we have: $sk, pk, info, sid, z, m, c \in \{0,1\}^*$.

Broadly, we consider a PAIR scheme to be correct if the recipient of a ciphertext always decrypts to the plaintext that was provided by the sender. Since these channels are stateful and progress over time through cooperation between the two parties, it is not immediately clear which particular values of a client's state should successfully decrypt which messages. We avoid this problem in the correctness definition by loosening the requirements: matching plaintexts are only required when the decryption succeeds. We make no correctness requirements for the state extractor algorithms. Although, meaningful state extractor algorithms are essential to capturing a meaningful understanding of the security guarantees a protocol provides against temporal compromise.

Definition 5.10 (Correctness of Pairwise Channels)**.** A scheme $\mathsf{PAIR} = (\mathsf{Init}, \mathsf{Enc}, \mathsf{Dec})$ is correct if, for all $sid, z, m, c \in \{0,1\}^*$,

$$\begin{aligned}\big(\ &\mathsf{Enc}(sk_i, pk_j, info_j, m) \mapsto (sid, z, c)\\ &\wedge\ \mathsf{Dec}(sk_j, pk_i, info_i, sid, z, c) \mapsto (\,\cdot\,, m')\\ &\wedge\ m' \neq \bot\ \big) \implies m = m'\end{aligned}$$

provided there exists ${sk_i}^*$, pk_i, sk_i, ${sk_j}^*$, pk_j and sk_j where

1) (sk_i^*, pk_i) are the output of a Init call and there exists a polynomial-step combination of Enc and Dec calls through which sk_i can be derived from ${sk_i}^*$, and

2) $({sk_j}^*, pk_j)$ are the output of a call to Init and there exists a polynomial-step combination Enc and Dec calls through which sk_j can be derived from ${sk_j}^*$.

Intuitively, we expect secure pairwise channels to provide confidentiality for messages as well as to guarantee their integrity and authenticity: it should not be possible for an adversary to modify a message or impersonate another participant. Additionally, we expect the protocol to be able to recover these security properties after the compromise of secret state, provided certain conditions have been met. The protocol-specific state extractor algorithms define the varying types of compromise, while these conditions are codified by security predicates. Specifically, the security definition utilises the state extractor algorithms to leak the correct information to the adversary under the different categories of compromise, while the security predicates codify when we would expect security to apply (given such compromises). Thus, the state extractor algorithms and security predicates work in tandem to define the expected security properties of a particular scheme for secure pairwise channels.

The security experiment captures confidentiality through the mechanism of challenge ciphertexts and authentication through a decryption oracle (triggering an immediate win). In both cases, we mediate the adversary's ability to win by checking the appropriate security predicate. To bootstrap the trust between

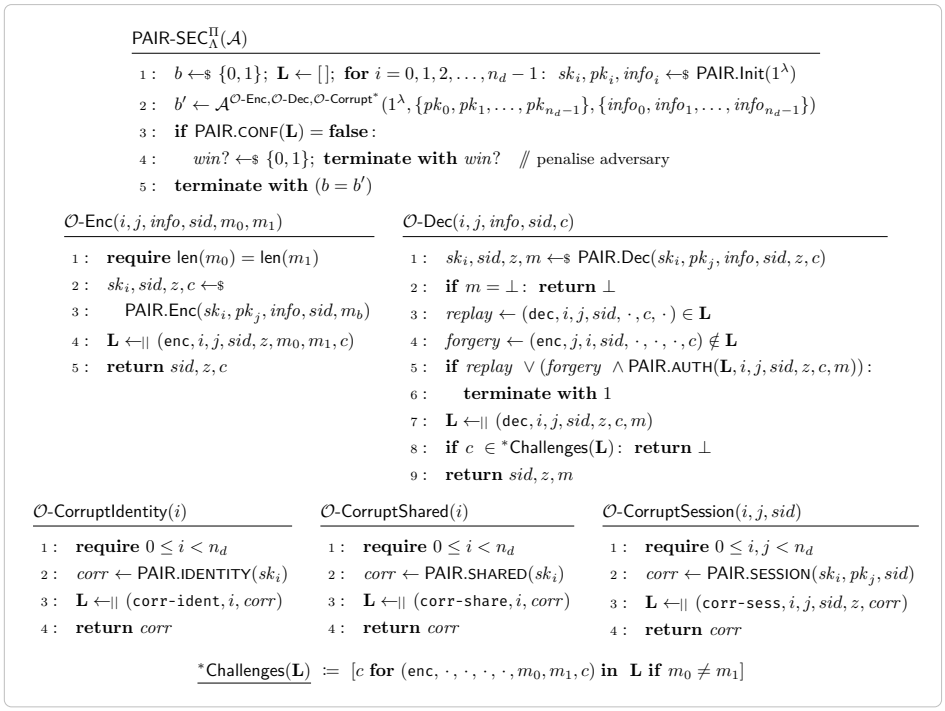


Figure 5.5. The security of pairwise channels.

participants, the challenger provides an initial distribution of each participant's public identifier.

Definition 5.11 (Security of Pairwise Channels)**.** A PAIR scheme is PAIR-SEC secure if any probabilistic polynomial-time adversary $\mathcal{A}$, with respect to security predicates PAIR.CONF and PAIR.AUTH and limited by the experiment parameterisation Λ, has a negligible decision-advantage of winning the PAIR-SEC$^{\Pi}_{\Lambda}(\mathcal{A})$ security experiment detailed in Figure 5.5. The experiment is parameterised by $\Lambda = (n_d, n_i, n_m)$ where

- n_d is the number of devices that interact within the experiment,
- n_i is the maximum number of sessions that each device may participate in with a single other device,
- n_m is the maximum number of messages exchanged within each of those sessions.

The *security predicates* codify precisely under what conditions the scheme should provide confidentiality and authentication.

1) PAIR.CONF takes an ordered log of actions within a security experiment and outputs a boolean indicating whether confidentiality holds:

$$conf? \leftarrow \text{PAIR.CONF}(\mathbf{L}).$$

2) $\mathsf{PAIR.AUTH}$ takes an ordered log of actions, the indices specifying a particular message within a security experiment, its ciphertext and the plaintext it decrypted to before outputting a boolean indicating whether authentication holds:

$$auth? \leftarrow \mathsf{PAIR.AUTH}(\mathbf{L}, i, j, sid, z, c, m).$$

Both predicates must be defined such that, once the predicate has been set to false within an experiment, it cannot be set to true. See the security game in Figure 5.5 for the structure of the experiment log, $\mathbf{L}$.

Note that the total number of messages that may be exchanged during the experiment is $N := \frac{1}{2} \cdot n_d \cdot (n_d - 1) \cdot n_i \cdot n_m$.

5.4.3 Group Messaging with Ratcheted Symmetric Signcryption

In this section we introduce a new variant of *symmetric signcryption*, capturing a single epoch of a Megolm or Sender Keys session, which we name *ratcheted symmetric signcryption* (RSS).

Related work. Closest to our work is the symmetric signcryption (SS) primitive defined by [JKS24]. On a high-level the changes compared to [JKS24] are as follows. First, the [JKS24] definition allows for multiple parallel groups, and multiple parallel users within a single group that share the same symmetric state but sign with different keys. This requires capturing both user and group identifiers within the formalism. In contrast, the constructions we study utilise a separate symmetric key for each sender. Thus, we are able to simplify our definition by focusing on a single unidirectional channel. This also simplifies the inputs to the primitive and more closely aligns with our setting. Second, the [JKS24] definition does not update their symmetric keys, and thus does not capture forward secrecy. We modify the formalism of the primitive to allow for the algorithms to output a symmetric state that ratchets forward, and update their OAE notion to capture forward secrecy. The FS-GAEAD formalism introduced in [ACDT21] is similar to ours in that it captures the ratcheting nature of a single epoch of secure group messaging. Although it does not capture the asymmetric nature of authentication present in the constructions we study here. As such, the formalism we present can be seen as capturing the natural combination of the symmetric ratchet captured in the FS-GAEAD primitive and the symmetric signcryption of the SS primitive.

We now turn to formalising our primitive.

Definition 5.12 (Ratcheted Symmetric Signcryption)**.** A ratcheted symmetric signcryption protocol is a tuple of algorithms $\mathsf{RSS} = (\mathsf{Gen}, \mathsf{Signcrypt}, \mathsf{Unsigncrypt})$.

1) The *key generation* algorithm, Gen, takes as input a security parameter λ and outputs an updatable sending state $st_s \in \mathcal{ST}$ and some updatable receiving state $st_r \in \mathcal{ST}$.

$$(st_s, st_r) \leftarrow\!\!\$\ \mathsf{Gen}(1^\lambda)$$

2) The *signcryption* algorithm, $\mathsf{Signcrypt}$, takes as input a sending state $st_s \in \mathcal{ST}$, an associated data field $ad \in \mathcal{AD}$, and a plaintext message $m \in \mathcal{M}$, and outputs an updated sending state $st'_s \in \mathcal{ST}$ and a ciphertext $c \in \mathcal{C}$.

$$(st'_s, c) \leftarrow\!\$\ \mathsf{Signcrypt}(st_s, m, ad)$$

3) The *unsigncryption* algorithm, $\mathsf{Unsigncrypt}$, takes as input a receiving state $st_r \in \mathcal{ST}$, a ciphertext message $c \in \mathcal{C}$ and an associated data field $ad \in \mathcal{AD}$ and outputs an updated receiving state $st'_r \in \mathcal{ST}$ and a plaintext message $m \in \mathcal{M}$ or a special failure symbol $\perp$.

$$(st'_r, m) \leftarrow \mathsf{Unsigncrypt}(st_r, c, ad)$$

These algorithms are defined with respect to a message space, $\mathcal{M}$, associated data space, $\mathcal{AD}$, ciphertext space, $\mathcal{C}$, and session state space $\mathcal{ST}$.

We define correctness as follows.

Definition 5.13 (Correctness of Ratcheted Symmetric Signcryption)**.** A ratcheted symmetric signcryption scheme, RSS, is correct if $\forall\ st_s^{(0)}, st_r^{(0)}$ from which $\mathsf{Gen}(1^\lambda) \to (st_s^{(0)}, st_r^{(0)})$, and $m^{(i)} \in \mathcal{M}$, $ad^{(i)} \in \mathcal{AD}$, then

$$st_r^{(j)}, m^{(i)} \leftarrow\!\$\ \mathsf{Unsigncrypt}(st_r^{(j-1)}, \mathsf{Signcrypt}(st_s^{(i)}, m^{(i)}, ad^{(i)}), ad^{(i)})$$

iff there exists no previous call

$$st_r^{(k)}, m^{(i)} \leftarrow\!\$\ \mathsf{Unsigncrypt}(st_r^{(j-1)}, \mathsf{Signcrypt}(st_s^{(i)}, m^{(i)}, ad^{(i)}), ad^{(i)}).$$

Intuitively, a secure ratcheted symmetric signcryption scheme should enforce the confidentiality messages from those without possession of either the sending or receiving state. Further, we expect the compromise of the current state (be that sending or receiving) maintains the security of historical ciphertexts.

Definition 5.14 (Confidentiality of Ratcheted Symmetric Signcryption)**.** Let RSS be a ratcheted symmetric signcryption scheme. We say that RSS provides ciphertext indistinguishability under chosen-plaintext attack with forward secrecy ($\mathsf{FS\text{-}IND\text{-}CPA}$ security) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible decision-advantage in winning the $\mathsf{FS\text{-}IND\text{-}CPA}^{\mathsf{RSS}}_{\lambda,n_q}(\mathcal{A})$ security experiment detailed in Figure 5.6 (when restricted to at most n_q queries to the send oracle).

For authentication, we expect that only the holder of the sending session state can produce messages that will be accepted by receiving sessions. We target a notion akin to strong existential unforgeability under chosen message attack.

Definition 5.15 (Unforgeability of Ratcheted Symmetric Signcryption)**.** Let RSS be a ratcheted symmetric signcryption scheme. We say that RSS provides strong-unforgeability under chosen-message attack ($\mathsf{SUF\text{-}CMA}$ security) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible search-advantage in winning the $\mathsf{SUF\text{-}CMA}^{\mathsf{RSS}}_{\lambda,n_q}(\mathcal{A})$ security experiment detailed in Figure 5.7 (when restricted to making at most n_q queries to the $\mathcal{O}\text{-}\mathsf{Send}$ oracle).

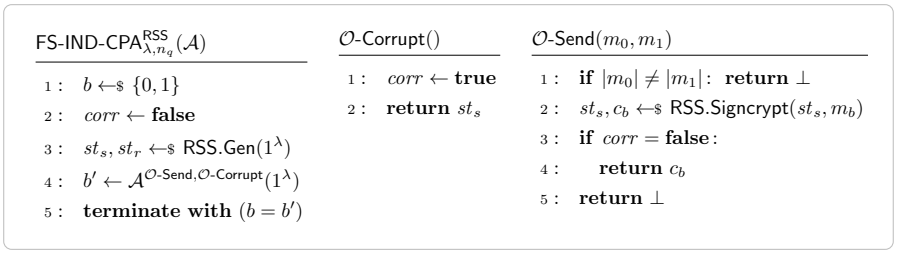


Figure 5.6. Confidentiality of ratcheted symmetric signcryption schemes.

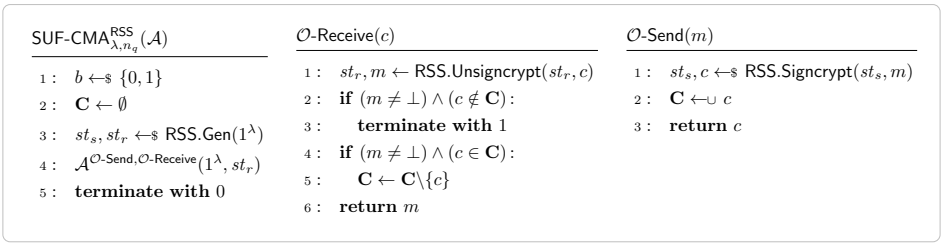


Figure 5.7. Unforgeability of ratcheted symmetric signcryption schemes.

Stronger authentication? While WhatsApp's use of Sender Keys seems to target the two notions above, Matrix' analogous construction (Megolm) seems to target a stronger authentication guarantee. In particular, its use of an authenticated encryption scheme in addition to digital signatures, in an Encrypt-then-MAC-then-Sign configuration, means that compromise of the current state should disallow the forgery of historical messages. This constructions seems to provide a property akin to forward secrecy, albeit for authentication: *forward authenticity*. Such a goal can provide stronger guarantees in contexts where history sharing is prevalent since it can prevent the modification of historical messages given compromise of the current sending state. Something the construction used by WhatsApp does not. However, Matrix' implementation of history sharing renders this particular use case mute.

CHAPTER 6

Security Analysis

We now proceed to analyse the security of the two protocols in our case study. Starting with Matrix, we express its multi-device group messaging functionality within our DOGM formalism, before analysing and proving its security. We do the same with WhatsApp, albeit in the DOGM with revocation formalism. Our analysis of Matrix is monolithic, while our analysis of WhatsApp utilises the components formalised in the prior chapter. In both cases, we discuss how these results can be interpreted in practice, as well as their limitations.

6.1 Matrix

In this section, we analyse the security of multi-device group messaging in Matrix within the device-oriented group messaging formalism.

6.1.1 Contributions

This section includes the following contribution.

Contribution 6.1. We prove the security guarantees of multi-device group messaging in Matrix within the DOGM model, which we specify through the predicates MX-DOGM.CONF and MX-DOGM.AUTH, demonstrating how session management, device management and history sharing interact with the security guarantees of the underlying messaging protocol.

Our analysis is performed in the random oracle model (see Section 2.3.1) and requires a variety of assumptions that we list in the statement of Theorem 6.4.

6.1.2 Scope & Limitations

The accuracy of our results fundamentally relies on the accuracy of our description of Matrix in Section 3.2. Thanks to its detailed public specification and open-source implementation, we can have relatively high confidence in the accuracy of our description. However, the vulnerabilities discovered in Section 3.3 demonstrate that we cannot assume the cryptographic components used in Matrix are perfect. Our description and, in turn, our analysis does not cover the implementations of cryptographic primitives.[1]

Broadly speaking, the scope of our analysis matches the subset of Matrix that naturally fits within the DOGM model. We specify a couple of notable scope limitations below.

Device state compromise. Whilst we do model the security of Olm channels, our analysis does not capture a nuanced understanding of secret state compromise with respect to pairwise channels. That is, rather than considering the implications of leaking long-term key material, medium-term key material, session state or randomness separately, we capture the all-or-nothing compromise of all secret material on the device (with the exception of Megolm sessions).

Since Matrix allows multiple active Olm channels between pairs of clients, access to a device's long-term key material enables attackers to establish new trusted Olm channels, regardless of whether *existing channels* have recovered using the asymmetric ratchet. In other words, much like WhatsApp, the session management layer partially undermines the post-compromise security of the underlying channels. As such, our analysis captures the forward secrecy of Olm channels but does not capture the post-compromise security they provide.

[1] Indeed, since this analysis was initially performed, a variety of issues in Matrix' implementation of these underlying primitives have been found (see [Soa24, Lot24]). Further still, the lack of side-channel resistance in `libolm` has been known since the audit performed by NCC Group in 2016 (see [MB16, Page 9]).

It follows that we are not able to model the exact nature of how temporal Olm state compromise interacts with the security of the Megolm channels above them. The result of the modelling trade-off we make is a simplified set of security predicates that do not capture some situations a message may have been exchanged over a healed channel. See [BCG23] for an analysis of similar interactions between Sender Keys that does capture the post-compromise security of Signal channels. Note, however, that while the analysis in [BCG23] does capture PCS, it does not consider the use of multiple sessions and, as such, comes with its own set of limitations.

Backups and out-of-band verification. This analysis do not cover Matrix' backup solutions for cryptographic key materials (neither Server-side Megolm Backups nor SSSS). Neither does our analysis cover the constituent protocols of the verification framework (such as the SAS protocol Figure 3.16}). This is unfortunate, since these components were the source of several of the vulnerabilities discovered in our case study. We hope that the description we provide in Chapter 3 will enable future work to analyse these components in depth.

The challenger in the DOGM security experiment pre-distributes the initial value of each user's public authenticator to the initial device state (see $\mathcal{O}$-Create in Figure 5.2). This makes the implicit assumption that Matrix clients are able to receive honestly distributed master cross-signing keys for all their communicating partners. Whilst it is possible for this assumption to be met, i.e. when out-of-band verification is enforced, this is not required by the Matrix specification [Mata]. Further, Element does allow for communication with unverified users/devices by default (with warnings in the user interface). An optional setting can disable this behaviour. We note that this is common behaviour among deployments of secure messaging protocols.

6.1.3 Primitives

We collect the cryptographic primitives used by Matrix, discuss the implicit security requirements placed on them, and specify the assumptions we make regarding these primitives in the analysis that follows.

Key derivation. Matrix uses both the HMAC and HKDF functions for various purposes related to key derivation. The security analysis in Section 6.1.5, i.e. Theorem 6.4 and its proof, model both HMAC and HKDF directly as random oracles. We now discuss how these two primitives are used within Matrix, the security assumptions available to us, and justify this particular choice.

Consider Olm, the protocol used to construct pairwise channels. Upon the initialisation of a new session, the result of the Triple Diffie-Hellman key exchange is passed through an HKDF call of the form '$\mathsf{HKDF}(0, ms, \texttt{OLM-RT}, 64\mathsf{B})$'. The definition of KDF security in [Kra10] assumes that a fresh, uniformly random salt is used with each piece of private key material. However, as is common in practice, Matrix uses HKDF in a number of places with a null salt. Since the salt is constant and the input key material is not pseudorandom, we are not able to apply a KDF security assumption. Further still, a PRF assumption is

similarly insufficient since the key material is not uniformly distributed. We follow a similar approach taken by prior work [CCD+20] and model this use of HKDF as a random oracle (see Section 2.3.1).

Upon the creation of a new messaging epoch, the result of a Diffie-Hellman key exchange is passed through the HKDF function, alongside the existing root key as salt, to derive the next root key and a new chain key (initialising the symmetric ratchet for this epoch). In this case, since the given root key is either the output of a random oracle (from the initial key exchange) or a previous HKDF call, we are able to require the salt to be pseudorandom. As such, it should be sufficient to model HDKF as a KDF in this context (see Definition 2.12). While such an assumption may be justified through the results of Krawczyk's analysis [Kra10], we must instead make a stronger assumption: that HKDF may be modelled directly as a random oracle and, in so doing, instantiates a secure KDF. This is necessitated by our modelling of HKDF as a random oracle at the start of Olm sessions (as discussed in the previous paragraph)

Within a single messaging epoch, the chain key is ratcheted forward using a call to the HMAC function, '$\mathsf{HMAC}(ck, \texttt{0x02})$', while the per-message key material is derived from the chain key using a similar call to HMAC, '$\mathsf{HMAC}(ck, \texttt{0x01})$'. Here, the chain key is simply used as a future input key to these same HMAC calls, while the per-message key material is utilised as input to a (constant-salt) HKDF call. As such, we require a primitive that takes a pseudorandom key and arbitrary message as input before outputting a pseudorandom value. Thus, we require HMAC to function as a PRF in this circumstance (see Definition 2.6). Utilising the work of [Bel06, Bel15] and, more recently, [BBGS23], we should be able to assume that HMAC instantiates a secure pseudorandom function under the assumption that the underlying compression function is itself a PRF and that it remains PRF-secure under related-key attack [BK03]. Similar to our modelling of HKDF, we instead assume that HMAC may be modelled directly as a random oracle and, in so doing, instantiates a secure PRF.

We now turn our attention to the Megolm ratchet. It is initialised with the system PRNG that we assume to provide pseudorandom bits of the requisite length. It then proceeds to ratchet its internal state in the schedule described by Figure 3.13. This process uses calls to the HMAC function for which the input key is pseudorandom and accompanied by a domain-separating message, and the output is expected to be pseudorandom. As such we can apply the PRF assumption to these calls. Per-message key material is derived through an HKDF call of the form '$\mathsf{HKDF}(0, ck, \texttt{MG-KDF}, 80\mathsf{B})$'. Here, ck is the aforementioned internal ratchet state (comprising of concatenated outputs of the HMAC calls mentioned above). Thus, we can assume that the initial key material input into the HKDF is pseudorandom. However, since the salt is constant, this usage of HKDF cannot be modelled as a key derivation function. Instead, this usage requires HKDF to implement a secure PRF.

However, we choose not to analyse the Megolm ratchet directly. Rather, our security analysis assumes it to implement a secure FF-PRG (as in Definition 2.43) [DJK22] when the underlying HKDF and HMAC functions are instantiated as random oracles. Nonetheless, the aforementioned assumptions

are prerequisites for the Megolm ratchet to implement a secure FF-PRG and, in turn, for our security analysis of Matrix to hold.

Key exchange. As we have seen, Matrix uses the XDH scheme for Curve25519-based DH key exchange [Ber06] in a variety of places. Most relevant to our analysis is the use of XDH within their construction of secure pairwise channels, both for initial key exchange (through their variant of Triple Diffie-Hellman key exchange) and within the Double Ratchet. Since we model the output of the initial key exchange as being input into a random oracle query (as a result of the discussion above), our security reduction requires access to a (partial) Decisional Diffie-Hellman oracle in order to correctly emulate a random oracle to the inner adversary. We require the Computational Diffie-Hellman assumption to hold despite this. As such, we rely on the Gap Diffie-Hellman (Gap DH) [OP01] assumption to hold on Curve25519 (see Definition 2.38).

Digital signatures. Matrix uses the Ed25519 digital signature scheme, Ed25519 = (Gen, Sign, Verify) [BDL+12], which we assume to provide strong existential unforgeability under chosen message attack (as in Definition 2.35). The work of Brendel, Cremers, Jackson, and Zhao somewhat justifies our assumptions [BCJZ21], albeit with the following caveat. Their proof of EUF-CMA security is against the reference implementation accompanying [BDL+12], while their proof of SUF-CMA security is against a variant standardised by the IETF in [JL17]. Matrix uses neither such variant in their implementations: further work would be required to determine whether this implementation provides SUF-CMA security.

Authenticated encryption with associated data. Matrix uses two schemes for AEAD: first, to protect messages in Olm sessions; and second, to protect messages in Megolm sessions. In both cases, Matrix uses a combination of AES-CBC and HMAC in an Encrypt-then-MAC configuration. In particular, they use AES-CBC with 256 bit keys to encrypt messages that are split into blocks using the PCS7 [Hou09] scheme. Note that Matrix truncates HMAC outputs to 64 bits[2] when calculating authentication tags, contrasting with the recommendations of the HMAC RFC [KBC97]. Despite this, we assume that both Olm and Megolm's constructions are IND$-CPA and EUF-CMA secure AEAD schemes (as in Definitions 2.27 and 2.28, respectively).

6.1.4 Matrix in the DOGM Model

We start by mapping a subset of Matrix' functionality into the DOGM model, capturing user and device cryptographic identity management, group messaging and history sharing. We do so as follows.

Definition 6.1. MX-DOGM is a device-oriented group messaging protocol that instantiates the DOGM formalism with algorithms (Gen, Reg, Init, Add, Rem, Enc, Dec, StateShare) in Figures 6.1 to 6.3.

[2] This issue was noted in an audit [AAD+22] and the developers claim this will be fixed in a future version of the Olm and Megolm specifications.

User and device management. We capture the cross-signing sub-protocol, which provides cryptographic identity management for users and their devices, within Gen and Reg:

- Gen captures the creation of a new cross-signing user identity. We execute the cross-signing initialisation algorithm, CrossSign.Init, with the given user identifier. We set the user's public authenticator value to their master cross-signing public key, while their secret authenticator value stores their secret cross-signing state as well as a reference copy of their public cross-signing state.

- Reg captures the setup and registration of a new device, simulating the interactive protocol between the user and their new device. First, the new device generates their cryptographic identity through a call to Olms.Gen. Next, the user's cross-signing identity signs the new device's public identity through a call to CrossSign.SignDevice.

  We initialise the new device's secret authenticator value with its identifiers, secret keys and cross-signing identity. In contrast, the public authenticator value contains its device key, identity key, and the relevant cross-signing structures to authenticate it.

Recall from Section 5.2 that the challenger distributes the initial public authenticators of every user to each device upon initialisation. In contrast, the public authenticators of devices have their distribution mediated by the adversary. It is important that our analysis captures each user's device composition as it

MX-DOGM (1)

Gen(uid) // Initial user setup

1 : $st_{cs}, \mathfrak{U} \leftarrow\!\$$ CrossSign.Init(uid)
2 : $upau \leftarrow \mathfrak{U}.mpk$
3 : $usau \leftarrow (st_{cs}, \mathfrak{U})$
4 : **return** $upau, usau$

Reg($uid, did, usau, upau$) // Register new device with user

1 : $dpau, dsau, c \leftarrow$ *GenDeviceIdentity(uid, did)
2 : $upau, usau, c \leftarrow$ *UserSignsDevice($uid, did, usau, upau, c$)
3 : $dpau, dsau \leftarrow$ *InitDeviceState($uid, did, dsau, dpau, c$)
4 : **return** $upau, usau, dpau, dsau$

*GenDeviceIdentity(uid, did)

1 : $\mathfrak{D}_{sk}, \mathfrak{D} \leftarrow$ Olms.Gen($uid, did, 10, 1$)
2 : $dsau \leftarrow \mathfrak{D}_{sk}$
3 : $dpau \leftarrow \mathfrak{D}$
4 : **return** $dpau, dsau, (dpau.dpk, dpau.ipk)$

*UserSignsDevice($uid, did, usau, upau, c$)

1 : $dpk, ipk \leftarrow c$
2 : $usau, \mathfrak{R} \leftarrow$ CrossSign.SignDevice(
3 : $usau, did, dpk, ipk$)
4 : **return** $upau, usau, (usau.\mathfrak{U}, \mathfrak{R})$

*InitDeviceState($uid, did, dsau, dpau, c$)

1 : $dpau.\mathfrak{U}, dpau.\mathfrak{R} \leftarrow c$
2 : $dsau' \leftarrow \langle$
3 : $mpk : upau.mpk,\ upk : upau.upk,$
4 : $dpk : dpau.dpk,\ dsk : dsau.\mathfrak{D}_{sk}.dsk,$
5 : $ipk : dpau.ipk,\ isk : dsau.\mathfrak{D}_{sk}.isk,$
6 : $mpks : \mathsf{Map}\{\ uid : mpk\ \},$
7 : $\sigma_\times : \mathsf{Map}\{\ uid : \sigma_\times\ \},$
8 : $st_{olm} : \mathsf{Map}\{\ ipk : st_{olm}\ \},$
9 : $\mathfrak{U} : \mathsf{Map}\{\ uid : dpau.\mathfrak{U}\ \},$
10 : $\mathfrak{D} : \mathsf{Map}\{\ did : dpau.\mathfrak{D}\ \},$
11 : $\mathfrak{R} : \mathsf{Map}\{\ did : dpau.\mathfrak{R}\ \}$
12 : $\rangle$
13 : **return** $dpau, dsau'$

Figure 6.1. Device management in Matrix expressed within the DOGM model.

changes over time and, importantly, how clients manage and verify the structures that define it. As such, we only include the user's master public key, the root of their cross-signing key hierarchy, in the public user authenticator. All other necessary information required to verify a device as belonging to this user must be distributed within that device's public authenticator value.

Note, additionally, that we consider an idealised Matrix implementation that will only interact with trusted devices, i.e. those whose public user authenticators have been pre-distributed.

Group messaging. We capture group messaging at the level of unidirectional Megolm sessions from one device to every other device in the group, as follows.

- Init models the initialisation of a new session representing a particular device in a particular group. Each session is either a sender or a recipient, matching our description of Megolm in Section 3.2.3 relatively closely. In normal operation, we expect a single device to have one sending session and many recipient sessions for each group they are a member of. However, we allow the adversary to orchestrate any variety of configurations outside of this expectation. This emulates their ability to provide inconsistent views of group membership to different sessions.[3]

- Add and Rem model the addition and removal of a device from a session, respectively. In the case of sending session, we capture the appropriate rotation and distribution of Megolm sessions. Recipient sessions only permit a single call to Add in order to set the sending user and device.

- Enc and Dec model sending and receiving messages using an outbound (or inbound) session using (respectively). The decryption algorithm additionally captures the reception of Megolm sessions through Olm ciphertexts, as well as the distribution of cross-signing information.

Note that we store pairwise session state within the secret device authenticator, rather than the DOGM session state. This is because, in practice, each device may have multiple active unidirectional sessions for a single group. In which case, a single set of Olm channels will be used to distribute Megolm sessions across multiple sessions and/or logical groups.

State sharing. We capture the Key Request protocol within the state-sharing functionality of our formalism.

The StateShare algorithm captures how Matrix clients share inbound Megolm session states using the Key Request protocol. It is consists of three distinct cases:

1) When the input is a message consisting of a group signing key and identity key, this triggers the executing session to initiate the Key Request protocol in pursuit of the inbound Megolm session identified by those keys. This

[3] This approach gives the adversary additional strengths. For example, they can create inconsistent views of group membership for sessions within a single device, something that is not possible in practice.

MX-DOGM (2)

Init($uid, did, dsau, \rho \overset{is}{=} \mathsf{send}, gid$)

$\mathfrak{S}_{snd}, \mathfrak{S}_{rcv}, \sigma_{mg} \leftarrow\$ \mathsf{Megolm.Init}()$
$st \leftarrow \langle$
  $\rho : \mathtt{send},\ \alpha : \mathtt{active},$
  $gid : gid,\ uid : uid, did : did,$
  $t : 0,\ z : 0, T : \mathsf{Map}\{\},$
  $CU : [\, uid\,],\ CD : [\,(uid, did)\,],$
  $st_{mg} : \mathfrak{S}_{snd}$
$\rangle$
return $dsau, st, gid$

Add($dsau, st \overset{is}{=} \langle \rho : \mathtt{send} \rangle, uid^\star, did^\star, \varnothing$)

require $(uid^\star, did^\star) \notin st.CD$
require $(uid^\star, did^\star) \notin \{(\bot, \bot), (st.uid, st.did)\}$
require CrossSign.VerifyDevice(
  $dsau, dsau.\mathfrak{R}[uid^\star, did^\star], dsau.\mathfrak{U}[uid^\star]$)
if $uid^\star \neq st.uid$: $st.CU \leftarrow\!\!\| [\, uid^\star\,]$
$st.CD \leftarrow\!\!\| [\,(uid^\star, did^\star)\,]$
$dsau, st, c \leftarrow {}^*\mathsf{ShareSession}(dsau, st, uid^\star, did^\star)$
return $dsau, st, c$

Rem($dsau, st \overset{is}{=} \langle \rho : \mathtt{send} \rangle, uid^\star, did^\star, \varnothing$)

$st.CD \leftarrow\!\!\backslash\ [(uid^\star, did^\star)]$
if $\nexists (uid', did')$ **in** $st.CD$ **st** $uid^\star = uid'$:
  $st.CU \leftarrow\!\!\backslash\ [\, uid^\star\,]$
$st.st_{mg}, \mathfrak{S}_{rcv}, \sigma_{mg} \leftarrow\$ \mathsf{Megolm.Init}()$
$st.t \leftarrow\!\!+\ st.t$
$cs \leftarrow [\,]$
for $(uid', did', dpau')$ **in** $st.CD$:
  $dsau, st, c \leftarrow {}^*\mathsf{ShareSession}(dsau, st, uid', did')$
  $cs \leftarrow\!\!\| [\, c\,]$
return $dsau, st, cs$

${}^*\mathsf{ShareSession}(dsau, st, uid^\star, did^\star)$

$\mathfrak{R}^\star \leftarrow dsau.\mathfrak{R}[uid^\star, did^\star]$
$\langle ver, i, R, gsk, gpk\rangle \leftarrow st.st_{mg}$
$\mathfrak{S}_{rcv} \leftarrow \langle ver, i, R, gpk\rangle$
$\sigma_{mg} \leftarrow \mathsf{Ed25519.Sign}(gsk, \mathfrak{S}_{rcv})$
$m \leftarrow \langle \mathtt{MG}, st.gid, \mathfrak{S}_{rcv}, \sigma_{mg}\rangle$
$dsau, c \leftarrow \mathsf{Olms.Enc}(dsau, \mathfrak{R}^\star.dpk, \mathfrak{R}^\star.ipk, m)$
return $dsau, st, c$

Enc($dsau, st \overset{is}{=} \langle \rho : \mathtt{send} \rangle, m$)

$st.st_{mg}, c \leftarrow \mathsf{Megolm.Enc}(st.st_{mg}, \langle \mathtt{PTXT}, st.gid, m\rangle)$
$c \leftarrow \langle \mathtt{CTXT}, \mathtt{MG}, st.st_{mg}.gpk, dsau.ipk, c\rangle$
$st.z \leftarrow st.st_{mg}.z$
$st.T[st.t, st.z] \leftarrow\!\!\| (st.uid, st.did, m, c)$
return $dsau, st, c$

Dec($dsau, st, C \overset{is}{=} \langle \mathtt{CS\text{-}USR},\ uid^\star, \mathfrak{U}^\star\rangle$)

if $(uid^\star \neq dsau.uid)$
  $\wedge\, (dsau.\mathfrak{U}[uid^\star].mpk = \mathfrak{U}^\star.mpk)$:
  $dsau.\mathfrak{U}[uid^\star] \leftarrow \mathfrak{U}^\star$
return $dsau, st, \bot$

Init($uid, did, dsau, \rho \overset{is}{=} \mathtt{recv}, gid$)

$st \leftarrow \langle$
  $\rho : \mathtt{recv}, \alpha : \mathtt{active},$
  $gid : gid, uid : uid, did : did,$
  $t : \varnothing, z : \varnothing, T : \mathsf{Map}\{\},$
  $CU : [\,\varnothing, uid\,], CD : [\,\varnothing, (uid, did)\,],$
  $st_{mg} : \mathsf{Map}\{\}, mgid : \mathsf{Map}\{\}$
$\rangle$ // $mgid$ assoc local stage index w/ Megolm session
return $dsau, st, gid$

Add($dsau, st \overset{is}{=} \langle \rho : \mathtt{recv} \rangle, uid^\star, did^\star, \varnothing$)

require $st.CD[0] = \varnothing$
require $(uid^\star, did^\star) \notin \{(\bot, \bot), (st.uid, st.did)\}$
require CrossSign.VerifyDevice(
  $dsau, dsau.\mathfrak{R}[uid^\star, did^\star], dsau.\mathfrak{U}[uid^\star]$)
if $uid^\star \neq st.uid$: $st.CU[0] \leftarrow uid^\star$
$st.CD[0] \leftarrow did^\star$
return $dsau, st, \varnothing$

Dec($dsau, st \overset{is}{=} \langle \rho : \mathtt{recv} \rangle, C \overset{is}{=} (c_P, c_U)$)

if $c_P \overset{is}{=} \langle \mathtt{CTXT}, \mathtt{OLM}, c\rangle$:
  $dsau, ipk^\star, m \leftarrow \mathsf{Olms.Dec}(dsau, c)$
  require $m \neq \bot$
  let $\mathfrak{R}^\star \in dsau.\mathfrak{R}$ **st** $\mathfrak{R}^\star.ipk = ipk^\star$
  let $\mathfrak{U}^\star \in dsau.\mathfrak{U}$ **st** $\mathfrak{U}^\star.uid = \mathfrak{R}^\star.uid$
  require CrossSign.VerifyDevice($dsau, \mathfrak{R}^\star, \mathfrak{U}^\star$)
  $\langle uid^\star, uid', dpk^\star, dpk', type, m\rangle \leftarrow m$
  require $uid^\star = \mathfrak{U}^\star.uid \wedge dpk^\star = \mathfrak{R}^\star.dpk$
  require $uid' = st.uid \wedge dpk' = dsau.dpk$
  require $type = \mathtt{MG\text{-}KEY}$
  require $\mathfrak{U}^\star.uid = st.CU[0]$
  require $(\mathfrak{R}^\star.uid, \mathfrak{R}^\star.did) = st.CD[0]$
  $\langle alg, gid, \mathfrak{S}_{rcv}, \sigma_{mg}\rangle \leftarrow m$
  require $alg = \mathtt{MG} \wedge gid = st.gid$
  $\mathfrak{S}^\star_{rcv} \leftarrow \mathsf{Megolm.Recv}(\mathfrak{S}_{rcv}, \sigma_{mg})$
  if $\mathfrak{S}^\star_{rcv} \neq \bot$:
    require $\mathfrak{S}^\star_{rcv}.gpk = c_U.gpk^\star$
    $\mathfrak{S}_{rcv} \leftarrow st.st_{mg}[\mathfrak{S}^\star_{rcv}.gpk]$
    if $\mathfrak{S}_{rcv} \neq \bot \wedge \mathfrak{S}^\star_{rcv}.i < \mathfrak{S}_{rcv}.i$:
      $st.st_{mg}[\mathfrak{S}^\star_{rcv}.gpk] \leftarrow \mathfrak{S}^\star_{rcv}$
      $st.t \leftarrow |st.st_{mg}|$
      $st.mgid[\mathfrak{S}^\star_{rcv}.gpk] \leftarrow st.t$
  $\langle \mathtt{CTXT}, \mathtt{MG}, gpk^\star, c\rangle \leftarrow c_U$
  $\mathfrak{S}_{rcv} \leftarrow st.st_{mg}[gpk^\star]$
  require $\mathfrak{S}_{rcv} \neq \bot$
  $\mathfrak{S}_{rcv}, (type, gid, m) \leftarrow \mathsf{Megolm.Dec}(\mathfrak{S}_{rcv}, c)$
  // (Note changes to $\mathfrak{S}_{rcv}$ not saved in session state)
  require $type = \mathtt{PTXT} \wedge gid = st.gid$
  if $st.mgid[\mathfrak{S}_{rcv}.gpk] = st.t$: $st.z \leftarrow \mathfrak{S}_{rcv}.z$
  $st.T[st.mgid[\mathfrak{S}_{rcv}.gpk], \mathfrak{S}_{rcv}.z] \leftarrow C$
  return $dsau, st, m$

Dec($dsau, st, C \overset{is}{=} \langle \mathtt{CS\text{-}DEV},\ \mathfrak{D}^\star, \mathfrak{R}^\star\rangle$)

if $(\mathfrak{D}^\star.uid = \mathfrak{R}^\star.uid)$
  $\wedge\, (\mathfrak{D}^\star.did = \mathfrak{R}^\star.did \neq dsau.did)$:
  $dsau.\mathfrak{D}[\mathfrak{D}^\star.uid, \mathfrak{D}^\star.did] \leftarrow \mathfrak{D}^\star$
  $dsau.\mathfrak{R}[\mathfrak{R}^\star.uid, \mathfrak{R}^\star.did] \leftarrow \mathfrak{R}^\star$
return $dsau, st, \bot$

Figure 6.2. Messaging in Matrix expressed within the DOGM model.

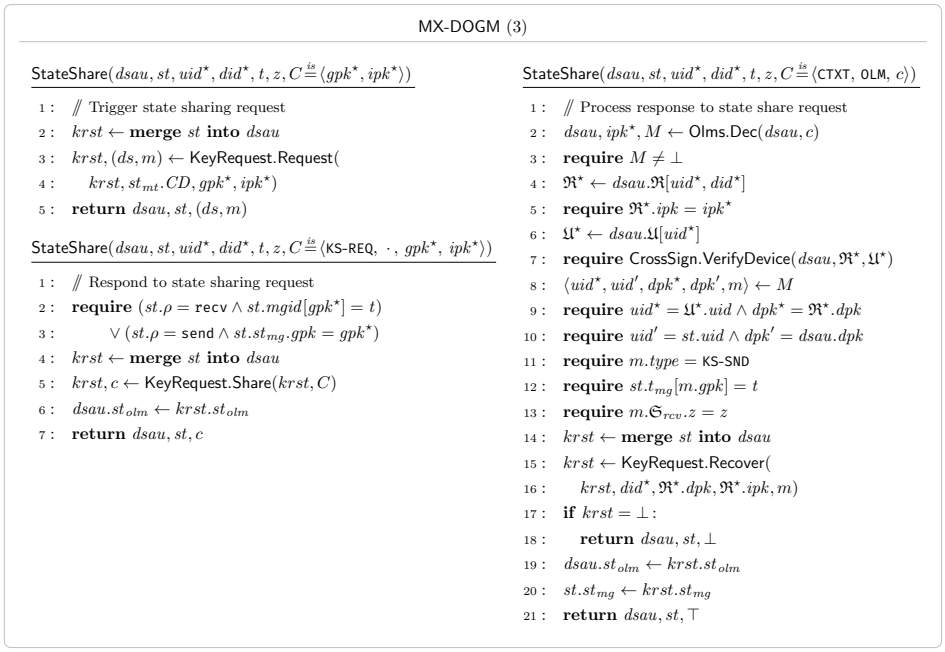


Figure 6.3. State sharing in Matrix expressed within the DOGM model.

case is implemented primarily through a call to the KeyRequest.Request algorithm. Note that this provides the adversary with the ability to trigger and orchestrate key requests between arbitrary sessions. This contrasts with practice, where clients will initiate key requests based on particular events.

2) When the input is a key request message, with type KS-REQ, the executing session attempts to process it as a key request message and, optionally, respond with the appropriate key share. This case is implemented primarily through a call to the KeyRequest.Share algorithm.

3) When the input is an Olm ciphertext, the executing session attempts to process it as a key sharing message. This case is implemented primarily through a call to the KeyRequest.Recover algorithm.

In practice, Matrix clients will accept inbound Megolm sessions at an earlier message index than they requested. However, in order to enable accurate tracking of state sharing, we require that the stage t and message index z input into the state sharing call match that of the session state received.

Upon completion of the state sharing protocol, the recipient of the key request outputs a final ciphertext of the form $\top$ to indicate they have accepted the share, and $\perp$ to indicate that the share failed. This allows the challenger to distinguish between unsuccessful, or incomplete, key request executions and those that succeeded.

6.1.5 Security Analysis

We initiate our security analysis of Matrix in the DOGM model by considering the situations within the security experiment where we expect the protocol to provide security (or not).

In the simplest case, we expect the protocol to provide security as long as no secrets have been compromised or corrupted. And, indeed, we will show that this is the case.[4] In other cases, however, we may still expect the protocol to provide security in the face of such compromises. For example, forward secrecy requires that compromise of the current state does not affect the security of previous messages. The DOGM security experiment simulates such leaks through the $\mathcal{O}$-CorruptUser, $\mathcal{O}$-CorruptDevice and $\mathcal{O}$-Compromise oracles. Of course, we cannot expect the protocol to provide security in the face of any leak. For example, some compromises may directly lead to the adversary being able to decrypt or forge a ciphertext. In such a case, we would call this a *trivial win* and the events that led to it a *trivial attack*. Other cases may be less direct, however. While these attacks may not be trivial, they may nonetheless invalidate the intended security guarantees of the protocol. Either way, both such cases describe situations in which Matrix is not able provide security within the experiment and, with the appropriate interpretation, in practice.

We now aim to determine in what manner compromise of one secret may propagate through the system, or how security may be recovered after a compromise. For example, Matrix' history sharing feature essentially removes all boundaries between a user and their verified devices, enabling security compromises to propagate throughout a user's devices and sessions. Since recipient sessions do not remove old key material, the ratcheting mechanism of sessions fails to provide FS when recipient session state has been compromised. Thanks to the allowance for fresh Olm channels to be initiated using long-term authentication keys, there is a virtual absence of meaningful PCS guarantees: there are very few situations where a group messaging session will be able to recover security after compromise of a participant's Olm secrets. While such cases should no longer be termed a trivial win, they still represent limitations in the design of the protocol whose repercussions should be understood, documented and formalised. We now enumerate the core techniques for propagating compromise or corruptions that result directly from the design of the protocol.

Creating new devices with user secrets. Whenever the adversary has access to a user's long-term secrets, they are able to generate a new device and perform the registration routine such that the new device passes verification (without the challenger's assistance). This device is then able to participate in the various sub-protocols as if they are an honest session, which they *are*, for all intents and purposes. As such, we expect any session for which a compromised user is a participant not to provide confidentiality. Not only is the adversary able to generate new devices, but they are also able to register a new cryptographic identity for an existing device. Such an identity will be accepted by existing

[4] Modulo any issues that are explicitly out of the scope of our analysis. Namely, the distribution of user's public cryptographic identities, server-control of group membership, and the backup scheme.

sessions, overwriting the cryptographic identity of the legitimate device in their store. However, any existing Megolm sessions for a particular stage will continue to provide authentication, until the particular session is rotated out of the recipient client's session store.

Initiating new Olm sessions with long-term device secrets. When the adversary has access to a device's long-term secrets, or has created their own device using access to a user's secrets as just described, they are able to initiate new Olm sessions with any other device in the experiment. As before, such sessions are equivalent to honest sessions, for all intents and purposes, and while it may be possible for an alternatively designed protocol to recover security, it is not possible for Matrix clients to do so. Restated, while any individual Olm channel may be able to recover security after compromise, the ability of the adversary to initialise a new channel in its place undermines such guarantees.

Propagating compromises with the Key Request protocol. The Key Request protocol allows any Olm session that originates from a verified device of the same user, or the same device as the sending session, to share incoming Megolm sessions to the matching recipient sessions. While clients choose when they request keys, the adversary can trigger such a request simply by sending a ciphertext from the soon-to-be-injected session. The target session will fail to process the ciphertext, triggering the session to send out a request. Note that, in our modelling, we provide the adversary with the additional strength of orchestrating the Key Request sub-protocol and, as such, the adversary need not go to these lengths *within the security experiment.* This allows impersonation attacks to be converted into confidentiality attacks, since an adversary can request copies of all inbound Megolm sessions the device they are impersonating has access to (even after their initial distribution). It also allows for an adversary that is able to impersonate a single device, to proceed to impersonate any device to the other devices of that user. This is done by injecting inbound Megolm sessions under the adversary's control into the user's other devices.

More explicitly, consider two cases. In the first case, the adversary is able impersonate a device, (A, j), that is a legitimate recipient of a Megolm session, $\mathfrak{S}_{rcv}$. The adversary may simply initiate an instance of the Key Request protocol between, say, (A, j) and the session (A, i) which has access to $\mathfrak{S}_{rcv}$. Since the adversary is able to initiate new Olm channels that authenticate as (A, j), Alice's other device (A, i) will simply share the session as requested. In the second case, the adversary is impersonating either another device of the target session or the device of the sending session. The attack proceeds by generating a new Megolm session, then triggering the target device to request key material for session gpk in epoch t^* by sending a forged ciphertext which they cannot decrypt nor authenticate: (A, i) then proceeds to request the key material from its trusted peer devices. The adversary may simply reply with their recently generated $\mathfrak{S}_{rcv}$ over Olm. The adversary must take an extra step to inject sessions into a device which is the claimed owner of that session. Here, the adversary calls $\mathcal{O}\text{-}\mathsf{Create}(A, k)$ for some $k \neq i$, which triggers A creating an honest device. The adversary then sends the forgery to (A, k), which requests the missing key material for gpk, which $\mathcal{A}$ provides using the compromised credentials from $\mathcal{O}\text{-}\mathsf{CorruptDevice}(A, i)$. Finally $\mathcal{A}$ sends the forgery to (A, i) which will request gpk, which will be provided by the honest device (A, k).

Lack of forward security in recipient Megolm sessions. As discussed in Chapter 3, clients keep the first copy of all incoming Megolm sessions they receive. As such, the compromise of session state leaks the decryption key for every past message. This is not the case for sending sessions, however, which do ratchet their symmetric state forward.

Having described the general techniques available to an adversary, we proceed to concretely describe how the various types of state compromise or corruption can affect the authentication and confidentiality guarantees of the protocol.

Authentication. In the following, consider the case of a successful forgery within the security experiment. Here, the session $\pi^s_{A,i}$ has accepted a Megolm ciphertext C at stage t and message index z that claims to originate from the device (B, j). However a ciphertext of the value C has not been sent by an honest session belonging to (B, j) for which (A, i) was an intended recipient. In what situations would we expect Matrix clients to permit such forgeries?

1) When the secret authenticator of the recipient user or any of their devices has been compromised, i.e. $\mathcal{O}$-CorruptUser(A) or $\mathcal{O}$-CorruptDevice(A, k) was issued $k \in [n_d]$ (including the case of $k = i$). The adversary may 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 $\pi^r_{B,j}$.

2) When the secret authenticator of the sending user or device has been compromised, i.e. a $\mathcal{O}$-CorruptUser(B) or $\mathcal{O}$-CorruptDevice(B, j) query was issued, before the recipient received the inbound Megolm session. In the first case, where the user's long-term secrets have been corrupted, the adversary can generate a new device (without the challenger's help) and use this device to initiate stage t. In the second case, where device (B, j)'s long-term secrets were corrupted before the initiation of stage t, the adversary can generate a fresh Megolm session, initiate a new Olm channel using ephemeral keys signed with the device's long-term keys and distribute the inbound session to $\pi^s_{A,i}$.

3) When the state of the sending session, $\pi^r_{B,j}$, has been directly compromised while at the appropriate stage, i.e. $\mathcal{O}$-Compromise(B, j, s) was issued when $\pi^r_{B,j}.t$ is partnered to the recipient session's stage t. Since this is an outbound Megolm session, it contains the necessary key material to encrypt, generate an authentication tag and then sign the forged message.

   However, as outbound sessions ratchet the symmetric key material forward after each encryption, access to the current outbound Megolm state alone does not allow the adversary to forge messages with a lower message index. Now, consider cases where the adversary also has access to an earlier copy of the symmetric secrets (e.g. from compromise of the inbound session). They can combine these symmetric secrets with *gsk* to forge ciphertexts from an earlier ratchet state. While it may be possible to capture such a distinction, doing so complicates our proof sufficiently that we choose not to. If we were to do so, we may allow session state compromise for partnered recipient sessions provided it is done so when it holds state for a

message index greater than the message index of the forgery. Instead, we simply require that no such compromise of the outbound Megolm session may occur when it is at stage t.

We codify these cases formally, and in the language of the DOGM experiment, in the definition that follows.

Definition 6.2. An instance of the DOGM security experiment with experiment log $\mathbf{L}$ fulfils the MX-DOGM.AUTH authentication predicate if, when processing a $\mathcal{O}$-Decrypt(A, i, s, c) query for the session, $\pi^s_{A,i}$, with sender (B, j) and ciphertext c at computed stage t and message index z, the following evaluates to true.

$$
\begin{aligned}
&\underline{\textsf{MX-DOGM.AUTH}(\mathbf{L}, A, i, s, t, z, B, j)} \;\coloneqq \\
&\quad \nexists\, (\texttt{corr-user}, A, \cdot)\ \textbf{in}\ \mathbf{L} \\
&\qquad \wedge\ \nexists\, k\ \textbf{st}\ (\texttt{corr-device}, A, k, \cdot)\ \textbf{in}\ \mathbf{L} \quad /\!/\ (1) \\
&\quad \wedge\ \nexists\, (\texttt{corr-user}, B, \cdot)\ \textbf{precedes}\ (\texttt{dec}, A, i, s, t, \cdot)\ \textbf{in}\ \mathbf{L} \\
&\qquad \wedge\ \nexists\, (\texttt{corr-device}, B, j, \cdot)\ \textbf{precedes}\ (\texttt{dec}, A, i, s, t, \cdot)\ \textbf{in}\ \mathbf{L} \quad /\!/\ (2) \\
&\quad \wedge\ \nexists\, r, t'\ \textbf{st}\ \big((\texttt{corr-sess}, B, j, r, \langle \rho : \texttt{send}, t : t', \cdot \rangle)\ \textbf{in}\ \mathbf{L} \\
&\qquad \wedge\ \pi^s_{A,i}.mgid[t] = \pi^r_{B,j}.mgid[t']\big) \quad /\!/\ (3)
\end{aligned}
$$

Confidentiality. In the following, consider an execution of the DOGM security experiment whereby the adversary has successfully guessed the challenge bit. Considering each challenge query in isolation, we ask: do we expect Matrix to provide confidentiality for this ciphertext? In particular, we ask this question for the challenge ciphertext c, the result of an encryption query $\mathcal{O}$-Encrypt(A, i, s, m_0, m_1), generated by the session $\pi^s_{A,i}$ with intended recipients $CU \coloneqq \pi^s_{A,i}.CU$ and $CD \coloneqq \pi^s_{A,i}.CD$ at stage $t \coloneqq \pi^s_{A,i}.t$ and message index $z \coloneqq \pi^s_{A,i}.z$. We expect Matrix *not* to provide confidentiality in the following cases.

1) When an intended recipient user, B, has had their secret authenticator compromised before the distribution of the inbound Megolm session by the sender. Here, the adversary may generate a new cryptographic device identity and link it to user B. They may either create a new device within the experiment (through a call to the $\mathcal{O}$-Create oracle) or replace the cryptographic identity of an existing device (by distributing an updated cross-signing state through $\mathcal{O}$-Decrypt queries). Additionally, if B's secret authenticator is compromised after the initial distribution of the inbound Megolm session but they were, nevertheless, an intended recipient (i.e. there exists a device (B, j) such that $B \in CU$ and $(B, j) \in CD$ at the time of session distribution) then the adversary is able to request a copy of the Megolm session from another of B's devices using the Key Request protocol.

2) Similarly, when an intended recipient device, (B, j), has had its secret authenticator compromised either before the distribution of the inbound Megolm session, or for which there existed at least one of B's devices,

(B, k) say, in the list of recipient devices, CD. These attacks proceed similarly to case (1), without the need to generate a new device.

3) When the adversary directly compromises the session state of an intended recipient, they can trivially decrypt the ciphertext. Since Matrix clients do not dispose of old inbound Megolm session states, something we include in our modelling, any compromise of a recipient session after it has received the relevant inbound Megolm session will compromise it directly

4) Finally, when the adversary directly compromises the session state of the sender during stage t and message index $z' < z$, they can trivially decrypt the ciphertext. We can restrict this case to sessions compromised at the exact stage and at message index before the challenge since Matrix clients do ratchet forward internal state for outbound Megolm sessions (as well as delete Megolm state relating to completed stages).

We codify these cases formally, and in the language of the DOGM experiment, in the definition that follows.

Definition 6.3. An instance of the DOGM security experiment with experiment log $\mathbf{L}$ fulfils the MX-DOGM.CONF confidentiality predicate provided the following evaluates to true for each challenge query, $chall \in {}^*\mathsf{Challenges}(\mathbf{L})$.

$$
\begin{aligned}
&\underline{\mathsf{MX\text{-}DOGM.CONF}(\mathbf{L}, chall)} \coloneqq \\
&\quad \textbf{let } chall = (\,\mathtt{enc}, A, i, s, CU, CD, t, z, m_0, m_1, c\,) : \\
&\qquad \nexists\, B \in CU \textbf{ st } (\mathtt{corr\text{-}user}, B, \cdot) \in \mathbf{L} \quad /\!/\ (1) \\
&\qquad \wedge\ \nexists\, B \in CU, j \in [n_d] \textbf{ st } (\mathtt{corr\text{-}device}, B, j, \cdot) \in \mathbf{L} \quad /\!/\ (2) \\
&\qquad \wedge\ \nexists\, (B, j, r, t', t^\dagger) \textbf{ st } \big((B, j) \in CD \ \wedge\ (\mathtt{corr\text{-}sess}, B, j, r, \langle t : t', \cdot \rangle) \textbf{ in } \mathbf{L} \\
&\qquad\quad \wedge\ t' \geq t^\dagger \ \wedge\ \pi^s_{A,i}.mgid[t] = \pi^r_{B,j}.mgid[t^\dagger] \ \wedge\ \pi^r_{B,j}.st_{mg}[t^\dagger].z < z \big) \quad /\!/\ (3) \\
&\qquad \wedge\ \nexists\, z' \textbf{ st } (\mathtt{corr\text{-}sess}, A, i, s, \langle t : t, z : z', \cdot \rangle) \textbf{ in } \mathbf{L} \ \wedge\ z' < z \quad /\!/\ (4)
\end{aligned}
$$

We now prove that our instantiation of multi-device group messaging in Matrix within the DOGM formalism, the MX-DOGM protocol, achieves security under these predicates.

Theorem 6.4 (Security of Matrix as a DOGM Protocol)**.** The MX-DOGM protocol specified in Definition 6.1 is a secure DOGM protocol with respect to the authentication predicate MX-DOGM.AUTH and confidentiality predicate MX-DOGM.CONF, given that message history is not stored in client state.

That is, for any probabilistic polynomial-time algorithm $\mathcal{A}$ playing the DOGM security experiment instantiated with MX-DOGM, we have that $\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{MX\text{-}DOGM},\mathcal{A}}(\Lambda)$ is negligible under the experiment parameters $\Lambda = (n_u, n_d, n_i, n_s, n_m)$ and the Olm usage parameters (n_e, n_f, n_p, n_q), in the following conditions.

1) HKDF may be modelled as a random oracle directly and, in so doing, instantiates an m-entropy preserving KDF such that $\mathsf{Adv}^{\mathsf{mKDF}}_{\mathsf{HKDF\text{-}RO},\mathcal{B}}(\lambda, n_q)$ is negligible in λ for any PPT adversary $\mathcal{B}$.

2) HMAC may be modelled as a random oracle directly and, in so doing, instantiates a secure PRF such that $\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC\text{-}RO},\mathcal{A}}(\lambda, n_q)$ is negligible in λ for any PPT adversary $\mathcal{B}$.

3) The Megolm ratchet, with its usage of HKDF and HMAC modelled as queries to random oracles, provides a secure FF-PRG such that $\mathsf{Adv}^{\mathsf{KIND}}_{\mathsf{MgRatchet},\mathcal{B}}(\lambda, n_q)$ is negligible in λ where λ is the bit-length of the keyspace and n_q limits the number of queries in the experiment, for any PPT adversary $\mathcal{B}$.

4) The Gap Diffie-Hellman assumption holds over Curve25519 when used by Olm for X25519 key exchange, such that $\mathsf{Adv}^{\mathsf{Gap\text{-}DH}}_{\mathsf{XDH},\mathcal{B}}(\lambda, n_q)$ is negligible in λ for any PPT adversary $\mathcal{B}$.

5) Ed25519 as used by Matrix instantiates a SUF-CMA secure digital signature scheme, i.e. $\mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{MgRatchet},\mathcal{B}}(\lambda, n_q)$ is negligible in λ for any PPT adversary $\mathcal{B}$.

6) The AEAD scheme instantiated by Olm, with its usage of HKDF and HMAC modelled as random oracles directly is IND$-CPA and EUF-CMA secure, i.e. $\mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{Olm\text{-}AEAD},\mathcal{B}}(\lambda, n_q)$ and $\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Olm\text{-}AEAD},\mathcal{B}}(\lambda, n_m)$ are negligible in λ, n_q and n_m for any PPT adversary $\mathcal{B}$.

7) The AEAD scheme instantiated by Megolm is IND$-CPA secure, i.e. $\mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{Megolm},\mathcal{B}}(\lambda, n_q)$ is negligible in λ for any PPT adversary $\mathcal{B}$.

Proof outline. We start by separating our analysis into two distinct cases. In the first, the adversary wins the security experiment through an authentication break, i.e. the adversary causes the challenger to terminate the experiment early (before returning a guess b') and return a value of 1. In the second, the adversary wins the security experiment through a confidentiality break, i.e. the adversary returns control to the challenger alongside a guess b' of the challenge bit b, and this guess is correct.

For the authentication case, we split our analysis further. We first consider the ability of the adversary to inject Megolm sessions under its control, before considering the adversary's ability to forge messages within an honest Megolm session.

We now handle the injection of adversarially-controlled Megolm sessions. We start by guessing the session and stage in which an honest session accepts a valid forgery (and abort if our guess is incorrect). We do the same for the claimed sending device of the forged ciphertext (and, similarly, abort if our guess is incorrect). We will use these guesses to embed the appropriate challenges as we perform the security reductions that bound each game hop.

We proceed to utilise the assumed SUF-CMA security of the Ed25519 digital signature scheme to ensure that our recipient and sending devices only accept honest user key packages for each another. These provide the cryptographic link from each user's master cross-signing key to the respective self-signing key. We do the same for the respective device record, providing a cryptographic link

from each user's self-signing key to the respective device key. At this point, the adversary cannot cause any sessions representing the sending or receiving device to accept a device key that does not represent the honest device.

We turn our attention to the Olm channels between the sending and receiving device. In particular, we are interested in the channel that was used to inject either an inbound Megolm session through either (a) a Megolm distribution message, or (b) a Key Request session sharing message. We start by guessing which of the two devices initiated this particular Olm channel. We utilise the SUF-CMA security of the Ed25519 digital signature scheme to ensure that the initiating device accepted an honest device key package for the other device (signed by its device key). We proceed to guess which of the non-initiating device's pre-keys (or fallback keys) was used to initiate the channel. We then note that the authentication predicate disallows either of our two target devices from being corrupted at any point in the experiment. We may now apply the Gap Diffie-Hellman security of XDH when applied to the identity key of the initiating device and the pre-key used for the non-initiating device. Our previous guesses allow us to correctly embed the Gap Diffie-Hellman challenge keys during our reduction and, since we can be sure that the adversary may not corrupt either device, our reduction will proceed soundly.

Additionally, by disallowing all such compromise of our target devices, we need not rely on Olm's asymmetric ratchet for security. Instead, we repeatedly apply the KDF security of HKDF to replace the Olm root key for each new epoch, until we arrive at the particular epoch used to exchange the relevant key material. We proceed to do something similar for the PRF security of HMAC, in order to replace the per-message key material with a uniformly random bit-string. We then utilise the assumed EUF-CMA security of Olm's AEAD scheme to then ensure that the message containing the inbound Megolm session was honestly distributed.

We now handle message forgeries within honestly distributed Megolm sessions. We start by guessing the sending session and stage for which the forgery occurs. We use this guess to embed a SUF-CMA challenge for the Ed25519 digital signature scheme, replacing the group signing key for that Megolm session. Noting that the authentication predicate prevents the sending session from being compromised directly when it is at the target stage, we can be sure that the reduction is sound, and any message forgery for an honestly generated Megolm session can be converted into a valid forgery in the SUF-CMA security game. This completes our proof of the authentication case.

For the confidentiality case, we start by utilising the SUF-CMA security of Ed25519 to ensure that every user's self-signing key is genuine and correct. We do the same for every device record of every device.

We proceed with a hybrid argument over all possible challenge encryptions (which we bound by the number of messages in the experiment). For each challenge, we guess at the beginning of the experiment the session, stage and message index of the challenge encryption (aborting if our guess is incorrect). We note that the confidentiality predicate in tandem with the changes in the first two games, ensures that all self-signing keys of recipient users, and device records of recipient devices, were honestly distributed.

We calculate the set of all pairwise channels for which we require confidentiality for the current challenge. We proceed with a similar argument to that in the authentication case to demonstrate that breaking the confidentiality of these channels requires breaking the security guarantees of one of the underlying primitives. The key difference from the authentication case is that we require the IND-CPA security of Olm's AEAD scheme, rather than any authentication guarantees.

Having determined the confidentiality provided by the relevant Olm channels, we turn our attention to the confidentiality of the Megolm session. We utilise the key indistinguishability of the Megolm ratchet, within the FF-PRG formalism, to replace the key material used to encrypt the ciphertext with a uniformly random bit string. To finish, we show that distinguishing the ciphertext produced by Megolm from a random replacement entails breaking the IND$-CPA security of the underlying encryption scheme.

Proof. 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 by correctly guessing the challenge bit through a confidentiality break. We bound the advantage of winning both cases and demonstrate that under certain assumptions, the adversary's advantage of winning overall is negligible.

Let $\mathsf{Adv}_{\mathsf{AUTH}}$ denote the advantage of the adversary winning the DOGM security game by triggering the early termination of the experiment, and let $\mathsf{Adv}_{\mathsf{CONF}}$ denote the advantage of the adversary winning the DOGM security game when it concludes with the adversary returning the guess of the challenge bit b'.

Since $\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{MX\text{-}DOGM},\Lambda}(\mathcal{A}) \leq \mathsf{Adv}_{\mathsf{AUTH}} + \mathsf{Adv}_{\mathsf{CONF}}$ we may bound the overall advantage of winning by considering each case in isolation.

Case 1: Adversary has triggered the authentication win condition

We split our analysis of *Case 1* in two. First, we consider *session injection* whereby a session accepts an inbound Megolm session that was not honestly generated by a session representing its assigned sending device (in $.CD[0]$) which the adversary proceeds to use to have the target session accept a message not honestly generated, triggering an authentication win. Let $\mathsf{Adv}_{\mathsf{AUTH\text{-}INJ}}$ be the advantage of the adversary in winning the DOGM security game in this case. Second, we consider *message forgery* whereby a session accepts a message from an honest Megolm session, but the message was not honestly generated itself. Let $\mathsf{Adv}_{\mathsf{AUTH\text{-}FRG}}$ be the advantage of the adversary in winning the DOGM security game in this case. Note that these are the only two situations in which the authentication win flag can be set to true and, once such an event has occurred in isolation then it is determined that the authentication win flag will be set. Thus, we have $\mathsf{Adv}_{\mathsf{AUTH}} \leq \mathsf{Adv}_{\mathsf{AUTH\text{-}INJ}} + \mathsf{Adv}_{\mathsf{AUTH\text{-}FRG}}$.

We bound the advantage of $\mathcal{A}$ in triggering the authentication win condition in the *session injection* case.

Game 0. This is the standard DOGM game in the session injection case, instantiated with the MX-DOGM protocol. Thus we have $\mathsf{Adv}_{\mathsf{AUTH\text{-}INJ}} = \mathsf{Adv}_{\mathsf{G0}}$.

Game 1. In this game, we guess a session $\pi^s_{A,i}$ and stage t (such that $\pi^s_{A,i}.T[-1] = (t, \cdot, \cdot)$) to be the session and stage that triggers an authentication win condition, and trigger an abort event if any session other than $\pi^s_{A,i}$ causes such a win. Specifically, at the beginning of the experiment we guess a tuple of values (A, i, s, t) and abort the game if the session processing a $\mathcal{O}$-Decrypt call is $\pi^u_{C,k}$ such that $(C, k, u) \neq (A, i, s)$ and $\pi^s_{C,k}.T[-1] \neq (t, \cdot, \cdot)$. Thus:

$$\mathsf{Adv}_{\mathsf{G0}} \;\leq\; n_u \cdot n_d \cdot n_i \cdot n_s \cdot \mathsf{Adv}_{\mathsf{G1}}$$

The authentication win flag can be set to true in one of two places. First, it may occur during the processing of a $\mathcal{O}$-Decrypt query. Second, it may occur during the processing of a $\mathcal{O}$-StateShare query. We denote the former case (a) and the latter case (b), throughout.

Game 2. In this game, we guess a device (B, j) to be the device that shares with (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). If we find ourselves in case (a), we abort the game if $\pi^s_{A,i}.CU[0] \neq B$ and $\pi^s_{A,i}.CD[0] \neq (B, j)$. In case (b), on the other hand, we abort the game if the values of the fourth and fifth inputs to the $\mathcal{O}$-StateShare call, sharing the names 'B' and 'j' respectively, are not equal to the B and j guessed at the beginning of the experiment. Thus:

$$\mathsf{Adv}_{\mathsf{G1}} \;\leq\; n_u \cdot n_d \cdot \mathsf{Adv}_{\mathsf{G2}}$$

Game 3. In this game, we introduce an abort event $abort_{\mathfrak{U}^\star}$ that triggers if party A has $\mathsf{CrossSign.VerifyDevice}(dsau, \mathfrak{R}_{B,j}, \mathfrak{U}_B)$ return true for which $\mathfrak{U}_B$ contains self-signing key spk_B, but B did not generate spk_B during initial user setup (i.e. while executing Gen).

Based on our partner guess from **Game 2**, at the beginning of the game we replace mpk_B with a verification key pk from a strongly unforgeable digital signature challenger $\mathcal{C}_{\mathsf{DS}}$. When B would otherwise sign the self-signing key linking message using msk_B, B instead requests a signature from $\mathcal{C}_{\mathsf{DS}}$. In particular, we replace the signature σ_s that links B's self-signing key, spk_B, to their user identity.

If party A receives a user key package $\mathfrak{U}_B$ containing a signature σ_s such that $\mathsf{Ed25519.Verify}(mpk_B, \langle \mathsf{SPK}, uid_B, spk_B \rangle, \sigma_s) \mapsto \mathbf{true}$ but spk_B was not generated honestly by B (and thus would trigger the abort event $abort_{\mathfrak{U}^\star}$), then σ_s is a valid forgery, and breaks the strong unforgeability of the digital signature scheme Ed25519. Thus:

$$\mathsf{Adv}_{\mathsf{G2}} \;\leq\; \mathsf{Adv}_{\mathsf{G3}} + \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Ed25519}}(\lambda, 3)$$

Game 4. In this game, we introduce an abort event $abort_{\mathfrak{R}^\star}$ that triggers if party A has $\mathsf{CrossSign.VerifyDevice}(dsau, \mathfrak{R}_{B,j}, \mathfrak{U}_B)$ return true for which $\mathfrak{R}_{B,j}$ contains device keys $dpk_{B,j}$ and $ipk_{B,j}$, but B did not generate both $dpk_{B,j}$ and $ipk_{B,j}$ within the relevant $\mathcal{O}\text{-}\mathsf{Create}(B, j)$ query.

Based on our partner guess from **Game 2**, at the beginning of the game we replace spk_B with pk from a strongly unforgeable digital signature challenger $\mathcal{C}_{\mathsf{DS}}$. When B is required to sign a device linking message of the form '$\langle \mathtt{DEV}, \mathtt{OLM}, uid_B, did_{B,j}, dpk_{B,j}, ipk_{B,j} \rangle$' using its self-signing key spk_B, B instead requests a signature from $\mathcal{C}_{\mathsf{DS}}$.

If party A receives a device record $\mathfrak{R}_{B,j}$ containing a signature σ_d such that $\mathsf{Ed25519.Verify}(spk_B, \langle \mathtt{DEV}, \mathtt{OLM}, uid_B, did_{B,j}, dpk_{B,j}, ipk_{B,j} \rangle, \sigma_d) \mapsto \mathbf{true}$ but $dpk_{B,j}$ and $ipk_{B,j}$ were not generated honestly by B (and thus would trigger the abort event $abort_{\mathfrak{R}^\star}$), then σ_d is a valid forgery, and breaks the strong unforgeability of the digital signature scheme Ed25519. Thus:

$$\mathsf{Adv}_{\mathsf{G3}} \leq \mathsf{Adv}_{\mathsf{G4}} + \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Ed25519}}(\lambda, n_d)$$

Game 5. In this game, we introduce an abort event $abort_{\mathfrak{U}}$ that triggers if party B has $\mathsf{CrossSign.VerifyDevice}(dsau, \mathfrak{R}_{A,i}, \mathfrak{U}_A)$ return true for which $\mathfrak{U}_A$ contains self-signing key spk_A, but A did not generate spk_A during initial user setup (i.e. while executing Gen).

Based on our user guess from **Game 1**, at the beginning of the game we replace mpk_A with a verification key pk from a strongly unforgeable digital signature challenger $\mathcal{C}_{\mathsf{DS}}$. Whenever A is required to sign the self-signing key linking message with mpk_A, A instead requests a signature from $\mathcal{C}_{\mathsf{DS}}$. In particular, we replace the signature σ_s that links A's self-signing key, spk_A, to their user identity.

If party B has such a call to CrossSign.VerifyDevice return true, such that $\mathsf{Ed25519.Verify}(mpk_A, \langle \mathtt{SPK}, uid_A, spk_A \rangle, \sigma_s) \mapsto \mathbf{true}$ but spk_A was not generated honestly by A (and thus would trigger the abort event $abort_{\mathfrak{U}}$), then σ_s is a valid forgery, and breaks the strong unforgeability of the digital signature scheme Ed25519. Thus:

$$\mathsf{Adv}_{\mathsf{G4}} \leq \mathsf{Adv}_{\mathsf{G5}} + \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Ed25519}}(\lambda, 3)$$

Game 6. In this game, we introduce an abort event $abort_{\mathfrak{R}}$ that triggers if party B has $\mathsf{CrossSign.VerifyDevice}(dsau, \mathfrak{R}_{A,i}, \mathfrak{U}_A)$ return true and for which $\mathfrak{R}_{A,i}$ contains device keys $dpk_{A,i}$ and $ipk_{A,i}$, but A did not generate both $dpk_{A,i}$ and $ipk_{A,i}$ within the relevant $\mathcal{O}\text{-}\mathsf{Create}(A, i)$ query.

Based on our user guess from **Game 1**, at the beginning of the game we replace spk_A with pk from a strongly unforgeable digital signature challenger $\mathcal{C}_{\mathsf{DS}}$. When A is required to sign a device linking message of the form '$\langle \mathtt{DEV}, \mathtt{OLM}, uid_A, did_{A,i}, dpk_{A,i}, ipk_{A,i} \rangle$' using its self-signing key spk_A, A instead requests a signature from $\mathcal{C}_{\mathsf{DS}}$.

If party B receives a device record $\mathfrak{R}_{A,i}$ containing a signature σ_d such that $\mathsf{Ed25519.Verify}(spk_A, \langle \mathtt{DEV}, \mathtt{OLM}, uid_A, did_{A,i}, dpk_{A,i}, ipk_{A,i} \rangle, \sigma_d)$ evaluates to true but $dpk_{A,i}$ and $ipk_{A,i}$ were not generated honestly by A (and thus would trigger the abort event $abort_{\mathfrak{R}^\star}$), then σ_d is a valid forgery, and breaks the strong unforgeability of the digital signature scheme Ed25519. Thus:

$$\mathsf{Adv}_{\mathsf{G5}} \leq \mathsf{Adv}_{\mathsf{G6}} + \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Ed25519}}(\lambda, n_d)$$

Game 7. In this game, we introduce an abort event $abort_{olm\text{-}init}$ that triggers if device (B, j) does not initiate the Olm channel between (A, i) and itself. We note that the proof will proceed identically in either case, up to a change in notation. Thus, repeating this argument across both cases gives

$$\mathsf{Adv}_{\mathsf{G6}} \leq 2 \cdot \mathsf{Adv}_{\mathsf{G7}}$$

Game 8. In this game, we introduce an abort event $abort_{\mathfrak{D}}$ that triggers if party B successfully evaluates $\mathsf{Olms.Enc}(st, dpk_{A,i}, ipk_{A,i}, m) \mapsto (st, c)$ without error and for which c is a pre-key message, with $c.type = 0$, but A did not generate the identity key $ipk_{A,i}$ or the ephemeral/fallback key $epk_{A,i,k}$ or $fpk_{A,i,k}$ that was used. Thanks to the signature checks in $\mathsf{Olms.Enc}$, it follows that the device key package containing these keys, which we denote $\mathfrak{D}$, contained signatures over the device keys and fallback/ephemeral key that were not honestly generated by A.

Based on our guess from **Game 1**, when A generates device (A, i) we replace $dpk_{A,i}$ with pk from a strongly unforgeable digital signature challenger $\mathcal{C}_{\mathsf{DS}}$. When A is required to sign a message of the form $\langle did_{A,i}, \mathtt{DPK}, dpk_{A,i}, \mathtt{IPK}, ipk_{A,i}\rangle$ for device and identity keys, $\langle \mathtt{EPK}, epk_{A,i,k}\rangle$ for ephemeral key $epk_{A,i,k}$, or $\langle \mathtt{FPK}, fpk_{A,i,k}\rangle$ for fallback key $fpk_{A,i,k}$, the challenger instead queries these messages to $\mathcal{C}_{\mathsf{DS}}$.

If party B receives a key package $\mathfrak{D} = \langle dpk, ipk, \sigma_d, epk, \sigma_e, fpk, \sigma_f\rangle$ for which either $epk_{A,i,k} = epk_0$ or $fpk_{A,i,k} = fpk_0$ and

1) $\mathsf{Ed25519.Verify}(dpk_{A,i}, \langle did_{A,i}, \mathtt{DPK}, dpk_{A,i}, \mathtt{IPK}, ipk_{A,i}\rangle, \sigma_d) \mapsto \mathbf{true}$,

2) $\mathsf{Ed25519.Verify}(dpk_{A,i}, \langle \mathtt{EPK}, epk_{A,i,k}\rangle, \sigma_e[k]) \mapsto \mathbf{true}$, or

3) $\mathsf{Ed25519.Verify}(dpk_{A,i}, \langle \mathtt{FPK}, fpk_{A,i,k}\rangle, \sigma_f[k]) \mapsto \mathbf{true}$,

but $ipk_{A,i}$ and $epk_{A,i,k}$ or $fpk_{A,i,k}$ were not generated honestly by A (and thus would trigger the abort event $abort_{\mathfrak{D}}$), then either σ_d, $\sigma_e[k]$ or $\sigma_f[k]$ is a valid forgery, and breaks the strong unforgeability of the digital signature scheme $\mathsf{Ed25519}$. Thus:

$$\mathsf{Adv}_{\mathsf{G7}} \leq \mathsf{Adv}_{\mathsf{G8}} + \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Ed25519}}(\lambda, 1 + n_e + n_f)$$

Game 9. In this game, the challenger guesses which ephemeral or fallback key pair will be used to initialise the Olm channel that is later used to distribute the inbound Megolm session which initiates stage t for $\pi^s_{A,i}$. In particular, the adversary guesses a random key pair from the set $\{(esk_{A,i,k}, epk_{A,i,k}) : k \in [n_e]\} \cup \{(fsk_{A,i,k}, fpk_{A,i,k}) : k \in [n_f]\}$.

In what follows, we write $(esk_{A,i,k}, epk_{A,i,k})$ to signify the guessed key pair, regardless of whether it refers to an ephemeral or fallback key pair. Indeed, since the authentication predicate discludes all cases that might differentiate between the use of an ephemeral or fallback key, we may treat both cases equally. Thus:

$$\mathsf{Adv}_{\mathsf{G8}} \leq (n_e + n_f) \cdot \mathsf{Adv}_{\mathsf{G9}}$$

Game 10. In this game, the challenger rejects calls to $\mathcal{O}$-CorruptDevice(A, i) and $\mathcal{O}$-CorruptDevice(B, j) for the devices (A, i) and (B, j) guessed in **Game 1** and **Game 2** respectively.

To start, we note that the authentication predicate disallows the adversary from issuing any call to $\mathcal{O}$-CorruptDevice(A, i) throughout the experiment.

To justify a similar change for (B, j), we consider three sub-cases. In case (a), the adversary does not issue a call to $\mathcal{O}$-CorruptDevice(B, j) during the experiment. In case (b), the adversary issues a call to $\mathcal{O}$-CorruptDevice(B, j) *before* $\pi^s_{A,i}$ has entered stage t. In case (c) the adversary issues a call to $\mathcal{O}$-CorruptDevice(B, j) *after* $\pi^s_{A,i}$ has entered stage t.

Note that the authentication predicate explicitly disallows case (b). In case (c), this is disallowed by the authentication predicate if the sender and recipient devices represent the same user, when $A = B$. If not, when $A \neq B$, $\pi^s_{A,i}$ has already accepted an inbound Megolm session for the stage t where the authentication break occurs and it will not accept any Megolm distribution or Key Request messages from B that would overwrite the Megolm session with one that has a *different signing key*.[5] It follows that we may disallow cases (b) and (c) without reducing the advantage of the adversary.

Thus:

$$\mathsf{Adv}_{\mathsf{G9}} \leq \mathsf{Adv}_{\mathsf{G10}}$$

Game 11. In this game we replace the values of rch_0 and ck_0 that are computed by (A, i) and (B, j) when they initiate the Olm session that will be used to distribute the challenge Megolm session with uniform random bit strings of the same length.

We bound the ability of an adversary to detect this change through the following security reduction. Let $\mathcal{O}$-RO(ms) denote the random oracle we use to replace calls of the form '$\mathsf{HKDF}(0, ms, \texttt{OLM-RT})$'. 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 $epk_{A,i,k}$ respectively. The change introduced in **Game 10** ensures that the challenger will not have to answer any queries that leak their secret counterparts, $isk_{B,j}$ or $esk_{A,i,k}$.

Whenever a session sets up their side of a new Olm channel, for which the Triple Diffie-Hellman key exchange includes at least one of $ipk_{B,j}$ and $epk_{A,i,k}$, the challenger replaces computation of the master secret, ms, and the subsequent key derivation of the root key, rch_0, and chain key, ck_0, by sampling two 32-byte random bit strings in their place. The challenger stores these samples in a table addressed by the public keys that would have been used to compute the master secret. Computations that do not include the target key pairs proceed as normal: calculating the master secret, ms, before feeding it into our emulated random oracle.

[5] Element clients will accept sessions at an earlier message index but since they use the group signing key to address sessions in the store, it is (somewhat tautologically) impossible to change a session's signing key once it is in the store.

Whenever the adversary queries our emulated random oracle, the challenger first checks if they need to respond using a value from the aforementioned table. To do so, they utilise the DDH oracle, $\mathcal{O}$-DDH, provided by $\mathcal{C}_{\mathsf{Gap\text{-}DH}}$ to check if the given value is a master secret, ms, that would have been computed using either $ipk_{B,j}$ or $epk_{A,i,k}$ but whose output was replaced with a random sample. Importantly, since the master secret contains the concatenation of three Diffie-Hellman key exchanges, we must also be sure to calculate and check the value for the other two Diffie-Hellman key exchanges. For each of the two remaining parts of ms, the challenger has access to at least one of the secret contributions and, thus, may compute the correct value. If all components match, then the challenger uses the randomly chosen keys from that list as the random oracle response, otherwise, it samples a new random value.

If, during this emulation process, the adversary causes $(ipk_{B,j}, epk_{A,i,k}, ms[0\mathsf{B} : 31\mathsf{B}])$ to be queried to the $\mathcal{O}$-DDH oracle and the response is 1, meaning they have queried $\mathcal{O}$-RO(ms) with a string ms that contains the value of $g^{isk_{B,j} \cdot esk_{A,i,k}}$, then the adversary has solved the Gap-DH problem and the challenger may submit $ms[0 : 31\mathsf{B}]$ to $\mathcal{C}_{\mathsf{Gap\text{-}DH}}$ under the knowledge that $ms[0 : 31\mathsf{B}] = g^{isk_{B,j} \cdot esk_{A,i,k}} = g^{xy}$.

On the other hand, if the adversary does not cause $(ipk_{B,j}, epk_{A,i,k}, ms[0\mathsf{B} : 31\mathsf{B}])$ to be queried to $\mathcal{O}$-DDH oracle, it follows that the values of rch_0 and ck_0 computed by (A, i) and (B, j) when initiating the target Olm session were sampled uniformly at random and are distributed in a manner independent of the protocol execution. Thus:

$$\mathsf{Adv}_{\mathsf{G10}} \;\leq\; \mathsf{Adv}_{\mathsf{G11}} + \mathsf{Adv}^{\mathsf{Gap\text{-}DH}}_{\mathsf{XDH}}(\lambda, \mathsf{poly}(\lambda))$$

Game 12. Let p be the Olm channel epoch within which our target inbound Megolm session is distributed. In this game, we replace the computation of rch_p and ck_p with uniformly random keys. To do so, we apply a hybrid argument to the series of games **Game 11.0**, **Game 11.1**, ..., **Game 11.*p***. In the h-th hybrid, we replace the first h values of the root and chain key after rch_0 and ck_0 ($rch_1, rch_2, \ldots, rch_h$ and $ck_1, ck_2, \ldots, ck_h$) with uniformly random keys such that **Game 11.0** is exactly **Game 11** and **Game 11.*p*** is exactly **Game 12**. We now consider the change in advantage between two consecutive hybrids, **Game 11.*h*** and **Game 11.(*h*+1)**.

Consider the following security reduction in which we construct an adversary against a KDF challenger, $\mathcal{C}_{\mathsf{mKDF}}$. First, note that the value of rch_h is uniformly distributed (by the previous hybrid) and thus can be replaced with the public salt provided by $\mathcal{C}_{\mathsf{mKDF}}$. It follows that we can replace the computation of keys (rch_{h+1}, ck_{h+1}) with a call to the challenge oracle provided by $\mathcal{C}_{\mathsf{mKDF}}$, '$rch_{h+1}, ck_{h+1} \leftarrow \mathcal{O}$-Challenge(OLM-RCH, 64B)'. Thus, we may bound the change in advantage between any two hybrids with:

$$\mathsf{Adv}_{\mathsf{G11.h}} \;\leq\; \mathsf{Adv}_{\mathsf{G11.(h+1)}} + \mathsf{Adv}^{\mathsf{mKDF}}_{\mathsf{HKDF\text{-}RO}}(\lambda, 0)$$

Note that this analysis does not rely on the authenticity of the ratchet keys exchanged alongside messages, nor does it rely on the security of the resulting

key exchange. The former, a lack of authenticity requirements, is because the use of an incorrect key contribution from the other party will simply cause the two parties to derive differing keys (and, as such, the protocol would fail). The latter, a lack of security requirements of the key exchange, is because the authentication predicate has disallowed any compromise of relevant Olm state. In other words, the continuous key exchange reduces to a symmetric ratchet in this context, relying on the security of the keys derived at the start: rch_0, ck_0. Since the epochs and message indices of Olm are independent and unrelated to the sessions, stages and message indices of the group messaging formalism, we bound the number of epochs within each Olm channel as an ad-hoc parameter, n_p. Together, we find that:

$$\mathsf{Adv}_{\mathsf{G11}} \;\leq\; \mathsf{Adv}_{\mathsf{G12}} + n_p \cdot \mathsf{Adv}^{\mathsf{mKDF}}_{\mathsf{HKDF\text{-}RO}}(\lambda, 0)$$

Game 13. In this game, we replace the computation of each chain key, $ck_{p,q}$, and message key, $mk_{p,q}$, in each epoch p with uniformly random replacements. To do so, we apply a hybrid argument to the series of games **Game 12.0**, **Game 12.1**, …, **Game 12.*pq***. In the h-th hybrid, we replace the first h values of the chain key and message key after $ck_{p,0}$ and $mk_{p,0}$ with uniformly random keys such that **Game 12.0** is exactly **Game 12** and **Game 12.*pq*** is exactly **Game 13**. We now consider the change in advantage between two consecutive hybrids, **Game 12.*h*** and **Game 12.(*h*+1)**.

Consider the chain key and message key generated at index $h + 1$. Rather than computing these keys through calls to $\mathsf{HMAC}(ck_{p,h}, \texttt{0x02})$ and $\mathsf{HMAC}(rch_{p,h}, \texttt{0x01})$, the challenger initialises a PRF challenger $\mathcal{C}_{\mathsf{PRF}}$, submits queries with `0x02` and `0x01`, then replaces ck_{h+1} and mk_{h+1} with the output of said query (respectively). We note that if the bit b sampled by $\mathcal{C}_{\mathsf{PRF}}$ is 0, then we are in **Game 12.*h***, else we are in **Game 12.*h*+1**. Thus:

$$\mathsf{Adv}_{\mathsf{G12.h}} \;\leq\; \mathsf{Adv}_{\mathsf{G12.(h+1)}} + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC\text{-}RO}}(\lambda, 2)$$

As in **Game 12**, we bound the number of messages in a single Olm epoch as an ad-hoc parameter that bounds usage of the Olm protocol, n_q. Together, we find that:

$$\mathsf{Adv}_{\mathsf{G12}} \;\leq\; \mathsf{Adv}_{\mathsf{G13}} + n_p \cdot n_q \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC\text{-}RO}}(\lambda, 2)$$

Game 14. In this game, we introduce an abort event that triggers if device (A, i) decrypts an Olm ciphertext (keyed by some $mk_{p,q}$), and accepts a Megolm inbound session $\mathfrak{S}'_{rcv} = (ver, i, R, gpk)$ but the ciphertext was not output by (B, j). As before, we consider a series of games, one for each possible message in the given Olm channel.

Consider the message encrypted in epoch p and message index q. Here, the challenger initialises a $\mathsf{EUF\text{-}CMA}$ challenger $\mathcal{C}_{\mathsf{EUF\text{-}CMA}}$ for a one-time AEAD scheme, which the challenger queries when it needs to encrypt or decrypt with $mk_{p,q}$. The abort event only triggers if the adversary can produce a valid ciphertext that decrypts under $mk_{p,q}$, and we can submit the ciphertext to $\mathcal{C}_{\mathsf{EUF\text{-}CMA}}$, breaking the $\mathsf{EUF\text{-}CMA}$ security of the one-time AEAD scheme. By

the changes introduced in **Game 13**, $mk_{p,q}$ is already uniformly random and independent and this replacement is sound. Any adversary that can trigger the abort event can be used by the challenger to break the EUF-CMA security of AEAD.

Thus:

$$\mathsf{Adv}_{\mathsf{G13}} \leq \mathsf{Adv}_{\mathsf{G14}} + n_p \cdot n_q \cdot \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Olm\text{-}AEAD}}(\lambda, 1)$$

Note that, from this point onwards, the success probability of the adversary in winning the security experiment through case (a), injecting a Megolm session through a Megolm distribution message, or (b), injecting a Megolm session through the state sharing oracle, is zero: $\mathsf{Adv}_{\mathsf{G14}} = 0$. This completes our analysis of session injections in *Case 1.*

We proceed to bound the advantage of $\mathcal{A}$ in triggering the authentication win condition in the *message forgery* case.

Game 0. This is the standard DOGM game in the message forgery case, instantiated with the MX-DOGM protocol. Thus we have $\mathsf{Adv}_{\mathsf{AUTH\text{-}FRG}} = \mathsf{Adv}_{\mathsf{G0}}$.

Note that, since we are in the message forgery case, all accepted inbound Megolm sessions were honestly generated by a session representing the assigned sending device (in $.CD[0]$). If this were not the case, then the acceptance of the adversarially-generated inbound Megolm session within the $\mathcal{O}$-Decrypt or $\mathcal{O}$-StateShare oracles would have triggered the abort event introduced in **Game 14** of the previous case.

Game 1. In this game, we guess the sending partner session, $\pi^r_{B,j}$, and stage, t' (such that $\pi^r_{B,j}.T[-1] = (t', \cdot, \cdot)$), of the first session that triggers a authentication win condition, 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, r, t'). Let $\pi^s_{A,i}$ be the first session that triggers an authentication win condition, and t the stage of the message it is processing when this occurs. We abort if $\pi^s_{A,i}.mgid[t] \neq \pi^r_{B,j}.mgid[t']$, $\pi^s_{A,i}.CU[0] \neq B$, $\pi^s_{A,i}.CD[0] \neq (B, j)$ or $\pi^r_{B,j}.\rho \neq \mathsf{send}$. Thus:

$$\mathsf{Adv}_{\mathsf{G0}} \leq n_u \cdot n_d \cdot n_i \cdot n_s \cdot \mathsf{Adv}_{\mathsf{G1}}$$

Game 2. 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 (B, j).

Specifically, we introduce the following reduction. When $\pi^r_{B,j}$ creates the Megolm inbound session $\mathfrak{S}_{rcv}$ for stage t', the challenger instead replaces gpk with the signing key, pk, from a strongly unforgeable digital signature challenger $\mathcal{C}_{\mathsf{DS}}$. Whenever the partner session is required to sign Megolm ciphertexts with gsk, it instead queries the ciphertext to $\mathcal{C}_{\mathsf{DS}}$. Thus, if $\pi^s_{A,i}$ receives a Megolm ciphertext c' such that $\mathsf{Ed25519.Verify}(gpk, (ver, i, c, \tau), \sigma) = 1$ but $c' = (ver, i, c, \tau, \sigma)$ was not generated honestly by the partner session (and thus

would trigger the abort event $abort_{auth}$), then σ is a valid forgery for message (ver, i, c, τ), and breaks the strong unforgeability of the digital signature scheme Ed25519. Thus:

$$\mathsf{Adv}_{\mathsf{G1}} \leq \mathsf{Adv}_{\mathsf{G2}} + \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Ed25519}}(\lambda, n_m)$$

As of **Game 2** it is not possible for the adversary to cause $\pi^s_{A,i}$ to accept a non-honest ciphertext in stage t (in the message forgery case). Thus, $\mathsf{Adv}_{\mathsf{G2}} = 0$. This completes our analysis of message forgeries in *Case 1*.

This completes our analysis of *Case 1*.

Case 2: Adversary terminates and outputs a bit b

Here, we bound the probability that the adversary, $\mathcal{A}$, correctly guesses the challenge bit b.

In the case where the adversary has returned a guess of the challenge bit, but has not satisfied the confidentiality predicate, the challenger flips a fair coin to determine whether the adversary has won the game, or not. This provides the adversary with a win probability of $\frac{1}{2}$, and an advantage of zero (in this case).

We proceed to consider the case where the adversary has satisfied the confidentiality predicate, and do so via the following sequence of games.

Game 0. This is the standard DOGM game in *Case 2*, under the assumption that the confidentiality predicate has been satisfied, instantiated with the MX-DOGM protocol. Thus we have $\mathsf{Adv}_{\mathsf{CONF}} = \mathsf{Adv}_{\mathsf{G0}}$.

Game 1. In this game, for each party A' that the adversary did not corrupt through a $\mathcal{O}$-CorruptUser(A') query in the initialisation stage of the experiment (`exp-init`), we introduce an abort event $abort_{\mathfrak{U}'}$ that triggers if any session has CrossSign.VerifyDevice($dsau, \mathfrak{R}_{A',i'}, \mathfrak{U}_{A'}$) return true for which $\mathfrak{U}_{A'}$ contains self-signing key $spk_{A'}$, but A' did not generate $spk_{A'}$ during initial user setup (i.e. while executing Gen).

For each uncorrupted party, the challenger replaces $mpk_{A'}$ with a verification key pk from a strongly unforgeable digital signature challenger $\mathcal{C}_{\mathsf{DS}}$ before beginning the main stage of the experiment (`exp-main`). When A' would otherwise required sign the self-signing key linking message using $msk_{A'}$, A' instead requests a signature from $\mathcal{C}_{\mathsf{DS}}$. In particular, we replace the signature σ_s that links A''s self-signing key, $spk_{A'}$, to their user identity.

If any session receives a user key package $\mathfrak{U}_{A'}$ containing a signature σ_s such that Ed25519.Verify($mpk_{A'}, \langle \mathsf{SPK}, uid_{A'}, spk_{A'} \rangle, \sigma_s$) $\mapsto$ **true** but $spk_{A'}$ was not generated honestly by A' (and thus would trigger the abort event $abort_{\mathfrak{U}'}$), then σ_s is a valid forgery, and breaks the strong unforgeability of the digital signature scheme Ed25519. Thus:

$$\mathsf{Adv}_{\mathsf{G0}} \leq \mathsf{Adv}_{\mathsf{G1}} + n_u \cdot \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Ed25519}}(\lambda, 3)$$

Game 2. In this game, for each party A' that the adversary did not corrupt through a $\mathcal{O}$-CorruptUser(A') query in the initialisation stage of the experiment (exp-init), we introduce an abort event $abort_{\mathfrak{R}'}$ that triggers if any session has CrossSign.VerifyDevice($dsau, \mathfrak{R}_{A',i'}, \mathfrak{U}_{A'}$) return true for which $\mathfrak{R}_{A'}$ contains device key $dpk_{A',i'}$ and identity key $ipk_{A',i'}$, but A' did not generate both $dpk_{A',i'}$ and $ipk_{A',i'}$ during device registration (i.e. while executing Reg).

For each uncorrupted party, the challenger replaces $spk_{A'}$ with a verification key pk from a strongly unforgeable digital signature challenger $\mathcal{C}_{\mathsf{DS}}$ before beginning the main stage of the experiment (exp-main). When A' is required to sign a device linking message with $spk_{A'}$, A' instead requests the signature from $\mathcal{C}_{\mathsf{DS}}$. In particular, we replace the device signature, σ_d, that links $dpk_{A',i'}$ and $ipk_{A',i'}$ with the self-signing key spk' for user A'.

If any session receives a device record $\mathfrak{R}_{A',i'}$ containing a signature σ_s such that Ed25519.Verify($spk_{A'}, \langle \mathtt{DEV}, \mathtt{OLM}, uid_{A'}, did_{A',i'}, dpk_{A',i'}, ipk_{A',i'} \rangle, \sigma_d$) $\mapsto$ **true** but either $dpk_{A',i'}$ or $ipk_{A',i'}$ was not generated honestly by A' (and thus would trigger the abort event $abort_{\mathfrak{R}'}$), then σ_d is a valid forgery, and breaks the strong unforgeability of the digital signature scheme Ed25519. Thus:

$$\mathsf{Adv}_{\mathsf{G1}} \;\leq\; \mathsf{Adv}_{\mathsf{G2}} + n_u \cdot \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Ed25519}}(\lambda, n_d)$$

Games 3, ..., N+2. We consider a series of hybrids **Game 3**, ..., **Game N+2** where $N \coloneqq n_u \cdot n_d \cdot n_i \cdot n_s \cdot n_m$; one for each possible challenge encryption that may be triggered in the game. For each hybrid **Game h** for $3 \leq h \leq N+2$, we look to bound the advantage of the adversary in determining the challenge bit through the $(h-3)$-th challenge encryption. We find, by a hybrid argument:

$$\mathsf{Adv}_{\mathsf{G2}} \;\leq\; \sum_{h=3}^{N+2} \mathsf{Adv}_{\mathsf{Gh}} \;=\; n_u \cdot n_d \cdot n_i \cdot n_s \cdot n_m \cdot \mathsf{Adv}_{\mathsf{Gh}}$$

In what follows, we bound the advantage of an adversary in a single such hybrid, **Game h**, and do so through the following sequence of games.

Game h.0. This is the h-th hybrid as described above: $\mathsf{Adv}_{\mathsf{Gh}} = \mathsf{Adv}_{\mathsf{Gh.0}}$.

Game h.1. In this game, we guess the session $\pi^s_{A,i}$ and stage t of the $(h-3)$-th challenge encryption, such that the adversary issues a $\mathcal{O}$-Encrypt(A, i, s, m_0, m_1) query for which $m_0 \neq m_1$. Specifically, we guess a tuple of values, (A, i, s, t), at the start of the experiment and abort if the $(h-3)$-th challenge encryption query is not made to $\pi^s_{A,i}$ within stage t. It follows that,

$$\mathsf{Adv}_{\mathsf{Gh.0}} \;\leq\; n_u \cdot n_d \cdot n_i \cdot n_s \cdot \mathsf{Adv}_{\mathsf{Gh.1}}$$

See that, by construction of the DOGM experiment, the adversary cannot win if the challenge encryption query we handle in this hybrid does not satisfy the confidentiality predicate. Thus, we may assume that CONF($\mathbf{L}, chall$) $\mapsto$ **true** when $\mathbf{L}$ is taken at the end of the experiment. Note that, letting $chall = ($ enc,

$A, i, s, CU, CD, t, z, m_0, m_1, c$), we can be sure of the values of A, i, s and t from the start of the experiment, but we do not necessarily know the values of CU, CD, z, m_0, m_1 or c ahead of the challenge.

Game *h*.2. We note that, by the first case of the confidentiality predicate, all users that $\pi^s_{A,i}$ intends as recipients when processing the $\mathcal{O}$-Encrypt(A, i, s, m_0, m_1) query must be uncorrupted and thus, as a result of **Game 2**, $\pi^s_{A,i}$ has received all such self-signing keys for the users in CU as well as all device and identity keys of the devices in CD without modification. Further, the second case of the confidentiality predicate requires that none of the devices representing any of the recipient users has been corrupted, even those which $\pi^s_{A,i}$ is not aware of (such as those which were added after initialisation of stage t). By **Game 2** and the confidentiality predicate, it follows that, all recipient devices, be they direct or indirect, have had their device and identity keys distributed to each other without modification.

In **Game *h*.2** to **Game *h*.8** we reason about an adversary breaking the security of the Olm channels used for either the initial Megolm key distribution, or its redistribution through state sharing. To do so, we first construct a set of the pairwise connections that must be secure:

$$\begin{aligned} \mathit{pairs} \coloneqq\ & \{\ \{\ (A,i),\ (B,j)\ \}\ :\ \forall\ (B,j) \in CD\ \} \\ & \cup\ \{\ \{\ (B,j),\ (B,k)\ \}\ :\ \forall\ B \in CU,\ 0 \leq j < \mathbf{D}[B],\ 0 \leq k < \mathbf{D}[B]\ \} \end{aligned}$$

We proceed to repeat the changes in **Game *h*.2** to **Game *h*.8** for each pair of devices in *pairs*, which we refer to as (C, k) and (D, l) in the following. We bound the number of users and devices by the respective experiment parameters, giving:

$$\mathsf{Adv}_{\mathsf{Gh.1}} \ \leq\ \big(\, n_u \cdot n_d + n_u \cdot (\frac{1}{2} \cdot n_d \cdot (n_d - 1))\,\big) \cdot \mathsf{Adv}_{\mathsf{Gh.2}}$$

Game *h*.3. In this game, we introduce an abort event $\mathit{abort}_{\mathit{olm\text{-}init}}$ that triggers if device (D, l) does not initiate the Olm channel between (C, k) and itself that is used to distribute a Megolm session. We note that the proof will proceed identically in either case, up to a change in notation. Thus, repeating this argument across both cases gives

$$\mathsf{Adv}_{\mathsf{Gh.2}} \ \leq\ 2 \cdot \mathsf{Adv}_{\mathsf{Gh.3}}$$

Game *h*.4. In this game, we introduce an abort event $\mathit{abort}_{\mathfrak{D}}$ that triggers if party D successfully evaluates $\mathsf{Olms.Enc}(st, \mathit{dpk}_{C,k}, \mathit{ipk}_{C,k}, m) \mapsto (st, c)$ without error and for which c is a pre-key message, with $c.\mathit{type} = 0$, but C did not generate the identity key $\mathit{ipk}_{C,k}$ or the ephemeral/fallback key $\mathit{epk}_{C,k,e}$ or $\mathit{fpk}_{C,k,e}$ that was used. Thanks to the signature checks in Olms.Enc, it follows that the device key package containing these keys, which we denote $\mathfrak{D}$, contained signatures over the device keys and fallback/ephemeral key that were not honestly generated by C.

When C generates device (C, k) we replace $\mathit{dpk}_{C,k}$ with pk from a strongly unforgeable digital signature challenger $\mathcal{C}_{\mathsf{DS}}$. When C is required to sign a

message of the form $\langle did_{C,k}, \mathtt{DPK}, dpk_{C,k}, \mathtt{IPK}, ipk_{C,k}\rangle$ for device and identity keys, $\langle \mathtt{EPK}, epk_{C,k,e}\rangle$ for ephemeral key $epk_{C,k,e}$, or $\langle \mathtt{FPK}, fpk_{C,k,e}\rangle$ for fallback key $fpk_{C,k,e}$, the challenger instead queries these messages to $\mathcal{C}_{\mathsf{DS}}$.

If party D receives a key package $\mathfrak{D} = \langle dpk, ipk, \sigma_d, epk, \sigma_e, fpk, \sigma_f\rangle$ for which either $epk_{C,k,e} = epk_0$ or $fpk_{C,k,e} = fpk_0$ and

1) $\mathsf{Ed25519.Verify}(dpk_{C,k}, \langle did_{C,k}, \mathtt{DPK}, dpk_{C,k}, \mathtt{IPK}, ipk_{C,k}\rangle, \sigma_d) \mapsto \mathbf{true}$,

2) $\mathsf{Ed25519.Verify}(dpk_{C,k}, \langle \mathtt{EPK}, epk_{C,k,e}\rangle, \sigma_e[e]) \mapsto \mathbf{true}$, or

3) $\mathsf{Ed25519.Verify}(dpk_{C,k}, \langle \mathtt{FPK}, fpk_{C,k,e}\rangle, \sigma_f[e]) \mapsto \mathbf{true}$,

but $ipk_{C,k}$ and $epk_{C,k,e}$ or $fpk_{C,k,e}$ were not generated honestly by C (and thus would trigger the abort event $abort_{\mathfrak{D}}$), then either σ_d, $\sigma_e[e]$ or $\sigma_f[e]$ is a valid forgery, and breaks the strong unforgeability of the digital signature scheme Ed25519. Thus:

$$\mathsf{Adv}_{\mathsf{Gh.3}} \;\leq\; \mathsf{Adv}_{\mathsf{Gh.4}} + \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Ed25519}}(\lambda, 1 + n_e + n_f)$$

Game *h*.5. In this game, the challenger guesses which ephemeral or fallback key pair will be used to initialise the Olm channel that will be used to either distribute the inbound Megolm session or state sharing message that initiates stage t for $\pi^s_{A,i}$. In particular, the adversary guesses a random key pair from the set $\{(esk_{C,k,e}, epk_{C,k,e}) : e \in [n_e]\} \cup \{(fsk_{C,k,f}, fpk_{C,k,f}) : f \in [n_f]\}$.

In what follows, we write $(esk_{C,k,e}, epk_{C,k,e})$ to signify the guessed key pair, regardless of whether it refers to an ephemeral or fallback key pair. Indeed, since the confidentiality predicate has disallowed all cases that might differentiate between the use of an ephemeral or fallback key, we can treat both cases equally. Thus:

$$\mathsf{Adv}_{\mathsf{Gh.4}} \;\leq\; (n_e + n_f) \cdot \mathsf{Adv}_{\mathsf{Gh.5}}$$

Game *h*.6. In this game we replace the values of rch_0 and ck_0 that are computed by (C, k) and (D, l) when they initiate the Olm session that will be used to distribute the challenge Megolm session with uniform random bit strings of the same length.

Let $\mathcal{O}\text{-}\mathsf{RO}(ms)$ denote the random oracle we use to replace calls of the form '$\mathsf{HKDF}(0, ms, \mathtt{OLM\text{-}RT})$'.

We bound the ability of an adversary to detect this change through the following security reduction. 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 $ipk_{D,l}$ and $epk_{C,k,e}$ respectively. The confidentiality predicate ensures that the challenger will not have to answer any queries that leak their secret counterparts, $isk_{D,l}$ or $esk_{C,k,e}$.

Whenever a session sets up their side of a new Olm channel, for which the Triple Diffie-Hellman key exchange includes at least one of $ipk_{D,l}$ and $epk_{C,k,e}$, the challenger replaces computation of the master secret, ms, and the subsequent key derivation of the root key, rch_0, and chain key, ck_0, by sampling two 32-byte

random bit strings in their place. The challenger stores these samples in a table addressed by the public keys that would have been used to compute the master secret. Computations that do not include the target key pairs proceed as normal: calculating the master secret, ms, before feeding it into our emulated random oracle.

Whenever the adversary queries our emulated random oracle, the challenger first checks if they need to respond using a value from the aforementioned table. To do so, they utilise the DDH oracle, $\mathcal{O}$-DDH, provided by $\mathcal{C}_{\mathsf{Gap\text{-}DH}}$ to check if the given value is a master secret, ms, that would have been computed using either $ipk_{D,l}$ or $epk_{C,k,e}$ but whose output was replaced with a random sample. Importantly, since the master secret contains the concatenation of three Diffie-Hellman key exchanges, we must also be sure to calculate and check the value for the other two Diffie-Hellman key exchanges. For each of the two remaining parts of ms, the challenger has access to at least one of the secret contributions and, thus, may compute the correct value. If all components match, then the challenger uses the randomly chosen keys from that list as the random oracle response, otherwise, it samples a new random value.

If, during this emulation process, the adversary causes $(ipk_{D,l}, epk_{C,k,e}, ms[0:31\mathsf{B}])$ to be queried to the $\mathcal{O}$-DDH oracle and the response is 1, meaning they have queried $\mathcal{O}$-RO(ms) with a string ms that contains the value of $g^{isk_{D,l} \cdot esk_{C,k,e}}$, then the adversary has solved the Gap-DH problem and the challenger may submit $ms[0\mathsf{B} : 31\mathsf{B}]$ to $\mathcal{C}_{\mathsf{Gap\text{-}DH}}$ under the knowledge that $ms[0:31\mathsf{B}] = g^{isk_{D,l} \cdot esk_{C,k,e}} = g^{xy}$.

On the other hand, if the adversary does not cause $(ipk_{D,l}, epk_{C,k,e}, ms[0\mathsf{B} : 31\mathsf{B}])$ to be queried to $\mathcal{O}$-DDH oracle, it follows that the values of rch_0 and ck_0 computed by (C,k) and (D,l) when initiating the target Olm session were sampled uniformly at random and are distributed in a manner independent of the protocol execution. Thus:

$$\mathsf{Adv}_{\mathsf{Gh.5}} \;\leq\; \mathsf{Adv}_{\mathsf{Gh.6}} + \mathsf{Adv}^{\mathsf{Gap\text{-}DH}}_{\mathsf{XDH}}(\lambda, \mathsf{poly}(\lambda))$$

Game *h*.7. Let p be the Olm channel epoch within which our target inbound Megolm session is distributed. In this game, we replace the computation of rch_p and ck_p with uniformly random keys. To do so, we apply a hybrid argument to the series of games **Game *h*.6.0**, **Game *h*.6.1**, ..., **Game *h*.6.*p***. In the g-th hybrid, we replace the first g values of the root and chain key after rch_0 and ck_0 ($rch_1, rch_2, \ldots, rch_h$ and $ck_1, ck_2, \ldots, ck_h$) with uniformly random keys such that **Game *h*.6.0** is exactly **Game *h*.6** and **Game *h*.6.*p*** is exactly **Game *h*.7**. We now consider the change in advantage between two consecutive hybrids, **Game *h*.6.*g*** and **Game *h*.6.(*g*+1)**.

Consider the following security reduction in which we construct an adversary against a KDF challenger, $\mathcal{C}_{\mathsf{mKDF}}$. First, note that the value of rch_g is uniformly distributed (by the previous hybrid) and thus can be replaced with the public salt provided by $\mathcal{C}_{\mathsf{mKDF}}$. It follows that we can replace the computation of keys (rch_{g+1}, ck_{g+1}) with a call to the challenge oracle provided by $\mathcal{C}_{\mathsf{mKDF}}$, '$rch_{g+1}, ck_{g+1} \leftarrow \mathcal{O}$-Challenge(OLM-RCH, 64B)'. Thus, we may bound the change

in advantage between any two hybrids with:

$$\mathsf{Adv}_{\mathsf{Gh.6.g}} \leq \mathsf{Adv}_{\mathsf{Gh.6.(g+1)}} + \mathsf{Adv}^{\mathsf{mKDF}}_{\mathsf{HKDF\text{-}RO}}(\lambda, 0)$$

Note that this analysis does not rely on the authenticity of the ratchet keys exchanged alongside messages, nor does it rely on the security of the resulting key exchange. The former, a lack of authenticity requirements, is because the use of an incorrect key contribution from the other party will simply cause the two parties to derive differing keys (and, as such, the protocol would fail). The latter, a lack of security requirements of the key exchange, is because the confidentiality predicate has disallowed any compromise of relevant Olm states. In other words, the continuous key exchange reduces to a symmetric ratchet in this context, relying on the security of the keys derived at the start: rch_0, ck_0. Since the epochs and message indices of Olm are independent and unrelated to the sessions, stages and message indices of the group messaging formalism, we bound the number of epochs within each Olm channel as an ad-hoc parameter that bounds usage of the Olm protocol, n_p. Together, we find that:

$$\mathsf{Adv}_{\mathsf{Gh.6}} \leq \mathsf{Adv}_{\mathsf{Gh.7}} + n_p \cdot \mathsf{Adv}^{\mathsf{mKDF}}_{\mathsf{HKDF\text{-}RO}}(\lambda, 0)$$

Game *h*.8. In this game, we replace the computation of each chain key, $ck_{p,q}$, and message key, $mk_{p,q}$, in each epoch p with uniformly random replacements. To do so, we apply a hybrid argument to the series of games **Game *h*.7.0**, **Game *h*.7.1**, …, **Game *h*.7.*pq***. In the g-th hybrid, we replace the first g values of the chain key and message key after $ck_{p,0}$ and $mk_{p,0}$ with uniformly random keys such that **Game *h*.7.0** is exactly **Game *h*.7** and **Game *h*.7.*pq*** is exactly **Game *h*.8**. We now consider the change in advantage between two consecutive hybrids, **Game *h*.7.*g*** and **Game *h*.7.(*g*+1)**.

Consider the chain key and message key generated at index $g + 1$. Rather than computing these keys through calls to $\mathsf{HMAC}(ck_{p,g}, \texttt{0x02})$ and $\mathsf{HMAC}(rch_{p,g}, \texttt{0x01})$, the challenger initialises a PRF challenger $\mathcal{C}_{\mathsf{PRF}}$, submits queries with `0x02` and `0x01`, then replaces ck_{g+1} and mk_{g+1} with the output of said query (respectively). We note that if the bit b sampled by $\mathcal{C}_{\mathsf{PRF}}$ is 0, then we are in **Game *h*.7.*g***, else we are in **Game *h*.7.*g*+1**. Thus:

$$\mathsf{Adv}_{\mathsf{Gh.7.g}} \leq \mathsf{Adv}_{\mathsf{Gh.7.(g+1)}} + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC\text{-}RO}}(\lambda, 2)$$

As in **Game *h*.7**, we bound the number of messages in a single Olm epoch as an ad-hoc parameter, n_q. Together, we find that:

$$\mathsf{Adv}_{\mathsf{Gh.7}} \leq \mathsf{Adv}_{\mathsf{Gh.8}} + n_p \cdot n_q \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC\text{-}RO}}(\lambda, 2)$$

Game *h*.9. In this game, we replace the ciphertext of either the Megolm session distribution or state sharing message with a random replacement of the same length. The challenger stores a mapping of such ciphertexts to their intended plaintext counterparts, which it uses to simulate decryption for the recipient.

Specifically, we introduce the following reduction. We initialise a IND$-CPA challenger, $\mathcal{C}_{\mathsf{IND\$\text{-}CPA}}$, for one-time AEAD schemes. When the relevant message (depending on the case we are in) is to be encrypted, the challenger replaces this with a call to the encryption oracle of the IND$-CPA challenger. If the bit b sampled by $\mathcal{C}_{\mathsf{IND\$\text{-}CPA}}$ is 0, we are in **Game *h*.8**, else we are in **Game *h*.9**. We note that any adversary $\mathcal{A}$, capable of distinguishing between the two games can be used to break the IND$-CPA security of the AEAD scheme. By **Game *h*.8**, the message key mk_l is already uniformly random and independent and this replacement is sound. Thus:

$$\mathsf{Adv}_{\mathsf{Gh.8}} \leq \mathsf{Adv}_{\mathsf{Gh.9}} + \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{Olm\text{-}AEAD}}(\lambda, 1)$$

Game *h*.10. In this game, we replace all outputs of the Megolm ratchet in epoch t with uniformly random keys.

Specifically, we introduce the following reduction. 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'. 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.Update. We note if the bit b sampled by $\mathcal{C}_{\mathsf{KIND}}$ is 0, we are in **Game *h*.9**, else we are in **Game *h*.10**. By **Game *h*.9**, the initial state of the Megolm ratchet maintained by $\pi^s_{A,i}$ and its communicating partners is already uniformly random and independent of the protocol execution, permitting this replacement. If a session state is compromised, we may utilise the KIND.Corrupt oracle provided by $\mathcal{C}_{\mathsf{KIND}}$ to reveal the internal ratchet state, then continue to execute this instance of MgRatchet honestly from this point onwards. Note that the confidentiality predicate disallows challenge encryptions from being issued after a session state has been compromised. It follows that 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. Thus:

$$\mathsf{Adv}_{\mathsf{Gh.9}} \leq \mathsf{Adv}_{\mathsf{Gh.10}} + \mathsf{Adv}^{\mathsf{KIND}}_{\mathsf{FF\text{-}PRG}}(\lambda, n_m)$$

Note that the Megolm ratchet, despite initialising an internal state with 1024 bits of randomness, generally relies on only 256 bits of this state being secret. Taking forward secrecy as an example, in the most common case the Megolm ratchet only applies the ratcheting operation to 256 bits of its internal state. We paper over this minor syntactic incompatibility with the FFPRG formalism by having the Megolm ratchet initialise its secret state with a value four times the bit-length of the FFPRG security parameter. Refer back to Figure 3.12 to see how this is done.

Game *h*.11. In this game, when the adversary issues the encryption query $\mathcal{O}$-Encrypt(A, i, s, m_0, m_1), the challenger instead initialises a confidentiality authenticated encryption challenger $\mathcal{C}_{\mathsf{IND\$\text{-}CPA}}$ for the AEAD scheme constructed by Megolm. The challenger forwards m_b to the $\mathcal{C}_{\mathsf{IND\$\text{-}CPA}}$ challenger for any challenge encryption query. We note that by **Game *h*.10**, 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 $\mathcal{C}_{\mathsf{IND\$\text{-}CPA}}$. It is straightforward to see that the advantage of our adversary, $\mathcal{A}$, in guessing the bit b in the DOGM security experiment is now equal to that of our reduction in guessing $\mathcal{C}_{\mathsf{IND\$\text{-}CPA}}$'s bit b:

$$\mathsf{Adv}_{\mathsf{Gh.10}} \leq \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{Megolm}}(\lambda, 1)$$

This completes our analysis of *Case 2*.

We combine the two cases above to arrive at an upper bound for the advantage of any PPT adversary.

$$
\begin{aligned}
&\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{MX\text{-}DOGM}}(n_u, n_d, n_i, n_s, n_m) \\
&\quad\leq\ (n_u \cdot n_d \cdot n_i \cdot n_s) \cdot \Big[\quad \textit{// Case 1: Authentication} \\
&\qquad (n_u \cdot n_d) \cdot \big[\\
&\qquad\quad 2 \cdot \big(\\
&\qquad\qquad \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Ed25519}}(\lambda, 3) \ +\ \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Ed25519}}(\lambda, n_d) \ + \\
&\qquad\qquad \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Ed25519}}(\lambda, 1 + n_e + n_f) \ + \\
&\qquad\qquad (n_e + n_f) \cdot \big[\\
&\qquad\qquad\quad \mathsf{Adv}^{\mathsf{Gap\text{-}DH}}_{\mathsf{XDH}}(\lambda, \mathsf{poly}(\lambda)) \ + \\
&\qquad\qquad\quad n_p \cdot \big(\\
&\qquad\qquad\qquad \mathsf{Adv}^{\mathsf{mKDF}}_{\mathsf{HKDF\text{-}RO}}(\lambda, 0) + n_q \cdot [\, \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC\text{-}RO}}(\lambda, 2) + \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Olm\text{-}AEAD}}(\lambda, 1) \,] \\
&\qquad\qquad\quad \big) \\
&\qquad\qquad \big] \\
&\qquad\quad \big) \\
&\qquad \big] \ +\ \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Ed25519}}(\lambda, n_m) \\
&\quad \Big] \ +\ \Big[\quad \textit{// Case 2: Confidentiality} \\
&\qquad n_u \cdot \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Ed25519}}(\lambda, 3) \ +\ n_u \cdot \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Ed25519}}(\lambda, n_d) \ + \\
&\qquad (n_u \cdot n_d \cdot n_i \cdot n_s \cdot n_m) \cdot (n_u \cdot n_d \cdot n_i \cdot n_s) \\
&\qquad \cdot (\, n_u \cdot n_d + n_u \cdot (\tfrac{1}{2} \cdot n_d \cdot (n_d - 1)) \,) \cdot 2 \cdot \big[\\
&\qquad\quad \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Ed25519}}(\lambda, 1 + n_e + n_f) \ + \\
&\qquad\quad (n_e + n_f) \cdot \big(\\
&\qquad\qquad \mathsf{Adv}^{\mathsf{Gap\text{-}DH}}_{\mathsf{XDH}}(\lambda, \mathsf{poly}(\lambda)) \ + \\
&\qquad\qquad n_p \cdot \big(\\
&\qquad\qquad\quad \mathsf{Adv}^{\mathsf{mKDF}}_{\mathsf{HKDF\text{-}RO}}(\lambda, 0) + n_q \cdot [\, \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC\text{-}RO}}(\lambda, 2) + \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{Olm\text{-}AEAD}}(\lambda, 1) \,] \\
&\qquad\qquad \big) \ +\ \mathsf{Adv}^{\mathsf{KIND}}_{\mathsf{FF\text{-}PRG}}(\lambda, n_m) + \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{Megolm}}(\lambda, 1) \\
&\qquad\quad \big) \\
&\qquad \big] \\
&\quad \Big]
\end{aligned}
$$

Observe that the above bound is a polynomial function of the experiment parameters, $\Lambda_{\mathsf{DOGM}} = (n_u, n_d, n_i, n_s, n_m)$, usage parameters of the Olm channels, (n_e, n_f, n_p, n_q), and the respective advantage against the security of each primitive used. It follows that the advantage of any PPT adversary is at most a negligible function of the security parameter used to instantiate each primitive.

This completes our proof. □

6.1.6 Interpretation

Our analysis shows that Matrix is able to provide confidentiality and authentication to its users, under certain circumstances, and providing that the vulnerabilities reported in Section 3.3 have been fixed. Furthermore, these guarantees would also *only* apply when composed with a formally analysed cryptographic backup solution, which is not captured by our model.

A caveat. This latter point is *essential* in order for Matrix to provide any authentication guarantees in practice. At the time of writing, the Matrix specification provides a single means of backing up Megolm sessions: "Server-side key backups" in the `m.megolm_backup.v1.curve25519-aes-sha2` configuration [Mata, Section 11.12.3.2.2]. This scheme does not authenticate the creator of the session backup, allowing a malicious server to trivially inject historical messages into any client. As we have seen, since the time and date of a message displayed to users is determined by the server, there is no meaningful separation between historical and "new" messages: this issue undermines authentication wholly.

Nonetheless, we stress that this issue is outside of our analysis, and press forward with an interpretation of our results.

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.[6]

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).[7]

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).

[6] 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 $\mathcal{O}$-AddMember and $\mathcal{O}$-RemoveMember oracles).

[7] This maps back to a particular sending session, such as $\pi^r_{B,j}$, within the DOGM experiment.

FS and PCS in primitives to enforce group membership. 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 [BSJ+17], and PCS ensure that messages sent *after* compromise remain secure (assuming the adversary remains passive) [CCG16]. Our security predicates in Section 6.1.5 essentially establish that neither security goal, as defined in the cryptographic literature, is attained by Matrix when long-term secrets are compromised.[8] 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.

6.2 WhatsApp

In this section, we analyse the security of multi-device group messaging in WhatsApp within the device-oriented group messaging with revocation formalism.

6.2.1 Contributions

This section includes the following contribution.

Contribution 6.2. We prove the security guarantees of WhatsApp's multi-device group messaging within the DOGM model, which we specify through the predicates WA-DOGM.CONF and WA-DOGM.AUTH, demonstrating how session management, device management and history sharing interact with the security guarantees of the underlying messaging protocol.

We express the multi-device group messaging functionality of WhatsApp within the DOGM with revocation model. We proceed to derive (and prove) the security guarantees of WhatsApp's multi-device group messaging within this model, capturing the interactions between the messaging channels, device management and history sharing sub-protocols. In doing so, we show that WhatsApp's use of device revocation allows a user to effectively recover security after a known compromise.

6.2.2 Scope & Limitations

Like our analysis of Matrix in the previous section, the accuracy of our results fundamentally relies on the accuracy of our description of WhatsApp in Section 4.2. As we discuss in Section 2.1, however, this is further amplified by the lack of access to WhatsApp's source code, requiring a reverse-engineering effort

[8] Our analysis shows that PCS can be achieved when only Megolm states are compromised.

to determine its functionality. As such, the same scope limitations discussed in Section 6.2.9 apply to our security analysis.

In addition, we *further* simplify WhatsApp's functionality (in comparison to our description in Section 4.2) in order to make the analysis tractable. We document this simplification primarily through our expression of WhatsApp in the Device-Oriented Group Messaging model in Section 6.2.7.

The results of our analysis are also limited by the constraints of our model. For example, while WhatsApp does provide immediate decryption [ACD19] in its pairwise channels, neither our description nor analysis reflect this. Similarly, while WhatsApp provides immediate decryption within group messaging, we do not capture this in our modelling (in contrast, [BCG23] *does*).

Notably, since pairwise channels are used to distribute group messaging sessions, the compromise of a single cached message key for the pairwise channel allows the adversary to establish a new Sender Keys session; subverting the *implied* limitations of such cached keys. That is, immediate decryption is expected to allow the processing of ciphertexts out-of-order while only modifying the temporal security guarantees of those messages which were skipped. Concretely, if the message at epoch p and index q has its message key, $mk[p, q]$ cached, we expect compromise of this key material should only affect the message at (p, q). However, since this message may distribute a Sender Keys session, the compromise of $mk[p, q]$ allows the adversary to either (a) inject an inbound Sender Keys session under their control, or (b) decrypt the inbound Sender Keys session state. The former case may be used to break authentication for a complete Sender Keys epoch, while the latter would break confidentiality for that epoch.

Security predicates. We also note that the security predicates we develop in this section are not necessarily *tight*. That is, the security predicates may mark certain situations as insecure, even in cases where the protocol does in fact guarantee security. We do so for two reasons. First, it enables us to simplify our proof in certain cases. Second, we hope that providing a simpler predicate aids in the interpretation of that predicate and, thus, the interpretation of this work. For example, while it is possible for a particular two-party channel to heal using its asymmetric ratchet, there is no way for a user to observe that a message has been exchanged over a secure channel rather than an insecure one. Where possible, our predicates focus on observable events that are exposed to a user or the client representing them.

Key distribution. Our analysis assumes trusted distribution of user identities. Doing so allows us to focus on other aspects of the protocol. We refer to WhatsApp's documentation on out-of-band verification [Wha23a, Page 24] and key transparency [Wha23b] whitepaper for WhatsApp's claimed assurances in this area. In keeping with the theme of this work, though, we caution against relying solely on whitepapers for establishing security guarantees and encourage such an independent analysis of this feature and its composition with the other protocols in WhatsApp.

As a consequence of the above, it is possible for the server to reset a user's cryptographic identity. Clients default to *not* displaying such a change to users.

As discussed, by sheer volume of the required work, we have to consider this out of scope of our analysis. We note that this weakness might be mitigated to some extent by WhatsApp's use of key transparency, but again caution that without a formal analysis the question of what guarantee can be expected is open.

Server-controlled group membership. As we discuss in Section 6.2.9, group membership is not cryptographically authenticated in WhatsApp, something which has been established in prior works [RMS17, BCG22, BCG23]. This is captured in our model by providing the adversary with the ability to orchestrate and schedule group management operations, a standard approach in the analysis of group key exchange and messaging models. Nonetheless, this greatly limits the real-world security that WhatsApp can provide and, thus, the security results we derive should be interpreted with this limitation in mind. Specifically, while a user may be sure that their messages are secure within the group of users in their user interface, they do not have meaningful control over this list of users and there is no guarantee that the group members have a consistent view of this list.

Note that, while both WhatsApp and the public-key orbits formalism from Section 5.4.1 support the ability to refresh a user's generation without making changes to their device composition, we choose not to capture this in our model. While this adds meaningful guarantees in practice, by providing a concrete time bound on how long it takes to detect device revocation, the DOGM does not capture this since there is no clock with which to implement such expiry. This is a limitation of our model, a choice we made in an effort to limit complexity.

6.2.3 Primitives

We collect the cryptographic primitives used by WhatsApp, discuss the implicit security requirements placed on them, and specify the assumptions we make regarding these primitives in the analysis that follows. Given the similarity in design between Matrix and WhatsApp, the primitives they use require similar assumptions.

Key derivation. WhatsApp uses both the HMAC and HKDF functions for various purposes related to key derivation.

The chain key for each Sender Keys session is initialised with the system PRNG, which we assume to provide pseudorandom bits. It is ratcheted forward through a call to the HMAC function, '$\mathsf{HMAC}(ck, \texttt{0x02})$', for which we require the output to also be pseudorandom. Per-message key material is derived, similarly, through a call to the HMAC function, '$\mathsf{HMAC}(ck, \texttt{0x01})$', for which we require the output to also be pseudorandom. Thus, we require HMAC to implement a secure PRF.

The initialisation vector and encryption key are then derived from the per-message key material through a call to the HKDF function, '$\mathsf{HKDF}(\varnothing, mk, \texttt{WhisperGroup}, 50\mathsf{B})$'. We have that the per-message key material, mk, is pseudorandom, and requires a pseudorandom output (albeit a longer output). The definition of KDF security in [Kra10] assumes that a fresh, uniformly random

salt is used with each piece of private key material. However, as is common in practice, WhatsApp uses HKDF with a constant salt (in this context). Thus, we cannot directly require that such use of HKDF achieves KDF security as defined in [Kra10]. If the private key material being used is already pseudorandom, however, we do not require the extraction functionality of the KDF. Instead, we simply require that the HKDF itself acts as a pseudorandom function. Thus, we require that HKDF implements a secure PRF.

As discussed below, and in Section 6.2.5, we rely on an existing analysis of Signal pairwise channels [CCD+20, Thereom 1] for our analysis of WhatsApp's pairwise channels which, in turn, relies on the Gap DH assumption [OP01] for Curve25519 and models two particular key derivations as random oracles. Since these key derivations make use of the HMAC and HKDF algorithms, we choose to model the HMAC and HKDF algorithms directly as random oracles (separate to those used for the aforementioned key derivations in WhatsApp's pairwise channels).

Joint Security of X25519 and XEd25519. As we have seen, WhatsApp uses the X25519 scheme for Curve25519-based DH key exchange [Ber06] within their construction of secure pairwise channels, which we denote XDH. In our security analysis, we model the underlying Signal pairwise channels as a MSKE protocol [CCD+20, Definition 1], and make the assumption that WhatsApp implements an MS-IND secure MSKE protocol. Thus, our analysis does not interact with XDH directly. Nonetheless, our intermediate result for WhatsApp's composed DM scheme, Theorem 6.9, and the main result, Theorem 6.18, rely on [CCD+20, Thereom 1] which, in turn, relies on the Gap DH assumption [OP01] for Curve25519 and models two particular key derivations as random oracles.

For signatures, WhatsApp uses the Curve25519 variant of XEdDSA [Per16], which we represent through XEd = (Gen, Sign, Verify) where XEd.Gen is equivalent to XDH.Gen. We assume that XEd provides SUF-CMA security, see Definition 2.35, when the keys are used solely for digital signatures. This is not always the case, however: identity keys are used jointly for XEdDSA signatures and X25519 key exchange. To avoid the explicit assumption of joint security, our analysis splits up the identity key into separate XEdDSA and X25519 key pairs. Of course, our analysis still implicitly relies on the assumption that they provide joint security for signatures and key exchange. See Remark 6.14.

Authenticated encryption with associated data. We assume that WhatsApp's use of AES-CBC and HMAC-SHA256, as we describe in both WA-AEAD and UNI, provides one-time IND$-CPA and EUF-CMA security for AEAD schemes (as defined in Section 2.3.5). Prior work [BN08, BDJR97, BR00] demonstrates that this is a reasonable assumption.

6.2.4 Device Management

We proceed to analyse the security of device management in WhatsApp. To do so, we express the device management sub-protocol of WhatsApp as a public key orbit (introduced in Section 5.4.1). Doing so requires minimal changes to

our description of device management in Section 4.2.1. Primarily, we make syntactic changes to ensure compatibility with the public key orbit formalism. See Figure 6.4 for the details of this instantiation.

Definition 6.5. WA-PO is a public key orbit that implements the PO formalism with algorithms (Setup, Attract, Repel, Refresh, Orbit) in Figure 6.4.

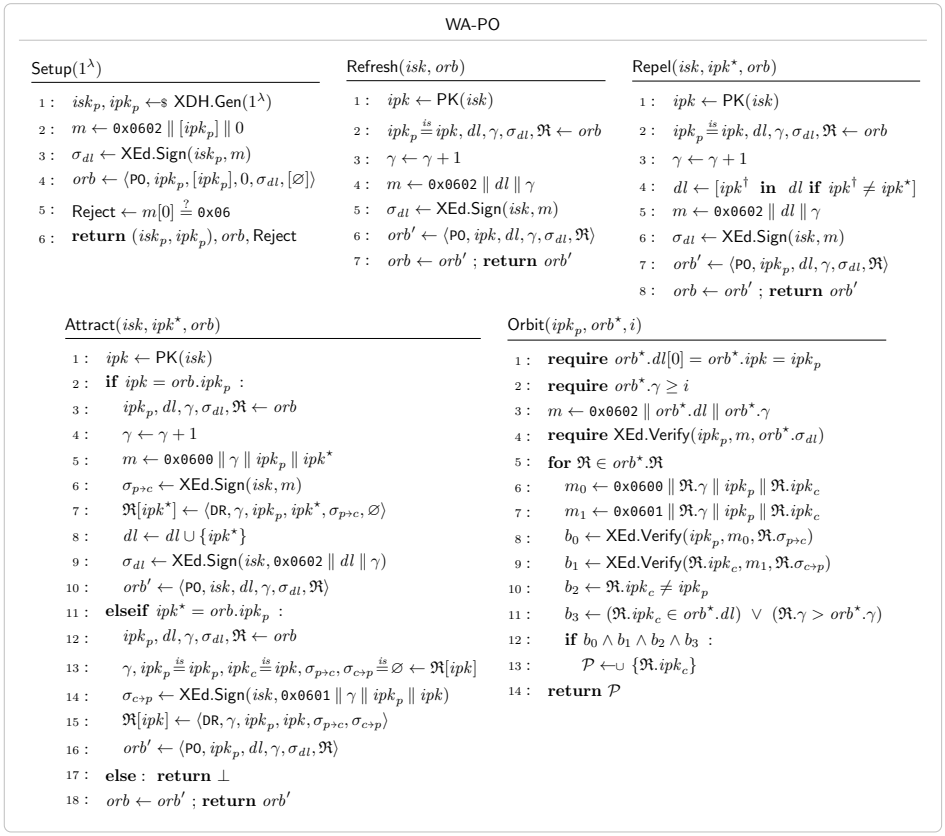


Figure 6.4. Device management in WhatsApp expressed as a public-key orbit.

We state and prove that our instantiation of device management in WhatsApp as a public key orbit, WA-PO, achieves weak public-key orbit security.

Theorem 6.6 (Security of WA-PO)**.** For any probabilistic polynomial-time algorithm $\mathcal{A}$ playing the wPO security game instantiated with the WA-PO scheme, making at most n_{ch} queries to $\mathcal{O}$-Challenge, at most n_σ queries to $\mathcal{O}$-Sign per signing key, and at most n_g cumulative queries to the $\mathcal{O}$-Attract, $\mathcal{O}$-Repel and $\mathcal{O}$-Refresh oracles, we have:

$$\mathsf{Adv}^{\mathsf{wPO}}_{\mathsf{WA\text{-}PO}}(\lambda, n_{ch}, n_\sigma, n_g) \ \leq \ (n_{ch} + 1) \cdot \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, n_\sigma + 2 \cdot n_g)$$

The instantiation we study hard codes the above security parameter to $\lambda = 256$.

Proof. We separate our proof into two cases. In *Case 1*, we restrict the adversary to winning the game by causing w_0 to be set to **true**, i.e. by successfully adding a companion device to the orbit that is not considered linked by the primary device. In *Case 2*, we restrict the adversary to winning the game by causing w_1 to be set to **true**, i.e. by adding an honest companion device to the orbit that was not linked by the companion device itself.

Let $\mathsf{Adv}_{\mathsf{w0}}$ and $\mathsf{Adv}_{\mathsf{w1}}$ be the respective advantages in *Case 1* and *Case 2*, giving:

$$\mathsf{Adv}^{\mathsf{wPO}}_{\mathsf{WA\text{-}PO},\mathcal{A}}(\lambda, n_{ch}, n_\sigma, n_g) \;\leq\; \mathsf{Adv}_{\mathsf{w0}} + \mathsf{Adv}_{\mathsf{w1}}$$

Case 1: Inject a companion device without the primary device

Game 0. We inline the WA-PO scheme into the wPO security game restricted to *Case 1*. This is a syntactic edit such that, $\mathsf{Adv}_{\mathsf{w0}} = \mathsf{Adv}_{\mathsf{G0}}$.

Game 1. The challenger proceeds to abort the game if a query to the $\mathcal{O}$-Compromise oracle is issued at any point in the experiment. Since the adversary may only win the game by causing w_0 to be set to **true**, and this requires that $corr =$ **false**, i.e. $\mathcal{O}$-Compromise has never been called, this change does not reduce the advantage of the adversary. Thus:

$$\mathsf{Adv}_{\mathsf{G0}} \;\leq\; \mathsf{Adv}_{\mathsf{G1}}$$

Game 2. We introduce an abort event, $abort_{forge}$, that is triggered if the challenger's execution of $\mathsf{WA\text{-}PO.Orbit}(pk_p,\ orb^\star,\ \gamma^\star)$ has a call to $\mathsf{XEd.Verify}(pk_p,\ m,\ \sigma)$ evaluate to **true** for a message m that was not honestly signed through a call to $\mathsf{XEd.Sign}(isk_p,\ m)$ in service of a $\mathcal{O}$-Sign, $\mathcal{O}$-Attract, $\mathcal{O}$-Repel or $\mathcal{O}$-Refresh query.

We bound the probability of $abort_{forge}$ occurring with the following security reduction. We construct an adversary, $\mathcal{B}$, against an EUF-CMA challenger for the DS digital signature scheme, which we denote $\mathcal{C}_{\mathsf{DS}}$. We proceed to emulate a variant of **Game 1** to our inner adversary, $\mathcal{A}$, but embed the challenge verification key, pk, output by $\mathcal{C}_{\mathsf{DS}}$ as the verification key of the primary device, pk_p. Whenever a signature must be produced using its signing counterpart, sk_p, we instead make an analogous call to $\mathcal{C}_{\mathsf{DS}}$'s $\mathcal{O}$-Sign oracle. We save the input message for each call in a set then, if at any point we have a call of the form $\mathsf{XEd.Verify}(pk_p,\ m,\ \sigma)$ evaluate to **true** for a message m that is not in our set, the message m and signature σ form a valid forgery. We proceed to return the pair (m, σ) to the EUF-CMA challenger, $\mathcal{C}_{\mathsf{DS}}$, and win the experiment.

It follows that we can bound the probability of the adversary $\mathcal{A}$ triggering $abort_{forge}$ by the advantage of any PPT adversary winning the EUF-CMA security experiment for the Ed25519 scheme, when restricted to at most $n_\sigma + 2 \cdot n_g$ signing queries. Thus:

$$\mathsf{Adv}_{\mathsf{G1}} \;\leq\; \mathsf{Adv}_{\mathsf{G2}} + \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, n_\sigma + 2 \cdot n_g)$$

Thanks to the changes introduced in **Game 2**, we can be sure that all message-signature pairs that verify as originating from pk_p were, indeed, honestly generated by the challenger. Further, since WA-PO.Reject? will prevent any calls to $\mathcal{O}$-Sign whose message starts with `0x06`, we can be sure that all message-signature pairs that verify as originating from pk_p during execution of WA-PO.Orbit were honestly generated while processing a WA-PO.Setup, WA-PO.Attract, WA-PO.Repel or WA-PO.Refresh call.

In order to win the game, the adversary must provide an orbit state, $orb^\star$, and generation, $\gamma^\star$, that when processed by PO.Orbit, outputs a set of devices $\mathcal{P}$ containing a companion device key that the primary device does not believe is in its orbit at the given generation.

Specifically, w_0 requires that there $\mathcal{P}$ contains a pk for which $pk \notin \mathcal{T}'$. If there exists such a pk, then the provided orbit state, $orb^\star$, must contain two signatures:

1) A device list signature, σ_{dl}, produced by pk_p. The signed message, m, must contain a device list, $m.dl$, for which $m.dl[0] = pk_p$ and $pk \in m.dl$; and a generation, $m.\gamma$, greater than or equal to the given generation, $\gamma^\star$.

2) An account signature, $\sigma_{p \to c}$, produced by pk_p. The signed message, m_0, must contain the companion device's verification key, $m_0.ipk_c = pk$, and the device list generation, $m_0.\gamma$, for which this signature was created.

Alternatively, if the account signature is newer than the device list signature, such that $m_0.\gamma > m.\gamma$, then WA-PO.Orbit will allow the companion key not to be present in the device list, i.e it will accept device lists for which $pk \notin m.dl$. We denote these as case (a) and (b), respectively, and show that if the adversary wins the game while either one is true, this leads to a contradiction.

Suppose that we are in case (a) and the adversary has triggered w_0. Let pk denote the key in $\mathcal{P}$ that causes w_0 to be satisfied. It follows that the device list in $orb^\star$ contains pk and is signed alongside a generation, $orb^\star.\gamma$, that is greater than or equal to $\gamma^\star$. This may only be the case if the primary device had pk in its device list at generation $orb^\star.\gamma$ and, thus, pk must have an active "to" link entry in $\mathcal{T}[orb^\star.\gamma]$. However, since w_0 was triggered, there does not exist a generation γ greater than or equal to $\mathsf{max}(orb^\star.\gamma, \gamma^\star)$ for which pk has an active "to" link entry in $\mathcal{T}[\gamma]$. In other words, we have found ourselves at a contradiction.

Now, suppose that we are in case (b) and the adversary has triggered w_0. Let pk denote the key in $\mathcal{P}$ that causes w_0 to be satisfied. It follows that there exists device record, $\mathfrak{R}$, in $orb^\star$ for pk that is signed alongside a generation, $m_0.\gamma$, that is greater than or equal to $\gamma^\star$. This may only be the case if the primary device had pk in its device list at generation $m_0.\gamma$ and, thus, pk must have an active "to" link entry in $\mathcal{T}[m_0.\gamma]$. However, since w_0 was triggered, there does not exist a generation γ greater than or equal to $\mathsf{max}(orb^\star.\gamma, \gamma^\star)$ for which pk has an active "to" link entry in $\mathcal{T}[\gamma]$. In other words, we have found ourselves at a contradiction.

It follows that it is not possible for the adversary to win **Game 2**, i.e. $\mathsf{Adv}_{\mathsf{G2}} = 0$, completing our analysis of *Case 1.* Note that we did not rely on the security of

signatures created by companion devices and, as such, need not consider the adversary's use of the $\mathcal{O}$-Eject oracle.

Case 2: Inject a companion device without the companion itself

Game 0. We inline the WA-PO scheme into the wPO security game restricted to *Case 2*. This is a syntactic edit such that, $\mathsf{Adv}_{\mathsf{w1}} = \mathsf{Adv}_{\mathsf{G0}}$.

Game 1. In this game, we guess one of the companion public keys that causes w_0 to be set to **true**, such that '$\mathcal{P} \cap \mathcal{C} \setminus \mathcal{F}$' is not empty. If the guess turns out to be incorrect at any point, the challenger aborts the game.

Note that the ejection of a device through the $\mathcal{O}$-Eject oracle removes the device from the set $\mathcal{C}$. It follows that if we guess a device that is later ejected, it cannot be a companion public key that causes w_0 to be set to true, our guess is incorrect, and it is determined that the challenger will abort the game.

Overall, there are at most n_{ch} possible choices of companion device, giving:

$$\mathsf{Adv}_{\mathsf{G0}} \leq n_{ch} \cdot \mathsf{Adv}_{\mathsf{G1}}$$

Game 2. Let (sk', pk') denote the key pair guessed in **Game 1**. We introduce an abort event, $abort_{forge}$, that is triggered if the challenger's execution of $\mathsf{PO.Orbit}(pk_p, orb^{\star}, orb.\gamma)$ has a call to $\mathsf{XEd.Verify}(pk', m, \sigma)$ evaluate to **true** for a message m that was not honestly signed through a call to $\mathsf{XEd.Sign}(sk', m)$ in service of a $\mathcal{O}$-Sign or $\mathcal{O}$-Attract query.

We bound the probability of $abort_{forge}$ occurring with an analogous security reduction to that which we use in **Game 2** of *Case 1*, albeit with the challenge embedded in the key pair guessed in **Game 1**: (sk', pk').

It follows that we can bound the probability of the adversary $\mathcal{A}$ triggering $abort_{forge}$ by the advantage of any PPT adversary winning the EUF-CMA security experiment for the Ed25519 scheme, when restricted to at most $n_\sigma + n_g$ signing queries. Thus:

$$\mathsf{Adv}_{\mathsf{G1}} \leq \mathsf{Adv}_{\mathsf{G2}} + \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, n_\sigma + n_g)$$

Thanks to the changes introduced in **Game 2**, we can be sure that all message-signature pairs that verify as originating from pk' were, indeed, honestly generated by the challenger on behalf of pk'. Further, since WA-PO.Reject? will prevent any calls to $\mathcal{O}$-Sign whose message starts with `0x06`, we can be sure that all message-signature pairs that verify as originating from pk' during execution of WA-PO.Orbit were honestly generated while processing a WA-PO.Attract call.

We now consider the advantage of our adversary in **Game 2**. Recall that, by the guess in **Game 1**, the signing key pk' was present in the computed public key orbit, such that $pk' \in \mathcal{P}$, and is a tracked companion key, such that $pk; \in \mathcal{C}$, *but* is not present in $\mathcal{F}$. Note that $\mathfrak{R}.\sigma_{c \nrightarrow p}$ cannot have been an output of $\mathcal{O}$-Sign since it starts with `0x06`. Further, since $pk' \notin \mathcal{F}$ we can determine that no

linking query was made from pk' to pk_p, i.e. the adversary did not issue the query $\mathcal{O}\text{-}\mathsf{Attract}(pk', pk_p)$ at any point in the experiment. It follows that the challenger did not execute $\mathsf{XEd.Sign}(sk', \texttt{0x0601} \,\|\, \gamma \,\|\, ipk_p \,\|\, pk')$ at any point the experiment. By **Game 2**, we know that no such valid signature σ' exists for which $\mathsf{XEd.Verify}(pk', \texttt{0x0601} \,\|\, \gamma \,\|\, ipk_p \,\|\, pk, \sigma')$ evaluates to true and, therefore, it is not possible for the public key orbit $\mathcal{P}$ output at the end of the experiment to contain pk'. Thus, the adversary cannot win, i.e. $\mathsf{Adv}_{\mathsf{G2}} = 0$, and we find:

$$\mathsf{Adv}_{\mathsf{G1}} \;\leq\; \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, n_\sigma + n_g)$$

This completes our analysis of *Case 2*.

Having bound the advantage of our adversary against *Case 1* and *Case 2* separately, we recombine them to find:[9]

$$\mathsf{Adv}^{\mathsf{wPO}}_{\mathsf{WA\text{-}PO},\mathcal{A}}(\lambda, n_{ch}, n_\sigma, n_g) \;\leq\; (n_{ch}+1) \cdot \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, n_\sigma + 2 \cdot n_g)$$

Observe that the above bound is a polynomial function of the experiment parameters (n_{ch}, n_σ, n_g), and the upper bound of the advantage of any PPT adversary against the EUF-CMA security of XEd. It follows that the advantage of any PPT adversary against the weak public key orbit security of WA-PO is at most a negligible function of the security parameter, providing that XEd is an EUF-CMA secure digital signature scheme.

This completes our proof. □

6.2.5 Pairwise Channels

In this section, we analyse the security of WhatsApp's pairwise channels with session management. We start by expressing WhatsApp's DM scheme within the pairwise channels formalism introduced in Section 5.4.2. We then proceed to derive and prove the security guarantees that this instantiation provides within the PAIR-SEC security experiment specified in Definition 5.11.

We now detail the simplifying assumptions and changes we have made while transforming WhatsApp's implementation into an instance of PAIR, which we name WA-PAIR.

Avoiding the joint security of X25519 and XEdDSA. First, we remove the pre-key signatures and their verification checks. We do this by allowing the challenger to distribute participant's signed pre-keys in a trusted manner at the beginning of the security experiment, in the same way that they distribute their identity keys. With this change it is not possible for an adversary to modify or replace a signed pre-key with one that gains them any advantage (in either learning the bit b or triggering an authentication win). Removing the pre-key signatures allows us to side-step the issue of identity keys being used for both signatures and key exchange. Instead, we may simply rely on the security of the

[9] We simplify the expression by noting that $\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, n_\sigma + n_g) \leq \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, n_\sigma + 2 \cdot n_g)$ in order to combine the two terms.

identity keys for key exchange. Nonetheless, applying our result to WhatsApp still requires the implicit assumption that such dual-use of identity keys does not affect the security of either the XDH key exchange and the XEd signatures. This mirrors a similar choice made in the prior analysis of X3DH that we rely on [CCD+20].

Message routing. In our description of WA-PAIR, reflecting the behaviour of WhatsApp, the session used to encrypt or decrypt a given message is determined by the client. To encrypt a messages, they use their most recently active session with that recipient. While, to decrypt a message, clients try each active session in the order of most recent use. In the PAIR-SEC model, the adversary is given the ability to select the particular session that the client should use. This transformation is safe in that it only provides the adversary with additional power: they are able to simulate the original functionality by triggering the appropriate sequence of operations with the appropriate session identifiers.

Rather than detecting pre-key ciphertexts during decryption, and passing them to the appropriate function, clients now route ciphertexts based on their local session state. If we assume that clients only process pre-key messages for sessions that are in the pre-key stage, and normal messages for sessions that have past the pre-key state, this change is simply a re-expression of the existing pseudocode.

Refreshing ephemeral keys. Finally, WhatsApp allows clients to upload new sets of ephemeral keys dynamically as the need arises. This functionality is not captured in our modelling of pairwise channels. Instead, we generate all such keys at the start of the security experiment. Thus, we set n_e in WA-PAIR to $n_d \cdot n_i$ from the PAIR-SEC experiment. This has the disadvantage of not capturing the ability for new ephemeral key pairs to be generated after a compromise has been recovered from (i.e. $\mathcal{O}$-CorruptShared reveals all future ephemeral key pairs to the adversary).

Reusing the Multi-Stage Key Exchange formalism. We re-use the existing analysis of Signal two-party key exchange of [CCD+20] in this analysis. In particular, we replace WhatsApp's implementation of a single two-party Signal session with an instance of Signal within the MSKE variant developed in [CCD+20]. Doing so necessitates a number of minor changes to the MSKE formalism, however. We proceed to briefly define our modified syntax, before referring the reader to [CCD+20, Section 4.2] for a security definition.

Definition 6.7 (Multi-stage Key Exchange Protocol)**.** A multi-stage key exchange protocol MSKE is a tuple of algorithms, along with a keyspace $\mathcal{K}$ and a security parameter λ_r indicating the number of bits of randomness each session requires. The algorithms are:

1) The $\mathsf{MSKE.KeyGen}() \mapsto (pk, sk)$ algorithm generates and outputs the long-term identity key pair (pk, sk) for a device.

2) The $\mathsf{MSKE.MedTermKeyGen}(sk) \mapsto (spk, ssk)$ algorithm takes as input the private long-term identity key sk, then generates and outputs a medium-term key pair (spk, ssk).

3) The $\mathsf{MSKE.EphemKeyGen}(sk) \mapsto (epk, esk)$ algorithm takes as input the private long-term identity key sk, then generates and outputs an ephemeral, single-use key pair (epk, esk).

4) The $\mathsf{MSKE.Activate}(sk, ssk, \rho, peerid) \mapsto (\pi', m')$ algorithm takes as input a long-term secret key sk, a medium-term secret key ssk, a role $\rho \in \{\texttt{init}, \texttt{resp}\}$, and optionally an identifier of its intended peer $peerid$ and outputs a state π' and (possibly empty) outgoing message m'.

5) The $\mathsf{MSKE.Run}(sk, ssk, \pi, m) \mapsto (\pi', m')$ that takes as input a long-term secret key sk, a medium-term secret key ssk, a state π and an incoming protocol or control message m and outputs an updated state π' and (possibly empty) outgoing protocol message m'.

The security analysis in [CCD+20] models the session responder's ephemeral keys as being generated on-the-fly at the start of each session. They implement this by requiring that the *responding device* executes Signal.Activate first, generating and outputting the one-time public key that the initiating device should use. This neatly communicates the purpose of the pre-keys, which essentially send the first message of the key exchange ahead of time. However, it differs from practice, where WhatsApp (and Signal) pre-generate a batch of one-time keys at registration time (adding new batches when all of the previous batch have been exhausted). The server is then expected to distribute these one-time keys to initiating devices appropriately. Our description of PAIR channels follows the implementation (using pre-generated batches of ephemeral key pairs).

There is little difference in the expressivity of these approaches, since the original MS-IND model allows exposing the random state used to generate these keys. Nonetheless, we have to modify the MSKE formalism, security definition and the representation of Signal two-party channels compared to that in [CCD+20]. We split the generation of one-time keys from the activation of a session by introducing the MSKE.EphemKeyGen algorithm that generates and returns a single ephemeral key pair. To activate a session, MSKE.Activate is provided with the ephemeral key to use (the session initiator is given the public key while the responder is provided with the private part). Since the pre-key output by Signal.Activate (with a responding role of `resp`) is not affected by the other inputs to the function, and Rev* queries may only be executed for sessions that have already been activated, we can reason that the results from [CCD+20] apply to this variant directly and without modification.

Next, the analysis in [CCD+20] applies to the $\mathsf{MS\text{-}IND}^{\Pi}_{\Lambda}(\mathcal{A})$ experiment that allows a single Test query. However, the PAIR-SEC security experiment allows the adversary to make multiple challenge queries. We require this multi-challenge security experiment since, when using the security of pairwise channels to reason about the security of the distribution of Sender Keys sessions, multiple pairwise channels are used to distribute a single inbound Sender Key sessions to the many recipients. Thus, we will assume that Signal's two-party channels are secure under a *multi-test* variant of the $\mathsf{MS\text{-}IND}^{\Pi}_{\Lambda}(\mathcal{A})$ security experiment. We posit that Theorem 1 in [CCD+20] can be extended to apply to multiple queries using a hybrid argument, introducing an additional factor in the number of challenge messages that are allowed (similar to the analysis in [DFGS21, Appendix A]). We do so without proof, however.

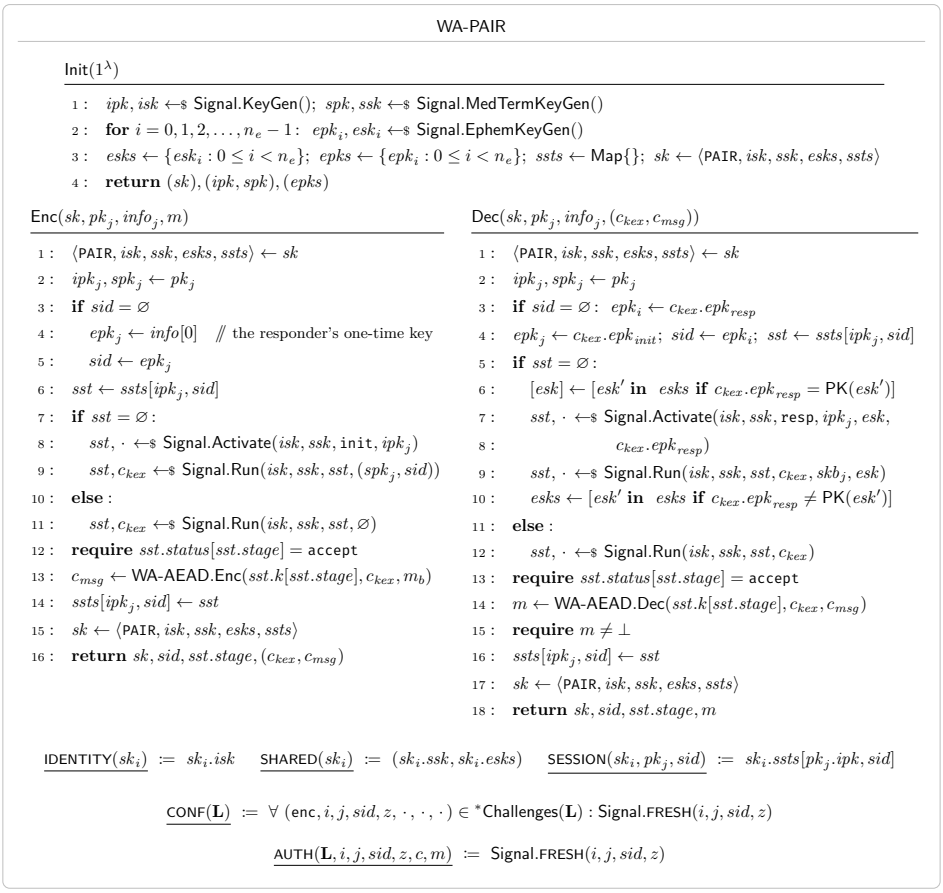


Figure 6.5. Pairwise channels in WhatsApp expressed within the PAIR formalism. See Appendix B.1 for a description of Signal.FRESH from [CCD+20] translated into the PAIR-SEC security experiment.

We define the scheme as follows. In doing so, we detail how we have chosen to map the varying forms of state corruption in the security experiment to the device state, and the security predicates we prove it secure with respect to.

Definition 6.8. WA-PAIR is a pairwise channel with session management that instantiates the PAIR formalism with algorithms (Init, Enc, Dec), state extractor algorithms (IDENTITY, SHARED, SESSION) and security predicates (CONF, AUTH) in Figure 6.5.

We state and prove that the WA-PAIR scheme instantiates a secure pairwise channel under the security predicates in Figure 6.5.

Theorem 6.9 (Security of WA-PAIR)**.** The WA-PAIR protocol specified in Definition 6.8 instantiates a secure pairwise channel, under security predicates WA-PAIR.AUTH and WA-PAIR.CONF, with the advantage of any adversary $\mathcal{A}$ in

winning the $\mathsf{PAIR\text{-}SEC}^{\mathsf{WA\text{-}PAIR}}_{\lambda,n_d,n_i,n_m}(\mathcal{A})$ security experiment bound by:

$$\begin{aligned}&\mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda, n_d, n_i, n_m)\\ &\leq \frac{1}{2} \cdot n_d \cdot (n_d - 1) \cdot n_i \cdot n_m \cdot \Big[\\ &\qquad \mathsf{Adv}^{\mathsf{MS\text{-}IND}}_{\mathsf{Signal}}(\lambda_r, n_d, 1, n_i, n_m) \;+\; \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{WA\text{-}AEAD}}(\lambda, 1) \;+\; \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{WA\text{-}AEAD}}(\lambda, 1)\\ &\quad \Big]\end{aligned}$$

under the following assumptions:

1) Signal is a multi-test $\mathsf{MS\text{-}IND}$ secure MSKE protocol for which the advantage of any PPT adversary is bound by

$$\mathsf{Adv}^{\mathsf{MS\text{-}IND}}_{\mathsf{Signal}}(\lambda_r, n_P, n_M, n_S, n_s)$$

with respect to the $\mathsf{Signal.FRESH}$ security predicate of [CCD+20] (see Appendix B.1), and for which its two KDFs (specified by Figure 7 of [CCD+20]) may be modelled as random oracles, and

2) $\mathsf{WA\text{-}AEAD}$ is an IND$-CPA and EUF-CMA secure one-time AEAD scheme for which the advantage of any PPT adversary is bound by

$$\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{WA\text{-}AEAD}}(\lambda, n_q) \quad \text{and} \quad \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{WA\text{-}AEAD}}(\lambda, n_q)$$

, when its underlying use of HMAC may be modelled as a random oracle.

The instantiation we study hard codes the above security parameters to $\lambda = 256$ with the exception of λ_r. This parameter, λ_r, represents the number of bits of randomness used by each Signal session, which we may bound as $3 \cdot n_d \cdot \lambda \leq \lambda_r \leq \mathsf{poly}(\lambda, n_d, n_i, n_m)$, where $\mathsf{poly}(\lambda)$ is a polynomial function of the given variables whose exact value depends on the number of epochs initiated within each two-party Signal session.

The proof consists of a series of games: $\mathsf{G0}$, $\mathsf{G1}$, $\mathsf{A0}$, $\mathsf{A1}$, $\mathsf{C0}$ and $\mathsf{C1}$. Starting with $\mathsf{G0}$, we inline the $\mathsf{WA\text{-}PAIR}$ protocol into the original $\mathsf{PAIR\text{-}SEC}^{\Pi}_{\Lambda}(\mathcal{A})$ experiment. The first hop, from $\mathsf{G0}$ to $\mathsf{G1}$, relies on the security of the underlying two-party Signal protocol as an $\mathsf{MS\text{-}IND}$ secure MSKE, allowing us to swap the keys that it outputs with random samples from the key space. We then split our analysis into two cases: an authentication break or a confidentiality break. In the case of an authentication break, handled in games $\mathsf{A0}$ to $\mathsf{A1}$, we rely on the one-time $\mathsf{EUF\text{-}CMA}$ security of the $\mathsf{WA\text{-}AEAD}$ scheme to bound the possibility of our adversary producing a forgery of a ciphertext. In the case of a confidentiality break, games $\mathsf{C0}$, $\mathsf{C1.0}$, $\mathsf{C1.1}$, ..., $\mathsf{C1.N} - 1$ rely on the one-time $\mathsf{IND\$\text{-}CPA}$ security of the $\mathsf{WA\text{-}AEAD}$ scheme to bound the possibility of the adversary correctly guessing the challenge bit.

Proof. The proof proceeds through a sequence of games, bounding the advantage of an adversary $\mathcal{A}$ in winning the $\mathsf{PAIR\text{-}SEC}^{\mathsf{WA\text{-}PAIR}}_{\Lambda}(\mathcal{A})$ experiment.

Game 0. We inline WhatsApp's $\mathsf{WA\text{-}PAIR}$ protocol (as described in Figure 6.5) into the $\mathsf{PAIR\text{-}SEC}^{\Pi}_{\Lambda}(\mathcal{A})$ security experiment. Thus:

$$\mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR},\mathcal{A}}(\lambda, n_d, n_i, n_m) = \mathsf{Adv}_{\mathsf{G0}}$$

We now aim to replace each message key with a random sample from its key space, and to bound the ability of any attacker to detect this change by their ability to break the underlying two-party Signal protocol. We do this by building an adversary against the MS-IND security experiment using our PAIR-SEC adversary. Our MS-IND adversary aims to emulate the **Game G0** security experiment to the **Game G0** adversary, all the while embedding the appropriate MS-IND challenges in place of real keys. For this security reduction to work, we must ensure that every case where our **Game G0** adversary wins the emulated **Game G0** experiment translates to a legitimate win of the MS-IND experiment. In particular, we need to ensure that the **Game G0** adversary is not provided with additional powers over the MS-IND adversary that we construct.

Ensuring the authenticity of key exchange messages. The MS-IND security definition does not model the encryption scheme nor does it rely on the scheme to provide authenticity for the key exchange messages. The $\mathsf{MS\text{-}IND}^{\Pi}_{\Lambda}(\mathcal{A})$ security experiment enforces that all key exchange messages are honestly distributed. In contrast, the PAIR-SEC security experiment does not. Since WhatsApp's pairwise channels, inheriting the design of Signal, tie the authenticity and integrity of key exchange messages to that of the application messages, any adversary is able to win the game if they are able to forge or modify the key exchange portion of a ciphertext. It therefore follows from our immediate win convention that all key exchange messages accepted during the experiment are honest, with the possible exception of the final message if the adversary has won through an authentication break. Leaving aside the possibility of state compromise, we can now determine that, at any point in the experiment, all message keys are the result of honestly generated and distributed key exchange messages, satisfying the MS-IND experiment's requirement that all key exchange messages are honest.

Satisfying Signal's security predicates. This leaves state compromises. We now have a class of permitted authentication breaks that do not end the experiment, but do mean that some key exchange messages (and the keys resulting from them) may not be honest. Theorem 1 of [CCD+20] specifies the set of such permitted authentication breaks for which the security guarantees still apply under the Signal.FRESH predicate. To ensure that our adversary has not had any advantages over an MS-IND adversary, we ensure that all keys we sample from the MS-IND adversary satisfy this freshness predicate. We capture this requirement in the WA-PAIR.AUTH and WA-PAIR.CONF security predicates in Figure 6.5, and do so with direct reference to the Signal freshness predicate. Since the session management is now performed by the protocol rather than the experiment, and our methods of state compromise differ slightly, we provide a syntactic translation of Signal's security predicate in Appendix B.1.

Matching sessions. Since WA-PAIR protocols are expected to manage sessions themselves, our security reduction needs to ensure that its session matching (and that of the WA-PAIR channel) is consistent with the session matching of the MS-IND security experiment for all sessions that are marked as clean within our security predicates. As seen in Figure 6.5, WhatsApp clients perform

this matching using a combination of the identity key xpk of their session partner and the ephemeral pre-key of the session responder epk_{resp}. In contrast, the MS-IND security experiment (that we are building an adversary against) matches sessions by the full sequence of keys that have been exchanged over the session, whereby two sessions match if one is a prefix of or equal to the other. As above, if no state compromise has occurred in this experiment, then either the adversary has won the game through a forgery or all of the previous key exchange messages were honest. In these cases, the session matching is equivalent. Consider the case where $\pi^s_{A,i}$ and $\pi^r_{B,j}$ are communicating, and the adversary compromises the session state of $\pi^r_{B,j}$ at index (p, q). If the adversary does not actively participate in the key exchange, opting to passively observe traffic or rewrite the application messages instead, we expect the asymmetric ratcheting mechanism to restore the security of message keys at the next epoch, $p + 1$. In this case, the sessions would continue to match in both the MS-IND experiment and the WA-PAIR protocol, and our security predicates would track the healing.

The session matching is not equivalent in all cases, however. If, instead, the adversary participates in the key exchange protocol by playing the role of $\pi^s_{A,i}$ to $\pi^r_{B,j}$ and/or $\pi^r_{B,j}$ to $\pi^s_{A,i}$, the viewpoints of the keys exchanged in the session will diverge between $\pi^r_{B,j}$ and $\pi^s_{A,i}$. This results in non-matching sessions in the MS-IND experiment and the security predicates provided by the analysis in [CCD+20]. In other words, as soon as the adversary actively participates in the key exchange of a session, we expect no security guarantees for that session for the remaining life of that session. Rather than relying on session matching, we capture this case in the security predicates. The '$\mathsf{Signal.CLEAN}($ $\mathtt{peerE}$, $\cdot$ $)$' case in Appendix B.1 determines whether a peer's contribution to a key exchange was clean. We do this by ensuring that the key exchange part of the ciphertext was generated honestly and not modified in transit.

Game 1. We replace the message keys output by the Signal two-party protocol with random samples from the keyspace. We justify this through the security reduction in Figure B.1, which builds the multi-test variant of the MS-IND experiment by emulating the PAIR-SEC experiment to our adversary $\mathcal{A}$. When the MS-IND experiment has its real-or-random bit b set to 0 (i.e. it outputs real keys), $\mathcal{A}$ is playing an instance of the **Game G0** experiment. While, when the MS-IND experiment has its real-or-random bit b set to 1 (i.e. it outputs random keys), $\mathcal{A}$ is playing an instance of the **Game G1** experiment. Thus, the ability of $\mathcal{A}$ to distinguish between the **Game G0** and **Game G1** experiments is bound by its ability to win the MS-IND experiment:

$$\mathsf{Adv}_{\mathsf{G0}} \;\leq\; \mathsf{Adv}_{\mathsf{G1}} \;+\; \frac{1}{2} \cdot n_d \cdot (n_d - 1) \cdot n_i \cdot n_m \cdot \mathsf{Adv}^{\mathsf{MS\text{-}IND}}_{\mathsf{Signal}}(\lambda_r, n_d, 1, n_i, n_m)$$

We gain the factor of '$\frac{1}{2} \cdot n_d \cdot (n_d - 1) \cdot n_i \cdot n_m$' by the assumption that our multi-test variant of the MS-IND experiment can be bound with respect to the single-test version by applying a hybrid argument over the maximum number of challenges.

We can now consider the security of the messages themselves. Let **Game A0** be a variant of the game where the adversary may not win by correctly guessing the

challenge bit (i.e. it is only possible to win the game through an authentication break). Let **Game C0** be a variant of the game where the adversary may not win through either a forgery or replay attack (i.e. it is only possible to win the game through a confidentiality break). In particular, we arrive at **Game A0** by removing the '$b = b' \wedge \mathsf{PAIR.CONF}(\mathbf{L})$' check from line 3 of the challenger algorithm, and **Game C0** by removing all instances early termination (outside of the challenger algorithm).

Applying the union bound we find that:

$$\mathsf{Adv}_{\mathsf{G1}} \;\leq\; \mathsf{Adv}_{\mathsf{A0}} + \mathsf{Adv}_{\mathsf{C0}}$$

Case 1: Authentication Break

We now consider the advantage of our adversary in winning **Game A0** variant of the security game.

Game A1. At the beginning of the experiment, the challenger guesses which message will trigger an authentication break. If the adversary wins the game through an authentication break for a different message, the challenger aborts the game.

$$\mathsf{Adv}_{\mathsf{A0}} \;\leq\; \frac{1}{2} \cdot n_d \cdot (n_d - 1) \cdot n_i \cdot n_m \cdot \mathsf{Adv}_{\mathsf{A1}}$$

Consider the following security reduction against a challenger for the EUF-CMA security of the WA-AEAD AEAD scheme, $\mathcal{C}_{\mathsf{AEAD}}$. We replace the encryption and decryption of our guessed message with the appropriate queries to the $\mathcal{C}_{\mathsf{AEAD}}$ challenger. If the **Game A1** adversary is able to produce a forgery of this message, it is also a valid forgery in the one-time EUF-CMA experiment. Thus:

$$\mathsf{Adv}_{\mathsf{A1}} \;\leq\; \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{WA\text{-}AEAD}}(\lambda, 1)$$

See Figure B.2 for explicit pseudocode detailing this reduction. This completes our analysis of *Case 1.*

Case 2: Confidentiality Break

We now consider the advantage of our adversary in winning **Game C0** variant of the security game.

In the case where the adversary has returned a guess of the challenge bit, but has not satisfied the confidentiality predicate, the challenger flips a fair coin to determine whether the adversary has won the game, or not. This provides the adversary with a win probability of $\frac{1}{2}$, and an advantage of zero (in this case).

We proceed to consider the case where the adversary has satisfied the confidentiality predicate, and do so through a sequence of $N = \frac{1}{2} \cdot n_d \cdot (n_d - 1) \cdot n_i \cdot n_m$ games, one for each possible encryption challenge in the experiment, and label them **Game C1.0** to **Game C1.*N*-1**. Consider the h-th game, **Game C1.*h***:

Game C1.*h*. We replace the h-th encryption challenge with a random ciphertext.

Consider the following security reduction against a challenger for the IND$-CPA security of the WA-AEAD AEAD scheme, $\mathcal{C}_{\mathsf{AEAD}}$. We replace the encryption and decryption of the h-th encryption challenge with the appropriate queries to the $\mathcal{C}_{\mathsf{AEAD}}$ challenger. When the $\mathcal{C}_{\mathsf{AEAD}}$ challenger's hidden bit is 0, we have that our reduction is playing **Game C1.*h***. When the $\mathcal{C}_{\mathsf{AEAD}}$ challenger's hidden bit is 1, we have that our reduction is playing **Game C1.*h*+1**.

It follows that an adversary that is capable of distinguishing two consecutive games, **Game C1.*h*** and **Game C1.*h*+1**, may be repurposed to win this instance of the IND$-CPA security experiment, giving:

$$\mathsf{Adv}_{\mathsf{C1.h+1}} \;\leq\; \mathsf{Adv}_{\mathsf{C1.h}} + \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{WA\text{-}AEAD}}(\lambda, 1)$$

See Figure B.3 for explicit pseudocode detailing this reduction.

The final game, **Game C1.N-1**, does not use the challenge bit b and, therefore, cannot leak any information about it to the adversary. Thus, the game cannot be won with any advantage, giving $\mathsf{Adv}_{\mathsf{C1.N-1}} = 0$. Summarising *Case 2*, we have that:

$$\mathsf{Adv}_{\mathsf{C0}} \;\leq\; \frac{1}{2} \cdot n_d \cdot (n_d - 1) \cdot n_i \cdot n_m \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{WA\text{-}AEAD}}(\lambda, 1)$$

This completes our analysis of *Case 2*.

We now assemble a final bound using the results above, finding:

$$\begin{aligned}
&\mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda, n_d, n_i, n_m) \\
&\quad\leq\; \frac{1}{2} \cdot n_d \cdot (n_d - 1) \cdot n_i \cdot n_m \cdot \big[\\
&\qquad \mathsf{Adv}^{\mathsf{MS\text{-}IND}}_{\mathsf{Signal}}(\lambda_r, n_d, 1, n_i, n_m) \;+\; \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{WA\text{-}AEAD}}(\lambda, 1) \;+\; \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{WA\text{-}AEAD}}(\lambda, 1) \\
&\quad\big]
\end{aligned}$$

Observe that the above bound is a polynomial function of the experiment parameters (n_d, n_i, n_m), and the respective advantage against the security of each primitive used. It follows that the advantage of any PPT adversary against the security of WA-PAIR is at most a negligible function of the security parameter.

This concludes our proof. □

Our security analysis has found that the pairwise channels in WhatsApp do provide confidentiality, integrity and authenticity for the messages sent over them, under varying state compromise scenarios (loss of long-term identity keys, medium-term secrets, and secret session state). In particular, we have shown that the session management layer does not, *directly*, undermine the security guarantees of the individual two-party sessions. We now consider the security guarantees of the session management layer at the conversation level, i.e. as if it were, itself, an individual two-party session. This composition

continues to provide forward secrecy: compromise of the long-term secrets, medium-term secrets, and secret session state does not compromise the security of old messages. The story for PCS is more complicated, however.

Recall that a protocol that provides PCS is able to recover its security guarantees after state compromise, providing that the adversary is passive while certain actions have been taken place. Following [CFKN20, CJN23], our model captures pairwise messaging schemes that allow multiple parallel sessions between parties. This provides the adversary with three new lines of attack. First, they may attempt to initiate new sessions after the two parties already have an active session between them. Second, they are able to manipulate how target parties decide which session to use by manipulating the delivery of ciphertexts.[10] Third, they are able to re-activate the use of old, compromised sessions at a time that suits them. We now provide two concrete examples of how an adversary can exploit these capabilities.

Example 1: Compromise of long-term keys. Consider an attack where the adversary compromises the long-term and medium-term keys of party uid_A, *after* uid_A and uid_B have initialised a session. In a single session setting, the adversary should not be able to compromise messages sent between these two parties, despite having access to one of their long-term keys. However, in the multiple session setting, the adversary simply initiates two new sessions, one between uid_A and itself (masquerading as uid_B using an unknown key share attack [FMB+16]) and another between uid_B and itself (masquerading as uid_A). This attack can be executed at any time in the future, even after the original session has recovered its security guarantees.

Example 2: Compromise of session state. Consider an attack where the adversary compromises the current session state of party uid_A (between uid_A and uid_B). They may then perform an active attack, whereby they forward application messages untouched, but modify the attached key exchange messages to use key material they have generated. In doing so, they have split the session into two: one between uid_A and itself (masquerading as uid_B) and another between uid_B and itself (masquerading as uid_A). If the adversary no longer has the resources to maintain an active mallory-in-the-middle attack, they may refuse to deliver messages sent to that session. This pauses the progression of the protocol, allowing the adversary to become passive. This result in the two clients setting up a new uncompromised two-party channel. This new session will be secure (and will continue to be unless another state compromise occurs). However, since the adversary can manipulate the network, they may coerce the two clients into reactivating the compromised channel at any point in the future, when they have the resources or inclination to do so.

Both of these attacks are possible against WhatsApp's pairwise channels, and are captured in the security predicates. Despite this, we have shown that the pairwise messaging scheme used by WhatsApp can provide confidentiality and authenticity of its messages in limited contexts. The notion of PCS that it provides is substantially weaker than the notion of PCS that a single-session variant of the two-party Signal pairwise messaging protocol would achieve.

[10] This is captured in the PAIR formalism by allowing the adversary to pick which session to use.

Effects on the security of group messaging. We now turn our attention to group messaging, and consider to what extent these issues affect the security of group messaging in WhatsApp. While compromise of a party's long-term identity, medium-term shared secrets, or the current secrets of a particular session all enable slightly different forms of compromise, taking a high-level view of the security of messages at the conversation level shows that there is little practical difference between these compromise cases. Additionally, since our analysis found no evidence of WhatsApp using a hardware security module to protect a device's long-term keys, it seems that compromise of a device's session state will generally entail the compromise of its long-term secrets as well.

Thus, we make the modelling trade-off not to capture fine-grained compromise for pairwise channels in the wider security analysis of multi-device group messaging (see Section 6.2.8). Instead, we map the corruption of a device (as in DOGM's $\mathcal{O}$-CorruptDevice query) to the leaking of all of a device's current secrets: their identity keys (through $\mathcal{O}$-CorruptIdentity), their medium-term shared secrets (through $\mathcal{O}$-CorruptShared) and the current values of their session secrets (through $\mathcal{O}$-CorruptSession). This greatly simplifies our analysis, at the cost of deriving overly-restrictive security predicates in Section 6.2.8.

6.2.6 Group Messaging

We now analyse the security of the ratcheted symmetric signcryption scheme that forms the basis of the unidirectional messaging channels in WhatsApp. We highlight that Theorem 6.12 relies on XEd25519, as it is used in WhatsApp and defined in [Per16], being a SUF-CMA secure signature scheme with advantage $\mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda)$.

Definition 6.10. WA-RSS is a ratcheted symmetric signcryption scheme that instantiates the RSS formalism with the algorithms (Gen, Signcrypt, Unsigncrypt) detailed in Figure 6.6.

We find that the WA-RSS scheme provides indistinguishability under chosen-plaintext attack with forward secrecy.

Theorem 6.11 (Confidentiality of WA-RSS)**.** The WA-RSS protocol instantiates a ratcheted symmetric signcryption scheme that provides FS-IND-CPA security. The advantage of any adversary $\mathcal{A}$ in winning the $\mathsf{FS\text{-}IND\text{-}CPA}^{\mathsf{WA\text{-}RSS}}_{\lambda,n_q}(\mathcal{A})$ security experiment is bound by

$$\mathsf{Adv}^{\mathsf{FS\text{-}IND\text{-}CPA}}_{\mathsf{WA\text{-}RSS}}(\lambda, n_q) \;\leq\; n_q \cdot \big(\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC}}(\lambda, 2) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF}}(\lambda, 1) + \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{AES\text{-}CBC}}(\lambda, 1) \big)$$

under the assumptions that

1) HMAC can be modelled as a PRF for which the advantage of any PPT adversary $\mathcal{B}$ in winning the $\mathsf{PRF}^{\mathsf{HMAC}}_{\lambda,2}(\mathcal{B})$ experiment is bound by $\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC}}(\lambda, 2)$,

2) HKDF can be modelled as a PRF for which the advantage of any PPT adversary $\mathcal{B}$ in winning the $\mathsf{PRF}^{\mathsf{HKDF}}_{\lambda,1}(\mathcal{B})$ experiment is bound by $\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF}}(\lambda, 1)$, and

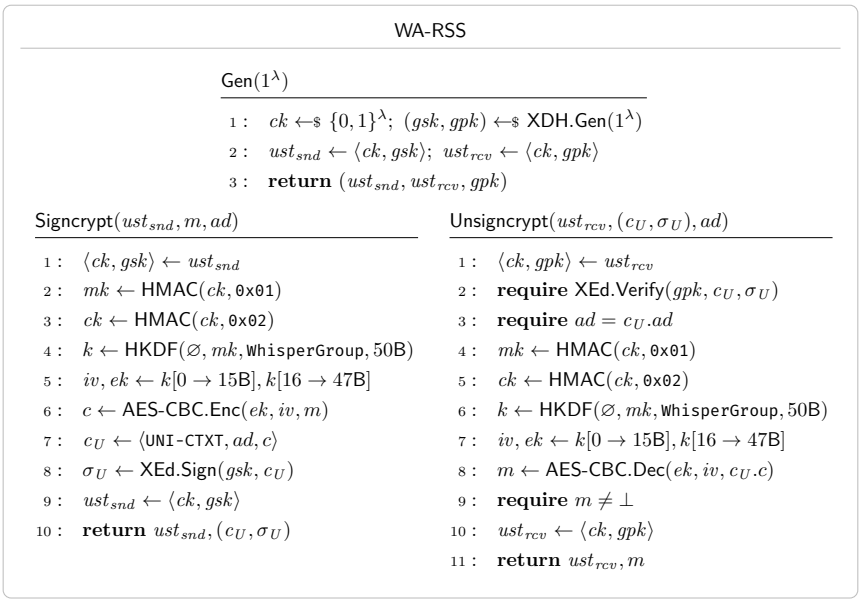


Figure 6.6. A single group messaging epoch in WhatsApp expressed within the RSS formalism. We modify the description of UNI in Figure 4.7 by removing the session identifier and message index, requiring the layer above to maintain and set the appropriate values for ad.

3) AES-CBC is an IND-CPA secure one-time symmetric encryption scheme for which the advantage of any PPT adversary $\mathcal{B}$ against the $\mathsf{IND\text{-}CPA}^{\mathsf{AES\text{-}CBC}}_{\lambda,1}(\mathcal{B})$ experiment is bound by $\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{AES\text{-}CBC}}(\lambda, 1)$.

The instantiation we study hard codes the above security parameter to $\lambda = 256$.

Proof. We bound the probability that the adversary, $\mathcal{A}$, correctly guesses the challenge bit b via the following sequence of games.

Game 0. This is the standard FS-IND-CPA security game for ratcheted symmetric signcryption schemes instantiated with WA-RSS. This gives:

$$\mathsf{Adv}^{\mathsf{FS\text{-}IND\text{-}CPA}}_{\mathsf{WA\text{-}RSS}}(\lambda, n_q) \;=\; \mathsf{Adv}_{\mathsf{G0}}$$

Game 1. In this game, we replace each message key mk and chain key ck with uniformly random values rmk and rck, until the point at which $\mathcal{A}$ issues a $\mathcal{O}$-Corrupt query.

We do so through a series of n_q reductions, $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_{n_q-1}$, each to the PRF security of HMAC. We apply this series of reductions iteratively in the form of a hybrid argument.

In the h-th hybrid, we modify the h-th $\mathcal{O}$-Send query to check if the $\mathcal{O}$-Corrupt query has been issued and, if not, replace the next computation[11] of ck and mk

[11] The h-th $\mathcal{O}$-Send query computes the $(h+1)$-th value of ck and the h-th value of mk.

with the outputs of the h-th PRF challenger, $\mathcal{C}_{\mathsf{PRF}}$. Specifically, when computing mk and ck as part of the h-th $\mathcal{O}$-Send query, $\mathcal{B}_h$ instead queries $\mathcal{C}_{\mathsf{PRF}}$ with `0x01` and `0x02`, using the outputs to replace mk and ck, which we denote rmk and rck (respectively).

Note that the PRF key, k, is sampled uniformly at random by $\mathcal{C}_{\mathsf{PRF}}$. In the first hybrid, this is identical to that of the original experiment where the initial value of ck is sampled as $ck \leftarrow\!\$\ \{0,1\}^\lambda$. If the bit b sampled by $\mathcal{C}_{\mathsf{PRF}}$ is 0, then $rmk = \mathsf{HMAC}(k, \texttt{0x01})$ and $rck = \mathsf{HMAC}(k, \texttt{0x02})$. Otherwise, rmk and rck are uniformly random values sampled by $\mathcal{C}_{\mathsf{PRF}}$. Thus, any adversary that can efficiently distinguish between two consecutive hybrids can be turned into an efficient algorithm against the PRF assumption on HMAC. We may iteratively apply the resulting bounds between each pair of consecutive hybrids to find that:

$$\mathsf{Adv}_{\mathsf{G0}} \;\leq\; n_q \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC}}(\lambda, 2) + \mathsf{Adv}_{\mathsf{G1}}$$

Game 2. In this game we replace each initialisation vector iv and encryption key k_e with uniformly random values riv and rk_e, until the point at which $\mathcal{A}$ issues a $\mathcal{O}$-Corrupt query.

We do so through a series of n_q reductions, $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_{n_q-1}$, each to the PRF security HKDF. We apply this series of reductions iteratively in the form of a hybrid argument.

In the h-th hybrid, we modify the h-th $\mathcal{O}$-Send query to check if the $\mathcal{O}$-Corrupt query has been issued and, if not, replace the next computation of iv and k_e with the outputs of the h-th PRF challenger, $\mathcal{C}_{\mathsf{PRF}}$, with an input length of 12 bytes and an output length of 50 bytes. Thus, when computing iv and k_e as part of the h-th $\mathcal{O}$-Send query, $\mathcal{B}_h$ instead queries $\mathcal{C}_{\mathsf{PRF}}$ with `WhisperGroup` as input, using the outputs to replace iv and k_e, which we denote riv and rk_e (respectively).

We note that if the bit b sampled by $\mathcal{C}_{\mathsf{PRF}}$ is 0, then $riv, rk_e = \mathsf{HKDF}(0, k, \texttt{WhisperGroup}, 50\mathsf{B})$. Here, k has been sampled uniformly at random from $\{0,1\}^\lambda$ by $\mathcal{C}_{\mathsf{PRF}}$ and is, thus, identically distributed to rmk thanks to the changes introduced in **Game 1**. It follows that, when the bit b sampled by $\mathcal{C}_{\mathsf{PRF}}$ is 0, riv and rk_e are identically distributed to those values they replace, $iv, k_e = \mathsf{HKDF}(0, rmk, \texttt{WhisperGroup}, 50\mathsf{B})$. If the bit b sampled by $\mathcal{C}_{\mathsf{PRF}}$ is 1, on the other hand, riv and rk_e are uniformly random values, and the iterative replacement is sound. Thus, any adversary that can efficiently distinguish between two consecutive hybrids can be turned into an efficient algorithm against the PRF assumption on HKDF.

We may iteratively apply the resulting bounds between each pair of consecutive hybrids to find that:

$$\mathsf{Adv}_{\mathsf{G1}} \;\leq\; n_q \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF}}(\lambda, 1) + \mathsf{Adv}_{\mathsf{G2}}$$

Game 3. In this game we demonstrate that any adversary that can distinguish between an encryption of m_0 and m_1 can be turned into an algorithm that breaks the IND-CPA security of AES-CBC.

Specifically, we introduce a reduction $\mathcal{B}$ that initialises a IND-CPA challenger $\mathcal{C}_{\text{IND-CPA}}$ whenever $\mathcal{A}$ issues a $\mathcal{O}$-Send(m_0, m_1) query. Since rk_e is sampled uniformly randomly by **Game 2**, the $\mathcal{C}_{\text{IND-CPA}}$'s sampling of rk_e internally proceeds identically as in **Game 2**. When computing $c \leftarrow \text{AES-CBC.Enc}(k_e, iv, m_b)$ as the result of a $\mathcal{O}$-Send query, $\mathcal{B}$ instead queries the most recently initialised $\mathcal{C}_{\text{IND-CPA}}$ with (m_0, m_1, iv), using the outputs to replace the computation of the ciphertext c. Upon computing rk_e, $\mathcal{B}$ initialises a new IND-CPA challenger. If $\mathcal{A}$ issues $\mathcal{O}$-Corrupt, then $\mathcal{B}$ simply returns the most recently computed rck and no longer initialises IND-CPA challengers.

We note that if the bit b sampled by $\mathcal{C}_{\text{IND-CPA}}$ is 0, then $c = \text{AES-CBC.Enc}(k_e, iv, m_0)$, otherwise, $c = \text{AES-CBC.Enc}(k_e, iv, m_1)$. Thus, any successful adversary that could detect any of these replacements can be turned into an efficient algorithm against the IND-CPA assumption. Bounding the advantage of $\mathcal{B}_3$ by the advantage of any PPT adversary limited to at most n_q encryption queries, then we have:

$$\text{Adv}_{\text{G2}} \;\leq\; n_q \cdot \text{Adv}^{\text{IND-CPA}}_{\text{AES-CBC}}(\lambda, 1)$$

This completes our proof. □

We find that the WA-RSS scheme provides authentication in the form of existential-unforgeability under chosen-message attack.

Theorem 6.12 (Unforgeability of WA-RSS)**.** The WA-RSS protocol instantiates a ratcheted symmetric signcryption scheme that provides SUF-CMA security for which the advantage of any adversary $\mathcal{A}$ in winning the $\text{SUF-CMA}^{\text{WA-RSS}}_{\lambda, n_q}(\mathcal{A})$ security experiment is bound by

$$\text{Adv}^{\text{SUF-CMA}}_{\text{WA-RSS}}(\lambda, n_q) \;\leq\; \text{Adv}^{\text{SUF-CMA}}_{\text{XEd}}(\lambda, n_q)$$

under the assumption that XEd25519 is itself a SUF-CMA secure digital signature scheme for which the advantage of any PPT adversary $\mathcal{B}$ in winning the $\text{SUF-CMA}^{\text{XEd}}_{\lambda, n_q}(\mathcal{B})$ experiment is bound by $\text{Adv}^{\text{SUF-CMA}}_{\text{XEd}}(\lambda, n_q)$. The instantiation we study hard codes the above security parameter to $\lambda = 256$.

Proof. We bound the advantage of $\mathcal{A}$ in triggering the authentication win condition via the following sequence of games.

Game 0. This is the standard SUF-CMA security game for ratcheted symmetric signcryption schemes instantiated with WA-RSS. This gives:

$$\text{Adv}^{\text{SUF-CMA}}_{\text{WA-RSS}}(\lambda, n_q) \;=\; \text{Adv}_{\text{G0}}$$

Game 1. In this game we abort if the adversary outputs a message that decrypts successfully, but was not the output of a $\mathcal{O}$-Send query.

Specifically, we introduce a reduction $\mathcal{B}$. At the beginning of the experiment, $\mathcal{B}$ initialises a SUF-CMA challenger $\mathcal{C}_{\text{SUF-CMA}}$, and replaces the generation of

the signing keys with $\mathcal{C}_{\mathsf{SUF\text{-}CMA}}$'s output public key pair. Whenever $\mathcal{A}$ queries $\mathcal{O}\text{-}\mathsf{Send}(m)$, instead of computing the signature σ itself, $\mathcal{B}$ instead queries $\mathcal{C}_{\mathsf{SUF\text{-}CMA}}$ with the ciphertext c. Finally, whenever $\mathcal{A}$ queries $\mathcal{O}\text{-}\mathsf{Receive}(c)$, $\mathcal{B}$ will use the output public key pk to verify the signature σ over c.

If the adversary makes outputs a message that decrypts successfully, but was not the output of a $\mathcal{O}\text{-}\mathsf{Send}$ query, then this means that the adversary has created a signature σ' that was not output by $\mathcal{C}_{\mathsf{SUF\text{-}CMA}}$, but verifies correctly, thus creating a forgery. Thus, if the adversary triggers our abort query, then it can be turned into a successful adversary against the SUF-CMA security of XEd. Bounding the advantage of $\mathcal{B}$ by the advantage of any PPT adversary, we find:

$$\mathsf{Adv}_{\mathsf{G0}} \leq \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, n_q) + \mathsf{Adv}_{\mathsf{G1}}$$

We note that since **Game 1** aborts whenever the adversary causes $win \leftarrow \mathbf{true}$, that it is not possible for the adversary to win in **Game 1** and thus we find:

$$\mathsf{Adv}_{\mathsf{G0}} \leq \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, n_q)$$

This completes our proof. □

6.2.7 WhatsApp in the DOGM Model

We start by mapping a subset of WhatsApp's functionality into the DOGM with revocation model, capturing user and device cryptographic identity management (incl. *device revocation*), group messaging and history sharing. We do so as follows.

Definition 6.13. WA-DOGM is a device-oriented group messaging protocol with revocation that instantiates the DOGMAR formalism with algorithms (Gen, Reg, Rev, Init, Add, Rem, Enc, Dec, StateShare) in Figures 6.7 to 6.9.

User and device management. We capture WhatsApp's multi-device functionality, which provides cryptographic identity management for users and their devices, using Gen, Reg and Rev:

- Gen captures the creation of a new primary device and its long-term keys (as described by PO.Setup), since a user in WhatsApp is cryptographically equivalent to their *primary device*. Note that, since Gen generates these identities, we are not able to determine the user identifier before calling the algorithm and require it to be null.

- Reg captures the device-specific setup of the primary device, as well as the initialisation and linking of new *companion devices*.

  Similar to Gen, no device identifier is given to this algorithm when registering a new companion device because the device identifier is not known at call time (since the public identity key used as the device identifier has not been generated yet). An exception to this is when registering the

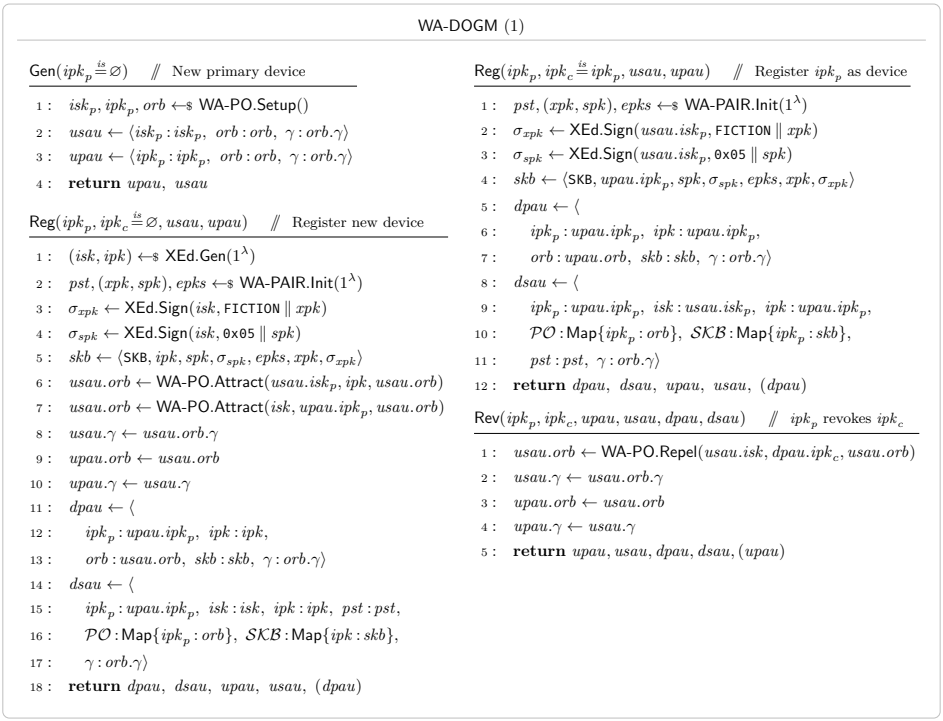


Figure 6.7. Device management in WhatsApp expressed within the DOGM with revocation formalism.

primary device, where the device identity is known to the caller. In this case, we expect it to match the user identifier.

We return the updated multi-device state to the adversary as a ciphertext. We also store and initialise storage for multi-device states, called $\mathcal{PO}$, and for Signal key bundles $\mathcal{SKB}$, which are kept in the device's secret authenticator, *dsau*. Note that, modulo cryptographic controls, the adversary can fill these at will by sending the appropriate ciphertexts of type PO or SKB, cf. WA-DOGM.Dec.

- Rev captures the revocation of a companion device. The challenger executes PO.Repel on behalf of the primary device, outputting an updated multi-device state. We return the updated state to the adversary as a ciphertext.

Recall that the challenger distributes the initial version of each user's public authenticator to each device in a trusted manner, while the public authenticators of devices are distributed through the adversary. Thus, how we choose to split information between the user and device authenticators will affect to what extent we capture device management in our security analysis.

Importantly, we would like to ensure that our analysis captures how clients manage and verify the structures that define one another's device composition. Each user public authenticator contains the identity key of their primary device, accompanied by their public orbit state and generation. Since the orbit state of

a user changes during the experiment, as new devices are registered and existing devices are revoked, it might be necessary for the adversary to propagate these changes to the devices of communicating partners. The decryption algorithm, described below, provides the adversary with a means to do just this, provided they can pass the requisite cryptographic checks. As they do so, the adversary may need to perform the work of the WhatsApp server in merging the various updates and provided the correct accompanying information. Thus, we capture the ability of a malicious server to attempt to distribute fake or erroneous multi-device states (but provide an initial distribution that is honest).

Remark 6.14. *As discussed in Sections 6.2.4 and 6.2.5, WhatsApp utilises device identity keys for both XEd signatures and XDH key exchange. Since a proof of the joint security of such constructions does not exist, we choose to model the dual use of such keys with two separate key pairs:* $(isk, ipk) \leftarrow\$ \mathsf{XEd.Gen}(1^\lambda)$ *and* $(xsk, xpk) \leftarrow\$ \mathsf{XDH.Gen}(1^\lambda)$*. This separation requires us to provide a cryptographic link from the signature key pair,* (isk, ipk)*, to the key exchange key pair,* (xsk, xpk)*. We highlight this modelling choice by using* `FICTION` *as the domain separator.*

Group messaging. Our description of WhatsApp in Section 4.2 follows the whitepaper and implementation in working with *logical groups*. However, the group messaging protocol in WhatsApp provides no guarantee that group members have a shared view of the group membership. The DOGM formalism, and our security analysis, follows this approach by capturing messaging sessions at the level of unidirectional channels. Each of the unidirectional sessions may have a different view of the group membership. Similarly, since it is not guaranteed that all sessions have the same view of a user's device composition, the formalism handles group membership at the device rather than user level.

- Init models the initialisation of a new DOGM session. That is, a series of UNI sessions for a particular device, in a particular group, and in a particular role (either as sender or recipient).

  This differs from our description in Section 4.2, which described the behaviour of a WhatsApp client, managing any number of simultaneous unidirectional sessions to construct a group chat. Along these lines, we remove the use of a logical group identifier *gid* from WA-DOGM altogether. Since any separation of state between logical groups is now handled by the DOGM challenger on behalf of the protocol, this identifier no longer serves a functional purpose.

- Add and Rem model the addition or removal of a device from a session, respectively. We do not expect WA-DOGM sessions to correctly track the list of verified devices for a user when performing group management functions, unlike WA.AddMember and WA.RemoveMember in Section 4.2. Instead we offload this responsibility to the challenger in the DOGM security experiment. Thus, these algorithms lean heavily on the SK and UNI primitives.

- Enc and Dec model sending and receiving messages using an outbound (or inbound) session (respectively). These algorithms act primarily as a wrapper around the SK and UNI primitives.

Figure 6.8. Group messaging in WhatsApp expressed within the DOGM with revocation formalism.

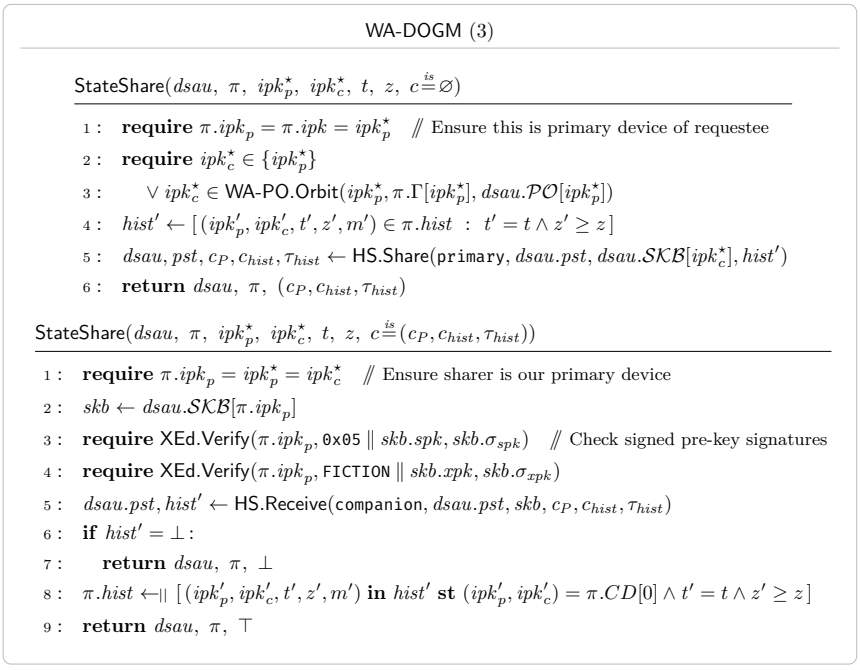


Figure 6.9. State sharing in WhatsApp expressed within the DOGM with revocation formalism.

Note that we require minor changes to the SK algorithms described in Figure 4.8. Specifically, we replace the use of PAIR with WA-PAIR and UNI with WA-RSS. For the latter change, we lift the initialisation and maintenance of the epoch identifier, *usid*, and message index, z, to the SK algorithms and ensure that they are included as additional data that is input to the WA-RSS.Signcrypt and WA-RSS.Unsigncrypt algorithms. We leave the pseudocode for these changes implicit.

State sharing. We capture WhatsApp's history sharing using the state share functionality in the DOGM model.

- StateShare models the sharing of message history between the primary device and new companion devices. Sharing and receiving message history is captured at the level of unidirectional Sender Keys sessions. This differs from WhatsApp's implementation, where the primary device shares all of its message history as a single bundle.

Using the state sharing functionality of the DOGM to capture history sharing in WhatsApp entails a number of modelling decisions. In particular, the DOGM enables the adversary to schedule and orchestrate state sharing sessions arbitrarily (up to their computation limitations). This differs from practice, where history sharing is only triggered in particular circumstances, but is necessary to capture the required security of the ciphertexts. We utilise the stage and message index, (t, z), inputs to the state sharing protocol to select all messages starting from index z in the stage t for sharing. We use these inputs to allow accurate tracking of history sharing in the security experiment. The

recipient in a history sharing session outputs a final ciphertext of the form $\top$ to indicate they have accepted the share, and $\bot$ to indicate that the share failed.

Remark 6.15. *As discussed in Section 6.2.5, we remove the need to check the signed pre-key signatures inside the* PAIR-SEC *experiment and lift this requirement to the layer above. Thus, our description of* WA-DOGM *checks the signed pre-key signature, σ_{spk}, when WhatsApp's* WA-PAIR *protocol would have done so. This can be seen in lines 5 and 6 of the ciphertext decryption case of* Dec *in Figure 6.8 and lines 3 and 4 of the receiving case of* StateShare *in Figure 6.9.*

6.2.8 Security Analysis

We now analyse the security of the WA-DOGM protocol within the DOGM with revocation formalism. In the simplest case, we expect the protocol to provide security as long as no secrets have been compromised or corrupted. And, indeed, we will show that this is the case. We proceed by detailing the situations in which WA-DOGM does not provide confidentiality and authenticity, and use these to motivate the confidentiality and authenticity security predicates (WA-DOGM.CONF and WA-DOGM.AUTH respectively), which restrict patterns of adversarial queries within the DOGM security experiment. As with our analysis of Matrix, these predicates capture a mixture of "trivial wins" in the security experiment, as well as breaks (that may not have been intended) in the design of the protocol. WhatsApp's whitepaper does not provide enough detail in its threat model [Wha23a, Defining End-to-End Encryption] to determine which cases should be categorised as expected or unexpected breaks. Either way, both such cases describe situations in which WhatsApp is not able provide security within the experiment and, with the appropriate interpretation, in practice.

Authentication. We first consider the case of an authentication break, whereby session $\pi^s_{A,i}$ accepts what it believes to be message z of stage t from sending session $\pi^r_{B,j}$.

The payload ciphertext of the Sender Keys session is authenticated via the WA-RSS ratcheted symmetric signcryption scheme RSS (which provides unforgeability as defined in Figure 5.7). Thus, we would expect authentication to hold within a single WA-RSS session unless the adversary has compromised the state of the matching sending session, $\pi^r_{B,j}$, at a point in time where it holds the sending counterpart to the recipients WA-RSS session. Doing so would expose the signing key which the adversary may then use to trivially forge a ciphertext.

Note that, since the pairwise channels cannot guarantee a consistent message order, there is similarly no guarantee that the stage indices used by $\pi^s_{A,i}$ and $\pi^r_{B,j}$ are consistent. Thus, we utilise the identifier for the WA-RSS session to determine partnering[12] and disallow authentication breaks where the partnered sending session was compromised. Putting this all together, when a recipient

[12] Recalling our description in the previous sections, we see that this mirrors the implementation. Note that the presence of collisions in WA-RSS identifiers does not gain the adversary an advantage, since such collisions only have the potential to add false positives to the predicate, thereby increasing its coverage. Additionally, these identifiers are not relied upon for authentication.

session, $\pi^s_{A,i}$, accepts a message as originating from $\pi^r_{B,j}$, we check whether the compromise oracle, $\mathcal{O}$-Compromise, was issued for the sending session at the point in time when it contained the partnered WA-RSS sending session.

As mentioned above, the WA-RSS public key that verifies the payload ciphertext is distributed to $\pi^s_{A,i}$ via the WA-PAIR scheme for pairwise channels. Thus, it relies on the authentication provided by those pairwise channels. The public key bundle used to establish the pairwise channel between devices (A, i) and (B, j) is authenticated by their companion public key ipk_c. Since WhatsApp allows for multiple parallel pairwise sessions, any adversary that corrupts the secret authenticator of device (B, j) can create, authenticate and distribute their own WA-RSS public key and proceed to forge payload ciphertexts. We disallow this attack by ensuring that the claimed sending device, (B, j), has not previously been corrupted. As we saw in our security analysis of the pairwise channels in Section 6.2.5, there exist (weaker) attacks when the pairwise session state of the *recipient* device has been compromised. Namely, thanks to the symmetric authentication within established pairwise channels, the compromise of one of the recipient device's session states enables the adversary to impersonate the sending party, forging a Sender Key distribution ciphertext that will be accepted by the recipient. We make the simplifying choice to capture an overly broad predicate, in this case, and disallow authentication breaks when the recipient device, (A, i), has previously been corrupted.

Each of the device public keys ipk_c are themselves authenticated by the primary device (as we capture with the public key orbit provided by the WA-PO scheme). Note that, in WhatsApp, corrupting a user's primary device provides the adversary with the same information as corrupting the user's long-term secrets. We must, therefore, also check if the adversary has corrupted the primary device through a $\mathcal{O}$-CorruptDevice call, and disallow both attacks. We contrast this with the analysis of Matrix in the previous section, where the modelling choice was made to separate the user's long-term secrets from the device state, despite both being distributed across all of a user's devices. Such a choice was not possible in the case of WhatsApp, since the same key material is required for both user- and device-level operations. WhatsApp achieves stronger security guarantees in this instance, despite the security predicates we will derive suggesting otherwise.

Finally, it is possible for the adversary to break authentication through state sharing. In WhatsApp, such breaks are captured when the adversary successfully forges a state sharing message over a pairwise channel. Since WhatsApp clients will only accept history sharing messages from their primary device, and over a pairwise channel, the only time we expect the adversary to be able to win the game by forging a history sharing message is if they have directly compromised the recipient device's long-term user identity (or, equivalently, their primary device).

We codify these requirements through the authentication predicate, specified in Definition 6.16. It requires that, for authentication to hold for a message being decrypted by the session $\pi^s_{A,i}$ with sender (B, j) and ciphertext c at computed stage t and message index z, none of the following conditions are true (at the point in time the forgery is detected).

1) The recipient user's secret authenticator has been compromised, i.e. either $\mathcal{O}$-CorruptUser(A) or $\mathcal{O}$-CorruptDevice$(A, 0)$ was issued at any point before the authentication break.

2) The sending user's secret authenticator has been compromised, i.e. either $\mathcal{O}$-CorruptUser(B) or $\mathcal{O}$-CorruptDevice$(B, 0)$ was issued before the recipient received the inbound Sender Keys session for that epoch.[13]

3) The sending or recipient device's secret authenticator has been compromised, i.e. either $\mathcal{O}$-CorruptDevice(B, j) or $\mathcal{O}$-CorruptDevice(A, i) was issued before the recipient received the inbound Sender Keys session for that epoch.[14]

4) The session state of the sender has been compromised, i.e. $\mathcal{O}$-Compromise(B, j, r) was previously issued at the point in time when the sending session $\pi^r_{B,j}$ was in the partnered Sender Keys stage for the received ciphertext.

Definition 6.16. An instance of the DOGM with revocation security experiment with experiment log $\mathbf{L}$ fulfils the WA-DOGM.AUTH authentication predicate if, when processing a $\mathcal{O}$-Decrypt(A, i, s, c) query for the session, $\pi^s_{A,i}$, with sender (B, j) and ciphertext c at computed stage t and message index z, the following evaluates to true.

$$
\begin{aligned}
&\underline{\mathsf{WA\text{-}DOGM.AUTH}(\mathbf{L}, A, i, s, t, z, \gamma, B, j)} \;\coloneqq \\
&\quad \nexists\, (\texttt{corr-user}, A, \cdot)\ \textbf{in}\ \mathbf{L} \\
&\qquad \wedge\ \nexists\, (\texttt{corr-device}, A, 0, \cdot)\ \textbf{in}\ \mathbf{L} \quad /\!/\ (1) \\
&\quad \wedge\ \nexists\, (\texttt{corr-user}, B, \cdot)\ \textbf{precedes}\ (\{\texttt{dec}, \texttt{share}\}, A, i, s, t, \cdot)\ \textbf{in}\ \mathbf{L} \\
&\qquad \wedge\ \nexists\, (\texttt{corr-device}, B, 0, \cdot)\ \textbf{precedes}\ (\{\texttt{dec}, \texttt{share}\}, A, i, s, t, \cdot)\ \textbf{in}\ \mathbf{L} \quad /\!/\ (2) \\
&\quad \wedge\ \nexists\, (\texttt{corr-device}, B, j, \cdot)\ \textbf{precedes}\ (\{\texttt{dec}, \texttt{share}\}, A, i, s, t, \cdot)\ \textbf{in}\ \mathbf{L} \\
&\qquad \wedge\ \nexists\, (\texttt{corr-device}, A, i, \cdot)\ \textbf{precedes}\ (\{\texttt{dec}, \texttt{share}\}, A, i, s, t, \cdot)\ \textbf{in}\ \mathbf{L} \quad /\!/\ (3) \\
&\quad \wedge\ \nexists\, r, t'\ \textbf{st}\ \big((\texttt{corr-sess}, B, j, r, \langle t : t', \cdot \rangle)\ \textbf{in}\ \mathbf{L} \ \wedge\ \pi^s_{A,i}.usid[t] = \pi^r_{B,j}.usid[t']\big) \quad /\!/\ (4)
\end{aligned}
$$

Confidentiality. We now consider confidentiality. Recall that the adversary may win the security experiment by correctly guessing the challenge bit, and that this challenge bit is explicitly used by the challenger in only one place during the adversary's execution: when encrypting challenge ciphertexts. We proceed to focus on the confidentiality of such ciphertexts.

If the adversary can forge messages for pairwise channels, they can convince honest parties to send them the requisite inbound Sender Keys session state. It follows that those conditions in the authentication predicate that capture

[13] We detect when a Sender Keys session for a particular epoch has been received by searching for the first log entry representing the decryption of a message in the appropriate stage. This will either be the log entry for the Sender Keys distribution message itself, or the log entry for the accompanying application ciphertext (if the two were distributed together).

[14] If the receiving device is aware that the sending device has been revoked, we would expect it to reject the message. We do not include this in the security predicate because it is enforced by the challenger, which will only trigger an authentication break when the recipient is aware of such a revocation.

impersonation at the level of pairwise channels will also apply to the confidentiality predicate. There are some key differences between the authentication and confidentiality cases, however. First, while only outbound Sender Keys sessions contain the key material necessary to forge messages, all Sender Keys sessions contain the key material necessary to decrypt messages. It follows that we must broaden the predicate to capture the intended recipients of a message. Second, confidentiality breaks can be retroactive, i.e. the adversary can break confidentiality through state compromises that occur after the message has been processed by intended recipients.

This is thanks to the history sharing sub-protocol. In particular, it is possible for a recipient session, having received a challenge encryption, to later share the resulting plaintext with another device.[15] If the recipient of a state share has had their secret device authenticator compromised, then the adversary can impersonate it at the level of pairwise channels, decrypt the state sharing ciphertext and, in turn, determine whether m_0 or m_1 was chosen by the challenger. Thus, we have the confidentiality predicate disallow the corruption of the intended recipient devices *at any point in the experiment.*[16]

Further still, since the device composition of a user can change after the distribution of a challenge ciphertext, it is possible that the state sharing protocol can be used to distribute (re-encrypted) challenges from an intended recipient device to another device of the same user. In the case of WhatsApp, history sharing should only occur from a primary device to one of its companion devices. Nonetheless, the sending session is not necessarily aware of all the possible recipient devices for its message: such a device would not be considered an intended recipient of the session and, thus, is not covered by the aforementioned checks. There are two approaches we could take. We could either (a) allow the corruption of a recipient user's secrets after initial distribution, providing that no such state sharing events have occurred, or (b) we can simply disallow the corruption of a recipient user's secrets at any point in the experiment. We take the second approach.

We codify these requirements through the confidentiality predicate, specified in Definition 6.17. It requires that, for every challenge ciphertext c in the security experiment, corresponding to an $\mathcal{O}\text{-}\mathsf{Encrypt}(A, i, s, m_0, m_1)$ query for the session, $\pi^s_{A,i}$, with intended recipients $CU \coloneqq \pi^s_{A,i}.CU$ and $CD \coloneqq \pi^s_{A,i}.CD$ at stage $t \coloneqq \pi^s_{A,i}.t$ and message index $z \coloneqq \pi^s_{A,i}.z$, none of the following conditions are true.

1) The sending session has been compromised, i.e. a query to $\mathcal{O}\text{-}\mathsf{Compromise}(A, i, s)$ was issued, within the same stage t and before the session has ratcheted forward past the challenge message index z.

2) A recipient session has been compromised, i.e. a query to $\mathcal{O}\text{-}\mathsf{Compromise}(B, j, r)$ was issued for a session representing a device (B, j) in the challenge

[15] This is, in essence, similar to an adversary making two identical calls to different encryption oracles in a security experiment, for which the first oracle triggers encryption using the outbound Sender Keys session and the second triggers encryption with the attachment encryption scheme.

[16] A more accurate approach would be to track when such state sharing events have actually occurred.

recipient list CD at a point in time where the session $\pi^r_{B,j}$ is partnered to the challenge stage and has not ratcheted forward past the challenge message index z.

3) The sending device has been compromised, i.e. a query to $\mathcal{O}$-CorruptDevice(A, i) was issued, before the inbound Sender Keys session for this stage was distributed.[17]

4) Any device representing a recipient user has been compromised, i.e. a query to $\mathcal{O}$-CorruptDevice(B, j) was issued for a user $B \in CU$ and any device $j \in [n_d]$.

5) The sending user has been compromised, i.e. a query to $\mathcal{O}$-CorruptUser(A) or $\mathcal{O}$-CorruptDevice$(A, 0)$ was issued, at any point in the experiment.

6) A recipient user has been compromised, i.e. a query to $\mathcal{O}$-CorruptUser(B) or $\mathcal{O}$-CorruptDevice$(B, 0)$ was issued for a user $B \in CU$, at any point in the experiment.

Definition 6.17. An instance of the DOGM with revocation security experiment with experiment log $\mathbf{L}$ fulfils the WA-DOGM.CONF confidentiality predicate provided the following evaluates to true for each challenge query, $chall \in {}^*\mathsf{Challenges}(\mathbf{L})$.

$$
\begin{aligned}
&\underline{\mathsf{WA\text{-}DOGM.CONF}(\mathbf{L}, chall)} \coloneqq \\
&\quad \textbf{let } chall = (\,\mathtt{enc}, A, i, s, \Gamma, CU, CD, t, z, m_0, m_1, c\,) : \\
&\qquad \nexists\, z' \textbf{ st } (\mathtt{corr\text{-}sess}, A, i, s, \langle t : t, z : z', \cdot \rangle) \textbf{ in } \mathbf{L} \ \wedge\ z' < z \quad /\!\!/ \ (1) \\
&\qquad \wedge\ \nexists\, (B, j, r, t', z') \textbf{ st } \big((\mathtt{corr\text{-}sess}, B, j, r, \langle t : t', z : z', \cdot \rangle) \textbf{ in } \mathbf{L} \\
&\qquad\quad \wedge\ (B, j) \in CD \ \wedge\ \pi^s_{A,i}.usid[t] = \pi^r_{B,j}.usid[t'] \ \wedge\ z' < z \big) \quad /\!\!/ \ (2) \\
&\qquad \wedge\ \nexists\, (\mathtt{corr\text{-}device}, A, i, \cdot) \textbf{ precedes } (\mathtt{enc}, A, i, s, \cdot, \cdot, \cdot, t, \cdot) \textbf{ in } \mathbf{L} \quad /\!\!/ \ (3) \\
&\qquad \wedge\ \nexists\, B \in CU, j \in [n_d] \textbf{ st } (\mathtt{corr\text{-}device}, B, j, \cdot) \textbf{ in } \mathbf{L} \quad /\!\!/ \ (4) \\
&\qquad \wedge\ \nexists\, (\mathtt{corr\text{-}user}, A, \cdot) \textbf{ in } \mathbf{L} \ \wedge\ \nexists\, (\mathtt{corr\text{-}device}, A, 0, \cdot) \textbf{ in } \mathbf{L} \quad /\!\!/ \ (5) \\
&\qquad \wedge\ \nexists\, B \in CU \textbf{ st } \big((\mathtt{corr\text{-}user}, B, \cdot) \textbf{ in } \mathbf{L} \ \vee\ (\mathtt{corr\text{-}device}, B, 0, \cdot) \textbf{ in } \mathbf{L} \big) \quad /\!\!/ \ (6)
\end{aligned}
$$

We now prove that our instantiation of multi-device group messaging in WhatsApp within the DOGM with revocation formalism, the WA-DOGM protocol, achieves security under these predicates.

Theorem 6.18 (Security of WA-DOGM)**.** The WA-DOGM protocol specified in Definition 6.13 is a secure DOGM with revocation protocol with respect to authentication predicate WA-DOGM.AUTH and confidentiality predicate WA-DOGM.CONF, given that state compromise does not reveal message history stored in '$\pi.hist$'.

That is, for any probabilistic polynomial-time algorithm $\mathcal{A}$ playing the DOGM with revocation security experiment instantiated with WA-DOGM we have that

[17] Note that device revocation of the *sending device* is no help here. It requires the sending session to have knowledge that its own device has been revoked.

$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{WA\text{-}DOGM}}(\Lambda)$ is negligible under the usage parameters $\Lambda = (n_u, n_d, n_i, n_s, n_m)$ in the following conditions.

1) AES-CBC instantiates an IND-CPA secure symmetric encryption scheme (as specified by Definition 2.15) with advantage $\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{AES\text{-}CBC}}(\lambda, n_{ch})$, which we assume to be negligible in the security parameter λ for any PPT adversary.

2) HKDF may be modelled as a random oracle and, in so doing, instantiates a secure PRF (as specified by Definition 2.6) with advantage $\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF\text{-}RO}}(\lambda, n_q)$, which we assume to be negligible in the security parameter λ for any PPT adversary.

3) HMAC may be modelled as a random oracle and, in so doing, instantiates a secure PRF (as specified by Definition 2.6) with advantage $\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC\text{-}RO}}(\lambda, n_q)$, as well as an EUF-CMA secure MAC (as specified by Definition 2.9) with advantage $\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{HMAC\text{-}RO}}(\lambda, n_q)$, both of which we assume to be negligible in the security parameter λ for any PPT adversary.

4) XEd instantiates a SUF-CMA secure digital signature scheme (as specified by Definition 2.35) with advantage $\mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, n_q)$, which we assume to be negligible in the security parameter λ for any PPT adversary.

5) WA-PO is a weakly secure public key orbit (as specified by Definition 5.8) with advantage $\mathsf{Adv}^{w\mathsf{PO}}_{\mathsf{WA\text{-}PO}}(\lambda, n_{ch}, n_\sigma, n_g)$. In Theorem 6.6, we derive an advantage term and prove it to be negligible in the security parameter λ for any PPT adversary, under certain assumptions.

6) WA-PAIR provides secure pairwise channels (as specified by Definition 5.11) with advantage $\mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda, n_d, n_i, n_m)$. In Theorem 6.9, we derive an advantage term and prove it to be negligible in λ for any PPT adversary, under certain assumptions (including that the two key derivations specified in Figure 7 of [CCD+20] may be modelled as random oracles).

7) UNI instantiates a FS-IND-CPA and SUF-CMA secure ratcheted symmetric signcryption scheme (as specified by Definitions 5.14 and 5.15) with advantages $\mathsf{Adv}^{\mathsf{FS\text{-}IND\text{-}CPA}}_{\mathsf{WA\text{-}RSS}}(\lambda, n_m)$ and $\mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{WA\text{-}RSS}}(\lambda, n_m)$ (respectively). In Theorems 6.11 and 6.12, we derive advantage terms and prove these to be negligible for any PPT adversary under certain assumptions. Note that, since we model HKDF and HMAC as random oracles in this result, the HKDF and HMAC advantage terms in Theorem 6.11 should be interpreted with respect to a random oracle.

Proof outline. We separate the proof into two cases. In *Case 1*, we consider the event where the adversary wins the game by breaking revocation or authentication. In *Case 2*, we consider the event where the adversary wins by guessing the challenge bit b. We bound the advantage of the adversary in both cases and demonstrate that under certain assumptions, $\mathcal{A}$'s advantage in winning overall is negligible.

In *Case 1* we begin by preventing the adversary from generating a public key orbit, *orb*, that verifies for any uncorrupted user's primary key ipk_p, or causing a user to output an *orb* containing companion public key ipk_c that it does not own. We demonstrate that doing either implies breaking the $w\mathsf{PO}$ security of the PO scheme. Note that, as a result of these changes, the adversary is no longer able to trigger a revocation win. Next, we prevent the adversary from forging signatures from a device's identity key, be that a primary or companion device. Doing so prevents the usage of forged key bundles for the WA-PAIR scheme. We then use the PAIR-SEC security of the WA-PAIR scheme to prevent the adversary from forging pairwise ciphertexts, and thus from injecting their own Sender Keys session with a public key under their control (for use in the WA-RSS scheme), or a state sharing ciphertext (to allow injecting messages through the history sharing scheme). We additionally demonstrate that the WA-PAIR leaks no information about the contents of its messages. We proceed to demonstrate that forging any WA-RSS message implies breaking the EUF-CMA security of the WA-RSS scheme. To prevent the adversary from forging HS ciphertexts, we replace the computation of the MAC key *ahk* used in the HS scheme with a uniformly random value, and finally show that any adversary capable of forging a HS ciphertext can be used to break the EUF-CMA security of the underlying MAC scheme.

In *Case 2* we reduce (by hybrid argument) the number of encryption challenge queries to a single query, and guess the "encryption challenge" session. We prevent the adversary from forging messages to this session through a WA-PAIR channel by the same arguments as in *Case 1*. Thanks to the confidentiality provided by the PAIR-SEC security of the WA-PAIR scheme, we replace the WA-RSS state sent by the challenge session, $\pi^s_{A,i}$, to its communicating partners in the guessed stage with random replacements. We do the same for any secret seed used to re-encrypt the challenge plaintext in a history sharing ciphertext. We proceed to replace the challenge plaintext m_b for the challenge WA-RSS and HS ciphertexts, arguing that the change is indistinguishable by the security of the respective underlying encryption scheme. In particular, we finish by demonstrating that any adversary that can win the game with non-negligible probability can be used to break the FS-IND-CPA security of the WA-RSS scheme.

Proof. We separate the proof into two cases. In *Case 1*, we consider the probability of the adversary winning the game through a revocation or authentication break. In *Case 2*, we consider the probability of the adversary winning the game by correctly guessing the challenge bit through a confidentiality break. We bound the advantage of winning both cases and demonstrate that under certain assumptions, the adversary's advantage of winning overall is negligible.

Let $\mathsf{Adv}_{\mathsf{AUTH}}$ denote the advantage of the adversary winning the security game by triggering an authentication or revocation win condition, and let $\mathsf{Adv}_{\mathsf{CONF}}$ denote the advantage of the adversary winning the security game when it concludes with the adversary returning the guess of the challenge bit b'.

Since $\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{WA\text{-}DOGM},\Lambda}(\mathcal{A}) \leq \mathsf{Adv}_{\mathsf{AUTH}} + \mathsf{Adv}_{\mathsf{CONF}}$ we may bound the overall advantage of winning by considering each case in isolation.

Case 1: Adversary has triggered either the authentication or revocation win condition

We bound the advantage of $\mathcal{A}$ in triggering an authentication or revocation win condition via the following sequence of games.

Game 0. This is the standard DOGMAR game in *Case 1*, instantiated with the WA-DOGM protocol. Thus we have $\mathsf{Adv}_{\mathsf{AUTH}} = \mathsf{Adv}_{\mathsf{G0}}$.

Game 1. We introduce a series of abort events, $abort^{A}_{w\mathsf{PO}}$, for each user $A \in [n_u]$. The event is triggered if any session, say $\pi^{r}_{B,j}$, has WA-PO.Orbit calculate an orbit $\mathcal{P}$ for the user A (which may be itself) at generation $\pi^{r}_{B,j}.\Gamma[A]$, neither user A nor device $(A, 0)$ have been corrupted at the time the orbit is calculated, and at least one of the following is true.

1) There exists an identity key ipk_c within $\mathcal{P}$ that was not added to A in the processing of an honest $\mathcal{O}$-Create query. Specifically, if there exists $ipk_c \in \mathcal{P}$ for which no query of the form $\mathcal{O}\text{-Create}(A, i) \mapsto dpau$ has been issued for some $i \in [n_d]$ such that $dpau.ipk = ipk_c$.

2) There exists an identity key ipk_c within $\mathcal{P}$ that has been revoked by A in the processing of an honest $\mathcal{O}$-Revoke query and the verifying session is aware of a sufficiently recent orbit generation. Specifically, if there exists $ipk_c \in \mathcal{P}$ for which a query of the form $\mathcal{O}\text{-Revoke}(A, i) \mapsto dpau$ has been issued for some $i \in [n_d]$ such that $dpau.ipk = ipk_c$ and $\pi^{r}_{B,j}.\Gamma[A] \geq dpau.\gamma$.

We bound the probability of each event occurring iteratively, for each user in turn, by a hybrid argument.

Consider the following security reduction, $\mathcal{B}$, which we construct against the weak public-key orbit security of the WA-PO scheme. Let $\mathcal{C}_{w\mathsf{PO}}$ denote the challenger. We replace the primary key of user A with the primary key, pk_p, provided by the challenger at the initialisation of our reduction. We proceed to emulate **Game 0** to our inner adversary with the following changes. Whenever each party needs to update orb_A, rather than computing it themselves, our reduction passes the appropriate query to the challenger, $\mathcal{C}_{w\mathsf{PO}}$: querying PO.Attract when adding companion keys, PO.Repel when removing companion keys, and PO.Refresh to refresh the state, orb_A. Further still, whenever a device's identity key is required to sign its key exchange counterpart or the medium-term pre-key, our reduction issues a query to the limited signing oracle exposed by $\mathcal{C}_{w\mathsf{PO}}$, $\mathcal{O}$-Sign.

Whenever the adversary issues a $\mathcal{O}\text{-CorruptUser}(A)$ or $\mathcal{O}\text{-CorruptDevice}(A, 0)$ query, we utilise the $\mathcal{C}_{w\mathsf{PO}}$ challenger's $\mathcal{O}$-Compromise query to reveal the secret key to the adversary. We do the same for the corruption of companion devices, making use of the $\mathcal{C}_{w\mathsf{PO}}$ challenger's $\mathcal{O}$-Eject query. The reduction tracks the expected values of the "to" and "from" sets of the wPO security experiment, $\mathcal{T}$ and $\mathcal{F}$, upon each change that is made through the aforementioned queries. They do so for user A at each generation of their orbit state.

If at any point, any session $\pi^{r}_{B,j}$ has WA-PO.Orbit calculate an orbit $\mathcal{P}^*$ for user A at generation γ^* and either '$\mathcal{P}^* \setminus \mathcal{T}' \neq^{?} \emptyset$' or '$\mathcal{P}^* \cap \mathcal{C} \setminus \mathcal{F} \neq^{?} \emptyset$' evaluates to

true for the values of $\mathcal{T}'$ and $\mathcal{F}$ at generation γ^*, and neither $\mathcal{O}\text{-}\mathsf{CorruptUser}(A)$ nor $\mathcal{O}\text{-}\mathsf{CorruptDevice}(A, 0)$ has been queried, then the orbit state and generation can be submitted to the challenger $\mathcal{C}_{w\mathsf{PO}}$ to win the experiment.

Applying the reduction above for each of the n_u users in the experiment and, in turn, each respective abort event, we therefore find:[18]

$$\mathsf{Adv}_{\mathsf{G0}} \;\leq\; \mathsf{Adv}_{\mathsf{G1}} + n_u \cdot \mathsf{Adv}^{w\mathsf{PO}}_{\mathsf{WA\text{-}PO}}(\lambda,\; n_d - 1,\; 2 \cdot n_d,\; 2 \cdot n_d - 1)$$

From this game onwards, it is not possible for the adversary to trigger an early termination breaking revocation.

Game 2. We define a set of events, $\{event^{A,i}_{ipk} \;:\; \forall\; A \in [n_u],\; i \in [n_d]\}$, one for every possible device in the experiment. For the device (A, i) with identity keys (isk, ipk), we introduce a set $\Sigma_{A,i}$ in which the challenger records every honest message-signature pair, (m, σ), produced through a call to $\mathsf{XEd.Sign}(isk, m)$ whilst processing a query. The $event^{A,i}_{ipk}$ event is triggered if, at any point in the experiment, a session has a call to $\mathsf{XEd.Verify}(ipk, m', \sigma')$ evaluate to true *but* the pair (m', σ') is not present in the set $\Sigma_{A,i}$ *and* the device (A, i) has not been corrupted at this point in the experiment (either through a call to $\mathcal{O}\text{-}\mathsf{CorruptDevice}(A, i)$ or $\mathcal{O}\text{-}\mathsf{CorruptUser}(A)$ when $i = 0$).

We then introduce an abort event, $abort_{ipk}$, that is triggered if one or more of the aforementioned events occurs, and look to bound the probability of the $abort_{ipk}$ abort event occurring. We do so through a sequence of $D \coloneqq n_u \cdot n_d$ hybrids: **Game 1.0**, **Game 1.1**, ..., **Game 1.D**. In the h-th hybrid, we track the signatures of the first h devices that are created in the experiment, and trigger the abort event as soon as a forgery is detected for an uncorrupted device (as described above). It follows that **Game 1.0** is exactly **Game 1** and **Game 1.D** is exactly **Game 2**. We now consider the change in advantage between two consecutive hybrids, **Game 1.h** and **Game 1.(h+1)**.

We construct an adversary $\mathcal{B}$ against a $\mathsf{SUF\text{-}CMA}$ challenger for the XEd signature scheme, which we denote $\mathcal{C}_{\mathsf{DS}}$. Our adversary proceeds to emulate the security experiment to our inner adversary, $\mathcal{A}$, with the following changes. Let (A, i) be the $(h + 1)$-th device created in the security experiment. We replace the generation of (isk, ipk) for this device with the challenge key pk provided by the $\mathcal{C}_{\mathsf{SUF\text{-}CMA}}$ challenger. Whenever $\mathcal{B}$ requires a signature under isk, rather than computing the signature σ itself, $\mathcal{B}$ instead queries $\mathcal{C}_{\mathsf{SUF\text{-}CMA}}$ with the message m. If any session receives a signature σ' that verifies correctly under pk but was not honestly generated by the device (A, i), then this means that the adversary has created a signature σ' that was not output by $\mathcal{C}_{\mathsf{SUF\text{-}CMA}}$, but verifies correctly. In other words, they have created a forgery. Note that the abort event will not be trigger if the adversary has corrupted the device (via

[18] We determine the parameterisation of each $w\mathsf{PO}$ experiment as follows. First, the number of challenges in the public key orbit experiment maps to the number of companion devices. Second, each primary or companion device's identity key makes use of the signing oracle twice in order to sign the medium-term pre-key and the (fictional) replacement of the identity key for key exchange. Third, the number of generations for each user is naturally limited to registering and revoking each possible companion device once (plus the initial generation consisting solely of the primary device).

a call to $\mathcal{O}$-CorruptDevice(A, i) or $\mathcal{O}$-CorruptUser(A) when $i = 0$), such that this reduction does not need to consider cases where the secret signing key is revealed to the adversary.

Applying this reduction to each hybrid in turn, we find that:

$$\mathsf{Adv}_{\mathsf{G1}} \leq \mathsf{Adv}_{\mathsf{G2}} + n_u \cdot n_d \cdot \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, 2)$$

Game 3. In this game, we abort if any session, $\pi^s_{A,i}$, accepts a pairwise ciphertext from a device, (B, j), *but* the message was not the honest output of that device *and* none of $\mathcal{O}$-CorruptUser(A), $\mathcal{O}$-CorruptDevice$(A, 0)$, $\mathcal{O}$-CorruptDevice(A, i), $\mathcal{O}$-CorruptUser(B), $\mathcal{O}$-CorruptDevice$(B, 0)$ and $\mathcal{O}$-CorruptDevice(B, j) have previously been issued by the adversary.

To do so, we introduce a reduction $\mathcal{B}$ against a challenger for the PAIR-SEC security of the WA-PAIR scheme. Denote this challenger by $\mathcal{C}_{\mathsf{PAIR}}$. Our reduction emulates **Game 2** to the inner adversary, $\mathcal{A}$, with the following changes. Our reduction proceeds to replace the pairwise channel keys for each device with a set output by the $\mathcal{C}_{\mathsf{PAIR}}$ challenge. Our adversary, $\mathcal{B}$, maintains a mapping between the device indices of the PAIR-SEC experiment and those of our emulated DOGM experiment.

Whenever a session $\pi^s_{A,i}$ initialises a new pairwise channel with a session $\pi^r_{B,j}$, the reduction instead simply queries $\mathcal{O}$-Encrypt(i, j, sid, m, m) to $\mathcal{C}_{\mathsf{PAIR}}$, where m is the message that would have been encrypted honestly. We maintain a record of the sessions that have been initiated between any two devices and the session identifiers they use. The output ciphertext c replaces the generation of the ciphertext by $\mathcal{B}$. Note that since we submit the same message as m_0 and m_1, then this is no different to $\mathcal{B}$ generating the ciphertext honestly.

Whenever the inner adversary, $\mathcal{A}$, calls $\mathcal{O}$-CorruptDevice(B', j') for some device (B', j'), our reduction uses its device identifier mapping to submit the appropriate $\mathcal{O}$-CorruptIdentity, $\mathcal{O}$-CorruptShared or $\mathcal{O}$-CorruptSession queries to $\mathcal{C}_{\mathsf{PAIR}}$, making sure to query $\mathcal{O}$-CorruptSession for all of that device's sessions. Similarly, whenever the inner adversary calls $\mathcal{O}$-CorruptUser(B') for some user B', our reduction uses its device identifier mapping to submit the appropriate $\mathcal{O}$-CorruptIdentity query to $\mathcal{C}_{\mathsf{PAIR}}$. Thus, whenever the abort event triggers, the resulting message is a valid forgery against the $\mathcal{C}_{\mathsf{PAIR}}$ challenger. We take the forged pairwise message m' accepted by $\pi^s_{A,i}$ and submit it to $\mathcal{C}_{\mathsf{PAIR}}$ by issuing the appropriate decryption query, resulting in $\mathcal{C}_{\mathsf{PAIR}}$ terminating the experiment early to indicate an authentication break.

It follows that anytime the inner adversary triggers the abort event, then it can be turned into an authentication break against the PAIR-SEC security of WA-PAIR and thus:

$$\mathsf{Adv}_{\mathsf{G2}} \leq \mathsf{Adv}_{\mathsf{G3}} + \mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda, n_u \cdot n_d, 2 \cdot n_e, n_p \cdot n_q)$$

In the bound above, we introduce three ad-hoc parameters, n_e, n_p and n_q, representing limits on the number of ephemeral key pairs a single device may

generate, epochs within a single Signal two-party channel and messages within each epoch (respectively). We use the former, n_e, to bound the number of two-party Signal channels that may be initialized between any pair of devices. The product of the latter two, $n_p \cdot n_q$, bounds the total number of messages exchanged within a single Signal two-party channel.

Note that, since we parameterise the DOGM security experiments primarily with respect to concepts most relevant to the application – i.e. through the number users, devices, Sender Key sessions, stages within those sessions and messages within those stages – the maximum number of messages sent over the underlying pairwise channels is somewhat unrelated to the experiment parameterisation. Thus, we introduce ad-hoc parameters that limit the usage of the underlying pairwise channels.[19]

Game 4. In this game, we abort if any session, $\pi^s_{A,i}$, accepts an inbound WA-RSS session, st_r, sent under a pairwise channel from the designated sending device, $(B, j) = \pi^s_{A,i}.CD[0]$, *but* st_r was not honestly output by that device *and* none of $\mathcal{O}$-CorruptUser(A), $\mathcal{O}$-CorruptDevice$(A, 0)$, $\mathcal{O}$-CorruptDevice(A, i), $\mathcal{O}$-CorruptUser(B),
$\mathcal{O}$-CorruptDevice$(B, 0)$ and $\mathcal{O}$-CorruptDevice(B, j) have previously been issued by the adversary. This is a purely syntactic change that follows directly from **Game 3**. Thus:

$$\mathsf{Adv}_{\mathsf{G3}} = \mathsf{Adv}_{\mathsf{G4}}$$

Game 5. Let $\pi^s_{A,i}$ be the first session that triggers an authentication win condition, upon acceptance of either (a) a ciphertext c from some inbound WA-RSS state st_r, or (b) a history sharing ciphertext $(c_P, c_{hist}, \tau_{hist})$. In this game, we introduce an abort event, $abort_{uni\text{-}forge}$, that is triggered if $\pi^s_{A,i}$ sets *win* to true upon acceptance of a WA-RSS message from the claimed sending session, $\pi^r_{B,j}$ for some r, but the message was not the honest output of device (B, j). In other words, we abort if case (a) is realised. Thus:

$$\mathsf{Adv}_{\mathsf{G4}} = \mathsf{Adv}_{\mathsf{G5}} + \mathsf{Pr}[abort_{uni\text{-}forge}]$$

We bound the probability of $abort_{uni\text{-}forge}$ occurring with **Game 5.1** and **Game 5.2**.

Game 5.1. In this game, we guess the sending session, $\pi^r_{B,j}$, that generated the inbound WA-RSS session state. We additionally guess the stage, t', of the sending session when the relevant inbound WA-RSS state was distributed. We trigger an abort event if any of these guesses are incorrect. Note that, by **Game 4**, we can be sure that such session state was generated by an honest session representing the device $(B, j) = \pi^s_{A,i}.CD[0]$ and that the inbound session state, st_r, was received by $\pi^s_{A,i}$ unmodified.

[19] We could, alternatively, look to limit these by the sum of the maximum number of Sender Key distribution messages and history sharing messages. However, such a bound would ignore that, in practice, these two sets of parameters are unrelated. This is especially true in the case of WhatsApp where, in practice, the underlying pairwise channels are used for direct messaging in addition to their use as signalling infrastructure for group messaging.

Specifically, at the beginning of the experiment we guess a tuple of values (B, j, r, t') and abort the game if $\pi^s_{A,i}$ uses some session, $st^\dagger_r \neq st_r$, to decrypt c. Thus, we find:

$$\mathsf{Pr}[abort_{uni\text{-}forge}] \;\leq\; n_u \cdot n_d \cdot n_i \cdot n_s \cdot \mathsf{Adv}_{\mathsf{G5.1}}$$

Game 5.2. In this game, we abort if $\pi^s_{A,i}$ triggers an authentication win condition upon acceptance of a WA-RSS message from the sending session guessed in the previous game, $\pi^r_{B,j}$, but the message was not the honest output of device (B, j).

We introduce the following reduction, $\mathcal{B}$. When the sending session $\pi^r_{B,j}$ generates a new WA-RSS state for the stage t', $\mathcal{B}$ instead initialises a strong unforgeability challenger for the RSS scheme, which we denote $\mathcal{C}_{\mathsf{SUF\text{-}CMA}}$. Whenever $\pi^r_{B,j}$ sends a message to the group in this stage, our reduction instead issues a call to the $\mathcal{O}$-Send(m) oracle, with the given message m, and distributes the resulting ciphertext c. The output ciphertext c replaces the generation of the ciphertext by $\mathcal{B}$. Similarly, whenever the challenger receives a predicate-respecting decryption, $\mathcal{B}$ simply queries $\mathcal{O}$-Receive(c), with the given ciphertext c, to $\mathcal{C}_{\mathsf{SUF\text{-}CMA}}$. Note that, since we abort only when a session $\pi^s_{A,i}$ accepts a malicious WA-RSS message while processing a predicate-respecting query, we do not need to handle the adversary's $\mathcal{O}$-Compromise queries for this particular WA-RSS state. It follows that, whenever the abort event triggers, we have submitted a forged WA-RSS ciphertext c to the $\mathcal{C}_{\mathsf{SUF\text{-}CMA}}$ challenger when submitting the decryption query, $\mathcal{O}$-Receive(c), on behalf of the $\pi^s_{A,i}$ recipient session.

Thus, any adversary that triggers this abort event, can be turned into a successful adversary against the SUF-CMA security of WA-RSS:

$$\mathsf{Adv}_{\mathsf{G5.1}} \;\leq\; \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{WA\text{-}RSS},\mathcal{A}}(\lambda, n_m)$$

We note here that the adversary can no longer win by causing a device to decrypt forged WA-RSS ciphertexts.

We now turn to considering the authenticity of history sharing, and can be sure that we are in case (b) whereby the first session that causes *win* to be set to true does so upon acceptance of a history sharing ciphertext $(c_P, c_{hist}, \tau_{hist})$. We will rely on the confidentiality of the random seed in the history sharing attachment pointer to ensure the authenticity of the attachment and, thus, ciphertext.[20]

Game 6. In this game, we introduce an abort event that is triggered if $\pi^s_{A,i}$ has DM.Dec successfully return an attachment pointer, m_{ptr}, decrypted from the ciphertext c_P whilst processing a state sharing query for its stage t *but* the attachment pointer was not honestly generated by one of device $(A, 0)$'s sessions. Since $\pi^s_{A,i}$ causes *win* to be set while processing this query, we can

[20] An alternative approach could be to utilise the already determined authentication of the pairwise channel ciphertexts alongside the hash $h \coloneqq \mathsf{SHA256}(c' \,\|\, \tau)$ included within them.

be sure that this query is predicate-respecting. Inspecting the implementation of DOGM.StateShare demonstrates that this is purely a syntactic change that follows directly from **Game 3**. Thus:

$$\mathsf{Adv}_{\mathsf{G5}} = \mathsf{Adv}_{\mathsf{G6}}$$

Game 7. In this game, we guess at the beginning of the experiment the user A and device i whose session, $\pi^s_{A,i}$, triggers the authentication win condition. We make this choice from up to n_u users and up to $n_d - 1$ of their companion devices. Thus:

$$\mathsf{Adv}_{\mathsf{G6}} \;\leq\; n_u \cdot (n_d - 1) \cdot \mathsf{Adv}_{\mathsf{G7}}$$

Game 8. In this game, we simulate the use of pairwise channels to distribute all history sharing attachment pointers from sessions of device $(A, 0)$ to sessions of device (A, i). Namely, we replace the plaintext of the relevant message, m, with a random bit-string of the same length, rm, and store this in the challenger's state. Whenever a session attempts to decrypt this pairwise ciphertext, we replace its output with our stored value.

Consider the following reduction, $\mathcal{B}$, which we construct against a challenger for the PAIR-SEC security of the WA-PAIR scheme. Denote this challenger by $\mathcal{C}_{\mathsf{PAIR}}$. Our reduction proceeds to emulate the security experiment to the inner adversary similarly to our reduction in **Game 3**, with a few changes. As before, it replaces the pairwise channel keys for every device in the experiment with those output by the $\mathcal{C}_{\mathsf{PAIR}}$ challenge. Our adversary, $\mathcal{B}$, maintains a mapping between the device indices of the PAIR-SEC experiment and those of our emulated DOGM experiment. Whenever a session needs to encrypt a message, m, over a pairwise channel, our reduction proceeds to forward the encryption request on as normal, with the following exception. If the plaintext is a history sharing attachment pointer for a message from device $(A, 0)$ to device (A, i), our reduction generates a random replacement for the plaintext, rm, and submits an encryption challenge to the $\mathcal{C}_{\mathsf{PAIR}}$ challenger with (m_0, m_1), where m_0 is the original message m and m_1 is its random replacement rm. Observe that the authentication predicate for WA-DOGM implies the confidentiality predicate for WA-PAIR in this case, such that our adversary may not corrupt either of these devices.

When the bit b sampled by $\mathcal{C}_{\mathsf{PAIR}}$ is 0, then the output ciphertext encrypts m honestly and we are in **Game 7**. However, if the bit b sampled by $\mathcal{C}_{\mathsf{PAIR}}$ is 1, then the output ciphertext encrypts rm instead and we are in **Game 8**. Thus, any adversary that can distinguish our change can be turned into a successful adversary against the PAIR-SEC security of WA-PAIR and we find:

$$\mathsf{Adv}_{\mathsf{G7}} \;\leq\; \mathsf{Adv}_{\mathsf{G8}} + \mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR},\mathcal{A}}(\lambda, n_u \cdot n_d, 2 \cdot n_e, n_p \cdot n_q)$$

Game 9. In this game, we guess at the beginning of the experiment which $\mathcal{O}$-StateShare query it is that produces the history sharing ciphertext whose processing triggers an authentication win condition, and abort if this guess is incorrect.

Observe that the number of history sharing ciphertexts that device (A, i) may receive is limited by the number of pairwise messages they are able to exchange with their primary device, $(A, 0)$. Thus, we guess from a pool of up to $H := 2 \cdot n_e \cdot n_p \cdot n_q$ possible $\mathcal{O}$-StateShare queries against session (A, i), an upper limit we arrive at if all possible pairwise messages between the two parties occur within the history sharing protocol. We implement our guess by picking an index between 0 and $H - 1$ (inclusive) uniformly at random and counting the history sharing queries. Our guess is the h-th such query, and we abort if our guess is incorrect. Thus:

$$\mathsf{Adv}_{\mathsf{G8}} \;\leq\; 2 \cdot n_e \cdot n_p \cdot n_q \cdot \mathsf{Adv}_{\mathsf{G9}}$$

Let $(c_P, c_{hist}, \tau_{hist})$ be the history sharing ciphertext output by the h-th state sharing query in the experiment (with the understanding that its specific contents are not known to the challenger until this query is made).

Game 10. In this game, we replace the key k that is used to encrypt and authenticate the guessed history sharing attachment sent by $(A, 0)$ with a uniformly random and independent value.

Specifically, we introduce a reduction $\mathcal{B}$ that interacts with a PRF challenger $\mathcal{C}_{\mathsf{PRF}}$ of key length 256 and output length 896: for the h-th StateShare query to $(A, 0)$, instead of computing k honestly $\mathcal{B}$ instead queries the $\mathcal{C}_{\mathsf{PRF}}$ challenger's $\mathcal{O}$-PRF query with t and replaces k with the output, which we denote k'.

By **Game 8**, the seed r used to compute k is a uniformly random value and independent of the protocol execution. Thus, when the bit b sampled by $\mathcal{C}_{\mathsf{PRF}}$ is 1, then the output key k' is identically distributed to k and we are in **Game 9**. However, if the bit b sampled by $\mathcal{C}_{\mathsf{PRF}}$ is 0, then $k' \leftarrow\$ \{0,1\}^{896}$ instead and we are in **Game 10**.

Any adversary that can distinguish our change can be turned into a successful adversary against the PRF security of HKDF,[21] and thus:

$$\mathsf{Adv}_{\mathsf{G9}} \;\leq\; \mathsf{Adv}_{\mathsf{G10}} + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF\text{-}RO}}(\lambda, 1)$$

Game 11. In this game, we introduce an abort event that triggers if the adversary forges the guessed history-sharing ciphertext to $\pi^s_{A,i}$.

Specifically, we introduce a reduction $\mathcal{B}$ that interacts with an EUF-CMA challenger $\mathcal{C}_{\mathsf{EUF\text{-}CMA}}$: whenever $\mathcal{B}$ needs to generate a MAC tag using ahk (truncated from k'), instead of computing the MAC tag honestly $\mathcal{B}$ instead queries $\mathcal{C}_{\mathsf{EUF\text{-}CMA}}$ with the ciphertext message c. Since ahk is already uniformly random and independent of the protocol flow, this change is sound. If the adversary triggers the abort event, then the adversary must have computed a fresh ciphertext c and MAC tag τ', breaking EUF-CMA security. $\mathcal{B}$ returns the history-sharing

[21] Recall that, thanks to our modelling of HKDF as a random oracle elsewhere in the protocol, this reduction (and the resulting bound) is made with respect to HKDF when replaced with a random oracle directly.

ciphertext c' and the MAC tag τ' to $\mathcal{C}_{\mathsf{EUF\text{-}CMA}}$ and aborts. Thus, any adversary that can trigger our abort event can be turned into a successful adversary against the EUF-CMA security of HMAC and thus:

$$\mathsf{Adv}_{\mathsf{G10}} \;\leq\; \mathsf{Adv}_{\mathsf{G11}} + \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{HMAC\text{-}RO}}(\lambda, 1)$$

We note here that the adversary can no longer win by causing a device to decrypt forged HS ciphertexts.

However, we must ensure that the primary device has not shared any already forged messages. In particular, we must show that $\pi^s_{A,i}$ is fresh at stage t and message index z, and that every partnered session belonging to its primary device, $(A, 0)$, is also fresh. Specifically, we must show that WA-DOGM.AUTH$(A, 0, p, t^\star, z, B, j) = \textbf{true}$ for all $p \in [n_i]$ for which there exists a $t^\star$ such that $\pi^p_{A,0}.usid[t^\star] = \pi^s_{A,i}.usid[t]$. Recall the authentication predicate to see that WA-DOGM.AUTH$(A, i, s, t, z, \gamma, B, j) = \textbf{true}$ implies the following:

1) $\nexists$ (`corr-user`, A, $\cdot$) **in L**

2) $\nexists$ (`corr-device`, $A, 0$, $\cdot$) **in L**

3) $\nexists$ (`corr-user`, B, $\cdot$) **precedes** ({`dec`, `share`}, A, i, s, t, $\cdot$) **in L**

4) $\nexists$ (`corr-device`, $B, 0$, $\cdot$) **precedes** ({`dec`, `share`}, A, i, s, t, $\cdot$) **in L**

5) $\nexists$ (`corr-device`, B, j, $\cdot$) **precedes** ({`dec`, `share`}, A, i, s, t, $\cdot$) **in L**

6) $\nexists\, r, t'$ **st** (`corr-sess`, $B, j, r, \langle t : t', \cdot \rangle$) **in L** $\wedge\ \pi^s_{A,i}.usid[t] = \pi^r_{B,j}.usid[t']$

It follows that WA-DOGM.AUTH$(A, 0, p, t^\star, z, B, j) = \textbf{true}$ for all partnered sessions belonging to device $(A, 0)$. Thanks to the changes in **Game 7**, we can be sure that the message shared by the primary device is itself genuine.

Thus, we find:

$$\mathsf{Adv}_{\mathsf{G11}} = 0$$

This completes our analysis of *Case 1*.

Case 2: Adversary terminates and outputs a bit b

We bound the probability of the adversary, $\mathcal{A}$, correctly guessing the challenge bit b.

In the case where the adversary has returned a guess of the challenge bit, but has not satisfied the confidentiality predicate, the challenger flips a fair coin to determine whether the adversary has won the game, or not. This provides the adversary with a win probability of $\frac{1}{2}$, and an advantage of zero (in this case).

We proceed to consider the case where the adversary correctly guesses the challenge bit b given that they have satisfied the confidentiality predicate. We do so via the following sequence of games.

Game 0. This is the standard DOGMAR game in *Case 2*, instantiated with the WA-DOGM protocol. Thus we have $\mathsf{Adv}_{\mathsf{CONF}} = \mathsf{Adv}_{\mathsf{G0}}$.

Since the confidentiality of challenge messages is reliant on the secure distribution of the inbound WA-RSS session state and the attachment pointers used for history sharing, our proof of confidentiality proceeds similarity to that of revocation and authentication. Specifically, the next three games look to prove that any message sent between two uncorrupted devices (at the time the message is sent and received) are both confidential and authentic. Note that we do so irrespective of any particular challenge.

Game 1. We make an analogous change to **Game 1** in *Case 1*. We introduce a series of abort events, $abort^{A}_{w\mathsf{PO}}$, for each user $A \in [n_u]$. The event is triggered if any session, say $\pi^r_{B,j}$, has WA-PO.Orbit calculate an orbit $\mathcal{P}$ for the user A (which may be itself) at generation $\pi^r_{B,j}.\Gamma[A]$, neither user A nor device $(A, 0)$ have been corrupted at the time the orbit is calculated, and at least one of the following is true.

1) There exists an identity key ipk_c within $\mathcal{P}$ that was not added to A in the processing of an honest $\mathcal{O}$-Create query. Specifically, if there exists $ipk_c \in \mathcal{P}$ for which no query of the form $\mathcal{O}\text{-Create}(A, i) \mapsto dpau$ has been issued for some $i \in [n_d]$ such that $dpau.ipk = ipk_c$.

2) There exists an identity key ipk_c within $\mathcal{P}$ that has been revoked by A in the processing of an honest $\mathcal{O}$-Revoke query and the verifying session is aware of a sufficiently recent orbit generation. Specifically, if there exists $ipk_c \in \mathcal{P}$ for which a query of the form $\mathcal{O}\text{-Revoke}(A, i) \mapsto dpau$ has been issued for some $i \in [n_d]$ such that $dpau.ipk = ipk_c$ and $\pi^r_{B,j}.\Gamma[A] \geq dpau.\gamma$.

We bound the probability of each event occurring iteratively, for each user in turn, by a hybrid argument.

Consider the following security reduction, $\mathcal{B}$, which we construct against the weak public-key orbit security of the WA-PO scheme. Let $\mathcal{C}_{w\mathsf{PO}}$ denote the challenger, for which we replace the primary key of user A with the primary key, pk_p, provided by the challenger at the initialisation of our reduction. We proceed to emulate **Game 0** to our inner adversary with the following changes. Whenever each party needs to update orb_A, rather than computing it themselves, our reduction passes the appropriate query to the challenger, $\mathcal{C}_{w\mathsf{PO}}$: querying PO.Attract when adding companion keys, PO.Repel when removing companion keys, and PO.Refresh to refresh the state, orb_A. Further still, whenever a device's identity key is required to sign its key exchange counterpart or the medium-term pre-key, our reduction issues a query to the $\mathcal{O}$-Sign limited signing oracle exposed by $\mathcal{C}_{w\mathsf{PO}}$.

Whenever the adversary issues a $\mathcal{O}\text{-CorruptUser}(A)$ or $\mathcal{O}\text{-CorruptDevice}(A, 0)$ query, we utilise the $\mathcal{C}_{w\mathsf{PO}}$ challenger's $\mathcal{O}$-Compromise query to reveal the secret key to the adversary. We do the same for the corruption of companion devices, making use of the $\mathcal{C}_{w\mathsf{PO}}$ challenger's $\mathcal{O}$-Eject query. The reduction tracks the expected values of the "to" and "from" sets of the wPO security experiment,

$\mathcal{T}$ and $\mathcal{F}$, upon each change that is made through the aforementioned queries. They do so for user A at each generation of their orbit state.

If at any point, any session $\pi^r_{B,j}$ has WA-PO.Orbit calculate an orbit $\mathcal{P}^*$ for user A at generation γ^* and either '$\mathcal{P}^* \setminus \mathcal{T}' \neq_? \emptyset$' or '$\mathcal{P}^* \cap \mathcal{C} \setminus \mathcal{F} \neq_? \emptyset$' evaluates to true for the values of $\mathcal{T}'$ and $\mathcal{F}$ at generation γ^*, and neither user A nor device $(A, 0)$ have been corrupted at the time the orbit is calculated, the orbit state and generation can be submitted to the challenger $\mathcal{C}_{w\mathsf{PO}}$ to win the experiment.

Applying the reduction above for each of the n_u users in the experiment and, in turn, each respective abort event, we therefore find:

$$\mathsf{Adv}_{\mathsf{G0}} \;\leq\; \mathsf{Adv}_{\mathsf{G1}} + n_u \cdot \mathsf{Adv}^{w\mathsf{PO}}_{\mathsf{WA\text{-}PO}}(\lambda,\; n_d - 1,\; 2 \cdot n_d,\; 2 \cdot n_d - 1)$$

Game 2. We make an analogous change to **Game 2** in *Case 1*. We define a set of events, $\{event^{A,i}_{ipk} \;:\; \forall\, A \in [n_u],\; i \in [n_d]\}$, one for every possible device in the experiment. For the device (A, i) with identity keys (isk, ipk), we introduce a set $\Sigma_{A,i}$ in which the challenger records every honest message-signature pair, (m, σ), produced through a call to $\mathsf{XEd.Sign}(isk, m)$ whilst processing a query. The $event^{A,i}_{ipk}$ event is triggered if, at any point in the experiment, a session has a call to $\mathsf{XEd.Verify}(ipk, m', \sigma')$ evaluate to true *but* the pair (m', σ') is not present in the set $\Sigma_{A,i}$ *and* the device (A, i) has not been corrupted at this point in the experiment (either through a call to $\mathcal{O}$-CorruptDevice(A, i) or $\mathcal{O}$-CorruptUser(A) when $i = 0$).

We then introduce an abort event, $abort_{ipk}$, that is triggered if one or more of the aforementioned events occurs, and look to bound the probability of the $abort_{ipk}$ abort event occurring. We do so through a sequence of $D \coloneqq n_u \cdot n_d$ hybrids: **Game 1.0**, **Game 1.1**, ..., **Game 1.D**. In the h-th hybrid, we track the signatures of the first h devices that are created in the experiment, and trigger the abort event as soon as a forgery is detected for an uncorrupted device (as described above). It follows that **Game 1.0** is exactly **Game 1** and **Game 1.D** is exactly **Game 2**. We now consider the change in advantage between two consecutive hybrids, **Game 1.h** and **Game 1.(h+1)**.

We construct an adversary $\mathcal{B}$ against a SUF-CMA challenger for the XEd signature scheme, which we denote $\mathcal{C}_{\mathsf{DS}}$. Our adversary proceeds to emulate the security experiment to our inner adversary, $\mathcal{A}$, with the following changes. Let (A, i) be the $(h+1)$-th device created in the security experiment. We replace the generation of (isk, ipk) for this device with the challenge key pk provided by the $\mathcal{C}_{\mathsf{SUF\text{-}CMA}}$ challenger. Whenever $\mathcal{B}$ requires a signature under isk, rather than computing the signature σ itself, $\mathcal{B}$ instead queries $\mathcal{C}_{\mathsf{SUF\text{-}CMA}}$ with the message m. If any session receives a signature σ' that verifies correctly under pk but was not honestly generated by the device (A, i), then this means that the adversary has created a signature σ' that was not output by $\mathcal{C}_{\mathsf{SUF\text{-}CMA}}$, but verifies correctly. In other words, they have created a forgery. Note that the abort event will not be trigger if the adversary has corrupted the device (via a call to $\mathcal{O}$-CorruptDevice(A, i) or $\mathcal{O}$-CorruptUser(A) when $i = 0$), such that this reduction does not need to consider cases where the secret signing key is revealed to the adversary.

Applying this reduction to each hybrid in turn, we find that:

$$\mathsf{Adv}_{\mathsf{G1}} \leq \mathsf{Adv}_{\mathsf{G2}} + n_u \cdot n_d \cdot \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, 2)$$

Game 3. We make a similar change to **Game 3** in *Case 1*. In this game, we abort if any session, $\pi^s_{A,i}$, accepts a pairwise ciphertext from a device, (B, j), *but* the message was not the honest output of that device *and* none of $\mathcal{O}$-CorruptUser(A), $\mathcal{O}$-CorruptDevice$(A, 0)$, $\mathcal{O}$-CorruptDevice(A, i), $\mathcal{O}$-CorruptUser(B), $\mathcal{O}$-CorruptDevice$(B, 0)$ and $\mathcal{O}$-CorruptDevice(B, j) have previously been issued by the adversary.

To do so, we introduce a reduction $\mathcal{B}$ against a challenger for the PAIR-SEC security of the WA-PAIR scheme. Denote this challenger by $\mathcal{C}_{\mathsf{PAIR}}$. Our reduction emulates **Game 2** to the inner adversary, $\mathcal{A}$, with the following changes. Our reduction proceeds to replace the pairwise channel keys for each device with a set output by the $\mathcal{C}_{\mathsf{PAIR}}$ challenge. Our adversary, $\mathcal{B}$, maintains a mapping between the device indices of the PAIR-SEC experiment and those of our emulated DOGM experiment.

Whenever a session $\pi^s_{A,i}$ initialises a new pairwise channel with a session $\pi^r_{B,j}$, the reduction instead simply queries $\mathcal{O}$-Encrypt(i, j, sid, m, m) to $\mathcal{C}_{\mathsf{PAIR}}$, where m is the message that would have been encrypted honestly. We maintain a record of the sessions that have been initiated between any two devices and the session identifiers they use. The output ciphertext c replaces the generation of the ciphertext by $\mathcal{B}$. Note that since we submit the same message as m_0 and m_1, then this is no different to $\mathcal{B}$ generating the ciphertext honestly.

Whenever the inner adversary, $\mathcal{A}$, calls $\mathcal{O}$-CorruptDevice(B', j') for some device (B', j'), our reduction uses its device identifier mapping to submit the appropriate $\mathcal{O}$-CorruptIdentity, $\mathcal{O}$-CorruptShared or $\mathcal{O}$-CorruptSession queries to $\mathcal{C}_{\mathsf{PAIR}}$, making sure to query $\mathcal{O}$-CorruptSession for all of that device's sessions. Similarly, whenever the inner adversary calls $\mathcal{O}$-CorruptUser(B') for some user B', our reduction uses its device identifier mapping to submit the appropriate $\mathcal{O}$-CorruptIdentity query to $\mathcal{C}_{\mathsf{PAIR}}$.

Note that, since we only trigger the abort event when none of $\mathcal{O}$-CorruptUser(B), $\mathcal{O}$-CorruptDevice$(B, 0)$ and $\mathcal{O}$-CorruptDevice(B, j) for the claimed sending device (B, j) have previously been issued by the challenger in the emulated experiment, the WA-PAIR security predicate is satisfied. Thus, whenever the abort event triggers, the resulting message is a valid forgery against the $\mathcal{C}_{\mathsf{PAIR}}$ challenger. We take the forged pairwise message m' accepted by $\pi^s_{A,i}$ and submit it to $\mathcal{C}_{\mathsf{PAIR}}$ by issuing the appropriate decryption query, resulting in $\mathcal{C}_{\mathsf{PAIR}}$ terminating the experiment early to indicate an authentication break.

It follows that anytime the inner adversary triggers the abort event, then it can be turned into an authentication break against the PAIR-SEC security of WA-PAIR and thus:

$$\mathsf{Adv}_{\mathsf{G2}} \leq \mathsf{Adv}_{\mathsf{G3}} + \mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda, n_u \cdot n_d, 2 \cdot n_e, n_p \cdot n_q)$$

The changes introduced in this game ensure that all pairwise channel messages exchanged between uncorrupted devices were honestly generated.

Games 4, ..., N+3. We consider a series of hybrids **Game 4**, ..., **Game N+3** where $N \coloneqq n_u \cdot n_d \cdot n_i \cdot n_s \cdot n_m$; one for each possible challenge encryption that may be triggered in the game. For each hybrid **Game h** for $4 \leq h \leq N+3$, we look to bound the advantage of the adversary in determining the challenge bit through the $(h-4)$-th challenge encryption. We find, by a hybrid argument:

$$\mathsf{Adv}_{\mathsf{G0}} \;\leq\; \sum_{h=4}^{N+3} \mathsf{Adv}_{\mathsf{Gh}} \;=\; n_u \cdot n_d \cdot n_i \cdot n_s \cdot n_m \cdot \mathsf{Adv}_{\mathsf{Gh}}$$

In what follows, we bound the advantage of an adversary in a single such hybrid, **Game h**, and do so through the following sequence of games.

Game h.0. This is the h-th hybrid as described above: $\mathsf{Adv}_{\mathsf{Gh}} = \mathsf{Adv}_{\mathsf{Gh.0}}$.

Game h.1. In this game, we guess the session $\pi^s_{A,i}$ and stage t of the $(h-3)$-th challenge encryption, such that the adversary issues a $\mathcal{O}$-$\mathsf{Encrypt}(A, i, s, m_0, m_1)$ query for which $m_0 \neq m_1$. We guess a tuple of values, (A, i, s, t), at the start of the experiment and abort if the $(h-3)$-th challenge encryption query is not made to $\pi^s_{A,i}$ within stage t. It follows that,

$$\mathsf{Adv}_{\mathsf{Gh.0}} \;\leq\; n_u \cdot n_d \cdot n_i \cdot n_s \cdot \mathsf{Adv}_{\mathsf{Gh.1}}$$

See that, by construction of the DOGMAR experiment, the adversary cannot win if the challenge encryption query we handle in this hybrid does not satisfy the confidentiality predicate. Thus, we may assume that $\mathsf{CONF}(\mathbf{L}, \mathit{chall}) \mapsto \mathbf{true}$ when $\mathbf{L}$ is taken at the end of the experiment. Note that, letting $\mathit{chall} = (\,\mathsf{enc}, A, i, s, \Gamma, CU, CD, t, z, m_0, m_1, c\,)$, we can be sure of the values of A, i, s and t from the start of the experiment, but we do not necessarily know the values of $\Gamma, CU, CD, z, m_0, m_1$ or c ahead of the challenge.

Game h.2. When $\pi^s_{A,i}$ initialises stage t, we record the list of communicating users and devices.

We proceed to simulate the use of pairwise channels to distribute messages privately between each pair of devices. For those pairwise messages that distribute the inbound WA-RSS session state, or a history sharing attachment pointer, that allows decryption of our challenge encryption, we replace the plaintext message, m, with a random bit-string of the same length, rm, and store this mapping in a table we maintain for each sending device. Whenever a session decrypts a pairwise ciphertext, we replace its output with appropriate value from our table (where applicable).

How do we determine which pairwise messages to replace? Any pairwise message distributing the inbound WA-RSS state for stage t when the challenge has not yet occurred must be replaced (since it will contain symmetric state that may be ratcheted forward to the message index of the challenge). However, after the challenge encryption has occurred, we must limit our replacement to those with a message index less than the challenge message index. History sharing messages are selected in the opposite fashion. No history sharing message can

give access to a challenge encryption until after the challenge encryption has occurred and, as such, they must not be replaced. Once the challenge encryption has occurred, we know that any history sharing message for a message index less than or equal to that of the challenge will provide access to its plaintext.[22]

Consider the following reduction, $\mathcal{B}$, which we construct against a challenger for the PAIR-SEC security of the WA-PAIR scheme. Denote this challenger by $\mathcal{C}_{\mathsf{PAIR}}$. Our reduction proceeds to emulate the security experiment to the inner adversary similarly to our reduction in **Game 3**, with a few changes. As before, it replaces the pairwise channel keys for each device with the set output by the $\mathcal{C}_{\mathsf{PAIR}}$ challenge. Our adversary, $\mathcal{B}$, maintains a mapping between the device indices of the PAIR-SEC experiment and those of our emulated DOGM experiment. Whenever a session needs to encrypt a message, m, over a pairwise channel, our reduction proceeds to forward the encryption request on as normal, with the following exception. If the plaintext is a WA-RSS session distribution message or an attachment pointer for a history sharing message of a challenge encryption, our reduction generates a random replacement for the plaintext, rm, and submits an encryption challenge to the $\mathcal{C}_{\mathsf{PAIR}}$ challenger with (m_0, m_1), where m_0 is the original message m and m_1 is its random replacement rm. Observe that the confidentiality predicate for WA-DOGM implies the confidentiality predicate for WA-PAIR.

When the bit b sampled by $\mathcal{C}_{\mathsf{PAIR}}$ is 0, then the output ciphertext encrypts m honestly and we are in **Game *h*.1**. However, if the bit b sampled by $\mathcal{C}_{\mathsf{PAIR}}$ is 1, then the output ciphertext encrypts rm instead and we are in **Game *h*.2**. Thus, any adversary that can distinguish our change can be turned into a successful adversary against the PAIR-SEC security of WA-PAIR and we find:

$$\mathsf{Adv}_{\mathsf{Gh.1}} \leq \mathsf{Adv}_{\mathsf{Gh.2}} + \mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda, n_u \cdot n_d, 2 \cdot n_e, n_p \cdot n_q)$$

Game *h*.3. In this game, we replace the key k that is used to encrypt and authenticate any history sharing attachment sent by a recipient primary device $(B, 0)$ for $B \in CU$ with a uniformly random and independent value. Since this may only occur after the initial distribution of the WA-RSS session state, the challenger has full knowledge of the set CU at the point in time these replacements need to be made.

Specifically, we introduce a reduction $\mathcal{B}$ that interacts with a PRF challenger $\mathcal{C}_{\mathsf{PRF}}$ of key length 256 and output length 896. We consider any $\mathcal{O}$-StateShare$(B, 0, p, B, j, t', z')$ query from a session $\pi^p_{B,0}$ for which $B \in CU$ and $\pi^p_{B,0}.usid[t'] = \pi^s_{A,i}.usid[t]$ to a device (B, j) for stage t' and message index z'. For each such query, rather than computing k honestly, $\mathcal{B}$ instead queries the $\mathcal{C}_{\mathsf{PRF}}$ challenger's $\mathcal{O}$-PRF oracle with r and replaces k with the output, which we denote rk.

By **Game *h*.2**, the seed r used to compute k is a uniformly random value and independent of the protocol execution. Thus, when the bit b sampled by $\mathcal{C}_{\mathsf{PRF}}$ is 1, then the output key rk is identically distributed to k and we are in

[22] See that our implementation of history sharing in WA-DOGM shares all plaintexts in the requested stage with a message index greater than or equal to the requested message index.

Game *h*.2. However, if the bit b sampled by $\mathcal{C}_{\mathsf{PRF}}$ is 0, then $rk \leftarrow\$ \{0,1\}^{896}$ instead and we are in **Game *h*.3**.

Thus, any adversary that can distinguish our change can be turned into a successful adversary against the PRF security of HKDF.[23] We apply this reduction iteratively, by a hybrid argument, for which the number of hybrids is the maximum number of such history sharing messages. We may calculate the maximum number of relevant history sharing ciphertexts in the same manner as is done in **Game 6** of *Case 1*. Thus, we find:

$$\mathsf{Adv}_{\mathsf{Gh.2}} \leq \mathsf{Adv}_{\mathsf{Gh.3}} + n_u \cdot (n_d - 1) \cdot 2 \cdot n_e \cdot n_p \cdot n_q \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF\text{-}RO}}(\lambda, 1)$$

Game *h*.4. In this game, we demonstrate that no history sharing ciphertext encrypting m_b leaks anything about the challenge plaintext, by the IND-CPA security of AES-CBC. We do so by replacing the ciphertext encrypting the history sharing attachment with a random bit string of the same length. As in **Game *h*.3**, since the creation of history sharing attachments occurs after the initialisation of stage t, the challenger has full knowledge of which plaintexts need replacement at the point in time these replacements are made.

Specifically, we introduce a reduction $\mathcal{B}$ that interacts with an IND-CPA challenger $\mathcal{C}_{\mathsf{IND\text{-}CPA}}$: when **Game *h*.4** would normally encrypt the plaintext m_b using aek (truncated from k') using a call to AES-CBC, our reduction $\mathcal{B}$ instead computes issues an encryption query for m_b to the $\mathcal{C}_{\mathsf{IND\text{-}CPA}}$ (m_b, m'_b) (where m'_b is a uniformly random string of the same length).. Since aek is already uniformly random and independent of the protocol flow by **Game *h*.3**, this change is sound. Note that when the bit b sampled by $\mathcal{C}_{\mathsf{IND\text{-}CPA}}$ is 0, then the challenge plaintext m_b is encrypted and we are in **Game *h*.3**. However, if the bit b sampled by $\mathcal{C}_{\mathsf{IND\text{-}CPA}}$ is 1, then the HS ciphertext is a uniformly random string instead and we are in **Game *h*.4**.

Any adversary that can distinguish our change can be turned into a successful adversary against the IND-CPA security of AES-CBC and thus:

$$\mathsf{Adv}_{\mathsf{Gh.3}} \leq \mathsf{Adv}_{\mathsf{Gh.4}} + n_u \cdot (n_d - 1) \cdot 2 \cdot n_e \cdot n_p \cdot n_q \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{AES\text{-}CBC}}(\lambda, 1)$$

Game *h*.5. In this game, we show that if the adversary, $\mathcal{A}$, can distinguish between the encryption of m_0 and m_1 by $\pi^s_{A,i}$ then they can be turned into a successful adversary against the FS-IND-CPA security of the WA-RSS scheme.

Specifically, we define a reduction $\mathcal{B}$ that acts identically to **Game *h*.5**. However, when the test session $\pi^s_{A,i}$ initialises the WA-RSS state at the beginning of stage T, $\mathcal{B}$ instead initialises a FS-IND-CPA challenger $\mathcal{C}_{\mathsf{FS\text{-}IND\text{-}CPA}}$ that generates a receiving state st_r for them. Since the receiving state is never sent in the pairwise channel (by **Game *h*.2**) then this replacement cannot be detected by the adversary. Whenever $\pi^s_{A,i}$ needs to encrypt using the WA-RSS sending

[23] Recall that this reduction (and the resulting bound) is made with respect to HKDF when replaced with a random oracle directly. This is due to our modelling of HKDF as a random oracle elsewhere in the protocol.

state st_s, $\mathcal{B}$ instead queries $\mathcal{C}_{\mathsf{FS\text{-}IND\text{-}CPA}}$ with $\mathcal{O}\text{-}\mathsf{Send}(m_0, m_1)$. If, after the challenge encryption query was made to $\mathcal{B}$, $\mathcal{A}$ calls $\mathcal{O}\text{-}\mathsf{Compromise}(A, i, s)$, $\mathcal{B}$ queries $\mathcal{C}_{\mathsf{FS\text{-}IND\text{-}CPA}}$ with $\mathcal{O}\text{-}\mathsf{Corrupt}$ and returns the output sending state st_s to $\mathcal{A}$. When $\mathcal{A}$ terminates, and outputs a bit b, $\mathcal{B}$ simply forwards it to $\mathcal{C}_{\mathsf{FS\text{-}IND\text{-}CPA}}$.

The success of $\mathcal{A}$ is bound by the success of $\mathcal{B}$, and thus we find:

$$\mathsf{Adv}_{\mathsf{Gh.4}} \;\leq\; \mathsf{Adv}^{\mathsf{FS\text{-}IND\text{-}CPA}}_{\mathsf{WA\text{-}RSS}}(\lambda, n_m)$$

This completes our analysis of *Case 2*.

We combine the two cases above to arrive at an upper bound for the advantage of any PPT adversary.

$$\begin{aligned}
&\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{WA\text{-}DOGM}}(n_u, n_d, n_i, n_s, n_m) \\
&\quad\leq \Big[\quad /\!/ \text{ Case 1: Authentication} \\
&\qquad n_u \cdot \mathsf{Adv}^{w\mathsf{PO}}_{\mathsf{WA\text{-}PO}}(\lambda,\ n_d - 1,\ 2 \cdot n_d,\ 2 \cdot n_d - 1) + \\
&\qquad n_u \cdot n_d \cdot \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, 2) + \\
&\qquad \mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda, n_u \cdot n_d, 2 \cdot n_e, n_p \cdot n_q) + \\
&\qquad n_u \cdot n_d \cdot n_i \cdot n_s \cdot \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{WA\text{-}RSS}}(\lambda, n_m) + \\
&\qquad n_u \cdot (n_d - 1) \cdot \Big[\\
&\qquad\quad \mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda, n_u \cdot n_d, 2 \cdot n_e, n_p \cdot n_q) + \\
&\qquad\quad 2 \cdot n_e \cdot n_p \cdot n_q \cdot \big(\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF\text{-}RO}}(\lambda, 1) \;+ \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{HMAC\text{-}RO}}(\lambda, 1) \big) \\
&\qquad \Big] \\
&\quad \Big] + \Big[\quad /\!/ \text{ Case 2: Confidentiality} \\
&\qquad n_u \cdot \mathsf{Adv}^{w\mathsf{PO}}_{\mathsf{WA\text{-}PO}}(\lambda,\ n_d - 1,\ 2 \cdot n_d,\ 2 \cdot n_d - 1) + \\
&\qquad n_u \cdot n_d \cdot \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, 2) + \\
&\qquad \mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda, n_u \cdot n_d, 2 \cdot n_e, n_p \cdot n_q) + \\
&\qquad (n_u \cdot n_d \cdot n_i \cdot n_s \cdot n_m) \cdot (n_u \cdot n_d \cdot n_i \cdot n_s) \cdot \Big[\\
&\qquad\quad \mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda, n_u \cdot n_d, 2 \cdot n_e, n_p \cdot n_q) + \\
&\qquad\quad \mathsf{Adv}^{\mathsf{FS\text{-}IND\text{-}CPA}}_{\mathsf{WA\text{-}RSS}}(\lambda, n_m) + \\
&\qquad\quad n_u \cdot (n_d - 1) \cdot 2 \cdot n_e \cdot n_p \cdot n_q \cdot \big(\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF\text{-}RO}}(\lambda, 1) \;+\; \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{AES\text{-}CBC}}(\lambda, 1) \big) \\
&\qquad \Big] \\
&\quad \Big]
\end{aligned}$$

Observe that the above bound is a polynomial function of the experiment parameters, $\Lambda_{\mathsf{DOGMAR}} = (n_u, n_d, n_i, n_s, n_m)$, and the respective advantage against the security of each primitive used. It follows that the advantage of any PPT adversary is at most a negligible function of the security parameter used to instantiate each primitive.

This completes our proof. □

6.2.9 Interpretation

We first reiterate the provided guarantees from the perspective of an individual device; the level at which WhatsApp gives meaningful guarantees. We then summarise caveats to these guarantees.

Confidentiality and authentication. Messages sent from a device are confidential between itself and the verified devices of a group's members. Similarly, our device will only accept messages from verified devices of the group's members. These guarantees are maintained throughout changes to both the group membership and each member's device composition, albeit with the caveat that this group membership is local to the perspective of the sender (for confidentiality) and the recipient (for authentication).

When a device is notified that a member has been added to the group, that user's verified devices will have access to future messages but not those from the past. When a device is notified that a member has been removed from the group, the removed user's devices will not have access to future messages. However, note that user group membership is controlled by the server.

When our device is notified that a member has added a new device, the new device will have access to future messages.[24] When our device is notified that a member has revoked a device, the revoked device will not have access to future messages. Unlike group membership, users control their own device lists. The inclusion of in-chat device consistency information in pairwise messages guarantees that device revocation can be detected if our device is able to securely communicate with at least one honest device of the other user. Additionally, while our analysis does not capture the automatic expiry of device lists, this mechanism provides an upper bound for the time taken for device revocations to be detected.

Device revocation. WhatsApp's device management sub-protocol enables a *user* to effectively recover from a *detected compromise* of any number of their companion devices. If a user suspects, say, their laptop was compromised, they can remove the device from their account using their phone. This will be propagated next time they send a a *direct* message to another user. This is because the ICDC information added to the follow-up message alerts the other user's devices that a new generation of the multi-device state is available. If the message is blocked by the adversary, the device list will eventually expire which make the primary device the only verified device for our user.

Lack of post-compromise security. Our analysis in Section 6.2.5 follows that of [CFKN20, CJN23] in demonstrating that WhatsApp's pairwise channels lack meaningful PCS guarantees considering their use of multiple parallel sessions between devices. When only pairwise session secrets have been compromised, recovery requires for each compromised session to either have healed (through the Double Ratchet) or to have been rotated out of the session list. The latter case requires the adversary to be passive while 40 new honest sessions

[24] Whilst the new device is not able to decrypt past ciphertexts, the history sharing feature gives it access to the same historical messages as the user's other devices.

are created.[25] When a device's identity key have been compromised, the multi-session setting enables the adversary to initialise a new compromised session at any point in the future. This contrasts with the single-session setting where compromise of the identity key does not affect the security of existing conversations [CCD+20].[26] Since there is little practical difference between compromise of a device's identity key and pairwise session state, our analysis in Section 6.2.7 combines these two cases.

In addition, WhatsApp clients keep the five most recent inbound Sender Keys sessions sent by each device, which limits the post-compromise security of messages in group chats, an issue that is compounded by the aforementioned lack of meaningful PCS in pairwise channels used to distribute Sender Keys sessions. In the case that only the Sender Keys session states are compromised, security can be restored after the sender has rotated the Sender Keys state five times, i.e. after five membership changes where the compromised sender has sent a message between each change (to trigger session rotation). However, if pairwise session state has been compromised, security is only guaranteed after the pairwise channels have healed and then the sender key has been rotated five times.[27]

Echoing our discussion of Matrix in Section 6.1.6, while this does contrast with the understanding of PCS guarantees within the cryptographic literature, this is not unusual in practice. Given that many deployed protocols lack meaningful PCS guarantees and that prior work established that these guarantees do not match up well with some higher-risk settings [ABJM21], we consider the question of what PCS guarantees *should* be targeted in design an exciting area for future multidisciplinary work. Here, we highlight WhatsApp's approach to tackle *detected* compromises via device revocation, which does not map to the PCS notions in the literature but seems to provide meaningful guarantees for some important settings.

FS and PCS in primitives to enforce group membership. Indeed, as we saw in our analysis of Matrix, the provision of forward secrecy and post-compromise security within Sender Keys enables the enforcement of group membership changes. Namely, the forward secrecy provided by the symmetric ratchet ensures that new group members cannot decrypt previous messages. Similarly, the rotation of Sender Key sessions upon the removal of a group member ensures that they cannot decrypt new messages.

[25] The WA-PAIR.AUTH and WA-PAIR.CONF security predicates state when we should expect the respective security guarantee to apply from the perspective of the challenger in the PAIR-SEC security experiment (with a global view of the protocol execution). In contrast, these statements describe when an honest client, with knowledge of the start and end of a compromise, can be sure that security has been restored.

[26] In the terminology of [CCG16], this setting does not provide *PCS via state* but does provide *PCS via weak compromise*, if all compromised sessions have healed or expired.

[27] When clients store previous Sender Keys sessions to allow out-of-order decryption, this issue can be mediated by including the number of messages sent in the last session when initiating a new session. The recipient can then derive the message keys for the missing messages from the previous session, allowing it to safely destroy the chain key. Indeed, a similar issue exists across consecutive epochs of Signal pairwise channels, where a similar improvement has been discussed [DGP22].

Adversarially-controlled group management. As mentioned above and in prior works, WhatsApp's design and implementation allow the server, i.e. the adversary, to control group membership. While a correctly implemented client will prevent the addition of *ghost*, i.e. invisible, users or devices,[28] this still means those reliant on WhatsApp do not have control over who they communicate with and to whom their devices will distribute session keys. Put differently, the adversary's presence in a group chat will not be invisible in correctly implemented clients, but an adversary calling itself "Alice" in a group chat of 1024 users is arguably reasonably well hidden from its surveillance targets.

Moreover, as a corollary, there is also no consistency guarantees among users or even devices of a user. For example, Alice may be given a different view of group membership to Bob and Alice's laptop may have a different view of group membership than Alice's phone. Alice checking for adversarially added members on her phone does not mean that none have been added to her laptop and Bob cannot validate group membership on behalf of the other group members. This is reflected in our model: the DOGM security experiment gives the adversary control over the group membership and does not guarantee a consistent view of group membership to individual sessions. Concretely, if Alice receives a confidential message from Bob, she cannot be sure that this message was confidential between herself, Bob and the other group members displayed in the user interface: Bob's client may have been instructed by the server to add Gilbert to the group.

Lack of domain separation. WhatsApp (like Signal) utilises the same key pair for the long-term device keys used for key exchange and signatures.[29] Following prior work, we do not model this lack of domain separation and stress that the security implications are unknown.

6.3 Discussion

Recalling the security predicates derived in Sections 6.1.5 and 6.2.8, we see that Matrix and WhatsApp provide broadly similar security guarantees. We proceed to highlight their similarities and differences.

Confidentiality. When a client sends a message for a set of intended recipient users, they can be sure that confidentiality is maintained between themselves and those users providing that the secret session state is not compromised at the given stage before this target message was sent and none of those recipients ever has their user secrets, or those of any of their present or future devices, corrupted.

Thanks to the history sharing functionality of Matrix and WhatsApp, the intended recipient *devices* have little effect on the confidentiality of messages. That is, if a sending session were to deliberately exclude a particular recipient

[28] Ghost users were a 2018 GCHQ proposal to circumvent end-to-end encryption [LR18].
[29] We note that WhatsApp adds another domain to the mix in comparison to Signal.

device, but another device of the same user is an intended recipient, the sender cannot expect messages to remain confidential from the excluded device.[30]

The device revocation feature of WhatsApp allows confidentiality to be restored even after a companion device has been compromised, providing the device has been revoked by the user's primary device and the sender has become aware of this revocation. This requires that the sending device has received a pairwise ciphertext message from an honest device representing the user that performed the revocation.

The restrictive history sharing in WhatsApp, whereby only primary devices share history with their own companion devices, works in tandem with the device revocation functionality to ensure that the history sharing feature does not undermine confidentiality against revoked devices. In particular, since it is only the primary device that revokes companions, and it is only the primary device that may share history, we can be sure that message history is never shared with a revoked device.

Authentication. When a client receives an application message that it believes was sent by a particular device in a particular group, the recipient can be sure that the message was sent by the given device for the given group, under the following circumstances. First, the sending session state cannot have been compromised whilst it was holding the signing key for the given stage. Second, neither the sending user nor sending device can have been corrupted before the state for this stage was received.

If the recipient user has been corrupted at any point in time, we cannot guarantee that the message was not injected through the history sharing functionality. Since Matrix allows for history sharing between any two of a user's devices, it requires that none of the recipient user's devices have been corrupted. In contrast, since WhatsApp only allows history sharing from the primary device to companion devices, we need only limit the primary device (i.e. the user secrets) from being corrupted.

Thanks to device revocation, WhatsApp can restore authentication guarantees in certain circumstances. We see that WhatsApp clients will not authenticate a message as originating from a particular user if that user has revoked the device (and the client is aware of that revocation). This is true for group messages, as well as pairwise messages, such as those used for the distribution of inbound group sessions or history sharing. Thus, WhatsApp clients can additionally provide authentication after a relevant device has been compromised (be that the sending device, sending user, recipient device or recipient user) providing they are aware of it.

Matrix and WhatsApp's differing implementations of history sharing functionality results in minor differences in the authentication guarantees they provide. Namely, since Matrix shares the inbound session state required to decrypt

[30] A minor exception, in the case of WhatsApp, is if the primary device is excluded, since this would prevent any sharing of message history. This is an artifact of our model, rather than a meaningful practical difference, since honest WhatsApp clients will automatically include the primary device (as we saw in Chapter 4).

the original ciphertext, the recipient loses authentication guarantees if any of their sibling devices has been compromised or if the original sending session state has been compromised. In other words, we rely on the security of both the sharing device and the original sender and, since any sibling device may share history, there are many devices that could be compromised. In contrast, since WhatsApp clients re-encrypt message history and will only accept history sharing messages from the primary device, they only require their own primary device to be uncorrupted.

Membership control and consistency. In the summary given above, we are careful to describe confidentiality as holding for the "intended recipient users" and for authentication to hold "for a particular device in a particular group". We would expect a natural description of confidentiality and authentication in the context of group messaging to be in terms of that group. That is, we would expect to say that confidentiality holds between the sender and the groups members. Similarly, we might expect that authentication to enforce group membership, such that clients do not accept messages from members outside of the group.

We avoid this language specifically because the protocols we study do not provide these guarantees. Neither protocol provides cryptographic control of the users in a group, nor do they guarantee a consistent view of a group's participants. Indeed, it seems as if the protocols we are studying do not implement "group messaging" at all.

Further still, neither of the implementations we studied seems to enforce the removal of a group member by removing the appropriate inbound session. We posit that this is to allow for out-of-order decryption of messages that are received after a group member has been removed. Indeed, this might seem like the expected behaviour, as long as the client is able to correctly place such messages within the message history. We demonstrated in Section 3.3 that this is not the case for Element, however. We have not examined this part of the WhatsApp client.

Device consistency. WhatsApp's inclusion of in-chat device consistency information is an important component in ensuring that communicating devices have a consistent view of the device composition of one another. In contrast, while Matrix is able to guarantee that any recipient devices were, at some point, verified devices of a particular user, they provide no guarantee that a particular device composition is complete or recent.

CHAPTER 7

Conclusion

We finish with a summary of the work completed in this thesis, which we follow with a retrospective, before concluding the thesis with some remarks.

7.1 A Quick Review

We have studied multi-device group messaging as it is deployed by two widely-used messaging applications: Matrix and WhatsApp. We began with a thorough examination of the documentation, specifications, and implementations of these two applications. These case studies culminated in a detailed description of the two protocols.

During this work, we identified six vulnerabilities in the design and implementation of Matrix. We demonstrated how these vulnerabilities can be exploited to construct five practical attacks against Matrix' flagship client, Element. We, additionally, highlight Matrix and WhatsApp's shared lack of *cryptographic membership control*; a severe limitation to the security guarantees they are able to provide.

Looking to better understand the security guarantees they can provide, we introduce the Device-Oriented Group Messaging (DOGM) model for multi-device group messaging. This model aims to capture the relationships between users, their devices, and the groups they belong to. We introduce two variants. The first captures a common subset of the functionality in Matrix and WhatsApp, while the second additionally captures the device revocation functionality provided by WhatsApp. In doing so, we develop formalisms for three common components for multi-device group messaging that were, to the best of our knowledge, not yet present in the literature.

We then proceed to derive and prove the security of multi-device group messaging in Matrix and WhatsApp, demonstrating that both applications ensure confidentiality and authentication within our formalism, in particular usage contexts and under certain assumptions. We discuss how the results of our

analysis can be interpreted in practice, and the extent to which they resist varying degrees of temporal state compromise. Notably, we find that both Matrix and WhatsApp provide weaker forward secrecy or post-compromise security guarantees than the primitives they compose.

We highlight the additional guarantees that WhatsApp is able to provide through its use of cryptographic *device revocation*. This feature set allows users to regain confidentiality and authentication in the face of complete device compromise.

7.2 A Lengthier Retrospective

Case Studies

In Chapters 3 and 4 we studied multi-device group messaging as it is deployed by Matrix and WhatsApp. In both cases, we combine the public documentation with an inspection of their implementation to produce a detailed description of how they implement multi-device group messaging.

Our case study of Matrix was greatly aided by the availability of source code, since the Matrix Foundation stewards the development of open source libraries alongside the specification. Not only did this provide us with greater certainty that our understanding of the protocol was correct, it also gave us the opportunity to take a broader view of the protocol. Take out-of-band verification, for example. While we did not include this functionality in our modelling, we were still able to provide a detailed formal description of the protocol. Indeed, this broader coverage enabled us to discover a variety of vulnerabilities.

In contrast, our case study of WhatsApp was hampered by a lack of public documentation and source code. The need to reverse-engineer the application necessarily limited the coverage of our case study. Examples of this can be seen throughout Chapter 4. Time constraints meant that we were not able to include the link between the cryptographic core and the user interface within the scope of our analysis. This proved troublesome in a couple of cases. In particular, we were not able to find evidence that the removal of group members was cryptographically enforced. However, we could not state this with certainty without following the code all the way to the user interface.[1] Despite these constraints, we were able to confirm that WhatsApp's implementation does follow the public documentation, for the most part.

On messaging protocols for key exchange. It seems as if Megolm was initially designed under the assumption that each pair of devices shared a singular Olm session with which to perform key exchange. Indeed, the specification of Megolm [Mat22a] suggests that it can achieve PCS through the regular rotation of sessions (inheriting the PCS guarantees of the underlying Olm channels). However, in the case of both Matrix and WhatsApp, we have seen that the PCS guarantees of the underlying pairwise channels do not necessarily translate to the group messaging layer. This is thanks to the session management layer that sits between them.

[1] During the Matrix case study, we found that verification of a sender's identity was only triggered at display time by the user interface.

Additionally, both protocols support out-of-order decryption at both the level of pairwise channels and within group messaging sessions. This is problematic when composing the two protocols because it undermines the intended restrictions of the cached keys of the pairwise channels. In particular, when a message is skipped inside an Olm or Signal two-party channel, the client will save just the per-message key material. Thanks to the key schedule, we can be sure that the compromise of this secret will only affect the security of the given message. However, since this message may be used to distribute an inbound session for group messaging, it enables the adversary to convert an authentication break against a single message to an, indefinitely long, group messaging session. Note that our formal security analysis does not cover out-of-ordering.

In both cases, these weaknesses are a consequence of adding features to the pairwise channels that are important when using them for *conversations*. That is, out-of-order messaging allows for recovery after a minor desynchronisation that does not break the channel, while session management allows for recovery from major desynchronisation events that do break the channel. As such, these features help increase the reliability of the channel used to exchange messages.

However, since these channels are also used to distribute group messaging keys, we see that these additional features can undermine the security provided by the group messaging layer in unexpected ways. Furthermore, the lack of canonical message order brings with it difficulties in modelling and reasoning about these constructions (in addition to the issues they bring to security).

Perhaps a more structured approach to this composition, one that distinguishes between the two-party channels used for key distribution and those used for application messages, would allow for stronger security guarantees?

Unused complexity. Both Matrix and WhatsApp make use of complex structures within primitives that, while they provide meaningful security features in isolation, are undermined when composed with the other components of the system. This can be seen most clearly in the relationship between the Double Ratchet and session management layers just discussed.

While Matrix puts great effort into constructing a key hierarchy between users and their devices, this separation is undermined by the practice of clients automatically sharing such secrets between all verified devices. Further, doing so would enable this key hierarchy to provide cryptographic device revocation (in a similar manner to WhatsApp).

Despite not providing meaningful security guarantees in their broader composition, much of this complex machinery must still be handled within our analysis and proofs. Could a simpler construction provide equivalent security guarantees while allowing for a simpler analysis?

Key reuse. WhatsApp's extensive use of the identity key throughout the protocol proved problematic for our security analysis. This required us to expose the signing oracle in our public key orbit formalism, as well as to introduce a fictional separate key pair for key exchange. Contrast this with Matrix, which derives and signs a fresh signing key for each specific purpose.

This latter approach eased our security analysis considerably, and avoided any unnecessary reliance on unproven assumptions.

(Lack of) Authentication in Matrix. As we discuss, the Matrix standard specifies a backup scheme that does not authenticate the party creating a backup. This single design choice serves to undermine every other effort Matrix has made to provide authentication that we have documented and studied in this thesis.

Formalism

In Chapter 5, we introduce two variants of the device-oriented group messaging model to provide a formal framework within which we may analyse Matrix and WhatsApp. We, additionally, introduce a few formalisms to capture the two protocols' common components: the public key orbit formalism captures device management; the PAIR formalism captures pairwise channels with session management; and the ratcheted symmetric signcryption formalism captures the security of a single group messaging epoch.

A stronger orientation towards devices? The notion of sessions in the DOGM security experiment maps to a single unidirectional channel. It does not provide a natural way to capture the large amount of dynamic state that is shared between the sessions of a single device. Rather than placing this shared state within a device's secret authenticator value, as we do in our construction of MX-DOGM and WA-DOGM, we may be better served by directly modelling device state as a distinct notion.

Such an approach would aid in capturing stronger security guarantees. For example, when a WhatsApp device receives notification that a device has been revoked, they are able to enact the requisite changes across all of their active sessions. In the DOGM formalism, however, capturing this would require providing the executing session with a global view of all sessions the device is participating in; something that is not available if the unit of execution is a single unidirectional session. As such, our modelling relies on the same updated multi-device state being applied repeatedly to each and every DOGM session. Further, the adversary may choose not to apply these updates to every one of a device's DOGM sessions at the same time. Doing so, would allow the adversary to induce weaker consistency guarantees than Matrix and WhatsApp achieve in practice. This can be seen most clearly in our description of WhatsApp in Figure 4.13, where the shared device state ensures that all of a device's sessions share the same view of each communicating partner's device list. An analogous issue can be seen for group membership. Thus, our formalism is not able to capture some of the implicit consistency guarantees that the shared device state, and the protocols we study, are able to provide.

This could, additionally, improve the granularity of the post-compromise security guarantees we are able to capture. As it stands, the DOGM security experiment provides a singular $\mathcal{O}$-CorruptDevice oracle that compromises of all of a device's state at once. In such an improved model, the security experiment would be able to offer separate oracle queries for the compromise of long-term, medium-term

and session state. This would enable us to prove the security of these protocols under more nuanced state compromise scenarios.

Alternative approaches? Although they differ in a number of places, we have seen that Matrix and WhatsApp take broadly similar approaches to implementing multi-device group messaging.

Of course, there exist a variety of alternative approaches to multi-device group messaging. Both the multi-party extension to OTR, *mpOTR* [GUVC09], and Keybase [JKS24, Key22] utilise existing pairwise channels to derive a shared symmetric key. A similar approach is taken by MLS, whereby a single shared symmetric key is derived per-epoch. In all such cases, while the manner in which this symmetric key is derived differs, these protocols have senders share a single symmetric key (with individual signing keys). Alternatively WhatsApp's previous approach to multi-device group messaging, or the design proposed for Signal in [CDDF20] are user-oriented. That is, messaging occurs primarily between users, who then proceed to synchronise messages between their respective devices in a separate protocol.

Can the DOGM model be meaningfully applied to these alternative approaches?

Security Analysis

Modular vs. monolithic analysis. Our security analysis of Matrix treats it as one monolithic whole. This resulted in a security analysis that was complex and intricate, motivating an alternative approach to our analysis of WhatsApp. In this case, we made use of the component formalisms developed in Section 5.4 to analyse the device management, pairwise channels and group messaging stages separately, before considering their composition within the broader protocol.

While such an approach did help to manage the complexity of our analysis, it increased the quantity of this analysis considerably. We leave the assessment of which approach provided the most clarity to the reader themselves.

Tighter bounds. It is likely that further work restructuring the proofs for Theorems 6.4 and 6.18 would result in a tighter advantage bound and, thus, a more accurate representation of the security these protocols provide.

This can be seen most clearly in our security analysis of Matrix. In the authentication case, we handle the injection of adversarially-controlled Megolm sessions separately from the forgery of a message within an honestly distributed Megolm session. Splitting these two cases is advantageous because the set of guesses the challenger makes in each case is largely independent from those made in the other. This results in the advantage term for the unforgeability of a single Megolm stage with a multiplication factor of '$n_u \cdot n_d \cdot n_i \cdot n_s$'. In contrast, we do not apply a similar strategy in the confidentiality case. This results in the analogous advantage term for the confidentiality of a single Megolm stage growing at far greater rate; something it inherits from the compounding guesses required to secure the pairwise channels.

This may also be seen in our development of the PAIR-SEC security notion, where we capture confidentiality and authentication jointly. Later, in our

proof of security for WhatsApp, we proceed to handle the confidentiality and authentication of pairwise channels separately. This results in an unnecessary factor of two in the final advantage term. Since our proof of security for WhatsApp's pairwise channels already considers these two cases separately, providing distinct security notions for confidentiality and authentication would have avoided this additional factor.

Precise predicates. The security predicates we derive in our analyses of Matrix and WhatsApp are overly restrictive: they disallow a variety of situations where the respective protocol may in fact offer security. Doing so helps to simplify our analysis and, we hope, ease its interpretation. It would, nonetheless, be valuable to derive and prove a more precise set of security predicates.

On forward secrecy and message history. In order to capture forward secrecy in our analysis of WhatsApp, we must remove the plaintext message history from session state when it is compromised. At first glance, this seems inconsistent. After all, history sharing requires access to plaintext messages in order to function. This is, nonetheless, necessary to prove that the key exchange layer of WhatsApp does provide forward secrecy.

Since Matrix clients implement history sharing by redistributing the original key (rather than messages), our instantiation of Matrix in the DOGM model does not store plaintext messages after decryption. However, since the original key material is stored in the session state, and we do not exclude this key material from compromise, we find that the key exchange layer does not provide forward secrecy. This approach additionally reduces the authentication guarantees of the protocol, as discussed in Section 6.3.

While we do derive differing guarantees for the forward secrecy provided by WhatsApp and Matrix, it is worth asking whether there is a meaningful distinction. Taking this further, what guarantees do we hope to get from forward secrecy in these contexts?

We should not expect that compromise of a device that currently has access to a message to be able to protect the contents of that message. However, forward secrecy does enable the secure deletion of such messages: consider the case where a message has been locally deleted from a device. If an adversary with access to its original ciphertext compromises a Matrix client after deletion, they may decrypt that ciphertext using the key material. In WhatsApp, however, neither the plaintext message or the key material to decrypt the ciphertext would be present on the device.

In this sense, our security analysis *does* capture a meaningful understanding of forward secrecy, in as much as it enables distinguishing between the guarantees provided by Matrix and WhatsApp. If Matrix clients were to enact the deletion of messages by additionally deleting the requisite key material, they would achieve the same level of forward secrecy as WhatsApp does, both in our model and, more importantly, in practice.

Missing pieces. Our results rely on a number of unproven assumptions (in addition to those fundamental to the security of our primitives, such as the requirement for the Gap Diffie-Hellman to hold for X25519).

Our security analysis of Matrix relies on the Megolm ratchet instantiating a secure FFPRG. We posit that such a proof would, in turn, rely on HKDF instantiating a secure PRF. The use of HKDF as a PRF is commonplace amongst the protocols we study: whenever HKDF is used to derive per-message key material from a symmetric ratchet. We posit that PRF security of HKDF follows relatively closely from recent work analyzing the security of HMAC [BBGS23].

WhatsApp, along with Signal, relies on the security of X25519 keys when used for both key exchange and digital signatures, via XEdDSA. To the best of our knowledge, no proof of joint security exists for these two primitives. While our security analysis of WhatsApp separates its usage into two separate key pairs, one for signatures and the other for key exchange, even this requires the (unproven) assumption that XEdDSA instantiates a secure signature scheme.

7.3 Concluding Remarks

While both Matrix and WhatsApp fall short of the post-compromise security guarantees expected in the cryptographic literature, this is only partially due to avoidable design flaws: it is partially also due to a deliberate design trade-off. That is, some of the limitations on FS/PCS guarantees we establish in this work match those intended by the protocol designers, who accept this behaviour in favour of utility in a chat context: making older messages available across devices and enabling messaging to recover after desynchronisation errors.

Whether this trade-off is correct is a question that falls outside the expertise of cryptography in a narrow sense. Given that many deployed protocols lack meaningful PCS guarantees and that prior work established that these guarantees do not match up well with some higher-risk settings [ABJM21], we consider the question of what PCS guarantees *should* be targeted in design an exciting area for future multidisciplinary work. Here, we highlight WhatsApp's approach to tackle *detected* compromises via device revocation, which does not map to the PCS notions in the literature but seems to provide meaningful guarantees for some important settings.

The difficulty of establishing a consistent global ordering of messages within such protocols we study 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 MLS [BBR$^+$23] 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. On this note, the lack of canonical ordering also poses a challenge for interpreting the security guarantees of our protocols. A formalism that restricts its security predicates to those events that are observable to users could provide interesting insights into how the security guarantees of such protocols surface to end users.

We conclude by stressing that the lack of user control over group membership is a critical limitation of the protocols we study in this thesis. We note that the provision of this feature set in MLS seems to indicate that this is not an insurmountable problem.

APPENDIX A

Constants

Throughout this thesis we make use of placeholder constants to represent real-world values that were otherwise too unwieldy to include directly within our pseudocode descriptions. The tables in this section provide a mapping from each placeholder to its concrete value.

A.1 Constants in Matrix

Constant	Identifier	Description
`m.olm.v1.curve25519-aes-sha2`	`OLM`	Value for the '`algorithm`' field in `m.room.encrypted` messages to indicate the message was encrypted with Olm.
`m.megolm.v1.aes-sha2`	`MG`	Value for the '`algorithm`' field in `m.room.encrypted` messages to indicate the message encrypted with Megolm.
`master`	`MPK`	Usage indicator in a master cross-signing key's signature over itself.
`self_signing`	`SPK`	Usage indicator in a master cross-signing key's signature over its self-signing key.
`user_signing`	`UPK`	Usage indicator in a master cross-signing key's signature over its user-signing key.
`device_keys`	`DEV`	Usage indicator in a self-signing key's signature over a device's keys (*dpk* and *ipk*).
`ed25519:<device-id>`	`DPK`	Identifier for the device key in a device key's signature over itself and the Olm identity key.
`curve25519:<device-id>`	`IPK`	Identifier for the Olm identity key in a device key's signature over the device keys.
`one_time_keys`	`EPK`	Usage indicator in a device key's signature over an ephemeral key.
`fallback_keys`	`FPK`	Usage indicator in a device key's signature over a fallback key.
`OLM_ROOT`	`OLM-RT`	Domain separator for initial derivation of the Olm root key.
`OLM_RATCHET`	`OLM-RCH`	Domain separator for ratcheting derivations of the new Olm root and chain keys.
`0x02`	`0x02`	Domain separator for the ratcheting derivation of new chain keys.
`0x01`	`0x01`	Domain separator for the derivation of per-message key material from a chain key.
`OLM_KEYS`	`OLM-KDF`	Domain separator for the stretching of per-message key material into the keys required by the AEAD scheme (in Olm).
`0x03`	`OLM-VER`	Version indicator within Olm ciphertexts.
`0x01`	`MG-SESS`	Version and format indicator for Megolm sessions.
`0x02`	`KS-SESS`	Version and format indicator for Megolm sessions in session export format.
`MEGOLM_KEYS`	`MG-KDF`	Domain separator for the stretching of per-message key material into the keys required by the AEAD scheme (in Megolm).
`request`	`REQ`	Value of the '`action`' field in `m.room_key_request` messages when requesting a Megolm session.
`request_cancellation`	`CNL`	Value of the '`action`' field in `m.room_key_request` messages a request has already been fulfilled.
`MATRIX_KEY_VERIFICATION_SAS`	`SAS-VERIF`	Domain separator in the KDF call used to calculate the short authenticated string during out-of-band verfication.

Table A.1. Constants used in Matrix and the identifiers we refer to them by.

Message Type	Identifier	Description
`m.room.message`	`PTXT`	Chat messages sent and received by users.
`m.room.encrypted`	`CTXT`	Container for encrypted messages.
`m.room_key`	`MG-KEY`	Container for an inbound Megolm session.
`m.room_key_request`	`KS-REQ`	Request for an inbound Megolm session (as part of the Key Request protocol).
`m.forwarded_room_key`	`KS-SND`	Container for an inbound Megolm session (as part of the Key Request protocol).
`m.room_key.witheld`	`KS-WTH`	Message signifying that the sending party has rejected the recipient party's key request.
`m.secret.request`	`4S-REQ`	Message requesting a secret (as part of the SSSS protocol).
`m.secret.send`	`4S-SND`	Message sharing a secret (as part of the SSSS protocol).

Table A.2. Message types in Matrix and the identifiers we refer to them by.

Server Endpoint	Identifier	Description
`/query/keys`	`CS-USR`	Server distribution of user cross-signing keys (containing the `user_signing_keys` and `self_signing_keys` fields) [Mata].
`/query/keys`	`CS-DEV`	Server distribution of device keys (containing `device_keys` field) [Mata].

Table A.3. Server responses received by Matrix clients and the identifiers we refer to them by.

A.2 Constants in WhatsApp

Constant	Description
`0x02`	Domain separator for the ratcheting derivation of new chain keys (in pairwise channels as well as groups).
`0x01`	Domain separator for the derivation of per-message key material from a chain key (in pairwise channels as well as groups).
`0x05`	Usage indicator in an identity key's signature over signed pre-key.
`0x0600`	Usage indicator for a primary device's signature over the linking metadata linking a new companion device.
`0x0601`	Usage indicator for the companion device's signature over the linking metadata linking itself with a primary device.
`0x0602`	Usage indicator for device list signatures.
`WhisperGroup`	Domain separator for the stretching of per-message key material when encrypting or decrypting group messages.
`WhatsApp History Keys`	Domain separator for key derivation when encrypting history sharing attachments.
`WhatsApp Image Keys`	Domain separator for key derivation when encrypting image attachments.

Table A.4. Constants used in WhatsApp.

APPENDIX B

Supporting Material for Security Proofs

We include some supporting material for the security proofs in this work.

B.1 Signal's Freshness Predicate

We give a translation of the two-party Signal protocol's freshness predicate, as derived in [CCD+20], in the syntax of PAIR channels and our PAIR-SEC security experiment.

$$\underline{\mathsf{Signal.FRESH}(i,j,sid,z)} \;\coloneqq\; \mathsf{Signal.CLEAN}(type,i,j,sid,z) \;\wedge \mathsf{Signal.VALID}(i,j,sid,z)$$

$$\underline{\mathsf{Signal.VALID}(i,j,sid,z)} \;\coloneqq$$
$$\quad {}^{*}\mathsf{PAIR.FindSession}(sk_i.ssts[ipk_j],sid)[0].status[z] = \texttt{accept} \wedge \mathsf{Signal.CLEAN}(\texttt{state},i,j,sid,z)$$

$$\underline{\mathsf{Signal.CLEAN}(\texttt{triple},i,j,sid,z \overset{is}{=} [0])} \;\coloneqq$$
$$\quad \mathsf{Signal.CLEAN}(\texttt{LM},i,j,sid) \;\vee \mathsf{Signal.CLEAN}(\texttt{EL},i,j) \;\vee \mathsf{Signal.CLEAN}(\texttt{EM},i,j)$$

$$\underline{\mathsf{Signal.CLEAN}(\texttt{triple+DHE},i,j,sid,z \overset{is}{=} [0])} \;\coloneqq$$
$$\quad \mathsf{Signal.CLEAN}(\texttt{triple},i,j,sid,z) \;\wedge \mathsf{Signal.CLEAN}(\texttt{EE},i,j,sid,0,0)$$

$$\underline{\mathsf{Signal.CLEAN}(\texttt{asym-ri},i,j,sid,z \overset{is}{=} [\textbf{asym-ri}:1])} \;\coloneqq$$
$$\quad \mathsf{Signal.CLEAN}(\texttt{EE},i,j,0,[\textbf{asym-ri}:1])$$
$$\qquad \vee (\mathsf{Signal.CLEAN}(\texttt{state},i,j,sid,z) \;\wedge \mathsf{Signal.CLEAN}(\texttt{state},i,j,sid,[0]))$$

$$\underline{\mathsf{Signal.CLEAN}(\texttt{asym-ri},i,j,sid,z \overset{is}{=} [\textbf{asym-ri}:x \geq 2])} \;\coloneqq$$
$$\quad \mathsf{Signal.CLEAN}(\texttt{EE},i,j,[\textbf{asym-ri}:x],[\textbf{asym-ir}:x-1]) \;\vee$$
$$\qquad (\mathsf{Signal.CLEAN}(\texttt{state},i,j,sid,z) \;\wedge \mathsf{Signal.CLEAN}(\texttt{asym-ir},i,j,sid,[\textbf{asym-ir}:x-1]))$$

$\underline{\mathsf{Signal.CLEAN}(\texttt{asym-ir}, i, j, sid, z \overset{is}{=} [\mathbf{asym\text{-}ir} : x])} \;\coloneqq$
$\quad \mathsf{Signal.CLEAN}(\texttt{EE}, i, j, [\mathbf{asym\text{-}ir} : x], [\mathbf{asym\text{-}ri} : x])$
$\quad\quad \vee (\mathsf{Signal.CLEAN}(\texttt{state}, i, j, sid, z) \;\wedge \mathsf{Signal.CLEAN}(\texttt{asym-ri}, i, j, sid, [\mathbf{asym\text{-}ri} : x]))$

$\underline{\mathsf{Signal.CLEAN}(\texttt{sym-ir}, i, j, sid, z \overset{is}{=} [\mathbf{sym\text{-}ir} : 0, 1])} \;\coloneqq$
$\quad \mathsf{Signal.CLEAN}(\texttt{state}, i, j, sid, z) \;\wedge \mathsf{Signal.CLEAN}(\cdot, i, j, sid, [0])$

$\underline{\mathsf{Signal.CLEAN}(\texttt{sym-ir}, i, j, sid, z \overset{is}{=} [\mathbf{sym\text{-}ir} : x \geq 1, y = 1])} \;\coloneqq$
$\quad \mathsf{Signal.CLEAN}(\texttt{state}, i, j, sid, z) \;\wedge \mathsf{Signal.CLEAN}(\texttt{asym-ir}, i, j, sid, [\mathbf{asym\text{-}ir} : x])$

$\underline{\mathsf{Signal.CLEAN}(\texttt{sym-ir}, i, j, sid, z \overset{is}{=} [\mathbf{sym\text{-}ir} : x \geq 0, y \geq 2])} \;\coloneqq$
$\quad \mathsf{Signal.CLEAN}(\texttt{state}, i, j, sid, z) \;\wedge \mathsf{Signal.CLEAN}(\texttt{sym-ir}, i, j, sid, [\mathbf{sym\text{-}ir} : x, y - 1])$

$\underline{\mathsf{Signal.CLEAN}(\texttt{sym-ri}, i, j, sid, z \overset{is}{=} [\mathbf{sym\text{-}ri} : x \geq 1, y = 1])} \;\coloneqq$
$\quad \mathsf{Signal.CLEAN}(\texttt{state}, i, j, sid, z) \;\wedge \mathsf{Signal.CLEAN}(\texttt{asym-ri}, i, j, sid, [\mathbf{asym\text{-}ri} : x])$

$\underline{\mathsf{Signal.CLEAN}(\texttt{sym-ri}, i, j, sid, z \overset{is}{=} [\mathbf{sym\text{-}ri} : x \geq 0, y \geq 2])} \;\coloneqq$
$\quad \mathsf{Signal.CLEAN}(\texttt{state}, i, j, sid, z) \;\wedge \mathsf{Signal.CLEAN}(\texttt{sym-ri}, i, j, sid, [\mathbf{sym\text{-}ri} : x, y - 1])$

$\underline{\mathsf{Signal.CLEAN}(\texttt{LM}, i, j, sid)}$

1 : $\cdot, sst \leftarrow {}^*\mathsf{PAIR.FindSession}(sk_i.ssts[ipk_j], sid)$
2 : **if** $sst.role = \texttt{init}$: **return** $(\texttt{corr-ident}, i, \cdot) \notin \mathbf{L} \;\wedge (\texttt{corr-share}, j, \cdot) \notin \mathbf{L}$
3 : **elseif** $sst.role = \texttt{resp}$: **return** $(\texttt{corr-share}, i, \cdot) \notin \mathbf{L} \;\wedge (\texttt{corr-ident}, j, \cdot) \notin \mathbf{L}$

$\underline{\mathsf{Signal.CLEAN}(\texttt{EL}, i, j, sid, z \overset{is}{=} [0])}$

1 : $\cdot, sst \leftarrow {}^*\mathsf{PAIR.FindSession}(sk_i.ssts[ipk_j], sid)$
2 : **if** $sst.role = \texttt{init}$: **return** $(\texttt{corr-sess}, i, j, sid, [0], \cdot) \notin \mathbf{L} \;\wedge (\texttt{corr-ident}, i, \cdot) \notin \mathbf{L}$
3 : **elseif** $sst.role = \texttt{resp}$: **return** $\mathsf{Signal.CLEAN}(\texttt{peerE}, i, j, sid, [0]) \;\wedge (\texttt{corr-ident}, i, \cdot) \notin \mathbf{L}$

$\underline{\mathsf{Signal.CLEAN}(\texttt{EM}, i, j, sid, z \overset{is}{=} [0])}$

1 : $\cdot, sst \leftarrow {}^*\mathsf{PAIR.FindSession}(sk_i.ssts[ipk_j], sid)$
2 : **if** $sst.role = \texttt{init}$: **return** $(\texttt{corr-sess}, i, j, sid, [0], \cdot) \notin \mathbf{L} \;\wedge (\texttt{corr-share}, i, \cdot) \notin \mathbf{L}$
3 : **elseif** $sst.role = \texttt{resp}$: **return** $\mathsf{Signal.CLEAN}(\texttt{peerE}, i, j, sid, [0]) \;\wedge (\texttt{corr-share}, i, \cdot) \notin \mathbf{L}$

$\underline{\mathsf{Signal.CLEAN}(\texttt{EE}, i, j, sid, z, z')}$

1 : $\cdot, sst \leftarrow {}^*\mathsf{PAIR.FindSession}(sk_i.ssts[ipk_j], sid)$
2 : **if** $sst.role = \texttt{init}$: **return** $(\texttt{corr-session}, i, j, sid, z, \cdot) \notin \mathbf{L} \;\wedge \mathsf{Signal.CLEAN}(\texttt{peerE}, i, j, sid, z')$
3 : **elseif** $sst.role = \texttt{resp}$: **return** $\mathsf{Signal.CLEAN}(\texttt{peerE}, i, j, sid, z) \;\wedge (\texttt{corr-session}, i, j, sid, z', \cdot) \notin \mathbf{L}$

$\underline{\mathsf{Signal.CLEAN}(\texttt{peerE}, i, j, sid, z \overset{is}{=} [t : x, y])}$

1 : $\cdot, sst \leftarrow {}^*\mathsf{PAIR.FindSession}(sk_j.ssts[ipk_i], sid)$
2 : **if** $sst = (\varnothing, \varnothing)$: **return false**
3 : **elseif** $x = 0 \;\wedge sst.role = \texttt{init}$: **return** $(\texttt{dec}, j, i, sid, \cdot)$ **precedes** $(\texttt{corr-share}, j, \cdot)$ **in** $\mathbf{L}$
4 : **else** : **return** $(\texttt{corr-session}, j, i, sid, z, \cdot) \notin \mathbf{L}$
5 : $\quad\quad \wedge \exists\, (\texttt{enc}, j, i, sid, z, (c_{kex}, \cdot), \cdot) \in \mathbf{L},\; (\texttt{dec}, i, j, sid, z, (c_{kex'}, \cdot), \cdot) \in \mathbf{L} \;:\; c_{kex} = c_{kex'}$

$\underline{\mathsf{Signal.CLEAN}(\texttt{state}, i, j, sid, z)} \;\coloneqq\; (\texttt{corr-session}, i, j, sid, z, \cdot) \notin \mathbf{L} \;\wedge (\texttt{corr-session}, j, i, sid, z, \cdot) \notin \mathbf{L}$

B.2 Security Reductions in the Proof for WhatsApp's Pairwise Channels

$\mathcal{B}^{\mathsf{Send},\mathsf{Test},\mathsf{Rev}^*}_{\mathsf{Signal},\mathcal{A}}(pubinfo)$

$b \leftarrow\$ \{0,1\}$; $win \leftarrow 0$; $\mathbf{L} \leftarrow [\,]$
$ssts \leftarrow \mathsf{Map}\{\}$; $\mathbf{MK} \leftarrow \mathsf{Map}\{\}$; $used\text{-}epks \leftarrow \mathsf{Map}\{\}$
$(ipk_0, \ldots, ipk_{n_u-1}), (spks_0, \ldots, spks_{n_u-1}), (epks_0, \ldots, epks_{n_u-1}) \leftarrow pubinfo$
$b' \leftarrow \mathcal{A}^{\mathcal{O}\text{-}\mathsf{Enc},\mathcal{O}\text{-}\mathsf{Dec},\mathcal{O}\text{-}\mathsf{Corrupt}^*}(\{(ipk_0, spks_0), \ldots, (ipk_{n_u-1}, spks_{n_u-1})\}, \{epks_0, \ldots, epks_{n_u-1}\})$
return $b = b' \wedge \mathsf{PAIR.CONF}(\mathbf{L})$

$\mathcal{O}\text{-}\mathsf{Enc}(i, j, info, sid, m_0, m_1)$

require $\mathsf{len}(m_0) = \mathsf{len}(m_1)$
if $sid = \varnothing$: $sid \leftarrow info[0]$
$s \leftarrow ssts[i, j, sid]$
if $s = \varnothing$:
  $s \leftarrow \mathsf{len}(ssts[i, \cdot])$
  $\mathsf{Send}(i, s, (ipk_j, 0, \texttt{init}))$
  $c_{kex} \leftarrow \mathsf{Send}(i, s, sid)$
  $ssts[i, j, sid] \leftarrow s$
else:
  $c_{kex} \leftarrow \mathsf{Send}(i, s, \varnothing)$
if $(i, j, sid, stage) \notin \mathbf{MK}$:
  $\mathbf{MK}[i, j, sid, stage] \leftarrow \mathsf{Test}(i, s, stage)$
$mk \leftarrow \mathbf{MK}[i, j, sid, stage]$
require $mk \neq \bot$
$c_{msg} \leftarrow \mathsf{WA\text{-}AEAD.Enc}(mk, c_{kex}, m_b)$
$c \leftarrow (c_{kex}, c_{msg})$
$\mathbf{L} \leftarrow\!\!\| (\texttt{enc}, i, j, sid, m_0, m_1, c)$
return sid, c

$\mathcal{O}\text{-}\mathsf{Dec}(i, j, info, sid, c)$

$c_{kex}, c_{msg} \leftarrow c$
if $sid = \varnothing$: $sid \leftarrow c_{kex}.epk_{resp}$
$s \leftarrow ssts[i, j, sid]$
if $s = \varnothing$:
  require sid **in** $epks_i \setminus used\text{-}epks[i]$
  $used\text{-}epks[i] \leftarrow\!\cup \{sid\}$
  $s \leftarrow \mathsf{len}(ssts[i, \cdot])$
  $ssts[i, j, sid] \leftarrow s$
  $\mathsf{Send}(i, s, (ipk_j, 0, sid, \texttt{resp}))$
  $\mathsf{Send}(i, s, c_{kex})$
else:
  $\mathsf{Send}(i, s, c_{kex})$
if $(j, i, sid, stage) \notin \mathbf{MK}$:
  $\mathbf{MK}[j, i, sid, stage] \leftarrow \mathsf{Test}(i, s, stage)$
$mk \leftarrow \mathbf{MK}[j, i, sid, stage]$
require $mk \neq \bot$
$m \leftarrow \mathsf{WA\text{-}AEAD.Dec}(mk, c_{kex}, c_{msg})$
if $m = \bot$: **return** $\bot$
$replay \leftarrow (\texttt{dec}, i, j, sid, c, \cdot) \in \mathbf{L}$
$forgery \leftarrow (\texttt{enc}, j, i, sid, \cdot, \cdot, c) \notin \mathbf{L}$
$win \leftarrow replay \vee (forgery \wedge \mathsf{PAIR.AUTH}(\mathbf{L}))$
$\mathbf{L} \leftarrow\!\!\| (\texttt{dec}, i, j, sid, c, m)$
if $c \in {}^*\mathsf{Challenges}(\mathbf{L})$: **return** $\bot$
return sid, m

$\mathcal{O}\text{-}\mathsf{CorruptIdentity}(i)$

require $0 \le i < n_u$
$corr \leftarrow \mathsf{RevLongTermKey}(i)$
$\mathbf{L} \leftarrow\!\!\| (\texttt{corr-ident}, i, corr)$
return $corr$

$\mathcal{O}\text{-}\mathsf{CorruptShared}(i)$

require $0 \le i < n_u$
$esks \leftarrow [\mathsf{RevEphemKey}(i, e)$
  for (e, epk) **in** $epks_i$
  if $epk \notin used\text{-}epks]$
$ssk \leftarrow \mathsf{RevMedTermKey}(i, 0)$
$corr \leftarrow ssk, esks$
$\mathbf{L} \leftarrow\!\!\| (\texttt{corr-share}, i, corr)$
return $corr$

$\mathcal{O}\text{-}\mathsf{CorruptSession}(i, j, sid)$

require $0 \le i, j < n_u$
$s \leftarrow ssts[i, j, sid]$
$sst \leftarrow \mathsf{RevState}(i, s, s.stage)$
$rchsk \leftarrow \mathsf{RevRand}(i, s, s.stage)$
$corr \leftarrow (sst, rchsk)$
$\mathbf{L} \leftarrow\!\!\| (\texttt{corr-sess}, i, j, sid, corr)$
return $corr$

Figure B.1. Security reduction demonstrating how we convert an adversary, $\mathcal{A}$, against the PAIR-SEC security of the WA-PAIR scheme into an adversary, $\mathcal{B}^{\mathsf{Send},\mathsf{Test},\mathsf{Rev}^*}_{\mathsf{Signal},\mathcal{A}}(pubinfo)$, against the key indistinguishability of the underlying Signal channels. It is used to reason about the changes introduced in **Game 1** of the proof of security for WhatsApp's pairwise channels (see Theorem 6.9).

$\mathcal{B}^{\text{EUF-CMA},\mathcal{O}\text{-Enc}'}_{\text{WA-AEAD},\mathcal{A}}()$

$forged\text{-}msg \leftarrow \varnothing$; $g_i \leftarrow\$ \{0 \ldots n_u - 1\}$; $g_j \leftarrow\$ \{0 \ldots n_u - 1\}$; $g_s \leftarrow\$ \{0 \ldots n_i - 1\}$; $g_z \leftarrow\$ \{0 \ldots n_m - 1\}$
$\mathbf{S} \leftarrow \mathsf{Map}\{\}$; $\mathbf{Z} \leftarrow \mathsf{Map}\{\cdot : 0\}$; $win \leftarrow 0$; $\mathbf{L} \leftarrow [\,]$; $ssts \leftarrow \mathsf{Map}\{\}$; $\mathbf{MK} \leftarrow \mathsf{Map}\{\}$
for $i = 0, 1, 2, \ldots, n_u - 1$:
  $ipk_i, isk_i \leftarrow\$ \mathsf{Signal.KeyGen}()$; $spk_i, ssk_i \leftarrow\$ \mathsf{Signal.MedTermKeyGen}()$
  for $e = 0, 1, 2, \ldots, n_e - 1$: $epk_e, esk_e \leftarrow\$ \mathsf{Signal.EphemKeyGen}()$
  $esks_i \leftarrow \{esk_e : 0 \le e < n_e\}$; $epks_i \leftarrow \{epk_e : 0 \le e < n_e\}$
$b' \leftarrow \mathcal{A}^{\mathcal{O}\text{-Enc},\mathcal{O}\text{-Dec},\mathcal{O}\text{-Corrupt}^*}(\{(ipk_0, spk_0), (ipk_1, spk_1), \ldots, (ipk_{n_u-1}, spk_{n_u-1})\}, \{epks_0, epks_1, \ldots, epks_{n_u-1}\})$
return $forged\text{-}msg$

$\mathcal{O}\text{-Enc}(i, j, info, sid, m_0, m_1)$

require $m_0 = m_1$
if $sid = \varnothing$:
  $epk_j \leftarrow info[0]$; $sid \leftarrow epk_j$
$sst \leftarrow ssts[i, j, sid]$
if $sst = \varnothing$:
  $sst, \cdot \leftarrow\$ \mathsf{Signal.Activate}(isk_i, ssk_i, \texttt{init}, ipk_j)$
  $sst, c_{kex} \leftarrow\$ \mathsf{Signal.Run}(isk_i, ssk_i, sst, (spk_j, sid))$
  $\mathbf{S}[i, j, sid] \leftarrow \mathsf{len}(\mathbf{S}[i, j, \cdot])$
else:
  $sst, c_{kex} \leftarrow\$ \mathsf{Signal.Run}(isk_i, ssk_i, sst, \varnothing)$
require $sst.status[sst.stage] = \texttt{accept}$
if $(i, j, sid, sst.stage) \notin \mathbf{MK}$:
  $\mathbf{MK}[i, j, sid, sst.stage] \leftarrow\$ \mathcal{K}$
$mk \leftarrow \mathbf{MK}[i, j, sid, sst.stage]$
$\mathbf{Z}[i, j, sid] \leftarrow \mathbf{Z}[i, j, sid] + 1$
if $(i, j, \mathbf{S}[i, j, sid], \mathbf{Z}[i, j, sid]) = (g_i, g_j, g_s, g_z)$:
  $c_{msg} \leftarrow \mathcal{O}\text{-Enc}'[c_{kex}, m_0]$
else:
  $c_{msg} \leftarrow \mathsf{WA\text{-}AEAD.Enc}(mk, c_{kex}, m_0)$
$c \leftarrow (c_{kex}, c_{msg})$
$ssts[i, j, sid] \leftarrow sst$
$\mathbf{L} \leftarrow\!\!\| (\texttt{enc}, i, j, sid, m_0, m_1, c)$
return sid, c

$\mathcal{O}\text{-Dec}(i, j, info, sid, c)$

$c_{kex}, c_{msg} \leftarrow c$
if $sid = \varnothing$: $epk_i \leftarrow c_{kex}.epk_{resp}$; $sid \leftarrow epk_i$
$sst \leftarrow ssts[i, j, sid]$
if $sst = \varnothing$:
  $[esk] \leftarrow [esk' \textbf{ in } esks_i \textbf{ if } epk_i = \mathsf{PK}(esk')]$
  $sst, \cdot \leftarrow\$ \mathsf{Signal.Activate}(isk_i, ssk_i, \texttt{resp}, ipk_j, esk)$
  $sst, \cdot \leftarrow\$ \mathsf{Signal.Run}(isk_i, ssk_i, sst, c_{kex})$
  $esks \leftarrow [esk' \textbf{ in } esks \textbf{ if } esk \neq esk']$
  $\mathbf{S}[i, j, sid] \leftarrow \mathsf{len}(\mathbf{S}[i, j, \cdot])$
else: $sst, \cdot \leftarrow\$ \mathsf{Signal.Run}(isk_i, ssk_i, sst, c_{kex})$
require $sst.status[sst.stage] = \texttt{accept}$
if $(j, i, sid, sst.stage) \notin \mathbf{MK}$:
  $\mathbf{MK}[j, i, sid, sst.stage] \leftarrow\$ \mathcal{K}$
$mk \leftarrow \mathbf{MK}[j, i, sid, sst.stage]$
$\mathbf{Z}[i, j, sid] \leftarrow \mathbf{Z}[i, j, sid] + 1$
$m \leftarrow \mathsf{WA\text{-}AEAD.Dec}(mk, c_{kex}, c_{msg})$
if $m = \perp$: **return** $\perp$
$ssts[i, j, sid] \leftarrow sst$
$replay \leftarrow (\texttt{dec}, i, j, sid, c, \cdot) \in \mathbf{L}$
$forgery \leftarrow (\texttt{enc}, j, i, sid, \cdot, \cdot, c) \notin \mathbf{L}$
$guess \leftarrow (j, i, \mathbf{S}[j, i, sid], \mathbf{Z}[i, j, sid]) = (g_i, g_j, g_s, g_z)$
if $replay \lor (forgery \land \mathsf{PAIR.AUTH}(\mathbf{L}))$:
  if $guess$: $forged\text{-}msg \leftarrow c$
  else: **abort**
$\mathbf{L} \leftarrow\!\!\| (\texttt{dec}, i, j, sid, c, m)$
return sid, m

$\mathcal{O}\text{-CorruptIdentity}(i)$

require $0 \le i < n_u$
$corr \leftarrow isk_i$
$\mathbf{L} \leftarrow\!\!\| (\texttt{corr-ident}, i, corr)$
return $corr$

$\mathcal{O}\text{-CorruptShared}(i)$

require $0 \le i < n_u$
$corr \leftarrow (ssk_i, esks_i)$
$\mathbf{L} \leftarrow\!\!\| (\texttt{corr-share}, i, corr)$
return $corr$

$\mathcal{O}\text{-CorruptSession}(i, j, sid)$

require $0 \le i, j < n_u$
$corr \leftarrow ssts[i, j, sid]$
$\mathbf{L} \leftarrow\!\!\| (\texttt{corr-sess}, i, j, sid, corr)$
return $corr$

Figure B.2. Security reduction demonstrating how we construct an adversary, $\mathcal{B}^{\text{EUF-CMA}}_{\text{WA-AEAD},\mathcal{A}}$, against the EUF-CMA security of the WA-AEAD AEAD scheme using an existing adversary, $\mathcal{A}$, against the PAIR-SEC security of the WA-PAIR scheme. We use this security reduction to bound the changes introduced in each hybrid of **Game C1** in the proof of security for WhatsApp's pairwise channels (see Theorem 6.9).

$\mathcal{B}^{\mathsf{IND\$\text{-}CPA},\mathcal{O}\text{-}\mathsf{Enc}'}_{\mathsf{WA\text{-}AEAD},\mathcal{A}}()$

$b \leftarrow\$ \{0,1\}$; $\mathbf{L} \leftarrow [\,]$; $ssts \leftarrow \mathsf{Map}\{\}$; $\mathbf{MK} \leftarrow \mathsf{Map}\{\}$
for $i = 0, 1, 2, \ldots, n_u - 1$:
  $ipk_i, isk_i \leftarrow\$ \mathsf{Signal.KeyGen}()$; $spk_i, ssk_i \leftarrow\$ \mathsf{Signal.MedTermKeyGen}()$
  for $e = 0, 1, 2, \ldots, n_e - 1$: $epk_e, esk_e \leftarrow\$ \mathsf{Signal.EphemKeyGen}()$
  $esks_i \leftarrow \{esk_e : 0 \le e < n_e\}$; $epks_i \leftarrow \{epk_e : 0 \le e < n_e\}$
$b' \leftarrow \mathcal{A}^{\mathcal{O}\text{-}\mathsf{Enc},\mathcal{O}\text{-}\mathsf{Dec},\mathcal{O}\text{-}\mathsf{Corrupt}^*}(\{(ipk_0, spk_0), (ipk_1, spk_1), \ldots, (ipk_{n_u-1}, spk_{n_u-1})\}, \{epks_0, epks_1, \ldots, epks_{n_u-1}\})$
return $(b = b' \wedge \mathsf{PAIR.CONF}(\mathbf{L}))$

$\mathcal{O}\text{-}\mathsf{Enc}(i, j, info, sid, m_0, m_1)$

require $\mathsf{len}(m_0) = \mathsf{len}(m_1)$
if $sid = \varnothing$:
  $epk_j \leftarrow info[0]$; $sid \leftarrow epk_j$
$sst \leftarrow ssts[i, jsid]$
if $sst = \varnothing$:
  $sst, \cdot \leftarrow\$ \mathsf{Signal.Activate}(isk_i, ssk_i, \mathtt{init}, ipk_j)$
  $sst, c_{kex} \leftarrow\$ \mathsf{Signal.Run}(isk_i, ssk_i, sst, (spk_j, sid))$
else :
  $sst, c_{kex} \leftarrow\$ \mathsf{Signal.Run}(isk_i, ssk_i, sst, \varnothing)$
require $sst.status[sst.stage] = \mathtt{accept}$
if $(i, j, sid, sst.stage) \notin \mathbf{MK}$:
  $\mathbf{MK}[i, j, sid, sst.stage] \leftarrow\$ \mathcal{K}$
$mk \leftarrow \mathbf{MK}[i, j, sid, sst.stage]$
if $m_0 \neq m_1 \wedge |{}^*\mathsf{Challenges}(\mathbf{L})| < x$:
  $c_{msg} \leftarrow\$ \mathcal{C}$
elseif $m_0 \neq m_1 \wedge |{}^*\mathsf{Challenges}(\mathbf{L})| = h$:
  $c_{msg} \leftarrow \mathcal{O}\text{-}\mathsf{Enc}'[c_{kex}, m_b]$
elseif $m_0 \neq m_1 \wedge |{}^*\mathsf{Challenges}(\mathbf{L})| > N$:
  abort
else :
  $c_{msg} \leftarrow \mathsf{WA\text{-}AEAD.Enc}(mk, c_{kex}, m_b)$
$c \leftarrow (c_{kex}, c_{msg})$
$ssts[i, j, sid] \leftarrow sst$
$\mathbf{L} \leftarrow\!\!\| (\mathtt{enc}, i, j, sid, m_0, m_1, c)$
return sid, c

$\mathcal{O}\text{-}\mathsf{Dec}(i, j, info, sid, c)$

$c_{kex}, c_{msg} \leftarrow c$
if $sid = \varnothing$:
  $epk_i \leftarrow c_{kex}.epk_{resp}$; $sid \leftarrow epk_i$
$sst \leftarrow ssts[i, j, sid]$
if $sst = \varnothing$:
  $[esk] \leftarrow [esk' \textbf{ in } esks_i \textbf{ if } epk_i = \mathsf{PK}(esk')]$
  $sst, \cdot \leftarrow\$ \mathsf{Signal.Activate}(isk_i, ssk_i, \mathtt{resp}, ipk_j, esk)$
  $sst, \cdot \leftarrow\$ \mathsf{Signal.Run}(isk_i, ssk_i, sst, c_{kex})$
  $esks \leftarrow [esk' \textbf{ in } esks \textbf{ if } esk \neq esk']$
else :
  $sst, \cdot \leftarrow\$ \mathsf{Signal.Run}(isk_i, ssk_i, sst, c_{kex})$
require $sst.status[sst.stage] = \mathtt{accept}$
if $(j, i, sid, sst.stage) \notin \mathbf{MK}$:
  $\mathbf{MK}[j, i, sid, sst.stage] \leftarrow\$ \mathcal{K}$
$mk \leftarrow \mathbf{MK}[j, i, sid, sst.stage]$
$m \leftarrow \mathsf{WA\text{-}AEAD.Dec}(mk, c_{kex}, c_{msg})$
if $m = \perp$: **return** $\perp$
$ssts[i, j, sid] \leftarrow sst$
$\mathbf{L} \leftarrow\!\!\| (\mathtt{dec}, i, j, sid, c, m)$
if $c \in {}^*\mathsf{Challenges}(\mathbf{L})$: **return** $\perp$
return sid, m

$\mathcal{O}\text{-}\mathsf{CorruptIdentity}(i)$

require $0 \le i < n_u$
$corr \leftarrow isk_i$
$\mathbf{L} \leftarrow\!\!\| (\mathtt{corr\text{-}ident}, i, corr)$
return $corr$

$\mathcal{O}\text{-}\mathsf{CorruptShared}(i)$

require $0 \le i < n_u$
$corr \leftarrow (ssk_i, esks_i)$
$\mathbf{L} \leftarrow\!\!\| (\mathtt{corr\text{-}share}, i, corr)$
return $corr$

$\mathcal{O}\text{-}\mathsf{CorruptSession}(i, j, sid)$

require $0 \le i, j < n_u$
$corr \leftarrow ssts[i, j, sid]$
$\mathbf{L} \leftarrow\!\!\| (\mathtt{corr\text{-}sess}, i, j, sid, corr)$
return $corr$

Figure B.3. Security reduction demonstrating how we construct an adversary, $\mathcal{B}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{WA\text{-}AEAD},\mathcal{A}}$, against the IND$-CPA security of the WA-AEAD AEAD scheme using an existing adversary, $\mathcal{A}$, against the PAIR-SEC security of the WA-PAIR scheme. We use this security reduction to bound the changes introduced in **Game A1** of the proof of security for WhatsApp's pairwise channels (see Theorem 6.9).

Bibliography

[AAD+22] Anna Kaplan, Ann-Christine Kycler, Denis Kolegov, Jan Winkelmann, and Rai Yang. *Vodozemac Security Audit Report*. Technical report, Least Authority, March 2022. https://matrix.org/media/Least Authority - Matrix vodozemac Final Audit Report.pdf.

[AAN+24] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, and Krzyzstof Pietrzak. *DeCAF: Decentralizable CGKA with Fast Healing*. In Clemente Galdi and Duong Hieu Phan, editors, SCN 24, Part II, volume 14974 of LNCS, pages 294–313. Springer, Cham, September 2024.

[ABJM21] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. *Collective Information Security in Large-Scale Urban Protests: the Case of Hong Kong*. In Michael Bailey and Rachel Greenstadt, editors, USENIX Security 2021, pages 3363–3380. USENIX Association, August 2021.

[ACD19] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. *The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol*. In Yuval Ishai and Vincent Rijmen, editors, EUROCRYPT 2019, Part I, volume 11476 of LNCS, pages 129–158. Springer, Cham, May 2019.

[ACDJ23] Martin R. Albrecht, Sofía Celi, Benjamin Dowling, and Daniel Jones. *Practically-exploitable Cryptographic Vulnerabilities in Matrix*. In Thomas Ristenpart and Patrick Traynor, editors, 44th IEEE Symposium on Security and Privacy, 2023.

[ACDT20] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. *Security Analysis and Improvements for the IETF MLS Standard for Group Messaging*. In Daniele Micciancio and Thomas Ristenpart, editors, CRYPTO 2020, Part I, volume 12170 of LNCS, pages 248–277. Springer, Cham, August 2020.

[ACDT21] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. *Modular Design of Secure Group Messaging Protocols and the Security of MLS*. In Giovanni Vigna and Elaine Shi, editors, ACM CCS 2021, pages 1463–1483. ACM Press, November 2021.

[ACJM20] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. *Continuous Group Key Agreement with Active Security*. In Rafael Pass and Krzysztof Pietrzak, editors, TCC 2020, Part II, volume 12551 of LNCS, pages 261–290. Springer, Cham, November 2020.

[ACMP10] Michel Abdalla, Céline Chevalier, Mark Manulis, and David Pointcheval. *Flexible Group Key Exchange with On-demand Computation of Subgroup Keys*. In Daniel J. Bernstein and Tanja Lange, editors, AFRICACRYPT 10, volume 6055 of LNCS, pages 351–368. Springer, Berlin, Heidelberg, May 2010.

[ADJ24] Martin R. Albrecht, Benjamin Dowling, and Daniel Jones. *Device-Oriented Group Messaging: A Formal Cryptographic Analysis of Matrix' Core*. In 45th IEEE Symposium on Security and Privacy, 2024.

[AES01] *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology, NIST FIPS PUB 197, U.S. Department of Commerce, November 2001.

[AHK+23] Joël Alwen, Dominik Hartmann, Eike Kiltz, Marta Mularczyk, and Peter Schwabe. *Post-Quantum Multi-Recipient Public Key Encryption*. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, ACM CCS 2023, pages 1108–1122. ACM Press, November 2023.

[AJM22] Joël Alwen, Daniel Jost, and Marta Mularczyk. *On the Insider Security of MLS*. In Yevgeniy Dodis and Thomas Shrimpton, editors, CRYPTO 2022, Part II, volume 13508 of LNCS, pages 34–68. Springer, Cham, August 2022.

[AMT23] Joël Alwen, Marta Mularczyk, and Yiannis Tselekounis. *Fork-Resilient Continuous Group Key Agreement*. In Helena Handschuh and Anna Lysyanskaya, editors, CRYPTO 2023, Part IV, volume 14084 of LNCS, pages 396–429. Springer, Cham, August 2023.

[AST98] Giuseppe Ateniese, Michael Steiner, and Gene Tsudik. *Authenticated Group Key Agreement and Friends*. In Li Gong and Michael K. Reiter, editors, ACM CCS 98, pages 17–26. ACM Press, November 1998.

[BBC+23] Josh Blum, Simon Booth, Brian Chen, Oded Gal, Maxwell Krohn, Julia Len, Karan Lyons, Antonio Marcedone, Mike Maxim, Merry Ember Mou, Armin Namavari, Jack O'Connor, Surya Rien, Miles Steele, Matthew Green, Lea Kissner, and Alex Stamos. *Zoom Cryptography Whitepaper*, November 2023. https://github.com/zoom/zoom-e2e-whitepaper/.

[BBGS23] Matilda Backendal, Mihir Bellare, Felix Günther, and Matteo Scarlata. *When Messages Are Keys: Is HMAC a Dual-PRF?* In Helena Handschuh and Anna Lysyanskaya, editors, CRYPTO 2023, Part III, volume 14083 of LNCS, pages 661–693. Springer, Cham, August 2023.

[BBL+23] Olivier Blazy, Ioana Boureanu, Pascal Lafourcade, Cristina Onete, and Léo Robert. *How fast do you heal? A taxonomy for post-compromise security in secure-channel establishment.* In Joseph A. Calandrino and Carmela Troncoso, editors, USENIX Security 2023, pages 5917–5934. USENIX Association, August 2023.

[BBM09] Timo Brecher, Emmanuel Bresson, and Mark Manulis. *Fully Robust Tree-Diffie-Hellman Group Key Exchange.* In Juan A. Garay, Atsuko Miyaji, and Akira Otsuka, editors, CANS 09, volume 5888 of LNCS, pages 478–497. Springer, Berlin, Heidelberg, December 2009.

[BBR18] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. *TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS).* Research report, Inria Paris, May 2018. https://hal.inria.fr/hal-02425247.

[BBR+23] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. *The Messaging Layer Security (MLS) Protocol*, July 2023.

[BCC+23] Khashayar Barooti, Daniel Collins, Simone Colombo, Loïs Huguenin-Dumittan, and Serge Vaudenay. *On Active Attack Detection in Messaging with Immediate Decryption.* In Helena Handschuh and Anna Lysyanskaya, editors, CRYPTO 2023, Part IV, volume 14084 of LNCS, pages 362–395. Springer, Cham, August 2023.

[BCG22] David Balbás, Daniel Collins, and Phillip Gajland. *Analysis and Improvements of the Sender Keys Protocol for Group Messaging.* In Daniel Sadornil Renedo, editor, XVII Reunión española sobre criptología y seguridad de la información (RECSI), 2022.

[BCG23] David Balbás, Daniel Collins, and Phillip Gajland. *WhatsUpp with Sender Keys? Analysis, Improvements and Security Proofs.* In Jian Guo and Ron Steinfeld, editors, ASIACRYPT 2023, Part V, volume 14442 of LNCS, pages 307–341. Springer, Singapore, December 2023.

[BCJ+24] Chris Brzuska, Cas Cremers, Håkon Jacobsen, Douglas Stebila, and Bogdan Warinschi. *Falsifiability, Composability, and Comparability of Game-based Security Models for Key Exchange Protocols.* Cryptology ePrint Archive, Report 2024/1215, 2024.

[BCJZ21] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. *The Provable Security of Ed25519: Theory and Practice.* In 2021 IEEE Symposium on Security and Privacy, pages 1659–1676. IEEE Computer Society Press, May 2021.

[BCK98] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. *A Modular Approach to the Design and Analysis of Authentication and Key Exchange Protocols (Extended Abstract).* In 30th ACM STOC, pages 419–428. ACM Press, May 1998.

[BCP01] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. *Provably Authenticated Group Diffie-Hellman Key Exchange – The Dynamic Case*. In Colin Boyd, editor, ASIACRYPT 2001, volume 2248 of LNCS, pages 290–309. Springer, Berlin, Heidelberg, December 2001.

[BCP02] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. *Dynamic Group Diffie-Hellman Key Exchange under Standard Assumptions*. In Lars R. Knudsen, editor, EUROCRYPT 2002, volume 2332 of LNCS, pages 321–336. Springer, Berlin, Heidelberg, April / May 2002.

[BCPQ01] Emmanuel Bresson, Olivier Chevassut, David Pointcheval, and Jean-Jacques Quisquater. *Provably Authenticated Group Diffie-Hellman Key Exchange*. In Michael K. Reiter and Pierangela Samarati, editors, ACM CCS 2001, pages 255–264. ACM Press, November 2001.

[BCV23] David Balbás, Daniel Collins, and Serge Vaudenay. *Cryptographic Administration for Secure Group Messaging*. In Joseph A. Calandrino and Carmela Troncoso, editors, USENIX Security 2023, pages 1253–1270. USENIX Association, August 2023.

[BDJR97] Mihir Bellare, Anand Desai, Eric Jokipii, and Phillip Rogaway. *A Concrete Security Treatment of Symmetric Encryption*. In 38th FOCS, pages 394–403. IEEE Computer Society Press, October 1997.

[BDL+12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. *High-speed high-security signatures*. Journal of Cryptographic Engineering, 2(2):77–89, September 2012.

[Bel98] Mihir Bellare. *Practice-Oriented Provable-Security (Invited Lecture)*. In Eiji Okamoto, George I. Davida, and Masahiro Mambo, editors, ISW'97, volume 1396 of LNCS, pages 221–231. Springer, Berlin, Heidelberg, September 1998.

[Bel06] Mihir Bellare. *New Proofs for NMAC and HMAC: Security without Collision-Resistance*. In Cynthia Dwork, editor, CRYPTO 2006, volume 4117 of LNCS, pages 602–619. Springer, Berlin, Heidelberg, August 2006.

[Bel14] Gary Belvin. *A Secure Text Messaging Protocol*. Cryptology ePrint Archive, Report 2014/036, 2014.

[Bel15] Mihir Bellare. *New Proofs for NMAC and HMAC: Security without Collision Resistance*. Journal of Cryptology, 28(4):844–878, October 2015.

[Ber06] Daniel J. Bernstein. *Curve25519: New Diffie-Hellman Speed Records*. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, PKC 2006, volume 3958 of LNCS, pages 207–228. Springer, Berlin, Heidelberg, April 2006.

[BFG+22] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. *A More Complete Analysis of the Signal Double Ratchet Algorithm*. In Yevgeniy Dodis and Thomas Shrimpton, editors, CRYPTO 2022, Part I, volume 13507 of LNCS, pages 784–813. Springer, Cham, August 2022.

[BGB04] Nikita Borisov, Ian Goldberg, and Eric Brewer. *Off-the-record communication, or, why not to use PGP*. In Proceedings of the 2004 ACM workshop on Privacy in the electronic society, pages 77–84, 2004. https://otr.cypherpunks.ca/otr-wpes.pdf.

[BK03] Mihir Bellare and Tadayoshi Kohno. *A Theoretical Treatment of Related-Key Attacks: RKA-PRPs, RKA-PRFs, and Applications*. In Eli Biham, editor, EUROCRYPT 2003, volume 2656 of LNCS, pages 491–506. Springer, Berlin, Heidelberg, May 2003.

[BM97] Mihir Bellare and Daniele Micciancio. *A New Paradigm for Collision-Free Hashing: Incrementality at Reduced Cost*. In Walter Fumy, editor, EUROCRYPT'97, volume 1233 of LNCS, pages 163–192. Springer, Berlin, Heidelberg, May 1997.

[BM08] Emmanuel Bresson and Mark Manulis. *Securing group key exchange against strong corruptions*. In Masayuki Abe and Virgil Gligor, editors, ASIACCS 08, pages 249–260. ACM Press, March 2008.

[BN08] Mihir Bellare and Chanathip Namprempre. *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm*. Journal of Cryptology, 21(4):469–491, October 2008.

[BPR22] Dan Boneh, Aditi Partap, and Lior Rotem. *Accountable Threshold Signatures with Proactive Refresh*. Cryptology ePrint Archive, Report 2022/1656, 2022.

[BR94] Mihir Bellare and Phillip Rogaway. *Entity Authentication and Key Distribution*. In Douglas R. Stinson, editor, CRYPTO'93, volume 773 of LNCS, pages 232–249. Springer, Berlin, Heidelberg, August 1994.

[BR00] Mihir Bellare and Phillip Rogaway. *Encode-Then-Encipher Encryption: How to Exploit Nonces or Redundancy in Plaintexts for Efficient Cryptography*. In Tatsuaki Okamoto, editor, ASIACRYPT 2000, volume 1976 of LNCS, pages 317–330. Springer, Berlin, Heidelberg, December 2000.

[BR04] Mihir Bellare and Phillip Rogaway. *Code-Based Game-Playing Proofs and the Security of Triple Encryption*. Cryptology ePrint Archive, Report 2004/331, 2004.

[BR06] Mihir Bellare and Phillip Rogaway. *The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs*. In Serge Vaudenay, editor, EUROCRYPT 2006, volume 4004 of LNCS, pages 409–426. Springer, Berlin, Heidelberg, May / June 2006.

[BSJ+17] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. *Ratcheted Encryption and Key Exchange: The Security of Messaging.* In Jonathan Katz and Hovav Shacham, editors, CRYPTO 2017, Part III, volume 10403 of LNCS, pages 619–650. Springer, Cham, August 2017.

[BVJ+23] Shi Bai, Iggy Van Hoof, Floyd Johnson, Tanja Lange, and Tran Ngo. *Concrete Analysis of Quantum Lattice Enumeration.* In Jian Guo and Ron Steinfeld, editors, ASIACRYPT 2023, Part III, volume 14440 of LNCS, pages 131–166. Springer, Singapore, December 2023.

[CCD+20] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. *A Formal Security Analysis of the Signal Messaging Protocol.* Journal of Cryptology, 33(4):1914–1983, October 2020.

[CCG16] Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. *On Post-compromise Security.* In Michael Hicks and Boris Köpf, editors, CSF 2016 Computer Security Foundations Symposium, pages 164–178. IEEE Computer Society Press, 2016.

[CCG+18] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. *On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees.* In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, ACM CCS 2018, pages 1802–1819. ACM Press, October 2018.

[CDDF20] Sébastien Campion, Julien Devigne, Céline Duguey, and Pierre-Alain Fouque. *Multi-Device for Signal.* In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, ACNS 2020, Part II, volume 12147 of LNCS, pages 167–187. Springer, Cham, October 2020.

[CDGM19] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. *SEEMless: Secure End-to-End Encrypted Messaging with less Trust.* In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, ACM CCS 2019, pages 1639–1656. ACM Press, November 2019.

[CDNO97] Ran Canetti, Cynthia Dwork, Moni Naor, and Rafail Ostrovsky. *Deniable Encryption.* In Burton S. Kaliski, Jr., editor, CRYPTO'97, volume 1294 of LNCS, pages 90–104. Springer, Berlin, Heidelberg, August 1997.

[CDV21] Andrea Caforio, F. Betül Durak, and Serge Vaudenay. *Beyond Security and Efficiency: On-Demand Ratcheting with Security Awareness.* In Juan Garay, editor, PKC 2021, Part II, volume 12711 of LNCS, pages 649–677. Springer, Cham, May 2021.

[CFKN20] Cas Cremers, Jaiden Fairoze, Benjamin Kiesl, and Aurora Naska. *Clone Detection in Secure Messaging: Improving Post-Compromise Security in Practice.* In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, ACM CCS 2020, pages 1481–1495. ACM Press, November 2020.

[CHK21] Cas Cremers, Britta Hale, and Konrad Kohbrok. *The Complexities of Healing in Secure Group Messaging: Why Cross-Group Effects Matter*. In Michael Bailey and Rachel Greenstadt, editors, USENIX Security 2021, pages 1847–1864. USENIX Association, August 2021.

[CJN23] Cas Cremers, Charlie Jacomme, and Aurora Naska. *Formal Analysis of Session-Handling in Secure Messaging: Lifting Security from Sessions to Conversations*. In Joseph A. Calandrino and Carmela Troncoso, editors, USENIX Security 2023, pages 1235–1252. USENIX Association, August 2023.

[CK01] Ran Canetti and Hugo Krawczyk. *Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels*. In Birgit Pfitzmann, editor, EUROCRYPT 2001, volume 2045 of LNCS, pages 453–474. Springer, Berlin, Heidelberg, May 2001.

[CL01] Jan Camenisch and Anna Lysyanskaya. *Efficient Revocation of Anonymous Group Membership*. Cryptology ePrint Archive, Report 2001/113, 2001.

[CL02] Jan Camenisch and Anna Lysyanskaya. *Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials*. In Moti Yung, editor, CRYPTO 2002, volume 2442 of LNCS, pages 61–76. Springer, Berlin, Heidelberg, August 2002.

[CPZ20] Melissa Chase, Trevor Perrin, and Greg Zaverucha. *The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption*. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, ACM CCS 2020, pages 1445–1459. ACM Press, November 2020.

[Den06] Alexander W. Dent. *A Note On Game-Hopping Proofs*. Cryptology ePrint Archive, Report 2006/260, 2006.

[DFG+23] Gareth T. Davies, Sebastian H. Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horváth, and Tibor Jager. *Security Analysis of the WhatsApp End-to-End Encrypted Backup Protocol*. In Helena Handschuh and Anna Lysyanskaya, editors, CRYPTO 2023, Part IV, volume 14084 of LNCS, pages 330–361. Springer, Cham, August 2023.

[DFGS21] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. *A Cryptographic Analysis of the TLS 1.3 Handshake Protocol*. Journal of Cryptology, 34(4):37, October 2021.

[DGGL21] Antonio Dimeo, Felix Gohla, Daniel Goßen, and Niko Lockenvitz. *SoK: Multi-Device Secure Instant Messaging*. Cryptology ePrint Archive, Report 2021/498, 2021.

[DGP22] Benjamin Dowling, Felix Günther, and Alexandre Poirrier. *Continuous authentication in secure messaging*. In European Symposium on Research in Computer Security, pages 361–381. Springer, 2022.

[DH76] Whitfield Diffie and Martin E. Hellman. *New Directions in Cryptography.* IEEE Transactions on Information Theory, 22(6):644–654, 1976.

[DJK22] Yevgeniy Dodis, Daniel Jost, and Harish Karthikeyan. *Forward-Secure Encryption with Fast Forwarding.* In Eike Kiltz and Vinod Vaikuntanathan, editors, TCC 2022, Part II, volume 13748 of LNCS, pages 3–32. Springer, Cham, November 2022.

[DJKM23] Yevgeniy Dodis, Daniel Jost, Balachandar Kesavan, and Antonio Marcedone. *End-to-End Encrypted Zoom Meetings: Proving Security and Strengthening Liveness.* In Carmit Hazay and Martijn Stam, editors, EUROCRYPT 2023, Part V, volume 14008 of LNCS, pages 157–189. Springer, Cham, April 2023.

[DSSW14] Yevgeniy Dodis, Adi Shamir, Noah Stephens-Davidowitz, and Daniel Wichs. *How to Eat Your Entropy and Have It Too - Optimal Recovery Strategies for Compromised RNGs.* In Juan A. Garay and Rosario Gennaro, editors, CRYPTO 2014, Part II, volume 8617 of LNCS, pages 37–54. Springer, Berlin, Heidelberg, August 2014.

[Dwo01] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Methods and Techniques.* Technical Report NIST Special Publication (SP) 800-38A, National Institute of Standards and Technology, December 2001. https://csrc.nist.gov/publications/detail/sp/800-38a/final.

[Fay] Faye Duxovni. *MSC3917: Cryptographically Constrained Room Membership.* https://github.com/matrix-org/matrix-spec-proposals/pull/3917.

[FG14] Marc Fischlin and Felix Günther. *Multi-Stage Key Exchange and the Case of Google's QUIC Protocol.* In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, ACM CCS 2014, pages 1193–1204. ACM Press, November 2014.

[FJ24] Rune Fiedler and Christian Janson. *A Deniability Analysis of Signal's Initial Handshake PQXDH.* Proceedings on Privacy Enhancing Technologies, 2024:907–928, 2024.

[FMB+16] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jörg Schwenk, and Thorsten Holz. *How Secure is TextSecure?* In IEEE European Symposium on Security and Privacy, EuroS&P 2016, pages 457–472, 2016.

[Gün90] Christoph G. Günther. *An Identity-Based Key-Exchange Protocol.* In Jean-Jacques Quisquater and Joos Vandewalle, editors, EUROCRYPT'89, volume 434 of LNCS, pages 29–37. Springer, Berlin, Heidelberg, April 1990.

[GUVC09] Ian Goldberg, Berkant Ustaoglu, Matthew Van Gundy, and Hao Chen. *Multi-party off-the-record messaging.* In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, ACM CCS 2009, pages 358–368. ACM Press, November 2009.

[HLW23] Lucjan Hanzlik, Julian Loss, and Benedikt Wagner. *Token meets Wallet: Formalizing Privacy and Revocation for FIDO2*. In 2023 IEEE Symposium on Security and Privacy, pages 1491–1508. IEEE Computer Society Press, May 2023.

[Hou09] Russ Housley. *RFC 5652: Cryptographic Message Syntax (CMS)*, September 2009.

[JBGH20] Florian Jacob, Luca Becker, Jan Grashöfer, and Hannes Hartenstein. *Matrix Decomposition: Analysis of an Access Control Approach on Transaction-based DAGs without Finality*. In Proceedings of the 25th ACM Symposium on Access Control Models and Technologies, SACMAT '20, page 81–92, New York, NY, USA, 2020. Association for Computing Machinery.

[JBHH21] Florian Jacob, Carolin Beer, Norbert Henze, and Hannes Hartenstein. *Analysis of the Matrix Event Graph Replicated Data Type*. IEEE Access, 9:28317–28333, 2021.

[JGH19] Florian Jacob, Jan Grashöfer, and Hannes Hartenstein. *A glimpse of the matrix: Scalability issues of a new message-oriented data synchronization middleware*. In Proceedings of the 20th International Middleware Conference Demos and Posters, pages 5–6, 2019.

[JKS24] Joseph Jaeger, Akshaya Kumar, and Igors Stepanovs. *Symmetric Signcryption and E2EE Group Messaging in Keybase*. In Marc Joye and Gregor Leander, editors, EUROCRYPT 2024, Part III, volume 14653 of LNCS, pages 283–312. Springer, Cham, May 2024.

[JL17] Simon Josefsson and Ilari Liusvaara. *RFC 8032: Edwards-Curve Digital Signature Algorithm (EdDSA)*, January 2017.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. *RFC 2104: HMAC: Keyed-Hashing for Message Authentication*, February 1997.

[KE10] H. Krawczyk and P. Eronen. *RFC 5869: HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*, May 2010.

[Key22] Keybase. *Meet your sigchain (and everyone else's)*, 2022. https://book.keybase.io/docs/server.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, November 2014.

[Koc98] Paul C. Kocher. *On Certificate Revocation and Validation*. In Rafael Hirschfeld, editor, FC'98, volume 1465 of LNCS, pages 172–177. Springer, Berlin, Heidelberg, February 1998.

[KPPW+21] Karen Klein, Guillermo Pascual-Perez, Michael Walter, Chethan Kamath, Margarita Capretto, Miguel Cueto, Ilia Markov, Michelle Yeo, Joël Alwen, and Krzysztof Pietrzak. *Keep the Dirt: Tainted TreeKEM, Adaptively and Actively Secure Continuous Group Key Agreement*. In 2021 IEEE Symposium on Security and Privacy, pages 268–284. IEEE Computer Society Press, May 2021.

[Kra10] Hugo Krawczyk. *Cryptographic Extraction and Key Derivation: The HKDF Scheme*. In Tal Rabin, editor, CRYPTO 2010, volume 6223 of LNCS, pages 631–648. Springer, Berlin, Heidelberg, August 2010.

[KY03] Jonathan Katz and Moti Yung. *Scalable Protocols for Authenticated Group Key Exchange*. In Dan Boneh, editor, CRYPTO 2003, volume 2729 of LNCS, pages 110–125. Springer, Berlin, Heidelberg, August 2003.

[KY07] Jonathan Katz and Moti Yung. *Scalable Protocols for Authenticated Group Key Exchange*. Journal of Cryptology, 20(1):85–113, January 2007.

[LCG+23] Julia Len, Melissa Chase, Esha Ghosh, Daniel Jost, Balachandar Kesavan, and Antonio Marcedone. *ELEKTRA: Efficient Lightweight multi-dEvice Key TRAnsparency*. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, ACM CCS 2023, pages 2915–2929. ACM Press, November 2023.

[LKMW19] Kevin Lewi, Wonho Kim, Ilya Maykov, and Stephen Weis. *Securing Update Propagation with Homomorphic Hashing*. Cryptology ePrint Archive, Report 2019/227, 2019.

[Lot24] Lotte. *Missing Salamanders: Matrix Media can be decrypted to multiple valid plaintexts using different keys*, August 2024. https://lotte.chir.rs/2024/08/17/Missing-Salamanders-Matrix-Media-can-be-decrypted-to-multiple-valid-plaintexts-using-different-keys/.

[LR18] Ian Levy and Crispin Robinson. *Principles for a More Informed Exceptional Access Debate*. Lawfare, November 2018. https://www.lawfareblog.com/principles-more-informed-exceptional-access-debate.

[Man06] Mark Manulis. *Security-Focused Survey on Group Key Exchange Protocols*. Cryptology ePrint Archive, Report 2006/395, 2006.

[Man07] Mark Manulis. *Provably Secure Group Key Exchange*, volume 5 of IT Security. Europäischer Universitätsverlag, Berlin, Bochum, Dülmen, London, Paris, August 2007. https://www.manulis.eu/papers/psgke.pdf.

[Man09] Mark Manulis. *Group Key Exchange Enabling On-Demand Derivation of Peer-to-Peer Keys*. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, ACNS 2009, volume 5536 of LNCS, pages 1–19. Springer, Berlin, Heidelberg, June 2009.

[Mar13a] Moxie Marlinspike. *Advanced cryptographic ratcheting*, November 2013. https://signal.org/blog/advanced-ratcheting/.

[Mar13b] Moxie Marlinspike. *Simplifying OTR deniability*, July 2013. https://signal.org/blog/simplifying-otr-deniability/.

[Mar14] Moxie Marlinspike. *Private Group Messaging*, May 2014. https://signal.org/blog/private-groups/.

[Mar16a] Moxie Marlinspike. *The Double Ratchet Algorithm*, November 2016. https://signal.org/docs/specifications/doubleratchet/.

[Mar16b] Moxie Marlinspike. *The X3DH Key Agreement Protocol*, November 2016. https://signal.org/docs/specifications/x3dh/, Revision 1.

[Mar17] Moxie Marlinspike. *The Sesame Algorithm: Session Management for Asynchronous Message Encryption*, April 2017. https://signal.org/docs/specifications/sesame/, Revision 2.

[Mata] Matrix.org Foundation. *Client-Server API*. https://spec.matrix.org/unstable/client-server-api/. Version: unstable.

[Matb] Matrix.org Foundation. *Matrix Client-Server SDK for JavaScript*. https://github.com/matrix-org/matrix-js-sdk/. Commit: 4721aa1d.

[Matc] Matrix.org Foundation. *Matrix SDK for React Javascript*. https://github.com/matrix-org/matrix-react-sdk/. Commit: 59b9d1e8.

[Matd] Matrix.org Foundation. *Server-Server API*. https://spec.matrix.org/unstable/server-server-api/. Version: unstable.

[Mate] Matrix.org Foundation. *Signature keys and user identity in libolm*. https://gitlab.matrix.org/matrix-org/olm/-/raw/master/docs/signing.md.

[Mat17] Matrix.org Foundation. *Olm: Implementation of the olm and megolm cryptographic ratchets*, April 2017. https://gitlab.matrix.org/matrix-org/olm.

[Mat21a] Matrix.org Foundation. *MSC1756: Cross-signing devices with device signing keys*, January 2021. https://github.com/matrix-org/matrix-doc/blob/5132f1ea01e9e95c7837bfdf885ababfe7e6908d/proposals/1756-cross-signing.md.

[Mat21b] Matrix.org Foundation. *MSC2732: Olm fallback keys*, August 2021. https://raw.githubusercontent.com/uhoreg/matrix-doc/refs/heads/fallback_keys/proposals/2732-olm-fallback-keys.md.

[Mat21c] Matrix.org Foundation. *MSC3270: Symmetric megolm backup*, July 2021. https://raw.githubusercontent.com/uhoreg/matrix-doc/refs/heads/symmetric-backups/proposals/3270-symmetric-megolm-backup.md.

[Mat22a] Matrix.org Foundation. *Megolm group ratchet*, May 2022. https://gitlab.matrix.org/matrix-org/olm/-/raw/master/docs/megolm.md. Version: unstable.

[Mat22b] Matrix.org Foundation. *Olm: A Cryptographic Ratchet*, May 2022. https://gitlab.matrix.org/matrix-org/olm/-/raw/master/docs/olm.md. Version: unstable.

[Mat23] Matrix.org Foundation. *End-to-End Encryption implementation guide*, February 2023. https://matrix.org/docs/guides/end-to-end-encryption-implementation-guide.

[MB16] Jake Meredith and Alex Balducci. *Matrix Olm Cryptographic Review*. Technical report, NCC Group, November 2016. https://www.nccgroup.com/us/research-blog/public-report-matrix-olm-cryptographic-review/. Version 2.0.

[MBB+15] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. *CONIKS: Bringing Key Transparency to End Users*. In Jaeyeon Jung and Thorsten Holz, editors, USENIX Security 2015, pages 383–398. USENIX Association, August 2015.

[MF21] Arno Mittelbach and Marc Fischlin. *The Theory of Hash Functions and Random Oracles: An Approach to Modern Cryptography*. Springer International Publishing AG, Cham, SWITZERLAND, 2021.

[MKKS+23] Harjasleen Malvai, Lefteris Kokoris-Kogias, Alberto Sonnino, Esha Ghosh, Ercan Oztürk, Kevin Lewi, and Sean F. Lawlor. *Parakeet: Practical Key Transparency for End-to-End Encrypted Messaging*. In NDSS 2023. The Internet Society, February 2023.

[New22] New Vector Ltd. *Element – A glossy Matrix collaboration client for the web*, May 2022. https://github.com/element-hq/element-web/. Commit: 479d4bf6.

[OP01] Tatsuaki Okamoto and David Pointcheval. *The Gap-Problems: A New Class of Problems for the Security of Cryptographic Schemes*. In Kwangjo Kim, editor, PKC 2001, volume 1992 of LNCS, pages 104–118. Springer, Berlin, Heidelberg, February 2001.

[Pag09] Thomas Page. *The application of hash chains and hash structures to cryptography*. PhD thesis, Royal Holloway, University of London, 2009.

[Per16] Trevor Perrin. *The XEdDSA and VXEdDSA Signature Schemes*, October 2016. https://signal.org/docs/specifications/xeddsa/, Revision 1.

[PP22] Jeroen Pijnenburg and Bertram Poettering. *On Secure Ratcheting with Immediate Decryption*. In Shweta Agrawal and Dongdai Lin, editors, ASIACRYPT 2022, Part III, volume 13793 of LNCS, pages 89–118. Springer, Cham, December 2022.

[PRSS21a] Bertram Poettering, Paul Rösler, Jörg Schwenk, and Douglas Stebila. *SoK: Game-Based Security Models for Group Key Exchange*. In Kenneth G. Paterson, editor, CT-RSA 2021, volume 12704 of LNCS, pages 148–176. Springer, Cham, May 2021.

[PRSS21b] Bertram Poettering, Paul Rösler, Jörg Schwenk, and Douglas Stebila. *SoK: Game-based Security Models for Group Key Exchange*. Cryptology ePrint Archive, Report 2021/305, 2021.

[RBBK01] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. *OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption*. In Michael K. Reiter and Pierangela Samarati, editors, ACM CCS 2001, pages 196–205. ACM Press, November 2001.

[RMS17] Paul Rösler, Christian Mainka, and Jörg Schwenk. *More is Less: How Group Chats Weaken the Security of Instant Messengers Signal, WhatsApp, and Threema*. Cryptology ePrint Archive, Report 2017/713, 2017.

[RMS18] Paul Rösler, Christian Mainka, and Jörg Schwenk. *More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema*. In IEEE European Symposium on Security and Privacy, EuroS&P 2018, pages 415–429, 2018.

[Rog02] Phillip Rogaway. *Authenticated-Encryption With Associated-Data*. In Vijayalakshmi Atluri, editor, ACM CCS 2002, pages 98–107. ACM Press, November 2002.

[Rog04] Phillip Rogaway. *On the role of Definitions in and Beyond Cryptography*. In Annual Asian computing science conference, pages 13–32. Springer, 2004.

[Rog09] Phillip Rogaway. *Practice-oriented provable security and the social construction of cryptography*. Unpublished essay, 2009. https://www.cs.ucdavis.edu/~rogaway/papers/cc.pdf.

[SA11a] Peter Saint-Andre. *RFC 6120: Extensible Messaging and Presence Protocol (XMPP): Core*, 2011.

[SA11b] Peter Saint-Andre. *RFC 6120: Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*, 2011.

[SHA02] *Secure Hash Standard*. National Institute of Standards and Technology, NIST FIPS PUB 180-2, U.S. Department of Commerce, August 2002.

[Sho99] Victor Shoup. *On Formal Models for Secure Key Exchange*. Cryptology ePrint Archive, Report 1999/012, 1999.

[Sho04] Victor Shoup. *Sequences of games: a tool for taming complexity in security proofs*. Cryptology ePrint Archive, Report 2004/332, 2004.

[Soa24] Soatok. *Security Issues in Matrix's Olm Library*, August 2024. https://soatok.blog/2024/08/14/security-issues-in-matrixs-olm-library/.

[TT01] Wen-Guey Tzeng and Zhi-Jia Tzeng. *Robust Forward-Secure Signature Schemes with Proactive Security*. In Kwangjo Kim, editor, PKC 2001, volume 1992 of LNCS, pages 264–276. Springer, Berlin, Heidelberg, February 2001.

[UDB+15] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. *SoK: Secure Messaging.* In 2015 IEEE Symposium on Security and Privacy, pages 232–249. IEEE Computer Society Press, May 2015.

[UG15] Nik Unger and Ian Goldberg. *Deniable Key Exchanges for Secure Messaging.* In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, ACM CCS 2015, pages 1211–1223. ACM Press, October 2015.

[Ung21] Nik Unger. *End-to-End Encrypted Group Messaging with Insider Security.* PhD thesis, University of Waterloo, August 2021. https://uwspace.uwaterloo.ca/bitstream/handle/10012/17196/Unger_Nik.pdf.

[Vau05] Serge Vaudenay. *Secure Communications over Insecure Channels Based on Short Authenticated Strings.* In Victor Shoup, editor, CRYPTO 2005, volume 3621 of LNCS, pages 309–326. Springer, Berlin, Heidelberg, August 2005.

[VGP12] Vinnie Moscaritolo, Gary Belvin, and Phil Zimmermann. *Silent Circle Instant Messaging Protocol Specification*, December 2012. https://silentcircle.com/sites/default/themes/silentcircle/assets/downloads/SCIMP_paper.pdf.

[VL16] Sebastian R. Verschoor and Tanja Lange. *(In-)Secure messaging with the Silent Circle instant messaging protocol.* Cryptology ePrint Archive, Report 2016/703, 2016.

[Wha23a] WhatsApp LLC. *WhatsApp Encryption Overview*, January 2023. https://whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf. Version 6.

[Wha23b] WhatsApp LLC. *WhatsApp Key Transparency Overview*, August 2023. https://www.whatsapp.com/security/WhatsApp-Key-Transparency-Whitepaper.pdf.

[Wha24] WhatsApp LLC. *WhatsApp Encryption Overview*, August 2024. https://whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf. Version 8.

[Won21] David Wong. *Real-world Cryptography.* Manning Publications, 2021. ISBN: 9781617296710.

[WPBB23] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. *TreeSync: Authenticated Group Management for Messaging Layer Security.* In Joseph A. Calandrino and Carmela Troncoso, editors, USENIX Security 2023, pages 1217–1233. USENIX Association, August 2023.

[ZAJC11] P. Zimmermann, Ed. A. Johnston, and J. Callas. *RFC 6189: RTP: Media Path Key Agreement for Unicast Secure RTP*, April 2011.

Articles, Websites & External Links

[L1] Paul Brown. *KDE is adding Matrix to its instant messaging infrastructure*, February 2019. https://dot.kde.org/2019/02/20/kde-adding-matrix-its-im-framework/ (archived on 2025-03-21).

[L2] Catalin Cimpanu. *French government releases in-house IM app to replace WhatsApp and Telegram use*. https://www.zdnet.com/article/french-government-releases-in-house-im-app-to-replace-whatsapp-and-telegram-use/ (archived on 2024-05-30).

[L3] Element. *Third Room – Open, decentralized, immersive worlds built on Matrix*. https://thirdroom.io/.

[L4] The Luxembourg Government's Ministry for Digitalisation. *dSam and eSam endorse Matrix for secure and federated communications in the Swedish public sector*, November 2022. https://gouvernement.lu/en/actualites/toutes_actualites/communiques/2022/11-novembre/16-hansen-lancement.html (archived on 2025-01-17).

[L5] Matrix.org Foundation. *Matrix.org – Servers*. https://matrix.org/ecosystem/servers/ (archived on 2025-03-28).

[L6] Mozilla Foundation. *Release Notes – Thunderbird Desktop Version 102.0*, June 2022. https://www.thunderbird.net/en-US/thunderbird/102.0/releasenotes/ (archived on 2025-02-11).

[L7] Matthew Hodgson. *Matrix's 'Olm' End-to-end Encryption security assessment released - and implemented cross-platform on Riot at last!*, November 2016. https://matrix.org/blog/2016/11/21/matrix-s-olm-end-to-end-encryption-security-assessment-released-and-implemented-cross-platform-on-riot-at-last/ (archived on 2024-12-21).

[L8] Matthew Hodgson. *Cross-signing and End-to-end Encryption by Default is HERE!!!*, May 2020. https://matrix.org/blog/2020/05/06/cross-signing-and-end-to-end-encryption-by-default-is-here/ (archived on 2025-01-22).

[L9] Matthew Hodgson. *Germany's national healthcare system adopts Matrix!*, July 2021. https://matrix.org/blog/2021/07/21/germany-s-national-healthcare-system-adopts-matrix/ (archived on 2024-12-09).

[L10] Matthew Hodgson. *Independent public audit of Vodozemac, a native Rust reference implementation of Matrix end-to-end encryption*, May 2022. https://matrix.org/blog/2022/05/16/independent-public-audit-of-vodozemac-a-native-rust-reference-implementation-of-matrix-end-to-end-encryption/ (archived on 2025-03-23).

[L11] Magnus Hult. *dSam and eSam endorse Matrix for secure and federated communications in the Swedish public sector*, December 2022. https://element.io/blog/dsam-och-esam-fororder-matrix-for-saker-och-federerad-kommunikation-inom-sveriges-offentliga-sektor/ (archived on 2025-02-21).

[L12] Neil Johnson. *Libolm Deprecation*, August 2024. https://matrix.org/blog/2024/08/libolm-deprecation/ (archived on 2025-03-22).

[L13] Sean Lawlor and Kevin Lewi. *Deploying key transparency at WhatsApp*, April 2023. https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/ (archived on 2025-03-25).

[L14] WhatsApp LLC. *About WhatsApp*. https://www.whatsapp.com/about/ (archived on 2025-01-08).

[L15] New Vector Ltd. *The Bundeswehr builds on Matrix.* https://element.io/case-studies/bundeswehr (archived on 2025-02-17).

[L16] New Vector Ltd. *France embraces Matrix to build Tchap.* https://element.io/case-studies/tchap (archived on 2025-03-27).

[L17] New Vector Ltd. *Element | Secure collaboration and messaging*, January 2025. https://element.io/ (archived on 2025-01-25).

[L18] Thibault Martin. *Hosting FOSDEM 2022 on Matrix*, February 2022. https://matrix.org/blog/2022/02/07/hosting-fosdem-2022-on-matrix/ (archived on 2025-01-12).

[L19] Thibault Martin. *This Week in Matrix 2022-09-30*, September 2022. https://matrix.org/blog/2022/09/30/this-week-in-matrix-2022-09-30 (archived on 2022-10-02).

[L20] Matrix.org Foundation. *Matrix Specification Proposals – Proposals for changes to the matrix specification.* https://github.com/matrix-org/matrix-spec-proposals (archived on 2025-01-23).

[L21] Meta Platforms, Inc. *akd – An implementation of an auditable key directory.* https://github.com/facebook/akd (archived on 2025-01-16).

[L22] Cade Metz. *Forget Apple vs. the FBI: WhatsApp Just Switched on Encryption for a Billion People*, April 2016. https://www.wired.com/2016/04/forget-apple-vs-fbi-whatsapp-just-switched-encryption-billion-people/ (archived on 2025-01-23).

[L23] *Matrix – MozillaWiki*, October 2020. https://wiki.mozilla.org/Matrix (archived on 2025-03-28).

[L24] Matrix.org Team. *We have discovered and addressed a security breach.*, April 2019. https://matrix.org/blog/2019/04/11/we-have-discovered-and-addressed-a-security-breach-updated-2019-04-12/ (archived on 2024-12-10).

[L25] Rocket.Chat Team. *Rocket.Chat Leverages The Matrix Protocol for Decentralized and Interoperable Communications*, May 2022. https://www.rocket.chat/press-releases/rocket-chat-leverages-matrix-protocol-for-decentralized-and-interoperable-communications (archived on 2025-03-08).

[L26] BWI GmbH (via Presse Portal). *BWI und Bundeswehr setzen auf Open-Source Matrix ist einheitlicher Messenger-Standard für Bundeswehrangehörige*. https://www.presseportal.de/pm/76712/4764023 (archived on 2024-07-21).

[L27] *CVE-2021-29471: Denial of service in Matrix Synapse*, May 2021. https://www.cve.org/CVERecord?id=CVE-2021-29471.

[L28] *CVE-2021-32622: File upload local preview can run embedded scripts after user interaction*, May 2021. https://www.cve.org/CVERecord?id=CVE-2021-32622.

[L29] *CVE-2021-32659: Automatic room upgrade handling can be used maliciously to bridge a room non-consentually*, June 2021. https://www.cve.org/CVERecord?id=CVE-2021-32659.

[L30] *CVE-2021-34813: Matrix libolm before 3.2.3 allows a malicious Matrix homeserver to crash a client*, June 2021. https://www.cve.org/CVERecord?id=CVE-2021-34813.

[L31] *CVE-2021-39163: Adding a private/unlisted room to a community exposes room metadata in an unauthorised manner.*, August 2021. https://www.cve.org/CVERecord?id=CVE-2021-39163.

[L32] *CVE-2021-39164: Improper authorisation of /members discloses room membership to non-members*, August 2021. https://www.cve.org/CVERecord?id=CVE-2021-39164.

[L33] *CVE-2021-40824: A logic error in the room key sharing functionality of Element Android […] allows the attacker to decrypt end-to-end encrypted messages sent by affected clients.*, September 2021. https://www.cve.org/CVERecord?id=CVE-2021-40824.

[L34] *CVE-2021-41281: Path traversal in Matrix Synapse*, November 2021. https://www.cve.org/CVERecord?id=CVE-2021-41281.

[L35] *CVE-2022-23597: Remote program execution with user interaction*, February 2022. https://www.cve.org/CVERecord?id=CVE-2022-23597.

[L36] *CVE-2022-31052: URL previews can crash Synapse media repositories or Synapse monoliths*, June 2022. https://www.cve.org/CVERecord?id=CVE-2022-31052.

[L37] *CVE-2022-39246: matrix-android-sdk2 vulnerable to impersonation via forwarded Megolm sessions*, September 2022. https://www.cve.org/CVERecord?id=CVE-2022-39246.

[L38] *CVE-2022-39248: matrix-android-sdk2 vulnerable to Olm/Megolm protocol confusion*, September 2022. https://www.cve.org/CVERecord?id=CVE-2022-39248.

[L39] *CVE-2022-39249: Matrix Javascript SDK vulnerable to impersonation via forwarded Megolm sessions*, September 2022. https://www.cve.org/CVERecord?id=CVE-2022-39249.

[L40] *CVE-2022-39250: Matrix JavaScript SDK vulnerable to key/device identifier confusion in SAS verification*, September 2022. https://www.cve.org/CVERecord?id=CVE-2022-39250.

[L41] *CVE-2022-39251: Matrix Javascript SDK vulnerable to Olm/Megolm protocol confusion*, September 2022. https://www.cve.org/CVERecord?id=CVE-2022-39251.

[L42] *CVE-2022-39252: When matrix-rust-sdk recieves forwarded room keys, the reciever doesn't check if it requested the key from the forwarder*, September 2022. https://www.cve.org/CVERecord?id=CVE-2022-39252.

[L43] *CVE-2022-39254: When matrix-nio receives forwarded room keys, the receiver doesn't check if it requested the key from the forwarder*, September 2022. https://www.cve.org/CVERecord?id=CVE-2022-39254.

[L44] *CVE-2022-39255: Matrix iOS SDK vulnerable ton Olm/Megolm protocol confusion*, September 2022. https://www.cve.org/CVERecord?id=CVE-2022-39255.

[L45] *CVE-2022-39257: Matrix iOS SDK vulnerable to impersonation via forwarded Megolm sessions*, September 2022. https://www.cve.org/CVERecord?id=CVE-2022-39257.

[L46] *CVE-2022-39264: nheko vulnerable to secret poisoning using MITM on secret requests by the homeserver*, September 2022. https://www.cve.org/CVERecord?id=CVE-2022-39264.