Formal Analysis of Multi-Device Group Messaging in WhatsApp

Martin R. Albrecht[1], Benjamin Dowling[1], and Daniel Jones[2]*

[1] King's College London, {martin.albrecht,benjamin.dowling}@kcl.ac.uk
[2] Royal Holloway, University of London, dan.jones@rhul.ac.uk

Abstract. 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*. We utilise this description to state and prove the security guarantees that this subset of WhatsApp provides. Our analysis is performed within a variant of the Device-Oriented Group Messaging model, which we extend to support *device revocation*. We discuss how to interpret these results, including the security WhatsApp provides as well as its limitations.

* The research of Albrecht and Dowling was supported by UKRI Grant EP/X017524/1, the research of Jones was supported by UKRI Grant EP/S021817/1. Part of this work was done while Dowling was at the University of Sheffield.

Table of Contents

1 Introduction

Group messaging in WhatsApp is based on the Signal two-party protocol and the Sender Keys multiparty extension [66, 8]. To date, in the academic literature, the ground truth for answering the question of how these building blocks are composed precisely is established by the WhatsApp security whitepaper [66] or unofficial third-party protocol implementations, cf. [60, 9, 8, 31]. Based upon these, [8] – appears at RECSI 2020 [9] – represents the most ambitious attempt so far to model WhatsApp's security. This work includes a security proof and models the interactions between the two-party Signal channels and Sender Keys.

1.1 Contributions

However, while a major step forward in establishing the security guarantees of the WhatsApp protocol, this work still comes with a series of caveats and limitations.

Scope. The security experiment does not capture multiple groups, which are ubiquitous in practice. Also, it covers only the group messaging functionalities of the Sender Keys protocol and does not consider how a user's multiple devices are managed, the security guarantees this provides and how these interact with group messaging. On the other hand, other prior works have demonstrated that features such as session management, multi-device management and history sharing can interact with, and undermine, the security guarantees of the underlying channels in unexpected ways [2, 60, 3, 30, 28].

Contribution 1: *We provide a formal description of WhatsApp's group messaging protocol that covers session management, multi-device management and history sharing.*

Implementation. As mentioned above, the analysis in [8] is based on the WhatsApp whitepaper but not WhatsApp's implementation, leaving open the question of how accurate the whitepaper is.

Contribution 2: *We provide pseudocode for the WhatsApp protocol which is based on both the WhatsApp whitepaper [66] and examining the minified JavaScript source code of the WhatsApp web client.*[3]

Model. We express our description of WhatsApp within a variant of the Device-Oriented Group Messaging (DOGM) [3] model. In doing so, we find that the model lacks support for *device revocation.*

Contribution 3: *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 post-compromise security (PCS) guarantees of the underlying messaging channels and, in turn, the security of user messages.*

[3] We also performed some spot checks using decompiled WhatsApp Android application.

Multiple Channels. The analysis in [8] relies on the assumption that the number of underlying two-party channels between any two parties is one. This is the configuration implied both by Signal's original description of the Sender Keys protocol [50] and WhatsApp's security whitepaper [66]. This is not the case, however. The `libsignal` library allows multiple active Signal channels between a single pair of devices [53, 30, 28], all of which can be used to distribute Sender Keys sessions. We confirm, in this work, that this translates into WhatsApp's implementation.[4] As explored in [30, 28], adversaries with the ability to initialise new sessions can undermine the security guarantees of an existing channel post-compromise. As discussed in [3], this has additional and compounding effects on the PCS guarantees of the Sender Keys protocol. To our knowledge, this is the first analysis of this session management layer in the computational setting; prior works were in the symbolic model.

Contribution 4: *Building on prior work modelling the interaction between Sender Keys and the underlying two-party channels it relies on, we derive and prove the forward secrecy guarantees WhatsApp group messaging provides in the face of multiple active two-party channels between a single pair of devices.*

In Section 5.2 we sketch attacks that undermine expected PCS guarantees.

Recovery. We proceed to derive (and prove) the security guarantees of WhatsApp's multi-device secure group messaging within the DOGM model, capturing the interactions between the messaging channels, the multi-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.

Contribution 5: *We prove security guarantees of WhatsApp's group messaging protocol in the DOGM model, demonstrating how the session management, device management and history sharing interact with the security guarantees of the underlying messaging protocol.*

Remark 1. We sent our work to WhatsApp for comments and received feedback, including that our description of the protocol is correct, by WhatsApp engineers.

1.2 Scope & Limitations

The accuracy of our results relies on the accuracy of our description of WhatsApp in Section 3. This description is based upon a reverse engineering of the client's implementation, a process which is necessarily imperfect without access to 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 [66].

To make the analysis tractable, we sometimes simplify WhatsApp's functionality and aim to document all such simplifications as they arise in the text.

[4] This is also the case in Matrix [2, 3].

Additionally, our description is not complete: we focus on functionality that is relevant to our modelling. For example, while WhatsApp does provide immediate decryption [4][5] in both its pairwise and group channels, neither our description nor analysis reflect this (in contrast, [8] *does*). We, additionally, do not capture immediate decryption within the pairwise channels that are used to distribute Sender Keys sessions. A single cached message key for the pairwise channel allows the adversary to establish a new Sender Keys session; subverting the limitations on such cached keys.

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 [66, Page 24] and key transparency [67] 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.

Critically, group membership is not cryptographically authenticated in WhatsApp, as already established in prior works [60, 9, 8]. Clients display group membership and thus participants in group chats *could* potentially review their groups regularly to mitigate the effect of this. However, as discussed in [2], this puts an unduly burden on those who (have to) rely on WhatsApp, especially in a setting of up to 1024 members per group.[6] 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. In our model this is captured as a trivial win, which should be interpreted as: "If WhatsApp addresses this issue then the protocol achieves the stated security guarantees".

1.3 Related Work

Direct Analysis. WhatsApp utilises the Signal protocol [66] which has seen varying analyses of its pairwise messaging protocol [25] and the Double Ratchet algorithm it uses [4, 17]. WhatsApp's key transparency implementation [67] builds

[5] A protocol that provides immediate decryption allows sessions to decrypt out-of-order messages immediately after receiving them, while maintaining the ability to any skipped messages in the future. This is achieved in WhatsApp by caching the per-message key material for a limited number of messages.

[6] We note that the same issue was reported as a vulnerability in [2], in response to which the Matrix developers committed to fixing the issue, at the time of writing this work is ongoing.

upon [44] the design of PARAKEET [49]. WhatsApp's application state synchronisation feature utilises [67] the LtHash homomorphic hashing algorithm [48, 13]. [31] analyse the security of WhatsApp's end-to-end encrypted backups.

Multi-Device Messaging. WhatsApp allows users to participate through multiple devices. [32] present a systemisation of knowledge on multi-device secure messaging, surveying existing deployments and their approaches. [23] propose a design for multi-device secure messaging based upon the underlying Signal pairwise messaging protocol. [3] study the security of Matrix, a communication platform that supports multiple devices and group messaging simultaneously, and introduce the Device-Oriented Group Messaging formalism to capture such protocols. [26] propose Asynchronous Ratchet Trees to provide asynchronous group key exchange and, in doing so, suggest the use subtrees to capture a user with multiple devices.

Group Messaging. Prior analysis of the group messaging protocol utilised by WhatsApp, the Sender Keys component of Signal, is sparse. [60] identified a number of weaknesses in deployed group messaging protocols, including WhatsApp, and made initial progress towards defining the security goals we might expect from them. [8] present a formalism and analysis of the Sender Keys protocol. [58] present a systemisation of knowledge on game-based models for group key exchange. The continuous group key agreement (CGKA) line of work, initiated in [5], focus on group key exchange in the context of the Message Layer Security (MLS) standard. [6] lifts such analysis from key exchange to messaging.

Compromise Recovery. The notion of PCS was formalised in [27]. The work focused on the compromise of long-term key material with a focus on single sessions (or ratchet chains). The PCS guarantees of Signal pairwise channels were analysed in [25, 4]. The resulting security guarantees apply to adversaries that at some point become passive and do not consistently interfere with protocol execution after a compromise. [10], building on prior works such as [21], consider PCS against active attackers (in a single session setting). The work 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. [28] demonstrate that the PCS guarantees of Signal pairwise channels are undermined if clients allow multiple sessions between parties. [30] formally analyse this setting and derive the resulting PCS guarantees. [18] introduce a model to capture protocols' ability to recover from varying levels of state compromise, from session state to long-term identity keys. [29] 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.

Device Management and Revocation. The *public key orbit* formalism introduced in Section 4 is intended to capture the device management features of WhatsApp. As such, it covers similar ground to that of Keybase's sigchain [38] and the related constructions used by Zoom [19] and ELEKTRA [46]. 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).

Similar problems have also been studied in adjacent areas, such as certificate revocation [39], proactive security in signature schemes [20, 64], revocation within anonymous credential schemes [22] and recent proposals for revocation in FIDO2 [35].

2 Preliminaries

We specify the notation and define the primitives used in this work. The main body starts with Section 3 on page 17.

2.1 Notation

Constants. Unless otherwise specified, the length of bit-strings are specified in bits (not bytes). We 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 '$xs[9\mathsf{B} \to 128\mathsf{b}]$' is equivalent to '$xs[72 \to 128]$'. When specifying slices, from a bit-string for example, we sometimes only specify the type once (in which case all values are implicitly the same), i.e. '$xs[9 \to 128\mathsf{B}]$' is equivalent to '$xs[9\mathsf{B} \to 128\mathsf{B}]$'.

Pseudocode. We make some of our pseudocode choices explicit.

Assertions and $\perp$ The syntax '**assert** *boolean-expression*' will evaluate the given expression. Then, if the expression evaluates to **false**, halt execution of the function and return $\perp$.

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, '**assert unique match** ' asserts that there is exactly one matching value.

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 '$\mathit{argument\text{-}name} \xleftarrow{\mathit{default}} \mathit{default\text{-}value}$'. 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 '$\mathit{argument\text{-}name} \overset{\mathit{is}}{=} \mathit{pattern\text{-}or\text{-}value}$'. 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 '$x_{tmp}, y_{tmp}, z_{tmp} \leftarrow (0, 1, 2);$ **assert** $x_{tmp} = 1; x, y, z \leftarrow (0, 1, 2)$'.

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** $[a, b, c]:$ $\ldots$', (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. '$[a, b, c][0 \to 1]$' evaluates to the list $[a, b]$. Slices are inclusive of the endpoint given, i.e. '$xs[s \to e]$' 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. '$[a, b, c][0 \to 10]$' evaluates to $[a, b, c]$. The length of a list can be computed with the $\mathsf{len}()$ 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 $\leftarrow_{app}$ assignment operator, i.e. '$xs \leftarrow [a, b];\ xs \leftarrow_{app} c$' 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 '$(1, b)$'.

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** $\mathsf{condition}(b)]$'. 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 '$\forall\ x, y \in xs : a(x) \land b(y) \implies x$ **precedes** y **in** xs'.

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 unordered list of tuples: '$m \leftarrow \mathsf{Map}\{(x,0),(y,1),(z,2)\}$'. They also support: (a) iteration i.e. '**for** (k,v) **in** m: …', (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: '$m \leftarrow \mathsf{Map}\{(0,10),(1,20)\}; x \leftarrow m[2]$' sets the value of x to $\varnothing$.

Objects We allow the creation of objects using the Obj algorithm, 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. Thus, 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. '$\mathsf{Obj}(\mathtt{example}, x = 1).x$' evaluates to '$\mathtt{1}$'. 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 WhatsApp's use of Protocol Buffers to encode messages.

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 $\underline{\text{i}}$dentity keys of a device and '(xsk, xpk)' to refer to keys used for key e$\underline{\text{x}}$change. Since WhatsApp re-uses keys for key exchange and signature schemes, we avoid using terms such as *signing key* and *verification key*. Within pseudocode, we use the PK algorithm to calculate the public key from a private key.

Security Experiments

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.

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 and returns $\perp$. We suppress type checks as these are not cryptographically enforced.

Instant Win We expect the challenger to only set the *win* flag after they are sure the adversary has won the game. For example, they are expected to check all relevant security predicates are satisfied beforehand. It is now easy to argue that the output of an experiment will always be 1 once the *win* flag has been set. It follows that the distribution of the experiment is not affected by ending the game immediately. Thus, we specify that the experiments in this work end immediately after the *win* is set. Making this explicit enables us to simplify our reasoning and exposition in some cases.

Changes between Games We highlight the changes between consecutive games in our proofs. Consider the code snippet '**assert** $b = b'$' as the starting point for our example. Additions are marked as '**assert** $b = b' \wedge \mathsf{CONF}$' and changes are marked as '**assert** $b = 1 \wedge \mathsf{CONF}$'. Code that has been removed is marked as '**assert** $b = 1$ ~~$\wedge$ CONF~~'.

Security Reductions When building an adversary as part of a security reduction, we highlight where the adversary embeds a challenge from the outer experiment into the inner adversary's simulation, marked as '$c \leftarrow \mathsf{AEAD.Enc}(h, m)$'.

Schemes & Protocols. We represent each scheme and/or protocol as a tuple of algorithms that represent its external interface. The interface of individual algorithms is defined by the supplied type signature, while their behaviour is defined through pseudocode. In most cases, this tuple of algorithms does not sufficiently describe the scheme and we must rely on the surrounding prose and figures to define how these algorithms interact with one another and, importantly, how they should be used.

2.2 Primitives

Message Authentication Codes. A message authentication code (MAC) allows parties that share a secret to exchange messages that guarantee (a) only those parties possessing the shared secret may create valid messages, and (b) such messages have not been modified in transit. We follow the definition and security notions of [37].

Definition 1 (Message Authentication Code). *A MAC scheme consists of two algorithms MAC = (MAC.Gen, MAC.Tag, MAC.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 (Correctness of MAC Schemes). *A MAC scheme is correct if, for all $k \in \mathcal{K}$ and $m \in \mathcal{M}$,*

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

Definition 3 (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 advantage of winning the $\mathsf{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC},n_q}(\mathcal{A})$ security experiment detailed in Figure 1. The experiment is parameterised by n_q which limits the number of Tag queries that the adversary may make before submitting their guess.*

```
$\mathsf{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC},n_q}(\mathcal{A})$
1 : $k \leftarrow\$ \mathcal{K};\ qs \leftarrow \emptyset$
2 : $(m, t) \leftarrow \mathcal{A}^{\mathsf{Tag}}()$
3 : return $m \notin qs \wedge \mathsf{MAC.Verify}(k, m, t) = 1$

$\mathsf{Tag}(m)$
1 : $t \leftarrow \mathsf{MAC.Tag}(k, m)$
2 : $qs \leftarrow\cup \{m\}$
3 : return $t$
```

Fig. 1: The EUF-CMA security experiment for MAC schemes.

Definition 4 (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 advantage of winning the $\mathsf{Exp}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{MAC},\mathcal{A}_{\mathsf{SUF\text{-}CMA}}}(\mathcal{A})$ security experiment detailed in Figure 2. The experiment is parameterised by n_q which limits the number of Tag queries that the adversary may make before submitting their guess.*

```
$\mathsf{Exp}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{MAC},\mathcal{A}_{\mathsf{SUF\text{-}CMA}}}(\mathcal{A})$
1 : $k \leftarrow\$ \mathcal{K};\ qs \leftarrow \emptyset$
2 : $(m, t) \leftarrow \mathcal{A}^{\mathsf{Tag}}()$
3 : return $(m, t) \notin qs \wedge \mathsf{MAC.Verify}(k, m, t) = 1$

$\mathsf{Tag}(m)$
1 : $t \leftarrow \mathsf{MAC.Tag}(k, m)$
2 : $qs \leftarrow\cup \{(m, t)\}$
3 : return $t$
```

Fig. 2: The SUF-CMA security experiment for MAC schemes.

The protocols we study use the $\mathsf{HMAC}(k, m)$ (or $\mathsf{HMAC\text{-}SHA256}$) Hash-based Message Authentication Code (HMAC) constructed with $\mathsf{SHA256}$ [63], taking as input a key k and message m [40]. In this work we assume that $\mathsf{HMAC\text{-}SHA256}$ provides SUF-CMA security Definition 4. Throughout, we use HMAC to refer to HMAC instantiated with $\mathsf{SHA256}$, i.e. $\mathsf{HMAC\text{-}SHA256}$.

Key Derivation and Pseudorandom Functions. We follow [37, Definition 3.25] in defining pseudorandom functions.

Definition 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 $\ell(n)$:*

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

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

Let $\mathsf{Func}(n, m)$ denote the set of all functions mapping n-bit strings to m-bit strings, and define the security of pseudorandom functions as follows.

Definition 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 $\mathcal{A}$ has a negligible advantage of winning the* $\mathsf{Exp}^{\mathsf{PRF}}_{\mathsf{PRF}}(\mathcal{A})$ *security experiment detailed in Figure 3.*

```
Exp^PRF_PRF(A)
1 : $k \leftarrow\!\!\$\ \{0,1\}^{\lambda_k};\ f \leftarrow\!\!\$\ \mathsf{Func}(n, \ell(n))$
2 : $b \leftarrow\!\!\$\ \{0,1\};\ b' \leftarrow \mathcal{A}^{\mathsf{Eval}}()$
3 : return $b = b'$

Eval(x)
1 : $y_0 \leftarrow \mathsf{PRF}(k, x)$
2 : $y_1 \leftarrow f(x)$
3 : return $y_b$
```

Fig. 3: The security of PRF with seed length n and expansion factor $\ell(n)$.

WhatsApp uses both HMAC and HKDF as pseudorandom functions. In this context, we express HMAC (or $\mathsf{HMAC\text{-}SHA256}$) [43] as $\mathsf{HMAC}(k, m)$, taking a 256-bit seed k and a variable length message m before outputting a 256-bit string. Utilising [11], we assume that $\mathsf{HMAC\text{-}SHA256}$ is a secure pseudorandom function as defined by [37, Definition 3.25].

Similarly, WhatsApp uses $\mathsf{HKDF\text{-}SHA256}(s, k, i, \ell)$, an implementation of HKDF [41, 42], taking as input a public salt s, private key material k, info i and length ℓ before returning a bit-string of length ℓ. Throughout, we use HKDF to refer to HKDF instantiated with $\mathsf{SHA256}$, i.e. $\mathsf{HKDF\text{-}SHA256}$.

The definition of KDF security in [42] assumes that a fresh, uniformly random salt is used with each piece of private key material. However, as is common in practice, WhatsApp uses $\mathsf{HKDF\text{-}SHA256}$ in a number of places with a null salt. While we cannot directly assume that such use of $\mathsf{HKDF\text{-}SHA256}$ achieves KDF security as defined in [42], if the private key material being used is already pseudorandom, we do not require the extraction functionality of the KDF. Instead, we simply require that the HKDF itself acts as a pseudorandom function. For this, we use the following lemma.

Lemma 1 (PRF Security of HKDF). *HKDF-SHA256 [41, 42], when used with a constant salt and pseudorandom initial key material, realizes a secure pseudorandom function as defined in Definition 6. Specifically, letting* $\mathsf{PRF}^{l}_{\mathsf{HKDF}}(k, x) \coloneqq \mathsf{HKDF}(\varnothing, k, x, l)$*, the advantage of any probabilistic polynomial-time adversary* $\mathcal{A}$ *in winning the PRF security experiment against a challenger instantiated with* $\mathsf{PRF}^{\ell(256)}_{\mathsf{HKDF}}$ *is negligible in the number of queries. We denote this advantage by* $\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF}}(\lambda_k, n_q)$.

Intuitively, this requires that the first use of HMAC within HKDF, in the extraction phase, does not reduce the ability of the second use of HMAC in the expand phase, to produce a pseudorandom bit-string of the required length. When HKDF is used in this context (with pseudorandom key material and a constant salt) the extraction stage is using HMAC as a swap-PRF and the expansion stage uses HMAC as a PRF. Both of these requirements are satisfied in some cases, as detailed in [7], and we expect that Lemma 1 follows closely. Nonetheless, we rely on this assumption without proof.

Symmetric Encryption. WhatsApp uses AES-CBC for symmetric encryption[7]. 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 [12].

Definition 7 (Symmetric Encryption Scheme). *A symmetric encryption scheme consists of two algorithms SE = (SE.Enc, SE.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 8 (Correctness of Symmetric Encryption). *A symmetric encryption 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.

Definition 9 (IND-CPA Security of Symmetric Encryption). *A symmetric encryption scheme SE provides* indistinguishability under chosen-plaintext attack *(IND-CPA security) if any probabilistic polynomial-time adversary* $\mathcal{A}$ *has*

[7] They utilise additional encryption schemes in other parts of the system. We skip these schemes since they are not part of our analysis.

a negligible decision-advantage of winning the $\mathsf{Exp}^{\mathsf{IND\text{-}CPA}}_{\mathsf{SE},n_q,n_{ch}}(\mathcal{A})$ *security experiment detailed in Figure 4. 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 SE.Enc for which* $m_0 \neq m_1$*).*

```
$\mathsf{Exp}^{\mathsf{IND\text{-}CPA}}_{\mathsf{SE}}{}_{\Lambda_{\mathsf{IND\text{-}CPA}}}(\mathcal{A})$
1 : $b \leftarrow\$ \{0,1\}$
2 : $k \leftarrow\$ \mathcal{K}$
3 : $b' \leftarrow \mathcal{A}^{\mathsf{Enc}}(1^\lambda)$
4 : return $b = b'$
```

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

Fig. 4: The IND-CPA security experiment for symmetric encryption.

Authenticated Encryption with Associated Data. We adapt the AEAD definition introduced in [59] to our setting. In doing so, we must pay close attention to the use of nonces in both the experiment and practice. The definition in [59] forbids the re-use of nonces, requiring the adversary to be *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. WhatsApp takes 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 a different security notion.

In what follows, we define a one-time Authenticated Encryption with Associated Data (AEAD) 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 10 (One-Time AEAD Scheme). *A one-time AEAD scheme consists of two algorithms AEAD = (AEAD.Enc, AEAD.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* $\mathcal{C} \subseteq \{0,1\}^*$*.*

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

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

Definition 11 (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 $m \in \mathcal{M}$,*

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

and the ciphertext output by $\mathsf{AEAD.Enc}$ is equal in length to the plaintext given to it (as measured by the linear-time computable length function l).

Definition 12 (IND\$-CPA Security of One-Time AEAD Schemes). *A one-time AEAD scheme AEAD = ($\mathsf{AEAD.Enc}$, $\mathsf{AEAD.Dec}$) provides indistinguishability from random under chosen plaintext attack (IND\$-CPA) if any probabilistic polynomial-time adversary $\mathcal{A}$ has a negligible advantage of winning the $\mathsf{Exp}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{AEAD},1}(\mathcal{A})$ security experiment detailed in Figure 5.*

Definition 13 (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 advantage of winning the $\mathsf{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{AEAD},1}(\mathcal{A})$ security experiment detailed in Figure 5.*

```
$\mathsf{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{AEAD},1}(\mathcal{A})$
    $k \leftarrow\$ \mathcal{K};\ q \leftarrow \varnothing$
    $(h, c) \leftarrow \mathcal{A}^{\mathsf{Enc}}()$
    $m^\star \leftarrow \mathsf{AEAD.Dec}(k, h, c)$
    return $m^\star \neq \bot \wedge (h, c) \neq q$

$\mathsf{Enc}(h, m)$
    assert $q = \varnothing$
    $c \leftarrow \mathsf{AEAD.Enc}(k, h, m)$
    $q \leftarrow (h, c)$
    return $c$

$\mathsf{Exp}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{AEAD},1}(\mathcal{A})$
    $b \leftarrow\$ \{0,1\}$
    $k \leftarrow\$ \mathcal{K}$
    $m^\star \leftarrow \varnothing$
    $b' \leftarrow \mathcal{A}^{\mathsf{Enc}}()$
    return $b = b'$

$\mathsf{Enc}(h, m)$
    assert $m^\star = \varnothing$
    $m^\star \leftarrow m$
    $c_1 \leftarrow \mathsf{AEAD.Enc}(k, h, m)$
    $c_0 \leftarrow\$ \{0,1\}^{l(m)}$
    return $c_b$
```

Fig. 5: The IND\$-CPA and EUF-CMA security experiments for one-time AEAD.

We assume that WhatsApp's combined use of AES-CBC and HMAC-SHA256, as we describe in $\mathsf{WA\text{-}AEAD}$, provides one-time IND\$-CPA and EUF-CMA security. Prior work [14, 12, 15] demonstrates that this is a reasonable assumption.

Signature Schemes. WhatsApp uses the Curve25519 variant of XEdDSA [57], which we represent through $\mathsf{XEd} = (\mathsf{XEd.Gen}, \mathsf{XEd.Sign}, \mathsf{XEd.Verify})$ where $\mathsf{XEd.Gen}$ is equivalent to $\mathsf{XDH.Gen}$. We assume that XEd provides $\mathsf{SUF\text{-}CMA}$ security, i.e. $\mathsf{Adv}^{\mathsf{SUF\text{-}CMA},\mathcal{A}}_{\mathsf{XEd}}(\lambda) \leq \mathsf{negl}(\lambda)$ for any probabilistic polynomial-time algorithm $\mathcal{A}$, when the keys are used solely for digital signatures (see Remark 3).

Definition 14 (Signature Scheme). *A signature scheme* DS *consists of three probabilistic polynomial-time algorithms* $(\mathsf{DS.Gen}, \mathsf{DS.Sign}, \mathsf{DS.Verify})$ *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 15 ((Strong) Existential Unforgeability under Chosen Message Attack). *A signature scheme provides (strong) existential unforgeability under chosen message attack ((E/S)UF-CMA) if any polynomial-time adversary $\mathcal{A}$ has a negligible advantage of winning the security experiment detailed in Figure 6. We write $\mathsf{Adv}_{\mathsf{DS}}^{\mathsf{EUF\text{-}CMA},\mathcal{A}}(\lambda)$ and $\mathsf{Adv}_{\mathsf{DS}}^{\mathsf{SUF\text{-}CMA},\mathcal{A}}(\lambda)$.*

```
(E/S)UF-CMA
1 : $\mathcal{Q} \leftarrow \emptyset$;   $sk, pk \leftarrow \mathsf{DS.Gen}(1^\lambda)$ ;   $(m^\star, \sigma^\star) \leftarrow \mathcal{A}^{\mathsf{S}}(pk)$
2 : return $(\mu^\star, \cdot) \notin \mathcal{Q} \wedge \mathsf{DS.Verify}(pk, \sigma^\star, m^\star) = 1$   // EUF-CMA
3 : return $(m^\star, \sigma^\star) \notin \mathcal{Q} \wedge \mathsf{DS.Verify}(pk, \sigma^\star, m^\star) = 1$   // SUF-CMA

S(m)
1 : $\sigma \leftarrow \mathsf{DS.Sign}(sk, m)$
2 : $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(m, \sigma)\}$
3 : return $\sigma$
```

Fig. 6: The EUF-CMA and SUF-CMA security experiments for signature schemes.

Key Exchange. WhatsApp uses the XDH scheme for Curve25519-based Diffie-Hellman (DH) key exchange [16], consisting of 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.

WhatsApp uses XDH within the Signal protocol for secure pairwise messaging. Thus, our security analysis does not interact with the scheme directly. Instead, we model the underlying Signal pairwise channels as a multi-stage key exchange protocol (MSKE) protocol [25, Definition 1], and make the assumption that WhatsApp implements an MS-IND secure MSKE protocol. Our intermediate result for the composed DM scheme, Theorem 2, and our main result, Theorem 5, rely on [25, Thereom 1] which, in turn, relies on the Gap Diffie-Hellman (Gap DH) assumption [56] for Curve25519 and models the KDFs as random oracles.

As described in Section 5, we modify the MSKE formalism for our setting. We now briefly provide the (modified) syntax of such protocols, and refer the reader to [25, Section 4.2] for a security definition. We describe these changes, and justify the security of Signal pairwise channels within it, in prose in Section 5.

Definition 16 (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* 1^λ *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* $\pi^{'}$ *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* $\pi^{'}$ *and (possibly empty) outgoing protocol message* m'*.*

3 Multi-Device Group Messaging in WhatsApp

In this section, we describe the results of our effort reverse engineering the WhatsApp web client. In doing so, we identify and define the sub-protocols that, together, make up its multi-device group messaging functionality. The description in this section is a compressed and incomplete description of the functionality it aims to describe. This is a necessary result of translating a large, production code base into pseudocode amenable to cryptographic analysis. For example, we may simplify data structures, algorithms or functionality. We do our best to signpost such changes. In particular, we distinguish between the WhatsApp sub-protocols and our formalism of these also by name, to highlight the distinction.

3.1 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, $\mathsf{MD} = (\mathsf{MD.Setup}, \mathsf{MD.Link}, \mathsf{MD.Unlink}, \mathsf{MD.Refresh}, \mathsf{MD.Devices})$. 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 primary 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 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 7 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 $\mathsf{MD.Setup}$ in Figure 7). 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 3.2). The public key, ipk_p, is uploaded to the server to register a

```
$\mathsf{MD.Setup}()$
1 :  $isk_p, ipk_p \leftarrow\!\$\ \mathsf{XDH.Gen}()$
2 :  $md \leftarrow \mathsf{Obj}(\texttt{MD}, ipk_p, [ipk_p], 0, \varnothing, [\varnothing])$
3 :  $md.\sigma_{dl} \leftarrow \mathsf{XEd.Sign}(isk_p, \texttt{0x0602} \parallel [ipk_p] \parallel 0)$
4 :  $\textbf{return } isk_p, ipk_p, md$

$\mathsf{MD.Link}(\rho \overset{is}{=} \texttt{primary}, isk, md, ipk_c)$
1 :  $\texttt{MD}, ipk_p \overset{is}{=} \mathsf{PK}(isk), dl, \gamma, \sigma_{dl}, \Delta \leftarrow md\ ;\ dl \leftarrow_{app} ipk_c$
2 :  $\sigma_{p \rightarrow c} \leftarrow \mathsf{XEd.Sign}(isk, \texttt{0x0600} \parallel \gamma + 1 \parallel ipk_p \parallel ipk_c)$
3 :  $\Delta[ipk_c] \leftarrow \mathsf{Obj}(\texttt{DR}, \gamma + 1, ipk_p, ipk_c, \sigma_{p \rightarrow c}, \varnothing)$
4 :  $\sigma_{dl} \leftarrow \mathsf{XEd.Sign}(isk, \texttt{0x0602} \parallel dl \parallel \gamma + 1)$
5 :  $md \leftarrow \mathsf{Obj}(\texttt{MD}, ipk_p, dl, \gamma + 1, \sigma_{dl}, \Delta)$
6 :  $\textbf{return } md$

$\mathsf{MD.Link}(\rho \overset{is}{=} \texttt{companion}, isk, md, ipk_p)$
1 :  $ipk \leftarrow \mathsf{PK}(isk)\ ;\ \texttt{MD}, ipk_p \overset{is}{=} ipk_p, dl, \gamma, \sigma_{dl}, \Delta \leftarrow md$
2 :  $\texttt{DR}, \gamma, ipk_p \overset{is}{=} ipk_p, ipk_c \overset{is}{=} ipk, \sigma_{p \rightarrow c}, \sigma_{c \rightarrow p} \overset{is}{=} \varnothing \leftarrow \Delta[ipk]$
3 :  $\sigma_{c \rightarrow p} \leftarrow \mathsf{XEd.Sign}(isk, \texttt{0x0601} \parallel \gamma \parallel ipk_p \parallel ipk)$
4 :  $\Delta[ipk] \leftarrow \mathsf{Obj}(\texttt{DR}, \gamma, ipk_p, ipk, \sigma_{p \rightarrow c}, \sigma_{c \rightarrow p})$
5 :  $md \leftarrow \mathsf{Obj}(\texttt{MD}, ipk_p, dl, \gamma, \sigma_{dl}, \Delta)\ ;\ \textbf{return } md$

$\mathsf{MD.Unlink}(\rho \overset{is}{=} \texttt{primary}, isk, md, ipk_c)$
1 :  $ipk \leftarrow \mathsf{PK}(isk)$
2 :  $\texttt{MD}, ipk_p \overset{is}{=} ipk, dl, \gamma, \sigma_{dl}, \Delta \leftarrow md$
3 :  $dl \leftarrow [ipk_* \textbf{ for } ipk_* \textbf{ in } dl \textbf{ if } ipk_* \neq ipk_c]$
4 :  $\sigma_{dl} \leftarrow \mathsf{XEd.Sign}(isk, \texttt{0x0602} \parallel dl \parallel \gamma + 1)$
5 :  $md \leftarrow \mathsf{Obj}(\texttt{MD}, ipk, dl, \gamma + 1, \sigma_{dl}, \Delta)$
6 :  $\textbf{return } md$

$\mathsf{MD.Devices}(ipk_p, \gamma, md)$
1 :  $\textbf{assert } md.dl[0] = md.ipk = ipk_p \wedge\ md.\gamma \geq \gamma$
2 :  $m \leftarrow \texttt{0x0602} \parallel md.dl \parallel md.\gamma$
3 :  $\textbf{assert } \mathsf{XEd.Verify}(ipk_p, m, md.\sigma_{dl})$
4 :  $ipks_{\checkmark} \leftarrow \{ipk_p\}$
5 :  $\textbf{for } dr \in md.\Delta$
6 :      $m_0 \leftarrow \texttt{0x0600} \parallel dr.\gamma \parallel ipk_p \parallel dr.ipk_c$
7 :      $m_1 \leftarrow \texttt{0x0601} \parallel dr.\gamma \parallel ipk_p \parallel dr.ipk_c$
8 :      $b_0 \leftarrow \mathsf{XEd.Verify}(ipk_p, m_0, dr.\sigma_{p \rightarrow c})$
9 :      $b_1 \leftarrow \mathsf{XEd.Verify}(dr.ipk_c, m_1, dr.\sigma_{c \rightarrow p})$
10 :     $b_2 \leftarrow dr.ipk_c \neq ipk_p$
11 :     $b_3 \leftarrow (dr.ipk_c \in md.dl) \vee (dr.\gamma > md.\gamma)$
12 :     $\textbf{if } b_0 \wedge b_1 \wedge b_2 \wedge b_3\ :\ ipks_{\checkmark} \leftarrow\!\cup \{dr.ipk_c\}$
13 :  $\textbf{return } ipks_{\checkmark}$

$\mathsf{MD.Refresh}(\rho \overset{is}{=} \texttt{primary}, isk, md)$
1 :  $ipk \leftarrow \mathsf{PK}(isk)\ ;\ \texttt{MD}, ipk_p \overset{is}{=} ipk, dl, \gamma, \sigma_{dl}, \Delta \leftarrow md\ ;\ \sigma_{dl} \leftarrow \mathsf{XEd.Sign}(isk, \texttt{0x0602} \parallel dl \parallel \gamma + 1)$
2 :  $md \leftarrow \mathsf{Obj}(\texttt{MD}, ipk, dl, \gamma + 1, \sigma_{dl}, \Delta)\ ;\ \textbf{return } md$
```

Fig. 7: Pseudocode describing how WhatsApp manages user devices; the multi-device sub-protocol MD.

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[8] 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 [65] between the two devices,

[8] The security experiment in Section 7.3 allows the challenger to provide a trusted mapping between user identities and their primary identity key ipk_p without interference from the adversary.

ensuring the authenticity of later messages in the registration sub-protocol that are routed through the server. We describe this process in Section 3.5.

The companion device starts by generating their own identity keys, isk_c and ipk_c (see line 1 of $\mathsf{WA.NewCompanionDevice}$ in Figure 16). 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 $\{0, 1, \ldots, max\}$.[9] 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 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 $\mathsf{MD.Link}(\rho \overset{is}{=} \texttt{primary}, \ldots)$ in Figure 7).

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 $\mathsf{MD.Link}(\rho \overset{is}{=} \texttt{primary}, \ldots)$). 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 $\mathsf{MD.Link}(\rho \overset{is}{=} \texttt{companion}, \ldots)$). 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 $\mathsf{MD.Unlink}$ in Figure 7). In order for other devices to enact this revocation, it is important that they are made aware of this change.

[9] 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.

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 7). 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.*[10]

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[11], 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.

2) A copy of a single device record that accompanies pre-key ciphertexts from 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

[10] 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.

[11] Whenever a device initiates a new pairwise Signal channel (see Section 3.2) 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.

```
$\mathsf{ICDC.Generate}(ipk_p, \Gamma, mem, \mathcal{MD})$
$meta \leftarrow \mathsf{Map}\{\ \}$
for $ipk_{pr}$ in $mem$ :
    $ipks_{\checkmark} \leftarrow \mathsf{MD.Devices}(ipk_{pr}, \Gamma[ipk_{pr}], \mathcal{MD}[ipk_{pr}])$
    for $ipk_r$ in $ipks_{\checkmark}$ : $meta[ipk_r] \leftarrow \mathsf{Obj}(\texttt{ICDC}, \Gamma[ipk_p], \Gamma[ipk_{pr}])$
return $meta$
```

```
$\mathsf{ICDC.Process}(ipk_p, \Gamma, ipk_s, meta)$
$\Gamma[ipk_s] \leftarrow \mathsf{max}(\Gamma[ipk_s], meta.\gamma_s)$
$\Gamma[ipk_p] \leftarrow \mathsf{max}(\Gamma[ipk_p], meta.\gamma_r)$
return $\Gamma$
```

Fig. 8: Pseudocode describing how WhatsApp's ICDC information is computed.

new pairwise channel with another device, so that the recipient may verify the sender.[12] 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 $\mathsf{MD.Devices}$ in Figure 7 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).[13]

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 messaging messages (described in Section 3.3). 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.

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

[13] Verifying that the user's cryptographic identity maps to the person they expect to communicate with is not trivial. Section 3.4 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 7.

We describe this behaviour through two algorithms, $\mathsf{ICDC} = (\mathsf{ICDC.Generate}, \mathsf{ICDC.Process})$.

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

WhatsApp's whitepaper describes the ICDC information 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[14], (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*[15] 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 $\mathsf{ICDC.Generate}$ in Figure 8). 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.

[14] 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 $\mathsf{SHA256}$ function.

[15] 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) [66].

```
$\mathsf{DM.Init}(isk, n_e)$
// Generate and sign medium-term pre-key
$spk, ssk \leftarrow\!\!\$\ \mathsf{SIGNAL.MedTermKeyGen}()$
$\sigma_{spk} \leftarrow \mathsf{XEd.Sign}(isk, \mathtt{0x05} \,\|\, spk)$
// Generate ephemeral/one-time keys
for $i = 0, 1, 2, \ldots, n_e - 1$ :
    $epk_i, esk_i \leftarrow\!\!\$\ \mathsf{SIGNAL.EphemKeyGen}()$
$esks \leftarrow \{esk_i : 0 \le i < n_e\}$
$epks \leftarrow \{epk_i : 0 \le i < n_e\}$
// Key bundle for distribution by server
$skb \leftarrow \mathsf{Obj}(\mathtt{SKB}, \mathsf{PK}(isk), spk, \sigma_{spk}, epks)$
// Initialise our private DM protocol state
$ssts \leftarrow \mathsf{Map}\{\}$    // Signal session states by $ipk$
$pst \leftarrow \mathsf{Obj}(\mathtt{DM}, isk, ssk, esks, ssts)$
return $pst, skb$
```

```
$\mathsf{DM.Dec}(pst, skb_s, c_P)$
if $c_P.c_{kex}.type = \mathtt{pkmsg}$ :
    $pst, m \leftarrow {}^*\mathsf{DM.DecPreKeyMsg}(pst, skb_s, c_P)$
else :
    $pst, m \leftarrow {}^*\mathsf{DM.DecNormalMsg}(pst, skb_s, c_P)$
return $pst, m$
```

```
$\mathsf{DM.Enc}(pst, skb_r, m)$
// Unpack our protocol state and recipient's key bundle
$\mathtt{DM}, isk, ssk, esks, ssts \leftarrow pst$
$ipk_r, spk_r, \sigma_{spk,r}, [epk_r] \leftarrow skb_r$
// (server allocates a single ephemeral key $epk_r$ from list)
// Initiate new Signal session if we have no pre-existing one
if $(ssts[ipk_r] = \varnothing)$ :
    assert $\mathsf{XEd.Verify}(ipk_r, \mathtt{0x05} \,\|\, spk_r, \sigma_{spk,r})$
    $sst, \cdot \leftarrow\!\!\$\ \mathsf{SIGNAL.Activate}(isk, ssk, \mathtt{init}, ipk_r)$
    $sst, c_{kex} \leftarrow\!\!\$\ \mathsf{SIGNAL.Run}(isk, ssk, sst, (spk_r, \sigma_{spk,r}, epk_r))$
else :    // Otherwise use the active matching session
    $sst \leftarrow ssts[ipk_r][0]$
    $sst, c_{kex} \leftarrow\!\!\$\ \mathsf{SIGNAL.Run}(isk, ssk, sst, \varnothing)$
assert $sst.status[sst.stage] = \mathtt{accept}$
$c_{msg} \leftarrow \mathsf{WA\text{-}AEAD.Enc}(sst.k[sst.stage], c_{kex}, m)$
$ssts[ipk_r] \leftarrow {}^*\mathsf{DM.UpdateSession}(ssts[ipk_r], \varnothing, sst)$
$pst \leftarrow \mathsf{Obj}(\mathtt{DM}, isk, ssk, esks, ssts)$
return $pst, (c_{kex}, c_{msg})$
```

Fig. 9: 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 10 for descriptions of *DM.DecNormalMsg and *DM.DecPreKeyMsg.

Instead, clients seem to immediately invalidate the relevant device records.[16] 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 8). Next time the list of verified devices is checked, they will use this new minimum device list generation (as input into MD.Devices).

3.2 Pairwise Channels

To secure pairwise channels, WhatsApp uses X3DH [52] for the initial key exchange and the Double Ratchet [51] 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).

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 [53] and recently

[16] Since we choose not to model time in Section 7, this inconsistency does not affect our analysis.

```
$^*\mathsf{DM.DecPreKeyMsg}(pst, skb_s, c_P)$
    // Unpack the DM ciphertext into its requisite pieces
    $c_{kex}, c_{msg} \leftarrow c_P$
    // Unpack protocol state and sender's key bundle
    $\texttt{DM}, isk, ssk, esks, ssts \leftarrow pst$
    $\texttt{SKB}, ipk_s, spk_s, \sigma_{spk,s}, epks_s \leftarrow skb_s$
    // Look for session with matching initiator ephemeral pk
    $k, sst \leftarrow {}^*\mathsf{DM.FindSession}(ssts[ipk_s], c_{kex}.epk_{init})$
    if $sst = \varnothing$ :    // for new (to us) session
        $[esk] \leftarrow [esk'$ in $esks$ if $c_{kex}.epk_{resp} = \mathsf{PK}(esk')]$
        $sst, \cdot \leftarrow\$ \mathsf{SIGNAL.Activate}(isk, ssk, \texttt{resp}, ipk_s, esk)$
        $sst, \cdot \leftarrow\$ \mathsf{SIGNAL.Run}(isk, ssk, sst, c_{kex})$
        $esks \leftarrow [esk'$ in $esks$ if $c_{kex}.epk_{resp} \neq \mathsf{PK}(esk')]$
    else :    // for existing session
        $sst, \cdot \leftarrow\$ \mathsf{SIGNAL.Run}(isk, ssk, sst, c_{kex})$
    assert $sst.status[sst.stage] = \texttt{accept}$
    $m \leftarrow \mathsf{WA\text{-}AEAD.Dec}(sst.k[sst.stage], c_{kex}, c_{msg})$
    assert $m \neq \perp$
    $ssts[ipk_s] \leftarrow {}^*\mathsf{DM.UpdateSession}(ssts[ipk_s], k, sst)$
    $pst \leftarrow \mathsf{Obj}(\texttt{DM}, isk, ssk, esks, ssts)$
    return $pst, m$
```

```
$^*\mathsf{DM.DecNormalMsg}(pst, skb_s, c_P)$
    $c_{kex}, c_{msg} \leftarrow c_P$
    $\texttt{DM}, isk, ssk, esks, ssts \leftarrow pst$
    $\texttt{SKB}, ipk_s, spk_s, \sigma_{spk,s}, epks_s \leftarrow skb_s$
    // Try to decrypt using each session
    // in turn (in order of most recent use)
    for $k, sst$ enum in $ssts[ipk_s]$ :
        $sst, \cdot \leftarrow\$ \mathsf{SIGNAL.Run}(isk, ssk, sst, c_P)$
        $status = sst.status[sst.stage]$
        if $status \neq \texttt{accept}$ : continue
        $m \leftarrow \mathsf{WA\text{-}AEAD.Dec}($
            $sst.k[sst.stage], c_{kex}, c_{msg})$
        if $m = \perp$ : continue    // failure
        else : break    // success
    assert $m \neq \perp$    // all failed
    // Save as the active session (at index 0)
    $ssts[ipk_s] \leftarrow {}^*\mathsf{DM.UpdateSession}($
        $ssts[ipk_s], k, sst)$
    $pst \leftarrow \mathsf{Obj}(\texttt{DM}, isk, ssk, esks, ssts)$
    return $pst, m$
```

```
$^*\mathsf{DM.FindSession}(ssts, epk_{init})$
    $match \leftarrow (\varnothing, \varnothing)$
    for $k, sst$ enum in $ssts$
        if $sst.epk_{init} = epk_{init}$ :
            assert $match = (\varnothing, \varnothing)$;  $match \leftarrow (k, sst)$
    return $match$
```

```
$^*\mathsf{DM.UpdateSession}(ssts, k, sst)$
    if $k \neq \varnothing$ :
        return $[sst] \parallel ssts[0 \rightarrow k-1] \parallel ssts[k+1 \rightarrow 39]$
    else :
        return $[sst] \parallel ssts[0 \rightarrow 39]$
```

Fig. 10: Helper functions completing our description of DM (cf. Figure 9).

analysed in the symbolic setting [30]. 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 [28, 30].

We capture WhatsApp's pairwise device-to-device communications, including their use of multiple channels, with the scheme $\mathsf{DM} = (\mathsf{DM.Init}, \mathsf{DM.Enc}, \mathsf{DM.Dec})$ in Figure 9. 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 pairwise channels internally. It initialises the channels, selects the appropriate channel to use, and rotates medium- and short-term keys as needed.

- $pst, skb \leftarrow\$ \mathsf{DM.Init}(isk, n_e)$ takes as input a private identity key, isk, and the number of ephemeral keys to generate, n_e. It proceeds by generating a key bundle consisting of 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.

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

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

Fig. 11: Pseudocode describing the AEAD scheme used by WhatsApp when sending messages over pairwise channels, WA-AEAD.

- $pst, c \leftarrow \mathsf{DM.Enc}(pst, skb_r, m)$ 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.
- $pst, m \leftarrow \mathsf{DM.Dec}(pst, skb_s, c_P)$ 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 existing pairwise 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 [53] but not in [66].

DM.Init captures a device's initial setup. It takes the devices 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 9, does not capture this ability. We discuss this decision further in Section 5. DM.Enc encrypts a message, taking as input an existing protocol state, pst, the recipients key bundle, skb_r, and the message as input before returning an updated protocol state, pst, and the ciphertext, c_P. The device unpacks the recipient's key bundle 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 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 two-party protocol as input into an AEAD scheme before saving any changes to the session 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 channel to perform the continuous key exchange. $\mathsf{DM.Dec}$ decrypts a message, taking as input an existing protocol state, pst, the senders key bundle, skb_s, and the ciphertext to be decrypted, c_P, before returning an updated protocol state, pst, and 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 pairwise ciphertexts are handled separately, in $^*\mathsf{DM.DecPreKeyMsg}$ and $^*\mathsf{DM.DecNormalMsg}$ 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 $^*\mathsf{DM.DecPreKeyMsg}$). 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[17] to each session in the order of most recent use (see lines 4 to 11 of $^*\mathsf{DM.DecNormalMsg}$). 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 $^*\mathsf{DM.UpdateSession}$).

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 the multi-stage key indistinguishability security model ($\mathsf{MS\text{-}IND}$) and results of [25], modelling the underlying Signal pairwise channels as a MSKE [25, Definition 1] with SIGNAL = ($\mathsf{SIGNAL.KeyGen}$, $\mathsf{SIGNAL.MedTermKeyGen}$, $\mathsf{SIGNAL.Activate}$, $\mathsf{SIGNAL.Run}$). Ciphertexts contain a '*type*' field that is set to the value `prekey` when it contains a pre-key message. 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 $\mathsf{SIGNAL.KeyGen}$. Additional minor changes are required for our security analysis in Section 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 $\mathsf{WA\text{-}AEAD}$ and detail in Figure 11.

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 3.2) 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 3.1) to determine the list of devices representing

[17] 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 [3, 2].

their communicating partner, then distribute the same application message over multiple independent Signal two-party channels.

3.3 Group Messaging

Group messaging is achieved through the *Sender Keys* multiparty extension to the Signal two-party protocol. Introduced in [50], and previously analysed within [8], 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 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.

Consider the UNI scheme described in Figure 12. It consists of four algorithms, $\mathsf{UNI} = (\mathsf{UNI.Init}, \mathsf{UNI.Enc}, \mathsf{UNI.Dec}, \mathsf{UNI.GenInbound})$, 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.

- $ust_{out}, ust_{in} \leftarrow\$ \mathsf{UNI.Init}()$ generates a new unidirectional Sender Keys session and outputs the outbound session, ust_{out}, and inbound session, ust_{in}.
- $ust_{out}, c \leftarrow \mathsf{UNI.Enc}(ust_{out}, m)$ takes as input an outbound session, ust_{out}, and plaintext message, m, before outputting an updated outbound session, ust_{out}, and the resulting ciphertext, c.
- $ust_{in}, m \leftarrow \mathsf{UNI.Dec}(ust_{in}, (c_U, \sigma_U))$ takes as input an inbound session, ust_{in}, and signed ciphertext, (c_U, σ_U), before outputting an updated inbound session, ust_{in}, and the resulting message, c.
- $ust_{in} \leftarrow \mathsf{UNI.GenInbound}(ust_{out})$ takes as input an outbound session, ust_{out}, and calculates its inbound counterpart, ust_{in}.

The sender initialises a new session with $\mathsf{UNI.Init}$. 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, 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).[18] 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).[19] 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 SK = (SK.Init, SK.Add, SK.Rem, SK.Enc, SK.Dec) scheme which we detail in Figure 13.

[18] 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 ck) 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.

[19] 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.

```
UNI.Init()
    $usid \leftarrow\$ \{1, 2, \ldots, 2^{31}\};\ z \leftarrow 0$
    $ck \leftarrow\$ \{0,1\}^{256};\ (gsk, gpk) \leftarrow\$ \mathsf{XDH.Gen}()$
    $ust_{out} \leftarrow \mathsf{Obj}(\texttt{UNI-OUT}, usid, z, ck, gsk)$
    $ust_{in} \leftarrow \mathsf{Obj}(\texttt{UNI-IN}, usid, z, ck, gpk)$
    $\textbf{return } (ust_{out}, ust_{in})$

UNI.Enc($ust_{out}, m$)
    $\texttt{UNI-OUT}, usid, z, ck, gsk \leftarrow ust_{out}$
    $z \leftarrow z + 1$
    $mk \leftarrow \mathsf{HMAC}(ck, \texttt{0x01})$
    $ck \leftarrow \mathsf{HMAC}(ck, \texttt{0x02})$
    $k \leftarrow \mathsf{HKDF}(\varnothing, mk, \texttt{WhisperGroup}, 50\mathsf{B})$
    $iv, ek \leftarrow k[0 \to 15\mathsf{B}], k[16 \to 47\mathsf{B}]$
    $c \leftarrow \mathsf{AES\text{-}CBC.Enc}(ek, iv, m)$
    $c_U \leftarrow \mathsf{Obj}(\texttt{UNI-CTXT}, usid, z, c)$
    $\sigma_U \leftarrow \mathsf{XEd.Sign}(gsk, c_U)$
    $ust_{out} \leftarrow \mathsf{Obj}(\texttt{UNI-OUT}, usid, z, ck, gsk)$
    $\textbf{return } ust_{out}, (c_U, \sigma_U)$

UNI.GenInbound($ust_{out}$)
    $\texttt{UNI-OUT}, usid, z, ck, gsk \leftarrow ust_{out}$
    $ust_{in} \leftarrow \mathsf{Obj}(\texttt{UNI-IN}, usid, z, ck, \mathsf{PK}(gsk))$
    $\textbf{return } ust_{in}$

UNI.Dec($ust_{in}, (c_U, \sigma_U)$)
    $\texttt{UNI-IN}, usid, z, ck, gpk \leftarrow ust_{in}$
    $\textbf{assert } usid = c_U.usid\ \wedge z = c_U.z$
    $\textbf{assert } \mathsf{XEd.Verify}(gpk, c_U, \sigma_U)$
    $mk \leftarrow \mathsf{HMAC}(ck, \texttt{0x01})$
    $ck \leftarrow \mathsf{HMAC}(ck, \texttt{0x02})$
    $k \leftarrow \mathsf{HKDF}(\varnothing, mk, \texttt{WhisperGroup}, 50\mathsf{B})$
    $iv, ek \leftarrow k[0 \to 15\mathsf{B}], k[16 \to 47\mathsf{B}]$
    $m \leftarrow \mathsf{AES\text{-}CBC.Dec}(ek, iv, c_U.c)$
    $\textbf{assert } m \neq \bot$
    $z \leftarrow z + 1$
    $ust_{in} \leftarrow \mathsf{Obj}(\texttt{UNI-IN}, usid, z, ck, gpk)$
    $\textbf{return } (ust_{in}, m)$
```

Fig. 12: Pseudocode description of the unidirectional channels used by WhatsApp for group messaging, the ratcheted symmetric signcryption scheme UNI.

- $skst \leftarrow\$ \mathsf{SK.Init}(\rho, ipk, mem)$ 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, $meta$, 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, $skst$.

- $sk st \leftarrow \mathsf{SK.Rem}(sk st, ipk_\star)$ removes a device as a recipient from a sending session. It takes as input the sending session state, $sk st$, and the device's public identity key, ipk, then outputs an updated sending session state, $sk st$.

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 3.2). 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 $\mathsf{SK.Add}$ and lines 8 to 12 of $\mathsf{SK.Enc}$).[20] 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 $\mathsf{SK.Rem}$ and lines 2 to 7 of $\mathsf{SK.Enc}$). 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.[21] This could be implemented, for example, by authenticating the sender and checking their identity against the server-provided member list before accepting the message.[22]

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 13, 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

[20] 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 [9, 8, 2, 3].

[21] Since our inspection of the implementation was not exhaustive, we are not confident in claiming that this functionality does not exist.

[22] 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.

```
$\mathsf{SK.Init}(\rho \overset{is}{=} \mathtt{snd}, ipk, mem)$
    $new \leftarrow [\,];\ refresh? \leftarrow \mathbf{false};\ t \leftarrow 0;\ ust_{out} = \varnothing$
    $skst \leftarrow \mathsf{Obj}(\mathtt{SK\text{-}SND}, mem, new, refresh?, t, ust_{out})$
    return $skst$
```

```
$\mathsf{SK.Init}(\rho \overset{is}{=} \mathtt{rcv}, ipk_s, \varnothing)$
    $usts_{in} = [\,]$   // sent with first ciphertext
    $skst \leftarrow \mathsf{Obj}(\mathtt{SK\text{-}RCV}, ipk_s, t, usts_{in})$
    return $skst$
```

```
$\mathsf{SK.Enc}(skst \overset{is}{=} \mathsf{Obj}(\mathtt{SK\text{-}SND}, \cdot), pst, \mathcal{SKB}, meta, m)$
    $\mathtt{SK\text{-}SND}, mem, new, refresh?, t, ust_{out} \leftarrow skst;\ \overrightarrow{c_P} \leftarrow [\,]$
    if $(ust_{out} = \varnothing) \cup (refresh? = \mathbf{true})$:
        $ust_{out}, ust_{in} \leftarrow\!\$\ \mathsf{UNI.Init}()$
        $t \leftarrow t + 1$
        for $(ipk_r$ in $mem)$:
            $M \leftarrow \mathsf{Obj}(\mathtt{SK\text{-}PTXT}, meta[ipk_r], \mathsf{Obj}(\mathtt{UNI\text{-}IN}, ust_{in}))$
            $pst, c_P \leftarrow\!\$\ \mathsf{DM.Enc}(pst, \mathcal{SKB}[ipk_r], m);\ \overrightarrow{c_P} \leftarrow_{app} c_P$
    else :   // If not, distribute the existing session to new members.
        $ust_{in} \leftarrow \mathsf{UNI.GenInbound}(ust_{out})$
        for $(ipk_r$ in $new)$:
            $M \leftarrow \mathsf{Obj}(\mathtt{SK\text{-}PTXT}, meta[ipk_r], \mathsf{Obj}(\mathtt{UNI\text{-}IN}, ust_{in}))$
            $pst, c_P \leftarrow\!\$\ \mathsf{DM.Enc}(pst, \mathcal{SKB}[ipk_r], m);\ \overrightarrow{c_P} \leftarrow_{app} c_P$
    $new \leftarrow [\,];\ ust_{out}, (c_U, \sigma_U) \leftarrow \mathsf{UNI.Enc}(ust_{out}, M)$
    $skst \leftarrow \mathsf{Obj}(\mathtt{SK\text{-}SND}, mem, new, refresh?, t, ust_{out})$
    return $skst, pst, \overrightarrow{c_P}, (c_U, \sigma_U)$
```

```
$\mathsf{SK.Dec}(skst \overset{is}{=} \mathsf{Obj}(\mathtt{SK\text{-}RCV}, \cdot), pst, skb_s, c_P, (c_U, \sigma_U))$
    if $(c_P \neq \varnothing)$:
        assert $skb_s.ipk = skst.ipk_s$
        $pst, M \leftarrow \mathsf{DM.Dec}(pst, skb_s, c_P)$
        if $(\mathsf{len}(skst.usts_{in}) > 4)$:
            $skst.usts_{in} \leftarrow skst.usts_{in}[0 \rightarrow 3] \parallel [M.ust_{in}]$
        else :
            $skst.usts_{in} \leftarrow_{app} M.ust_{in}$
        $skst.t \leftarrow skst.t + 1$
        $meta \leftarrow M.meta$
    else : $meta \leftarrow \varnothing$
    $i, ust_{in} \leftarrow {}^*\mathsf{SK.FindSession}(skst.usts_{in}, c_U.usid)$
    $ust_{in}, m \leftarrow \mathsf{UNI.Dec}(ust_{in}, (c_U, \sigma_U))$
    $skst.usts_{in}[i] \leftarrow ust_{in}$
    return $skst, pst, meta, m$
```

```
$\mathsf{SK.Add}(skst \overset{is}{=} \mathsf{Obj}(\mathtt{SK\text{-}SND}, \cdot), ipk_*)$
    $skst.mem \leftarrow_{app} ipk_*$
    $skst.new \leftarrow_{app} ipk_*$
    return $skst$
```

```
$\mathsf{SK.Rem}(skst \overset{is}{=} \mathsf{Obj}(\mathtt{SK\text{-}SND}, \cdot), ipk_*)$
    $skst.mem \leftarrow skst.mem \setminus [ipk_*]$
    if $ipk_*$ in $skst.new$:
        $skst.new \leftarrow skst.new \setminus [ipk_*]$
    else : $skst.refresh? \leftarrow \mathbf{true}$
    return $skst$
```

```
${}^*\mathsf{SK.FindSession}(usts_{in}, usid)$
    $i \leftarrow 0$
    for $(ust_{in}$ in $usts_{in})$:
        if $(ust_{in}.usid = usid)$:
            return $(i, ust_{in})$
        $i \leftarrow i + 1$
    assert false
```

Fig. 13: Pseudocode describing WhatsApp's variant of the Sender Keys protocol, capturing rotation of unidirectional channels to manage recipients.

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 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 [34].

Group management. 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 [60, 9, 8].

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 3.5 reflects the former, while our security analysis in Section 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. 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 this functionality 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 $\mathsf{ATTACH.Enc}$ and $\mathsf{ATTACH.Dec}$ 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 $\mathsf{ATTACH.Dec}$ 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.

```
ATTACH.Enc($m, t$)
1 :  $r \leftarrow\$ \{0,1\}^{32\mathsf{B}}$
2 :  $k \leftarrow \mathsf{HKDF}(\varnothing, r, t, 112\mathsf{B})$
3 :  $aiv \leftarrow k[0 \rightarrow 15\mathsf{B}];\ aek \leftarrow k[16 \rightarrow 47\mathsf{B}]$
4 :  $ahk \leftarrow k[48 \rightarrow 79\mathsf{B}];\ ark \leftarrow k[80 \rightarrow 111\mathsf{B}]$
5 :  $c \leftarrow \mathsf{AES\text{-}CBC.Enc}(aek, aiv, m)$
6 :  $\tau \leftarrow \mathsf{HMAC}(ahk, c)[0 \rightarrow 9\mathsf{B}]$
7 :  $c' \leftarrow c[16\mathsf{B} \rightarrow \ldots]$   // remove iv
8 :  $h \leftarrow \mathsf{SHA256}(c' \parallel \tau)$
9 :  $m_{ptr} \leftarrow \mathsf{Obj}(\mathsf{ATTACH}, t, r, h)$
10 : return $m_{ptr}, c', \tau$
```

```
ATTACH.Dec($m_{ptr}, c', \tau$)
1 :  $(t, r, h) \leftarrow \mathsf{Obj}(\mathsf{ATTACH}, m_{ptr})$
2 :  if $(h = \mathsf{SHA256}(c' \parallel \tau))$
3 :    $k \leftarrow \mathsf{HKDF}(\varnothing, r, t, 112\mathsf{B})$
4 :    $aiv \leftarrow k[0 \rightarrow 15\mathsf{B}];\ aek \leftarrow k[16 \rightarrow 47\mathsf{B}]$
5 :    $ahk \leftarrow k[48 \rightarrow 79\mathsf{B}];\ ark \leftarrow k[80 \rightarrow 111\mathsf{B}]$
6 :    if $(\tau = \mathsf{HMAC}(ahk, aiv \parallel c')[0 \rightarrow 9\mathsf{B}])$
7 :      $m \leftarrow \mathsf{AES\text{-}CBC.Dec}(aek, aiv, c')$
8 :      return $m$
9 :  return $\perp$
```

Fig. 14: Pseudocode describing how attachments are secured in WhatsApp.

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 [66].

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 '$\rho =$ `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 '$\rho =$ `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.

```
HS.Share($\rho \overset{is}{=}$ primary, $pst, skb, T$)
1 : $m_{ptr}, c_{hist}, \tau_{hist} \leftarrow\$$ ATTACH.Enc($T$,
2 :     WhatsApp History Keys)
3 : $pst, c_P \leftarrow\$$ DM.Enc($pst, skb, m_{ptr}$)
4 : return $pst, c_P, c_{hist}, \tau_{hist}$
```

```
HS.Receive($\rho \overset{is}{=}$ companion, $pst, skb, c_P, c_{hist}, \tau_{hist}$)
1 : $pst, m_{ptr} \leftarrow$ DM.Dec($pst, skb, c_P$)
2 : assert $m_{ptr}.t =$ WhatsApp History Keys
3 : $T \leftarrow$ ATTACH.Dec($m_{ptr}, c_{hist}, \tau_{hist}$)
4 : return $pst, T$
```

Fig. 15: Pseudocode describing how history is shared in WhatsApp.

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 our informal description in this section, with explicit pseudocode in Section 3.5.

3.4 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 [45], whitepaper [68], open-source implementation `akd` [55] and the academic works it is built upon [49, 24, 54].

3.5 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 16. 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 3.4, 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.

```
$\mathsf{WA.NewPrimaryDevice}()$
    $isk, ipk, md \leftarrow\$ \mathsf{MD.Setup}()$
    $pst, skb \leftarrow\$ \mathsf{DM.Init}(isk, n_e)$
    $\Gamma \leftarrow \mathsf{Map}\{(ipk, md.\gamma)\};\ gi \leftarrow 0$
    $wst \leftarrow \mathsf{Obj}(\texttt{WA}, \texttt{primary}, isk, ipk, pst, \Gamma, md, gi)$
    $\textbf{return}\ wst, skb$

$\mathsf{WA.NewCompanionDevice}()$
    $isk, ipk \leftarrow\$ \mathsf{XDH.Gen}()$
    $pst, skb \leftarrow\$ \mathsf{DM.Init}(isk, n_e);\ \Gamma \leftarrow \mathsf{Map}\{\}$
    $lk \leftarrow\$ \{0,1\}^{32\mathsf{B}};\ link \leftarrow (ipk, lk);\ gi \leftarrow 0$
    $wst \leftarrow \mathsf{Obj}(\texttt{WA}, \texttt{companion}, isk, \varnothing, pst, \Gamma, \varnothing, gi, lk)$
    $\textbf{return}\ wst, qr, skb$

$\mathsf{WA.LinkDevice}(wst \overset{is}{=} \mathsf{Obj}(\texttt{WA}, \texttt{primary}, \cdot), link \overset{is}{=} (ipk_c, lk))$
    $wst.md \leftarrow \mathsf{MD.Link}(\texttt{primary}, wst.isk, wst.md, ipk_c)$
    $dr \leftarrow wst.md.\Delta[ipk_c];\ \tau_{link} \leftarrow \mathsf{HMAC}($
        $lk, \mathsf{Obj}(\texttt{L-DATA}, dr.\gamma \parallel dr.ipk_c \parallel dr.ipk_p \parallel dr.\sigma_{p \to c}))$
    $\textbf{return}\ wst, wst.md, \tau_{link}$

$\mathsf{WA.LinkDevice}(wst \overset{is}{=} \mathsf{Obj}(\texttt{WA}, \texttt{companion}, \cdot), md, \tau_{link})$
    $dr \leftarrow wst.md.\Delta[\mathsf{PK}(wst.isk)]$
    $\textbf{assert}\ \tau_{link} = \mathsf{HMAC}($
        $lk, \mathsf{Obj}(\texttt{L-DATA}, dr.\gamma \parallel \mathsf{PK}(wst.isk) \parallel dr.ipk_p \parallel dr.\sigma_{p \to c}))$
    $wst.md \leftarrow \mathsf{MD.Link}(\texttt{companion}, wst.isk, md, dr.ipk_p)$
    $\textbf{return}\ wst, wst.md$

$\mathsf{WA.UnlinkDevice}(wst \overset{is}{=} \mathsf{Obj}(\texttt{WA}, \texttt{primary}, \cdot), ipk_c)$
    $wst.md \leftarrow \mathsf{MD.Unlink}(\texttt{primary}, wst.isk, wst.md, ipk_c)$
    $\textbf{return}\ wst, wst.md$

$\mathsf{WA.RefreshDeviceList}(wst \overset{is}{=} \mathsf{Obj}(\texttt{WA}, \texttt{primary}, \cdot))$
    $wst.md \leftarrow \mathsf{MD.Refresh}(\texttt{primary}, wst.isk, wst.md)$
    $\textbf{return}\ wst, wst.md$

$\mathsf{WA.NewGroup}(wst \overset{is}{=} \mathsf{Obj}(\texttt{WA}, \cdot), mem, \mathcal{MD})$
    $r_1 \leftarrow\$ \{0,1\}^{16};\ r_2 \leftarrow\$ \{0,1\}^{16}$
    $wst.gi \leftarrow wst.gi + 1$
    $gid \leftarrow \mathsf{Obj}(\texttt{GID}, r_1, r_2, gi)$
    $wst \leftarrow \mathsf{WA.JoinGroup}(wst, gid, mem, \mathcal{MD})$
    $\textbf{return}\ wst, gid$

$\mathsf{WA.JoinGroup}(wst \overset{is}{=} \mathsf{Obj}(\texttt{WA}, isk, \cdot), gid, mem, \mathcal{MD})$
    $ipk \leftarrow \mathsf{PK}(isk)$
    $\textbf{assert}\ ipk\ \textbf{in}\ mem$
    $wst.mem[gid] \leftarrow mem$
    $wst.skts_{snd}[gid] \leftarrow\$ \mathsf{SK.Init}(\texttt{snd}, ipk, [\,])$
    $\textbf{for}\ ipk_{p*}\ \textbf{in}\ wst.mem[gid]:$
        $wst \leftarrow \mathsf{WA.AddMember}(wst, gid, ipk_{p*}, \mathcal{MD}[ipk_{p*}])$
    $\textbf{return}\ wst$

$\mathsf{WA.AddMember}(wst \overset{is}{=} \mathsf{Obj}(\texttt{WA}, \cdot), gid, ipk_*, md_*)$
    $wst.mem[gid] \leftarrow\cup \{ipk_{p*}\}$
    $ipks_{\checkmark} \leftarrow \mathsf{MD.Devices}(ipk_{p*}, wst.\Gamma[ipk_{p*}], md_*)$
    $\textbf{for}\ ipk_{\dagger}\ \textbf{in}\ ipks_{\checkmark}:$
        $wst.skts_{snd}[gid] \leftarrow \mathsf{SK.Add}(wst.skts_{snd}[gid], ipk_{\dagger})$
        $wst.skts_{rcv}[gid, ipk_{p*}, ipk_{\dagger}] \leftarrow\$ \mathsf{SK.Init}(\texttt{rcv}, ipk_{\dagger})$
    $\textbf{return}\ wst$

$\mathsf{WA.RemoveMember}(wst \overset{is}{=} \mathsf{Obj}(\texttt{WA}, \cdot), gid, ipk_*)$
    $\textbf{for}\ (ipk_{\dagger}, \cdot)\ \textbf{in}\ wst.skts_{rcv}[gid, ipk_{p*}, \cdot]:$
        $wst.skts_{snd}[gid] \leftarrow \mathsf{SK.Rem}(wst.skts_{snd}[gid], ipk_{\dagger})$
        // (inbound sessions from $ipk_*$ are not removed)
    $wst.mem[gid] \leftarrow wst.mem[gid] \setminus \{ipk_{p*}\}$
    $\textbf{return}\ wst$

$\mathsf{WA.SendGroup}(wst \overset{is}{=} \mathsf{Obj}(\texttt{WA}, isk, ipk_p, \Gamma, mem, \cdot), gid, \mathcal{MD}, \mathcal{SKB}, m)$
    $wst \leftarrow {}^*\mathsf{WA.ProcessDL}(wst, gid, \mathcal{MD})$
    $meta \leftarrow \mathsf{ICDC.Generate}(ipk_p, \Gamma, mem[gid], \mathcal{MD})$
    $skst_{snd} \leftarrow wst.skts_{snd}[gid];\ pst \leftarrow wst.pst$
    $skst_{snd}, pst, \overrightarrow{c_P}, c_U \leftarrow\$ \mathsf{SK.Enc}(skst_{snd}, pst, \mathcal{SKB}, meta, m)$
    $wst.skts_{snd}[gid] \leftarrow skst_{snd};\ wst.pst \leftarrow pst$
    $\textbf{return}\ wst, \overrightarrow{c_P}, c_U$

$\mathsf{WA.ReceiveGroup}(wst \overset{is}{=} \mathsf{Obj}(\texttt{WA}, \cdot), gid, ipk_s, c_P, c_U, \mathcal{MD}, \mathcal{SKB})$
    $skst_{rcv} \leftarrow wst.skts_{rcv}[gid, ipk_s];\ pst \leftarrow wst.pst$
    $\textbf{if}\ skts_{rcv}[gid, ipk_s] = \bot:\ \textbf{return}\ wst, \bot$
    $skst_{rcv}, pst, meta, m \leftarrow \mathsf{SK.Dec}(skst_{rcv}, pst, \mathcal{SKB}[ipk_s], c_P, c_U)$
    $\textbf{if}\ m = \bot:\ \textbf{return}\ wst, \bot$
    $wst.skts_{rcv}[gid, ipk_s] \leftarrow skst_{rcv};\ wst.pst \leftarrow pst$
    $wst.\Gamma \leftarrow \mathsf{ICDC.Process}(wst.ipk_p, wst.\Gamma, ipk_s, meta, \mathcal{MD})$
    $wst \leftarrow {}^*\mathsf{WA.ProcessDL}(wst, gid, \mathcal{MD})$
    $\textbf{if}\ wst.skts_{rcv}[gid, ipk_s] = \bot:\ \textbf{return}\ wst, \bot$
    $\textbf{return}\ wst, m$

${}^*\mathsf{WA.ProcessDL}(wst \overset{is}{=} \mathsf{Obj}(\texttt{WA}, \cdot), gid, \mathcal{MD})$
    $\textbf{for}\ (ipk_*, md)\ \textbf{in}\ \mathcal{MD}:$
        $\gamma \leftarrow wst.\Gamma[ipk_*]$
        $ipks_{\checkmark} \leftarrow \mathsf{MD.Devices}(ipk_*, \gamma, md)$
        $\textbf{if}\ ipks_{\checkmark} \neq \bot:\ wst.\Gamma[ipk_*] \leftarrow \gamma$
        $\textbf{for}\ (ipk_{\dagger}, skst)\ \textbf{in}\ wst.skts_{rcv}[gid, ipk_*, \cdot]:$
            $\textbf{if}\ ipk_{\dagger}\ \textbf{not in}\ ipks_{\checkmark}:$
                $wst.skts_{snd}[gid] \leftarrow \mathsf{SK.Rem}(wst.skts_{snd}[gid], ipk_{\dagger})$
                $wst.skts_{rcv}[gid, ipk_*, ipk_{\dagger}] \leftarrow \varnothing$
        $\textbf{for}\ ipk_{\dagger}\ \textbf{in}\ ipks_{\checkmark}:$
            $\textbf{if}\ (gid, ipk_*, ipk_{\dagger})\ \textbf{not in}\ wst.skts_{rcv}:$
                $wst.skts_{snd}[gid] \leftarrow \mathsf{SK.Add}(wst.skts_{snd}[gid], ipk_{\dagger})$
                $wst.skts_{rcv}[gid, ipk_*, ipk_{\dagger}] \leftarrow\$ \mathsf{SK.Init}(\texttt{rcv}, ipk_{\dagger})$
    $\textbf{return}\ wst$

$\mathsf{WA.ShareHistory}(wst \overset{is}{=} \mathsf{Obj}(\texttt{WA}, \cdot), ipk_c, \mathcal{MD}, \mathcal{SKB}, hist)$
    $\texttt{WA}, \texttt{primary}, ipk_p, pst, \Gamma, \cdot \leftarrow wst$
    $\textbf{assert}\ ipk_c \in \mathsf{MD.Devices}(ipk_p, \Gamma[ipk_p], \mathcal{MD}[ipk_p])$
    $wst.pst, c_P, c_{hist}, \tau_{hist}$
        $\leftarrow \mathsf{HS.Share}(\texttt{primary}, pst, \mathcal{SKB}[ipk_c], hist)$
    $\textbf{return}\ wst, c_P, c_{hist}, \tau_{hist}$

$\mathsf{WA.ReceiveHistory}(wst \overset{is}{=} \mathsf{Obj}(\texttt{WA}, \cdot), ipk_s, \mathcal{MD}, \mathcal{SKB}, c_P, c_{hist}, \tau_{hist})$
    $\texttt{WA}, \texttt{companion}, ipk_p, pst, \cdot \leftarrow wst$
    $\textbf{assert}\ ipk_s = ipk_p$
    $wst.pst, hist \leftarrow$
        $\mathsf{HS.Receive}(\texttt{companion}, pst, \mathcal{SKB}[ipk_p], c_P, c_{hist}, \tau_{hist})$
    $\textbf{return}\ wst, hist$
```

Fig. 16: Pseudocode describing multi-device group messaging in WhatsApp, WA.

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 (lines 3 and 4).[23] 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 security analysis captures the security of the cryptographic link that is made *without* capturing how the creation of that link is secured. In particular, our analysis does not capture the security of the registration sub-protocol within an adversarial network. Rather, the security experiment simulates the sub-protocol without providing the adversary with access to, or control over, those communications. See Section 4.

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.

[23] WhatsApp has since introduced an alternative to QR codes for device pairing utilising confirmation codes [69]. The resulting protocol is substantively different and is not covered here.

Group management. The WA.NewGroup algorithm describes the creation of a new group. 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 Sender Keys 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 3.1, 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 of 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 3.3, 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 [61, 62]. 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 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 3.1, 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 Device Management with Public Key Orbits

In this section, we seek to capture and analyse WhatsApp's device management functionality (described in Section 3.1). We introduce *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 multi-device setup (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.

We define the syntax of a public key orbit as follows.

Definition 17 (Public Key Orbit). *Let* $\mathsf{DS} := (\mathsf{DS.Gen}, \mathsf{DS.Sign}, \mathsf{DS.Verify})$ *be a digital signature scheme. A PO scheme is a five-tuple of algorithms, (PO.Setup, PO.Attract, PO.Repel, PO.Refresh, PO.Orbit).*

1) *The setup algorithm,* $\mathsf{PO.Setup}$*, 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* $\mathsf{PO.Reject?}$ *accepting a message* m *and outputting a bit* $\{0,1\}$*. This is a probabilistic algorithm.*

$$sk_p, pk_p, orb, \mathsf{PO.Reject?} \leftarrow\!\!\$\ \mathsf{PO.Setup}(1^\lambda)$$

2) *The attraction algorithm,* $\mathsf{PO.Attract}$*, takes in some 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* $\mathsf{DS.Gen}$*. This is a probabilistic algorithm.*

$$orb' \leftarrow\!\!\$\ \mathsf{PO.Attract}(sk, pk, orb)$$

3) *The repelling algorithm,* $\mathsf{PO.Repel}$*, takes in the signing key* sk_p*, a verification key* pk *and a state orb and outputs a new state* orb' *or* $\perp$*. This is a probabilistic algorithm.*

$$orb' \leftarrow\!\!\$\ \mathsf{PO.Repel}(sk_p, pk, orb)$$

4) *The refresh algorithm,* $\mathsf{PO.Refresh}$*, takes in the signing key* sk_p*, a state orb and outputs a new state* orb' *or* $\perp$*. This is a probabilistic algorithm.*

$$orb' \leftarrow\!\!\$\ \mathsf{PO.Refresh}(sk_p, orb)$$

5) *The orbit calculation algorithm,* $\mathsf{PO.Orbit}$*, takes in the verification key* pk_p*, a state orb and a generation* i *and returns a set of verification keys* $\mathcal{P}$ *or* $\perp$*. This is a deterministic algorithm.*

$$\mathcal{P} \leftarrow\!\!\$\ \mathsf{PO.Orbit}(pk_p, orb, i)$$

Remark 2. Note that this definition does not permit some (sk_c, pk_c) pair to "repel" itself from (sk_p, pk_p).

```
$\mathsf{Exp}^{w\mathsf{PO}}_{\mathsf{PO},(\lambda,n_{ch},n_\sigma,n_g)}(\mathcal{A})$
    $\mathcal{SK} \leftarrow \emptyset$  // Initialise set to store signing keys,
    $\mathcal{T} \leftarrow \mathsf{Map}\{0:\emptyset\}$  // record "to" links per gen,
    $\mathcal{F} \leftarrow \emptyset$  // all "from" links in single set,
    $\mathcal{C} \leftarrow \emptyset$  // challenge companion keys, and
    $corr \leftarrow \mathbf{false}$  // if primary key compromised.
    $sk_p, pk_p, orb, \mathsf{PO.Reject?} \leftarrow \mathsf{PO.Setup}(1^\lambda)$
    $\mathcal{SK} \leftarrow_\cup \{sk_p\}$
    $orb^\star, \gamma^\star \leftarrow \mathcal{A}^{\mathcal{O}}(1^\lambda, pk_p, orb, \mathsf{PO.Reject?})$
    $\mathcal{P} \leftarrow \mathsf{PO.Orbit}(pk_p, orb^\star, \gamma^\star)$
    // Compare orbit against expected value
    $\gamma' \leftarrow \mathsf{max}[orb^\star.\gamma, \gamma^\star]$
    $\mathcal{T}' \leftarrow \mathcal{T}[\gamma'] \cup \mathcal{T}[\gamma'+1] \cup \ldots \cup \mathcal{T}[\mathsf{PO}.\gamma]$
    $w_0 \leftarrow (corr = \mathbf{false}) \wedge (\mathcal{P} \setminus \mathcal{T}' \neq_? \emptyset)$
    $w_1 \leftarrow (\mathcal{P} \cap \mathcal{C} \setminus \mathcal{F} \neq_? \emptyset)$
    return $w_0 \vee w_1$
```

```
$\mathsf{Attract}(pk_{self}, pk_{other})$
    // Is this a "to" or "from" link?
    $op \leftarrow \bot$
    if $pk_{self} = pk_p$ :
        $op \leftarrow to$
    elseif $pk_{other} = pk_p \wedge pk_{self} \in \mathcal{C}$ :
        $op \leftarrow from$
    else : return $\bot$
    // Find signing key then create link
    let $sk_{self} \in \mathcal{SK}$ st $\mathsf{PK}(sk_{self}) = pk_{self}$
    $orb' \leftarrow \mathsf{PO.Attract}(sk_{self}, pk_{other}, orb)$
    assert $orb' \neq \bot$
    // Record new link and update state
    if $op = to$ :
        $\mathcal{T}[orb'.\gamma] \leftarrow \mathcal{T}[orb.\gamma] \cup \{pk_{other}\}$
    else :
        $\mathcal{T}[orb'.\gamma] \leftarrow \mathcal{T}[orb.\gamma]$
        $\mathcal{F} \leftarrow \mathcal{F} \cup \{pk_{self}\}$
    $orb \leftarrow orb'$
    return $orb$
```

```
$\mathsf{Repel}(pk)$
    $orb' \leftarrow \mathsf{PO.Repel}(sk_p, pk, orb)$
    assert $orb' \neq \bot$
    $\mathcal{T}[orb'.\gamma] \leftarrow \mathcal{T}[orb.\gamma] \setminus \{pk\}$
    $orb \leftarrow orb'$
    return $orb$
```

```
$\mathsf{Refresh}()$
    $orb' \leftarrow \mathsf{PO.Refresh}(sk_p, orb)$
    assert $orb' \neq \bot$
    $orb \leftarrow orb'$
    return $orb$
```

```
$\mathsf{Compromise}()$
    $corr \leftarrow \mathbf{true}$
    return $sk_p$
```

```
$\mathsf{Sign}(pk, m)$
    if $\mathsf{PO.Reject?}(m)$ :
        return $\bot$
    let $sk \in \mathcal{SK}$ st $\mathsf{PK}(sk) = pk$
    return $\mathsf{DS.Sign}(sk, m)$
```

```
$\mathsf{Challenge}()$
    $sk, pk \leftarrow \mathsf{DS.Gen}(1^\lambda)$
    $\mathcal{SK} \leftarrow_\cup \{sk\}$
    $\mathcal{C} \leftarrow_\cup \{pk\}$
    return $pk$
```

```
$\mathsf{Eject}(pk)$
    assert $pk \neq pk_p$
    $\mathcal{C} \leftarrow_\setminus \{pk\}$
    let $sk \in \mathcal{SK}$ st $\mathsf{PK}(sk) = pk$
    return $sk$
```

Fig. 17: Security experiment capturing Weak Public Key Orbit security.

We define a series of correctness properties.

Definition 18 (Correctness of Public Key Orbits). *Let* $\mathsf{PO} = (\mathsf{PO.Setup}, \mathsf{PO.Attract}, \mathsf{PO.Repel}, \mathsf{PO.Refresh}, \mathsf{PO.Orbit})$. *Let* $z = \mathsf{poly}(\lambda)$ *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* $\forall\ (sk_p, pk_p)$, $orb, \mathsf{PO.Reject?} \leftarrow \mathsf{PO.Setup}(1^\lambda)$*, we have*

1) ***(Operations with*** sk_p ***increase gen)****. For* $\mathsf{op} \in \{\mathsf{PO.Attract}, \mathsf{PO.Repel}, \mathsf{PO.Refresh}\}$ *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* $\mathsf{PO.Attract}$, $\mathsf{PO.Repel}, \mathsf{PO.Refresh}$ *after a* $\mathsf{PO.Setup}$. *Then for* $\mathcal{P} \leftarrow \mathsf{PO.Orbit}(pk_p, orb, k)$ *we have* $\mathcal{P} \neq \perp$ *if* $orb \neq \perp$ *and* $k = orb.\gamma$.

3) ***(Attracting without repelling includes)*** *Let* $0 \leq i < z$ *be some index such that* $\mathsf{PO.Attract}(sk_p, pk_c, orb_i)$ *was called. Let* $i < j < z$ *the smallest index such that* $\mathsf{PO.Repel}(sk_p, pk_c, orb_j)$ *was called. Let* $j = z$ *if no such call occurred. Let* i' *be the smallest index* i' *such that* $\mathsf{PO.Attract}(sk_c, pk_p, orb_{i'})$ *was called. It holds that* $\forall\ \max(i, i') < k \leq j$
$$pk_c \in \mathsf{PO.Orbit}(pk_p, orb_k, k).$$

4) ***(Repelling removes)*** *Let* i *be the smallest index such that* $\mathsf{PO.Repel}(sk_p, pk_c, orb_i)$ *or* $i = 0$ *if no such call happened. Let* $j > i$ *be the smallest index such that both* $\mathsf{PO.Attract}(sk_p, pk_c, orb)$ *and* $\mathsf{PO.Attract}(sk_c, pk_p, orb')$ *have been called for some* $orb.\gamma \geq j$. *It holds that*
$$pk_c \notin \mathsf{PO.Orbit}(pk_p, orb_k, k)\, \forall i < k \leq j.$$

5) ***(Refresh does not change list)*** *If* $orb' \leftarrow \mathsf{PO.Refresh}(sk_p, orb)$ *was called then for* $\mathcal{P} \leftarrow \mathsf{PO.Orbit}(sk_p, orb, orb.\gamma)$ *and* $\mathcal{P}' \leftarrow \mathsf{PO.Orbit}(pk_p, orb', orb'.\gamma)$ *we have* $\mathcal{P} = \mathcal{P}'$.

We define $w\mathsf{PO}$ security 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 $\mathsf{PO.Attract}$ or were followed by an $\mathsf{PO.Repel}$ call.

We insist on this guarantee even in the presence of a signing oracle that will sign any message except those specified by $\mathsf{PO.Reject?}$.[24] 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 $\mathsf{Compromise}$ 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.

[24] This will allow us to model the lack of domain separation at the level of signing keys in WhatsApp.

Definition 19 (Weak Public Key Orbit Security). *Let* $\mathsf{PO} = (\mathsf{PO.Setup}, \mathsf{PO.Attract}, \mathsf{PO.Repel}, \mathsf{PO.Refresh}, \mathsf{PO.Orbit})$. *The advantage of an adversary* $\mathcal{A}$ *breaking* $w\mathsf{PO}$ *security is defined as* $\mathsf{Adv}^{\mathsf{wPO},\mathcal{A}}_{\mathsf{PO}}(\lambda) \coloneqq \Pr\left[\mathsf{Exp}^{w\mathsf{PO},\mathcal{A}}_{\mathsf{PO}}(\lambda)\right]$ *where* $\mathsf{Exp}^{w\mathsf{PO}}_{\mathsf{PO},\Lambda}(\mathcal{A})$ *is defined in Figure 17.*

```
$\mathsf{WA\text{-}PO.Setup}(1^\lambda)$
$isk_p, ipk_p \leftarrow\$ \mathsf{XDH.Gen}(1^\lambda)$
$m \leftarrow \mathtt{0x0602} \,\|\, [ipk_p] \,\|\, 0$
$\sigma_{dl} \leftarrow \mathsf{XEd.Sign}(isk_p, m)$
$orb \leftarrow \mathsf{Obj}(\mathtt{PO}, ipk_p, [ipk_p], 0, \sigma_{dl}, [\varnothing])$
$\mathsf{Reject} \leftarrow m[0] \overset{?}{=} \mathtt{0x06}$
$\mathbf{return}\ (isk_p, ipk_p), orb, \mathsf{Reject}$
```

```
$\mathsf{WA\text{-}PO.Refresh}(isk, orb)$
$ipk \leftarrow \mathsf{PK}(isk)$
$ipk_p \overset{is}{=} ipk, dl, \gamma, \sigma_{dl}, \Delta \leftarrow orb$
$\gamma \leftarrow \gamma + 1$
$m \leftarrow \mathtt{0x0602} \,\|\, dl \,\|\, \gamma$
$\sigma_{dl} \leftarrow \mathsf{XEd.Sign}(isk, m)$
$orb' \leftarrow \mathsf{Obj}(\mathtt{PO}, ipk, dl, \gamma, \sigma_{dl}, \Delta)$
$orb \leftarrow orb'$ ; $\mathbf{return}\ orb'$
```

```
$\mathsf{WA\text{-}PO.Repel}(isk, ipk_*, orb)$
$ipk \leftarrow \mathsf{PK}(isk)$
$ipk_p \overset{is}{=} ipk, dl, \gamma, \sigma_{dl}, \Delta \leftarrow orb$
$\gamma \leftarrow \gamma + 1$
$dl \leftarrow [ipk_\dagger\ \mathbf{in}\ dl\ \mathbf{if}\ ipk_\dagger \neq ipk_*]$
$m \leftarrow \mathtt{0x0602} \,\|\, dl \,\|\, \gamma$
$\sigma_{dl} \leftarrow \mathsf{XEd.Sign}(isk, m)$
$orb' \leftarrow \mathsf{Obj}(\mathtt{PO}, ipk_p, dl, \gamma, \sigma_{dl}, \Delta)$
$orb \leftarrow orb'$ ; $\mathbf{return}\ orb'$
```

```
$\mathsf{WA\text{-}PO.Attract}(isk, ipk_*, orb)$
$ipk \leftarrow \mathsf{PK}(isk)$
$\mathbf{if}\ ipk = orb.ipk_p$ :
    $ipk_p, dl, \gamma, \sigma_{dl}, \Delta \leftarrow orb$
    $\gamma \leftarrow \gamma + 1$
    $m \leftarrow \mathtt{0x0600} \,\|\, \gamma \,\|\, ipk_p \,\|\, ipk_*$
    $\sigma_{p \to c} \leftarrow \mathsf{XEd.Sign}(isk, m)$
    $\Delta[ipk_*] \leftarrow \mathsf{Obj}(\mathtt{DR}, \gamma, ipk_p, ipk_*, \sigma_{p \to c}, \varnothing)$
    $dl \leftarrow dl \cup \{ipk_*\}$
    $\sigma_{dl} \leftarrow \mathsf{XEd.Sign}(isk, \mathtt{0x0602} \,\|\, dl \,\|\, \gamma)$
    $orb' \leftarrow \mathsf{Obj}(\mathtt{PO}, isk, dl, \gamma, \sigma_{dl}, \Delta)$
$\mathbf{elseif}\ ipk_* = orb.ipk_p$ :
    $ipk_p, dl, \gamma, \sigma_{dl}, \Delta \leftarrow orb$
    $\gamma, ipk_p \overset{is}{=} ipk_p, ipk_c \overset{is}{=} ipk, \sigma_{p \to c}, \sigma_{c \to p} \overset{is}{=} \varnothing \leftarrow \Delta[ipk]$
    $\sigma_{c \to p} \leftarrow \mathsf{XEd.Sign}(isk, \mathtt{0x0601} \,\|\, \gamma \,\|\, ipk_p \,\|\, ipk)$
    $\Delta[ipk] \leftarrow \mathsf{Obj}(\mathtt{DR}, \gamma, ipk_p, ipk, \sigma_{p \to c}, \sigma_{c \to p})$
    $orb' \leftarrow \mathsf{Obj}(\mathtt{PO}, ipk_p, dl, \gamma, \sigma_{dl}, \Delta)$
$\mathbf{else}$ : $\mathbf{return}\ \bot$
$orb \leftarrow orb'$ ; $\mathbf{return}\ orb'$
```

```
$\mathsf{WA\text{-}PO.Orbit}(ipk_p, orb^\star, i)$
$\mathbf{assert}\ orb^\star.dl[0] = orb^\star.ipk = ipk_p$
$\mathbf{assert}\ orb^\star.\gamma \geq i$
$m \leftarrow \mathtt{0x0602} \,\|\, orb^\star.dl \,\|\, orb^\star.\gamma$
$\mathbf{assert}\ \mathsf{XEd.Verify}(ipk_p, m, orb^\star.\sigma_{dl})$
$\mathbf{for}\ dr \in orb^\star.\Delta$
    $m_0 \leftarrow \mathtt{0x0600} \,\|\, dr.\gamma \,\|\, ipk_p \,\|\, dr.ipk_c$
    $m_1 \leftarrow \mathtt{0x0601} \,\|\, dr.\gamma \,\|\, ipk_p \,\|\, dr.ipk_c$
    $b_0 \leftarrow \mathsf{XEd.Verify}(ipk_p, m_0, dr.\sigma_{p \to c})$
    $b_1 \leftarrow \mathsf{XEd.Verify}(dr.ipk_c, m_1, dr.\sigma_{c \to p})$
    $b_2 \leftarrow dr.ipk_c \neq ipk_p$
    $b_3 \leftarrow (dr.ipk_c \in orb^\star.dl)\ \vee\ (dr.\gamma > orb^\star.\gamma)$
    $\mathbf{if}\ b_0 \wedge b_1 \wedge b_2 \wedge b_3$ :
        $\mathcal{P} \leftarrow_\cup \{dr.ipk_c\}$
$\mathbf{return}\ \mathcal{P}$
```

Fig. 18: Device management in WhatsApp expressed as a public-key orbit.

We now 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. Doing so required minimal changes to our description of device management in Section 3. These changes are primarily syntactic in order to ensure compatibility with the public key orbit formalism. See Figure 18 for the details of this instantiation.

Definition 20. *WA-PO is a public key orbit that implements the PO formalism with algorithms (WA-PO.Setup, WA-PO.Attract, WA-PO.Repel, WA-PO.Refresh, WA-PO.Orbit) in Figure 18.*

We state and prove that our instantiation of device management in WhatsApp as a public key orbit, $\mathsf{WA\text{-}PO}$, achieves weak public-key orbit security.

Theorem 1 (Security of $\mathsf{WA\text{-}PO}$). *For any probabilistic polynomial-time algorithm $\mathcal{A}$ playing the $w\mathsf{PO}$ security game instantiated with the $\mathsf{WA\text{-}PO}$ scheme, making at most n_{ch} queries to $\mathsf{Challenge}$, at most n_σ queries to Sign per signing key, and at most n_g cumulative queries to the $\mathsf{Attract}$, Repel and $\mathsf{Refresh}$ oracles, we have:*

$$\mathsf{Adv}^{\mathsf{wPO}}_{\mathsf{WA\text{-}PO}}(\lambda, n_{ch}, n_\sigma, n_g) \;\leq\; (n_{ch}+1)\cdot \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, n_\sigma + 2\cdot n_g)$$

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 $\mathsf{WA\text{-}PO}$ scheme into the $w\mathsf{PO}$ 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 $\mathsf{Compromise}$ 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 $corr = \textbf{false}$, i.e. $\mathsf{Compromise}$ 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 Sign, $\mathsf{Attract}$, Repel or $\mathsf{Refresh}$ query.

We bound the probability of $abort_{forge}$ occurring with the following security reduction. We construct an adversary, $\mathcal{B}$, against an $\mathsf{EUF\text{-}CMA}$ 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 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 Ed 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 $\mathsf{WA\text{-}PO.Reject?}$ will prevent any calls to 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 $\mathsf{WA\text{-}PO.Orbit}$ were honestly generated while processing a $\mathsf{WA\text{-}PO.Setup}$, $\mathsf{WA\text{-}PO.Attract}$, $\mathsf{WA\text{-}PO.Repel}$ or $\mathsf{WA\text{-}PO.Refresh}$ call.

In order to win the game, the adversary must provide an orbit state, $orb^\star$, and generation, $\gamma^\star$, that when processed by $\mathsf{PO.Orbit}$, 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 $\mathsf{WA\text{-}PO.Orbit}$ 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, dr, 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 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 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 Sign or 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 Ed scheme, when restricted to at most $n_\sigma + n_g$ signing queries. Thus:

$$\mathsf{Adv}_{\mathsf{G1}} \leq \mathsf{Adv}_{\mathsf{G2}} + \mathsf{Adv}_{\mathsf{XEd}}^{\mathsf{EUF\text{-}CMA}}(\lambda, n_\sigma + n_g)$$

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 Sign whose message starts with `0x06`, we can be sure that all message-signature pairs that verify as originating from pk' during execution of $\mathsf{WA\text{-}PO.Orbit}$ were honestly generated while processing a $\mathsf{WA\text{-}PO.Attract}$ 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 $dr.\sigma_{c \to p}$ cannot have been an output of 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 $\mathsf{Attract}(pk', pk_p)$ at any point in the experiment. It follows that the challenger did not execute $\mathsf{XEd.Sign}(sk', \texttt{0x0601} \parallel \gamma \parallel ipk_p \parallel pk')$ at any point the experiment. By **Game 2**, we know that no such valid signature σ' exists for which $\mathsf{XEd.Verify}(pk', \texttt{0x0601} \parallel \gamma \parallel ipk_p \parallel pk, \sigma')$ 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:[25]

$$\mathsf{Adv}^{\mathsf{wPO}}_{\mathsf{WA\text{-}PO},\mathcal{A}}(\lambda, n_{ch}, n_\sigma, n_g) \leq (n_{ch} + 1) \cdot \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, n_\sigma + 2 \cdot n_g)$$

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 $\mathsf{WA\text{-}PO}$ 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. □

5 Pairwise Channels with Session Management

As we saw in Section 3.2, WhatsApp makes use of a collection of two-party channels for communication between pairs of devices. In particular, clients allow for up to 40 simultaneously active two-party sessions between themselves and another device. This mirrors Signal's use of the Sesame session management protocol, the implications of which have previously been explored in [28, 30]. The formalism we present here follows the general approach of [30], albeit in the computational setting.

Additionally, the resulting formalism is not too dissimilar from the MSKE model introduced in [25], since the security experiment also captures multiple parallel sessions executing in parallel. Indeed, the core differentiator between the

[25] 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.

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 briefly discuss the MSKE formalism of [25] and discuss how the two formalisms differ in Section 5.1.

We start by defining the syntax of such protocols.

Definition 21 (Pairwise Channels). *A PAIR scheme for secure pairwise channels is a three-tuple of algorithms, ($\mathsf{PAIR.Init}$, $\mathsf{PAIR.Enc}$, $\mathsf{PAIR.Dec}$) and a three-tuple of* state extractor *algorithms, ($\mathsf{PAIR.}\textsc{identity}$, $\mathsf{PAIR.}\textsc{shared}$, $\mathsf{PAIR.}\textsc{session}$).*

1) *The* initialisation *algorithm, $\mathsf{PAIR.Init}$, 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, $\mathsf{PAIR.Enc}$, 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, $\mathsf{PAIR.Dec}$, 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) *$\mathsf{PAIR.}\textsc{identity}$ takes in a private state and outputs the long-term identity secrets it contains: ident-sk $\leftarrow \mathsf{PAIR.}\textsc{identity}(sk)$.*

2) *$\mathsf{PAIR.}\textsc{shared}$ takes in a private state and outputs any secret state that is shared across multiple sessions: share-sk $\leftarrow \mathsf{PAIR.}\textsc{shared}(sk)$.*

3) *$\mathsf{PAIR.}\textsc{session}$ takes in a private state and session identifier and outputs the current state of the specified session: sess-sk $\leftarrow \mathsf{PAIR.}\textsc{session}(sk, sid)$.*

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 22 (Correctness of Pairwise Channels). *A scheme* $\mathsf{PAIR} = (\mathsf{PAIR.Init}, \mathsf{PAIR.Enc}, \mathsf{PAIR.Dec})$ *is correct if, for all* $sid, z, m, c \in \{0,1\}^*$,

$$\begin{aligned} &\big(\,\mathsf{PAIR.Enc}(sk_i, pk_j, info_j, m) \mapsto (sid, z, c)\\ &\wedge\ \mathsf{PAIR.Dec}(sk_j, pk_i, info_i, sid, z, c) \mapsto (\,\cdot\,, m')\\ &\wedge\, m' \neq \bot\big) \implies m = m' \end{aligned}$$

provided there exists pk_i, ${sk_i}^*$, sk_i, pk_j, ${sk_j}^*$ *and* sk_j *where*

1) (sk_i^*, pk_i) *are the output of a* PAIR.Init *call and there exists a polynomial-step combination of* PAIR.Enc *and* PAIR.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* PAIR.Init *and there exists a polynomial-step combination* PAIR.Enc *and* PAIR.Dec *calls through which* sk_j *can be derived from* sk_j^*.

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 the conditions in which security is guaranteed 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 participants, the challenger provides an initial distribution of each participant's public identifier.

Definition 23 (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* $\mathsf{Exp}^{\mathsf{PAIR\text{-}SEC}}_{\Pi,\Lambda}(\mathcal{A})$

```
$\mathsf{Exp}^{\mathsf{PAIR\text{-}SEC}}_{\Pi,\Lambda}(\mathcal{A})$
    $b \leftarrow\$ \{0,1\};\ win \leftarrow 0;\ \mathbf{L} \leftarrow [\,];$ for $i = 0,1,2,\dots,n_p-1:\ sk_i, pk_i, info_i \leftarrow\$ \mathsf{PAIR.Init}(1^\lambda)$
    $b' \leftarrow \mathcal{A}^{\mathsf{Enc},\mathsf{Dec},\mathsf{Corrupt}^*}(\{pk_0, pk_1, \dots, pk_{n_p-1}\}, \{info_0, info_1, \dots, info_{n_p-1}\})$
    return $b = b' \wedge \mathsf{PAIR.CONF}(\mathbf{L})$
```

```
$\mathsf{Enc}(i, j, info, sid, m_0, m_1)$
    assert $\mathsf{len}(m_0) = \mathsf{len}(m_1)$
    $sk_i, sid, z, c \leftarrow\$$
        $\mathsf{PAIR.Enc}(sk_i, pk_j, info, sid, m_b)$
    $\mathbf{L} \leftarrow_{app} (\mathtt{enc}, i, j, sid, z, m_0, m_1, c)$
    return $sid, z, c$
```

```
$\mathsf{Dec}(i, j, info, sid, c)$
    $sk_i, sid, z, m \leftarrow\$ \mathsf{PAIR.Dec}(sk_i, pk_j, info, sid, z, c)$
    if $m = \bot$: return $\bot$
    $replay \leftarrow (\mathtt{dec}, i, j, sid, \cdot, c, \cdot) \in \mathbf{L}$
    $forgery \leftarrow (\mathtt{enc}, j, i, sid, \cdot, \cdot, \cdot, c) \notin \mathbf{L}$
    $win \leftarrow replay \vee (forgery \wedge \mathsf{PAIR.AUTH}(\mathbf{L}, i, j, sid, z, c, m))$
    $\mathbf{L} \leftarrow_{app} (\mathtt{dec}, i, j, sid, z, c, m)$
    if $c \in {}^*\mathsf{Challenges}(\mathbf{L})$: return $\bot$
    return $sid, z, m$
```

```
$\mathsf{CorruptIdentity}(i)$
    assert $0 \le i < n_p$
    $corr \leftarrow \mathsf{PAIR.IDENTITY}(sk_i)$
    $\mathbf{L} \leftarrow_{app} (\mathtt{corr\text{-}ident}, i, corr)$
    return $corr$
```

```
$\mathsf{CorruptShared}(i)$
    assert $0 \le i < n_p$
    $corr \leftarrow \mathsf{PAIR.SHARED}(sk_i)$
    $\mathbf{L} \leftarrow_{app} (\mathtt{corr\text{-}share}, i, corr)$
    return $corr$
```

```
$\mathsf{CorruptSession}(i, j, sid)$
    assert $0 \le i, j < n_p$
    $corr \leftarrow \mathsf{PAIR.SESSION}(sk_i, pk_j, sid)$
    $\mathbf{L} \leftarrow_{app} (\mathtt{corr\text{-}sess}, i, j, sid, z, corr)$
    return $corr$
```

```
$^*\mathsf{Challenges}(\mathbf{L}) := [c$ for $(\mathtt{enc}, \cdot, \cdot, \cdot, \cdot, m_0, m_1, c)$ in $\mathbf{L}$ if $m_0 \neq m_1]$
```

Fig. 19: Pairwise Channel Security Game, $\mathsf{Exp}^{\mathsf{PAIR\text{-}SEC}}_{\Pi,\Lambda}(\mathcal{A})$.

security experiment detailed in Figure 19. 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:*

$$conf? \leftarrow \mathsf{PAIR}.\mathit{CONF}(\mathbf{L}).$$

2) *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:*

$$auth? \leftarrow \mathsf{PAIR}.\mathit{AUTH}(\mathbf{L}, i, j, sid, z, c, m).$$

See the security game in Figure 19 for the structure of the experiment log, $\mathbf{L}$.

Note that the total number of messages that may be exchanged during the experiment is $N := \frac{1}{2} \cdot n_d \cdot (n_d - 1) \cdot n_i \cdot n_m$.

5.1 Security of WhatsApp's Pairwise Channels

```
$\underline{\mathsf{WA\text{-}PAIR.Init}(1^\lambda)}$
$ipk, isk \leftarrow\$ \mathsf{SIGNAL.KeyGen}();\ spk, ssk \leftarrow\$ \mathsf{SIGNAL.MedTermKeyGen}()$
$\textbf{for } i = 0, 1, 2, \ldots, n_e - 1:\ epk_i, esk_i \leftarrow\$ \mathsf{SIGNAL.EphemKeyGen}()$
$esks \leftarrow \{esk_i : 0 \le i < n_e\};\ epks \leftarrow \{epk_i : 0 \le i < n_e\};\ ssts \leftarrow \mathsf{Map}\{\};\ sk \leftarrow \mathsf{Obj}(\texttt{PAIR}, isk, ssk, esks, ssts)$
$\textbf{return } (sk), (ipk, spk), (epks)$
```

```
$\underline{\mathsf{WA\text{-}PAIR.Enc}(sk, pk_j, info_j, m)}$
$\texttt{PAIR}, isk, ssk, esks, ssts \leftarrow sk$
$ipk_j, spk_j \leftarrow pk_j$
$\textbf{if } sid = \varnothing$
    $epk_j \leftarrow info[0]$    // the responder's one-time key
    $sid \leftarrow epk_j$
$sst \leftarrow ssts[ipk_j, sid]$
$\textbf{if } sst = \varnothing:$
    $sst, \cdot \leftarrow\$ \mathsf{SIGNAL.Activate}(isk, ssk, \texttt{init}, ipk_j)$
    $sst, c_{kex} \leftarrow\$ \mathsf{SIGNAL.Run}(isk, ssk, sst, (spk_j, sid))$
$\textbf{else}:$
    $sst, c_{kex} \leftarrow\$ \mathsf{SIGNAL.Run}(isk, ssk, sst, \varnothing)$
$\textbf{assert } sst.status[sst.stage] = \texttt{accept}$
$c_{msg} \leftarrow \mathsf{WA\text{-}AEAD.Enc}(sst.k[sst.stage], c_{kex}, m_b)$
$ssts[ipk_j, sid] \leftarrow sst$
$sk \leftarrow \mathsf{Obj}(\texttt{PAIR}, isk, ssk, esks, ssts)$
$\textbf{return } sk, sid, sst.stage, (c_{kex}, c_{msg})$
```

```
$\underline{\mathsf{WA\text{-}PAIR.Dec}(sk, pk_j, info_j, (c_{kex}, c_{msg}))}$
$\texttt{PAIR}, isk, ssk, esks, ssts \leftarrow sk;\ ipk_j, spk_j \leftarrow pk_j$
$\textbf{if } sid = \varnothing:\ epk_i \leftarrow c_{kex}.epk_{resp}$
$epk_j \leftarrow c_{kex}.epk_{init};\ sid \leftarrow epk_i;\ sst \leftarrow ssts[ipk_j, sid]$
$\textbf{if } sst = \varnothing:$
    $[esk] \leftarrow [esk' \textbf{ in } esks \textbf{ if } c_{kex}.epk_{resp} = \mathsf{PK}(esk')]$
    $sst, \cdot \leftarrow\$ \mathsf{SIGNAL.Activate}(isk, ssk, \texttt{resp}, ipk_j, esk,$
                $c_{kex}.epk_{resp})$
    $sst, \cdot \leftarrow\$ \mathsf{SIGNAL.Run}(isk, ssk, sst, c_{kex}, skb_j, esk)$
    $esks \leftarrow [esk' \textbf{ in } esks \textbf{ if } c_{kex}.epk_{resp} \ne \mathsf{PK}(esk')]$
$\textbf{else}:$
    $sst, \cdot \leftarrow\$ \mathsf{SIGNAL.Run}(isk, ssk, sst, c_{kex})$
$\textbf{assert } sst.status[sst.stage] = \texttt{accept}$
$m \leftarrow \mathsf{WA\text{-}AEAD.Dec}(sst.k[sst.stage], c_{kex}, c_{msg})$
$\textbf{assert } m \ne \bot$
$ssts[ipk_j, sid] \leftarrow sst$
$sk \leftarrow \mathsf{Obj}(\texttt{PAIR}, isk, ssk, esks, ssts)$
$\textbf{return } sk, sid, sst.stage, m$
```

$$\underline{\mathsf{WA\text{-}PAIR.IDENTITY}(sk_i)} := sk_i.isk \qquad \underline{\mathsf{WA\text{-}PAIR.SHARED}(sk_i)} := (sk_i.ssk, sk_i.esks)$$

$$\underline{\mathsf{WA\text{-}PAIR.SESSION}(sk_i, pk_j, sid)} := {}^*\mathsf{WA\text{-}PAIR.FindSession}(sk_i.ssts[pk_j.ipk], sid)[0]$$

$$\underline{\mathsf{WA\text{-}PAIR.CONF}(\mathbf{L})} := \forall\ (\mathsf{enc}, i, j, sid, z, \cdot, \cdot, \cdot) \in {}^*\mathsf{Challenges}(\mathbf{L}) : \mathsf{SIGNAL.FRESH}(i, j, sid, z)$$

$$\underline{\mathsf{WA\text{-}PAIR.AUTH}(\mathbf{L}, i, j, sid, z, c, m)} := \mathsf{SIGNAL.FRESH}(i, j, sid, z)$$

Fig. 20: WhatsApp's DM sub-protocol expressed as a pairwise channel in the WA-PAIR formalism, with state extractor algorithms (WA-PAIR.IDENTITY, WA-PAIR.SHARED, WA-PAIR.SESSION) and security predicates (WA-PAIR.CONF, WA-PAIR.AUTH). See Appendix A.1 for a description of SIGNAL.FRESH from [25] translated into the PAIR-SEC security experiment.

We now proceed with our security analysis of WhatsApp's pairwise channels within our newly introduced formalism. We start with a description 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 setting the *win* flag). 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 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 [25].

Message routing. In our description of WhatsApp's pairwise channels, 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. CorruptShared reveals all future ephemeral key pairs to the adversary).

Reusing the Multi-Stage Key Exchange formalism. As mentioned, we re-use the existing analysis of Signal two-party key exchange of [25] 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 formalism of [25]. We then proceed to prove the security of $\mathsf{WA\text{-}PAIR}$ with respect to the $\mathsf{MS\text{-}IND}$ security of those underlying Signal MSKE channels. However, doing so necessitates a number of minor changes to the MSKE formalism. We briefly define the modified syntax, and refer the reader to [25, Section 4.2] for a security definition.

Definition 24 (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* $\rho \in \{\mathtt{init}, \mathtt{resp}\}$*, and optionally an identifier of its intended peer peerid and outputs a state* $\pi^{'}$ *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* $\pi^{'}$ *and (possibly empty) outgoing protocol message* m'*.*

The security analysis in [25] 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 $\mathsf{SIGNAL.Activate}$ 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 WhatsApp's pairwise 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 $\mathsf{MS\text{-}IND}$ 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 [25]. We split the generation of one-time keys from the activation of a session by introducing the $\mathsf{MSKE.EphemKeyGen}$ algorithm that generates and returns a single ephemeral key pair. To activate a session, $\mathsf{MSKE.Activate}$ 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 $\mathsf{.Activate}$ (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 [25] apply to this variant directly and without modification.

Next, the analysis in [25] applies to the $\mathsf{Exp}^{\mathsf{MS\text{-}IND}}_{\Pi,\Lambda}(\mathcal{A})$ experiment that allows a single Test query. However, the $\mathsf{PAIR\text{-}SEC}$ 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 $\mathsf{Exp}^{\mathsf{MS\text{-}IND}}_{\Pi,\Lambda}(\mathcal{A})$ security experiment. We posit that Theorem 1 in [25] 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 [33, Appendix A]). We do so without proof, however.

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. For example, $\mathsf{PAIR.SESSION}$ defines compromise of a client's *shared state* as revealing their medium-term secret key and their unused secret ephemeral keys.

Definition 25. *$\mathsf{WA\text{-}PAIR}$ is a pairwise channel with session management that instantiates the PAIR formalism with algorithms ($\mathsf{WA\text{-}PAIR.Init}$, $\mathsf{WA\text{-}PAIR.Enc}$, $\mathsf{WA\text{-}PAIR.Dec}$), state extractor algorithms ($\mathsf{WA\text{-}PAIR.IDENTITY}$, $\mathsf{WA\text{-}PAIR.SHARED}$, $\mathsf{WA\text{-}PAIR.SESSION}$) and security predicates ($\mathsf{WA\text{-}PAIR.CONF}$, $\mathsf{WA\text{-}PAIR.AUTH}$) in Figure 20.*

We state and prove that the $\mathsf{WA\text{-}PAIR}$ scheme instantiates a secure pairwise channel under the security predicates in Figure 20.

Theorem 2. *The $\mathsf{WA\text{-}PAIR}$ protocol specified in Definition 25 instantiates a secure pairwise channel, under security predicates $\mathsf{WA\text{-}PAIR.AUTH}$ and $\mathsf{WA\text{-}PAIR.CONF}$, with the advantage of any probabilistic polynomial-time adversary $\mathcal{A}$ in winning* *the* $\mathsf{Exp}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR},\lambda,n_d,n_i,n_m}(\mathcal{A})$ *security experiment bound by:*

$$\begin{aligned}&\mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda, n_d, n_i, n_m)\\ &\quad\leq\ \frac{1}{2}\cdot n_d\cdot(n_d-1)\cdot n_i\cdot n_m\cdot\big[\\ &\qquad\quad \mathsf{Adv}^{\mathsf{MS\text{-}IND}}_{\mathsf{SIGNAL}}(\lambda_r, n_d, 1, n_i, n_m)\ +\ \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{WA\text{-}AEAD}}(\lambda, 1)\ +\ \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{WA\text{-}AEAD}}(\lambda, 1)\\ &\quad\ \ \big]\end{aligned}$$

under the following assumptions:

1) *SIGNAL is a multi-test* MS-IND *secure MSKE protocol for which the advantage of any PPT adversary is bound by*

$$\mathsf{Adv}^{\mathsf{MS\text{-}IND}}_{\mathsf{SIGNAL}}(\lambda, n_P, n_M, n_S, n_s)$$

with respect to security predicate SIGNAL.FRESH as defined in [25],[26] *and*

2) *WA-AEAD is an IND\$-CPA and EUF-CMA secure one-time AEAD scheme for which the advantage of any PPT adversary is bound by*

$$\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{WA\text{-}AEAD}}(\lambda, n_q)\ \ \textit{and}\ \ \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{WA\text{-}AEAD}}(\lambda, n_q).$$

The proof consists of a series of games: G0, G1, A0, A1, C0 and C1. Starting with G0, we inline the WA-PAIR protocol into the original $\mathsf{Exp}^{\mathsf{PAIR\text{-}SEC}}_{\Pi,\Lambda}(\mathcal{A})$ experiment. The first hop, from G0 to G1, relies on the security of the underlying two-party Signal protocol as an MS-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 A0 to A1, we rely on the one-time EUF-CMA security of the WA-AEAD scheme to bound the possibility of our adversary producing a forgery of a ciphertext. In the case of a confidentiality break, games C0, C1.0, C1.1, ..., $\mathsf{C1.N}-1$ rely on the one-time IND\$-CPA security of the WA-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 $\mathsf{Exp}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR},\Lambda}(\mathcal{A})$ experiment.

Game 0. We inline WhatsApp's WA-PAIR protocol (as described in Figure 20) into the $\mathsf{Exp}^{\mathsf{PAIR\text{-}SEC}}_{\Pi,\Lambda}(\mathcal{A})$ security experiment. Thus:

$$\mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR},\mathcal{A}}(\lambda, n_d, n_i, n_m) = \mathsf{Adv}_{\mathsf{G0}}$$

[26] The security parameter in the original experiment, 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.

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 $\mathsf{Exp}^{\mathsf{MS\text{-}IND}}_{\Pi,\mathcal{A}}(\mathcal{A})$ 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 [25] 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 20, 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 A.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 20, 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 [25]. 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 A.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 27, 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:

$$\mathsf{Adv}_{\mathsf{G0}} \leq \mathsf{Adv}_{\mathsf{G1}} + \frac{1}{2} \cdot n_d \cdot (n_d - 1) \cdot n_i \cdot n_m \cdot \mathsf{Adv}_{\mathsf{SIGNAL}}^{\mathsf{MS\text{-}IND}}(\lambda_r, n_d, 1, n_i, n_m)$$

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 of the *win* flag.

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 28 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. We 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 29 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:

$$\begin{aligned} &\mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR},\mathcal{A}}(\lambda, n_d, n_i, n_m) \\ &\leq \frac{1}{2} \cdot n_d \cdot (n_d - 1) \cdot n_i \cdot n_m \cdot \Big[\\ &\qquad \mathsf{Adv}^{\mathsf{MS\text{-}IND}}_{\mathsf{SIGNAL}}(\lambda_r, n_d, 1, n_i, n_m) + \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{WA\text{-}AEAD}}(\lambda, 1) + \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{WA\text{-}AEAD}}(\lambda, 1) \\ &\quad \Big] \end{aligned}$$

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. □

5.2 Discussion

Our security analysis shows that the pairwise channels in WhatsApp are able to 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 doing so, we have shown that they are able to provide a form of forward security: compromise of the long-term secrets, medium-term secrets, and secret session state does not compromise the security of old messages (for the most part). The story for PCS is more complicated, however.

Intuitively, 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 [28, 30], our model captures pairwise messaging schemes that allow multiple parallel sessions between parties. We need to do this because the group messaging functionality in WhatsApp, Sender Keys, relies on the pairwise messaging scheme for key distribution. Previous work on Sender Keys in WhatsApp does not cover this functionality. In doing so, our model captures a stronger adversary with new capabilities. First, the adversary 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. 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. In this attack, the adversary compromises the long-term and medium-term keys of party A after A and B have initialised a session. In a single session setting, the adversary should not be able to compromise messages sent between A and B, 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 A and itself (masquerading as B using an unknown key share attack) and another between B and itself (masquerading as 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. In this attack, the adversary compromises the current session state of party A (between A and 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 A and itself (masquerading as B) and another between B and itself (masquerading as 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. While this may result in clients setting up a new uncompromised session, the adversary can manipulate the network, coercing them into using 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 protocol would achieve.

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.

In order for the group messaging component to regain security post-compromise, we require all pairwise channels used to distribute inbound Sender Keys sessions to have regained both confidentiality and authenticity. Thus, an adversary may apply the aforementioned attack strategies to either distribute an inbound session under their control (to break authenticity) or to capture a legitimate inbound session as it is distributed (to break confidentiality). Only once these pairwise links have healed, and all previously compromised Sender Keys sessions have been rotated out of the cache, can security start to recover.

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 would (generally) entail the compromise of its long-term secrets. It follows that practical compromise of the device state would, in most cases, compromise all secret key material it currently contains.

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 7.5). Instead, we map the corruption of a device (as in DOGM's $\mathsf{CorruptDevice}$ query) to the leaking of all of a device's current secrets: their identity keys (through $\mathsf{CorruptIdentity}$), their medium-term shared secrets (through $\mathsf{CorruptShared}$) and the current values of their session secrets (through $\mathsf{CorruptSession}$). This greatly simplifies our analysis, at the cost of deriving overly-restrictive security predicates in Section 7.5.

6 Ratcheted Symmetric Signcryption

In this section we introduce a new variant of the *symmetric signcryption* (SS) primitive defined in [36]: *ratcheted symmetric signcryption* (RSS). This captures a single stage of a unidirectional Sender Keys session, i.e. an instance of the UNI scheme as we describe in Section 3.3.

On a high-level the changes compared to [36] are as follows. First, the [36] 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 [36] 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. Thus, the FS-GAEAD formalism introduced in [6] is similar to ours in that it captures the ratcheting nature of a single stage 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 FS-GAEAD and the symmetric signcryption in SS.

We now turn to formalising our primitive.

Definition 26 (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 27 (Correctness of Ratcheted Symmetric Signcryption). *A ratcheted symmetric signcryption scheme, RSS, is correct if $\forall\ st_s^{(0)}, st_r^{(0)}$ from which $\mathsf{Gen}(1^\lambda) \rightarrow (st_s^{(0)}, st_r^{(0)})$, 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 (either sending or receiving) to maintain the security of historical ciphertexts.

Definition 28 (Confidentiality of Ratcheted Symmetric Signcryption). *Let RSS be a ratcheted symmetric signcryption scheme. We define the advantage of a probabilistic polynomial-time algorithm $\mathcal{A}$ in breaking the $\mathsf{PFS\text{-}OAE}$-security of RSS as*

$$\mathsf{Adv}^{\mathsf{PFS\text{-}OAE}}_{\mathsf{RSS},\mathcal{A}}(\lambda, n_q) = \Pr[\mathsf{Exp}^{\mathsf{PFS\text{-}OAE}}_{\mathsf{RSS},\lambda,n_q}(\mathcal{A}) = 1]$$

where $\mathsf{Exp}^{\mathsf{PFS\text{-}OAE}}_{\mathsf{RSS},\lambda,n_q}(\mathcal{A})$ is defined in Figure 21.

We say that RSS provides online authenticated-encryption with perfect-forward secrecy if $\mathsf{Adv}^{\mathsf{PFS\text{-}OAE}}_{\mathsf{RSS},\lambda,n_q}(\mathcal{A})$ is negligible in the security parameter λ for all PPT $\mathcal{A}$ (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 29 (Unforgeability of Ratcheted Symmetric Signcryption). *Let* RSS *be a ratcheted symmetric signcryption scheme. We define the advantage of a probabilistic polynomial-time algorithm $\mathcal{A}$ in breaking the* SUF-CMA*-security of* RSS *as*

$$\mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{RSS},\mathcal{A}}(\lambda, n_q) = \mathsf{Pr}[\mathsf{Exp}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{RSS},\lambda,n_q}(\mathcal{A}) = 1]$$

where $\mathsf{Exp}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{RSS},\lambda,n_q}(\mathcal{A})$ *is defined in Figure 21.*

We say that RSS *provides strong-unforgeability under chosen-message attack if* $\mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{RSS},\lambda,n_q}(\mathcal{A})$ *is negligible in the security parameter* λ *for all PPT* $\mathcal{A}$ *(when restricted to at most* n_q *queries to the* Send *oracle).*

```
$\mathsf{Exp}^{\mathsf{PFS\text{-}OAE}}_{\mathsf{RSS},\lambda,n_q}(\mathcal{A})$
1 : $b \leftarrow\!\$ \{0,1\};\ \mathit{corr\text{-}user} \leftarrow \mathbf{false}$
2 : $st_s, st_r \leftarrow\!\$ \mathsf{RSS.Gen}(\lambda)$
3 : $b' \leftarrow \mathcal{A}^{\mathsf{Send},\mathsf{Corrupt}}(st_r.pk)$
4 : $\mathbf{return}\ (b = b')$

Corrupt
1 : $\mathit{corr\text{-}user} \leftarrow \mathbf{true}$
2 : $\mathbf{return}\ st_s$

$\mathsf{Send}(m_0, m_1)$
1 : $\mathbf{if}\ |m_0| \neq |m_1| :\ \mathbf{return}\ \perp$
2 : $(st_s, c_b) \leftarrow\!\$ \mathsf{RSS.Signcrypt}(st_s, m_b)$
3 : $\mathbf{if}\ \mathit{corr\text{-}user} = \mathbf{false} :\ \mathbf{return}\ c_b$
4 : $\mathbf{return}\ \perp$
```

```
$\mathsf{Exp}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{RSS},\lambda,n_q}(\mathcal{A})$
1 : $b \leftarrow\!\$ \{0,1\};\ \mathit{win} \leftarrow 0; \mathbf{C} \leftarrow \emptyset$
2 : $st_s, st_r \leftarrow\!\$ \mathsf{RSS.Gen}(\lambda)$
3 : $\mathcal{A}^{\mathsf{Send},\mathsf{Receive}}(st_r)$
4 : $\mathbf{return}\ (\mathit{win})$

Receive (c)
1 : $st_r, m \leftarrow \mathsf{RSS.Unsigncrypt}(st_r, c)$
2 : $\mathbf{if}\ (m \neq \perp) \wedge (c \notin \mathbf{C}) :\ \mathit{win} \leftarrow 1$
3 : $\mathbf{if}\ (m \neq \perp) \wedge (c \in \mathbf{C}) :\ \mathbf{C} \leftarrow \mathbf{C} \backslash \{c\}$
4 : $\mathbf{return}\ m$

Send (m)
1 : $(st_s, c) \leftarrow\!\$ \mathsf{RSS.Signcrypt}(st_s, m)$
2 : $\mathbf{C} \leftarrow_{\cup} c$
3 : $\mathbf{return}\ c$
```

Fig. 21: Security Experiments for RSS.

We now analyse the security of UNI as a ratcheted symmetric signcryption scheme. We highlight that Theorem 4 relies on XEd25519, as it is used in WhatsApp and defined in [57], being a SUF-CMA-secure signature scheme with advantage $\mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda)$.

Definition 30. *WA-RSS is a ratcheted symmetric signcryption scheme that instantiates the RSS formalism with the algorithms (WA-RSS.Gen, WA-RSS.Signcrypt, WA-RSS.Unsigncrypt) detailed in Figure 22.*

We state and prove the security guarantees that WA-RSS provides.

Theorem 3 (Confidentiality of WA-RSS). *The WA-RSS protocol instantiates a ratcheted symmetric signcryption scheme that provides PFS-OAE security for which the advantage of any adversary* $\mathcal{A}$ *in winning the* $\mathsf{Exp}^{\mathsf{PFS\text{-}OAE}}_{\mathsf{WA\text{-}RSS},\lambda,n_q}(\mathcal{A})$ *security experiment is bound by*

$$\mathsf{Adv}^{\mathsf{PFS\text{-}OAE}}_{\mathsf{WA\text{-}RSS}}(\lambda, n_q) \ \leq\ n_q \cdot \big(2 \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC}}(\lambda, 1) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF}}(\lambda, 1) + \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{AES\text{-}CBC}}(\lambda, 1) \big)$$

under the assumptions that

```
WA-RSS.Gen(1^λ)
1 : $ck \leftarrow_\$ \{0,1\}^\lambda$; $(gsk, gpk) \leftarrow_\$ \mathsf{XDH.Gen}(1^\lambda)$
2 : $ust_{out} \leftarrow \mathsf{Obj}(ck, gsk)$; $ust_{in} \leftarrow \mathsf{Obj}(ck, gpk)$
3 : return $(ust_{out}, ust_{in})$
```

```
WA-RSS.Signcrypt(ust_out, m, ad)
1 : $\mathsf{Obj}(ck, gsk) \leftarrow ust_{out}$
2 : $mk \leftarrow \mathsf{HMAC}(ck, \texttt{0x01})$
3 : $ck \leftarrow \mathsf{HMAC}(ck, \texttt{0x02})$
4 : $k \leftarrow \mathsf{HKDF}(\varnothing, mk, \texttt{WhisperGroup}, 50\mathsf{B})$
5 : $iv, ek \leftarrow k[0 \rightarrow 15\mathsf{B}], k[16 \rightarrow 47\mathsf{B}]$
6 : $c \leftarrow \mathsf{AES\text{-}CBC.Enc}(ek, iv, m)$
7 : $c_U \leftarrow \mathsf{Obj}(\texttt{UNI-CTXT}, ad, c)$
8 : $\sigma_U \leftarrow \mathsf{XEd.Sign}(gsk, c_U)$
9 : $ust_{out} \leftarrow \mathsf{Obj}(ck, gsk)$
10 : return $ust_{out}, (c_U, \sigma_U)$
```

```
WA-RSS.Unsigncrypt(ust_in, (c_U, σ_U), ad)
1 : $\mathsf{Obj}(ck, gpk) \leftarrow ust_{in}$
2 : assert $\mathsf{XEd.Verify}(gpk, c_U, \sigma_U)$
3 : assert $ad = c_U.ad$
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 : $iv, ek \leftarrow k[0 \rightarrow 15\mathsf{B}], k[16 \rightarrow 47\mathsf{B}]$
8 : $m \leftarrow \mathsf{AES\text{-}CBC.Dec}(ek, iv, c_U.c)$
9 : assert $m \neq \perp$
10 : $ust_{in} \leftarrow \mathsf{Obj}(ck, gpk)$
11 : return $ust_{in}, m$
```

Fig. 22: A single group messaging stage in WhatsApp expressed within the RSS formalism. We modify the description of UNI in Figure 12 by removing the session identifier and message index, requiring the layer above to maintain and set the appropriate values for ad.

1) *HMAC can be modelled as a PRF for which the advantage of any PPT adversary $\mathcal{B}$ in winning the $\mathsf{Exp}^{\mathsf{PRF}}_{\mathsf{HMAC},\lambda,1}(\mathcal{B})$ experiment is bound by $\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC}}(\lambda, 1)$,*

2) *HKDF can be modelled as a PRF for which the advantage of any PPT adversary $\mathcal{B}$ in winning the $\mathsf{Exp}^{\mathsf{PRF}}_{\mathsf{HKDF},\lambda,1}(\mathcal{B})$ experiment is bound by $\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF}}(\lambda, 1)$, and*

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 $\mathsf{Exp}^{\mathsf{IND\text{-}CPA}}_{\mathsf{AES\text{-}CBC},\lambda,1}(\mathcal{B})$ experiment is bound by $\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{AES\text{-}CBC}}(\lambda, 1)$.*

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 PFS-OAE security game for ratcheted symmetric signcryption schemes instantiated with WA-RSS. This gives:

$$\mathsf{Adv}^{\mathsf{PFS\text{-}OAE}}_{\mathsf{WA\text{-}RSS}}(\lambda, n_q) \;=\; \mathsf{Adv}_{\mathsf{G0}}$$

Game 1. In this game we replace the message key mk and the chain key ck with uniformly random values $\widetilde{mk}$ and $\widetilde{ck}$ until $\mathcal{A}$ issues a Corrupt query.

Specifically, we introduce a reduction $\mathcal{B}_1$ that initialises a PRF challenger $\mathcal{C}_{\mathsf{PRF}}$ whenever $\mathcal{A}$ issues a $\mathsf{Send}(m_0, m_1)$ query. Since the initial ck is sampled uniformly at random, the initial $\mathcal{C}_{\mathsf{PRF}}$'s sampling of ck internally proceeds identically as the original security experiment. When computing mk and ck as the result of a Send query, $\mathcal{B}_1$ instead queries the most recently initialised $\mathcal{C}_{\mathsf{PRF}}$ with `0x01` and `0x02`, using the outputs to replace mk and ck (respectively). Upon receiving $\tilde{ck}$, $\mathcal{B}_1$ initialises a new PRF challenger. If $\mathcal{A}$ issues $\mathsf{Corrupt}$, then $\mathcal{B}_1$ simply returns the most recently computed $\tilde{ck}$ and no longer initialises PRF challengers.

We note that if the bit b sampled by $\mathcal{C}_{\mathsf{PRF}}$ is 0, then $\tilde{mk} = \mathsf{HMAC}(ck, \texttt{0x01})$ and $\tilde{ck} = \mathsf{HMAC}(ck, \texttt{0x02})$. Otherwise, $\tilde{mk}, \tilde{ck}$ are uniformly random values, and the iterative replacement is sound. Thus, any adversary that can efficiently distinguish between **Game 1** and **Game 0** can be turned into an efficient algorithm against the PRF assumption. Bounding the advantage of $\mathcal{B}_1$ by that of any such PPT adversary limited to at most n_q encryption queries, then we have:

$$\mathsf{Adv}_{\mathsf{G0}} \leq 2 \cdot n_q \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC}}(\lambda, 1) + \mathsf{Adv}_{\mathsf{G1}}$$

Game 2. In this game we replace the initialisation vector iv and encryption key k_e with uniformly random values $\tilde{iv}$, $\tilde{k}_e$ until $\mathcal{A}$ issues a $\mathsf{Corrupt}$ query.

Specifically, we introduce a reduction $\mathcal{B}_2$ that initialises a PRF challenger $\mathcal{C}_{\mathsf{PRF}}$ whenever $\mathcal{A}$ issues a $\mathsf{Send}(m_0, m_1)$ query. Since $\tilde{mk}$ is sampled uniformly by **Game 1**, the $\mathcal{C}_{\mathsf{PRF}}$'s sampling of $\tilde{mk}$ internally proceeds identically as in **Game 1**. When computing iv and k_e as the result of a Send query, $\mathcal{B}_2$ instead queries the most recently initialised $\mathcal{C}_{\mathsf{PRF}}$ with PRF(`WhisperGroup`), using the outputs to replace iv and k_e respectively. Upon computing $\tilde{mk}$, $\mathcal{B}_2$ initialises a new PRF challenger. If $\mathcal{A}$ issues $\mathsf{Corrupt}$, then $\mathcal{B}_2$ simply returns the most recently computed $\tilde{ck}$ and no longer initialises PRF challengers.

We note that if the bit b sampled by $\mathcal{C}_{\mathsf{PRF}}$ is 0, then $\tilde{iv}, \tilde{k}_e = \mathsf{HKDF}($ $0, mk, \texttt{WhisperGroup}, 50\mathsf{B})$. Otherwise, $\tilde{iv}$ and $\tilde{k}_e$ are uniformly random values, and the iterative replacement is sound. Thus, any adversary that can efficiently distinguish between **Game 2** and **Game 1** can be turned into an efficient algorithm against the PRF assumption on HKDF. Bounding the advantage of $\mathcal{B}_2$ by the advantage of any PPT adversary limited to at most n_q encryption queries, then we have:

$$\mathsf{Adv}_{\mathsf{G1}} \leq n_q \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF}}(\lambda, 1) + \mathsf{Adv}_{\mathsf{G2}}$$

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 $\mathsf{IND\text{-}CPA}$ security of $\mathsf{AES\text{-}CBC}$.

Specifically, we introduce a reduction $\mathcal{B}_3$ that initialises a $\mathsf{IND\text{-}CPA}$ challenger $\mathcal{C}_{\mathsf{IND\text{-}CPA}}$ whenever $\mathcal{A}$ issues a $\mathsf{Send}(m_0, m_1)$ query. Since $\tilde{k}_e$ is sampled uniformly randomly by **Game 2**, the $\mathcal{C}_{\mathsf{IND\text{-}CPA}}$'s sampling of $\tilde{k}_e$ internally proceeds identically as in **Game 2**. When computing $c \leftarrow \mathsf{AES\text{-}CBC.Enc}(k_e, iv, m_b)$ as the result of a Send query, $\mathcal{B}_3$ instead queries the most recently initialised $\mathcal{C}_{\mathsf{IND\text{-}CPA}}$ with (m_0, m_1, iv), using the outputs to replace the computation of the ciphertext c. Upon computing $\tilde{k}_e$, $\mathcal{B}_3$ initialises a new $\mathsf{IND\text{-}CPA}$ challenger. If $\mathcal{A}$ issues $\mathsf{Corrupt}$, then $\mathcal{B}_3$ simply returns the most recently computed $\tilde{ck}$ and no longer initialises $\mathsf{IND\text{-}CPA}$ challengers.

We note that if the bit b sampled by $\mathcal{C}_{\mathsf{IND\text{-}CPA}}$ is 0, then $c = \mathsf{AES\text{-}CBC.Enc}(k_e, iv, m_0)$, otherwise, $c = \mathsf{AES\text{-}CBC.Enc}(k_e, iv, m_1)$. Thus, any successful adversary that could detect any of these replacements can be turned into an efficient algorithm against the $\mathsf{IND\text{-}CPA}$ 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:

$$\mathsf{Adv}_{\mathsf{G2}} \;\leq\; n_q \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{AES\text{-}CBC}}(\lambda, 1)$$

This completes our proof. □

We find that the $\mathsf{WA\text{-}RSS}$ scheme provides authentication in the form of existential-unforgeability under chosen-message attack.

Theorem 4 (Unforgeability of $\mathsf{WA\text{-}RSS}$). *The $\mathsf{WA\text{-}RSS}$ protocol instantiates a ratcheted symmetric signcryption scheme that provides SUF-CMA security for which the advantage of any adversary $\mathcal{A}$ in winning the $\mathsf{Exp}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{WA\text{-}RSS},\lambda,n_q}(\mathcal{A})$ security experiment is bound by*

$$\mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{WA\text{-}RSS}}(\lambda, n_q) \;\leq\; \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, n_q)$$

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 $\mathsf{Exp}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{XEd},\lambda,n_q}(\mathcal{B})$ experiment is bound by $\mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, n_q)$.

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 $\mathsf{SUF\text{-}CMA}$ security game for ratcheted symmetric signcryption schemes instantiated with $\mathsf{WA\text{-}RSS}$. This gives:

$$\mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{WA\text{-}RSS}}(\lambda, n_q) \;=\; \mathsf{Adv}_{\mathsf{G0}}$$

Game 1. In this game we abort if the adversary outputs a message that decrypts successfully, but was not the output of a Send query.

Specifically, we introduce a reduction $\mathcal{B}$. At the beginning of the experiment, $\mathcal{B}$ initialises a $\mathsf{SUF\text{-}CMA}$ challenger $\mathcal{C}_{\mathsf{SUF\text{-}CMA}}$, and replaces the generation of the signing keys with $\mathcal{C}_{\mathsf{SUF\text{-}CMA}}$'s output public key pair. Whenever $\mathcal{A}$ queries $\mathsf{Send}(m)$, 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 $\mathsf{Receive}(c)$, $\mathcal{B}$ 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 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 $\mathsf{SUF\text{-}CMA}$ 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 $win \leftarrow$ `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. □

7 WhatsApp in the DOGM Model

Thanks to the architectural similarities between WhatsApp and Matrix, we adapt the DOGM model introduced in [3] for our security analysis. The DOGM models group messaging protocols for which each user may have multiple devices, capturing the relationship between users and their devices, as they change over time.

We briefly describe our extensions to the DOGM model that capture device revocation. We then describe the model and experiment in detail, highlighting any changes from the original. Next, we describe WhatsApp's multi-device group messaging functionality in terms of this model, and proceed to analyse its security.

7.1 Capturing Device Revocation

We augment the DOGM model to capture device revocation by adding the $\mathsf{DOGM.Rev}$ algorithm, which enables a user to revoke one of their linked devices. It is executed by the user, taking as input their user identifier and long-term secrets, generating a new public authenticator value for themselves and the device, an optional ciphertext that may be used to notify other protocol sessions and updating their private state.

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 $\mathsf{DOGM.Gen}$, 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 $\mathsf{DOGM.Reg}$ or $\mathsf{DOGM.Rev}$ 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 A revokes device $D_{A,i}$ 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 $D_{A,i}$ as originating from A.[27]

[27] A limitation of this model, as it stands, is that revoked devices are not able to be re-registered to a user in the future.

7.2 Device-Oriented Group Messaging Protocols

A DOGM protocol is a tuple of algorithms $\mathsf{DOGM} = (\mathsf{Gen}, \mathsf{Reg}, \mathsf{Init}, \mathsf{Add}, \mathsf{Remove}, \mathsf{Encrypt}, \mathsf{Decrypt})$ with additional functionality ($\mathsf{StateShare}$) [3]. For example, WhatsApp Sender Keys sessions add and remove *devices*, not users. They similarly send messages between groups of devices. As we increase the level of abstraction, operations work with users rather than devices (e.g. *users* primarily interact with the application at the level of users, not devices).

One can imagine an alternative design for multi-device group messaging for which users are the primary unit of action. In the case of WhatsApp, this could take the form of Sender Keys sessions between users, where a user's devices would share the necessary key material and/or plaintexts themselves, using a separate protocol.[28]

The DOGM model bridges the gap between lower-level device-oriented protocols and higher-level 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. This latter point becomes important when implementing features such as *history sharing*. Thus, we start by describing the $\mathsf{DOGM.Gen}$ and $\mathsf{DOGM.Reg}$ algorithms which are used to manage the cryptographic identities of users, their devices and the links between them:

- The *user creation* algorithm, Gen, models the generation of a user's long-term cryptographic identity.

  $$pk_A, sk_A \leftarrow\$ \mathsf{Gen}(A)$$

  It takes as input a user identifier, $A \in \mathcal{U}_{id}$, before outputting a public and secret authenticator pair, $pk_A \in \mathcal{P}_k$ and $sk_A \in \mathcal{S}_k$.

  This is instantiated in WhatsApp by the $\mathsf{WA.NewPrimaryDevice}$ algorithm.

  In the security experiment, the resulting user cryptographic identities are distributed honestly by the challenger. In practice, users rely on a combination of their trust in the WhatsApp server, manual out-of-band verification between users, or WhatsApp's key transparency mechanism to ensure this mapping is correct. Thus the provision of such public-key infrastructure is outside the scope of our analysis.

- 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).

  $$pk_A, sk_A, dpk_{A,i}, dsk_{A,i}, c \leftarrow\$ \mathsf{Reg}(A, D_{A,i}, sk_A, pk_A)$$

[28] 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 (and used a different protocol) to communication between users. Additionally, [23] have previously proposed a user-oriented design to implement multi-device functionality for Signal.

It takes as input a user identifier, $A \in \mathcal{U}_{id}$, the chosen device identifier, $D_{A,i} \in \mathcal{D}_{id}$ and the user's secret and public authenticators, $pk_A \in \mathcal{P}_k$ and $sk_A \in \mathcal{S}_k$, before returning the updated user authenticators, $pk_A \in \mathcal{P}_k$ and $sk_A \in \mathcal{S}_k$, new device authenticators, $dpk_{A,i} \in \mathcal{P}_k$ and $dsk_{A,i} \in \mathcal{S}_k$, alongside an optional ciphertext, $c \in \mathcal{C}$ (or a failure state $\perp$).

WA.NewCompanionDevice and WA.LinkDevice instantiate DOGM.Reg.

We do not model the out-of-band verification between the primary device and companion devices. Instead, the challenger simulates this functionality within the security experiment.

We augment the DOGM model [3] to allow users to revoke their linked devices via DOGM.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. 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 3.1).

- The *revocation* algorithm, Rev, enables a user to revoke one of their linked devices.

$$pk_A, sk_A, dpk_{A,i}, dsk_{A,i}, c \leftarrow\$ \; \mathsf{Rev}(A, D_{A,i}, pk_A, sk_A, dpk_{A,i}, dsk_{A,i})$$

It takes as input a user identifier, $A \in \mathcal{U}_{id}$, the identifier of the device to be removed, $D_{A,i} \in \mathcal{D}_{id}$, a user public and secret authenticator, $pk_A \in \mathcal{P}_k$ and $sk_A \in \mathcal{S}_k$, and a device public and secret authenticator, $dsk_{A,i} \in \mathcal{S}_k$ and $dpk_{A,i} \in \mathcal{S}_k$, then returns updated public and secret authenticator values for the user, $pk_A \in \mathcal{P}_k$ and $sk_A \in \mathcal{S}_k$, updated public and secret authenticator values for the device, $dpk_{A,i} \in \mathcal{P}_k$ and $dsk_{A,i} \in \mathcal{S}_k$, as well as an optional ciphertext, $c \in \mathcal{C}$.

Note that, while both WhatsApp and the public-key orbits formalism from Section 4 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 order to control the complexity of the model.

In the DOGM model, each group messaging session captures a series of unidirectional channels between one sender and many recipients.[29] A group can

[29] This maps closely to Sender Keys style group messaging, but does not naturally generalise outside of it. For example, MLS style group messaging protocols derive a shared secret used by all members of the group to send messages, which may be more naturally modelled as a single shared channel.

then be captured as a composition of these unidirectional channels, one for each group member.[30]

In the DOGM experiment, the user and device identities created using $\mathsf{DOGM.Gen}$ and $\mathsf{DOGM.Reg}$ can be re-used across many group messaging sessions, both within the same logical group and across multiple logical groups, all of which may run simultaneously. The following algorithms are used to initialise, manage and use such messaging sessions in the DOGM:

- 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.

  $$dsk_{A,i}, st, gid \leftarrow\!\!\$\ \mathsf{Init}(A, D_{A,i}, \rho, dsk_{A,i}, gid)$$

  This algorithm should be run once for each device that participates in a session, before Add, Remove, $\mathsf{Encrypt}$ or $\mathsf{Decrypt}$. It takes as input the user identifier, $A \in \mathcal{U}_{id}$, device identifier, $D_{A,i} \in \mathcal{D}_{id}$, role of the session $\rho \in \{\mathtt{snd}, \mathtt{rcv}\}$, a secret authenticator, $dsk_{A,i} \in \mathcal{S}_k$ and (optional) group identifier, $gid \in \mathcal{G}_{id}$, before returning an updated secret authenticator, $dsk_{A,i} \in \mathcal{S}_k$, session state, $st \in \mathcal{ST}$, and group identifier (or a failure state $\perp$).

- The *membership management* algorithms, Add and Remove, 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.

  $$dsk_{A,i}, st, c' \leftarrow\!\!\$\ \mathsf{Add}(dsk_{A,i}, st, A, D_{A,i}, c)$$
  $$dsk_{A,i}, st, c' \leftarrow\!\!\$\ \mathsf{Remove}(dsk_{A,i}, st, A, D_{A,i}, c)$$

  They take as input a secret authenticator, $dsk_{A,i} \in \mathcal{S}_k$, session state, $st \in \mathcal{ST}$, user identifier, $A \in \mathcal{U}_{id}$, device identifier, $D_{A,i} \in \mathcal{D}_{id}$, and (optional) ciphertext, $c \in \mathcal{C}$, before returning an updated secret authenticator, $dsk_{A,i} \in \mathcal{S}_k$, session state, $st \in \mathcal{ST}$, and (optional) ciphertext, $c \in \mathcal{C}$ (or a failure state $\perp$).

- The *messaging* algorithms, $\mathsf{Encrypt}$ and $\mathsf{Decrypt}$, model sending and receiving messages using an outbound (or inbound) session (respectively).

  $$dsk_{A,i}, st, c \leftarrow\!\!\$\ \mathsf{Encrypt}(dsk_{A,i}, st, m)$$
  $$dsk_{A,i}, st, m \leftarrow \mathsf{Decrypt}(dsk_{A,i}, st, c)$$

[30] Our security notion, described in Section 7.3, does not capture logical groups explicitly, however. Thus, 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 [8], where the security experiment ensures that sessions have a synchronized viewpoint of group membership.

The encryption algorithm takes as input a secret authenticator, $dsk_{A,i} \in \mathcal{S}_k$, session state, $st \in \mathcal{ST}$, and the message to be sent, $m \in \mathcal{M}$, before returning an updated secret authenticator, $dsk_{A,i} \in \mathcal{S}_k$, session state, $st \in \mathcal{ST}$, and ciphertext, $c \in \mathcal{C}$.

The decryption algorithm takes as input a secret authenticator, $dsk_{A,i} \in \mathcal{S}_k$, session state, $st \in \mathcal{ST}$, and ciphertext, $c \in \mathcal{C}$, before returning an updated secret authenticator, $dsk_{A,i} \in \mathcal{S}_k$, 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 DOGM model captures history sharing, where devices owned by the same user can share their messaging history. As we saw in Section 3.3, WhatsApp implements history sharing by directly sharing the plaintext transcript in a newly encrypted message.[31] State sharing functionality is captured with DOGM.StateShare:

- 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.

$$dsk_{A,i}, st, c \leftarrow\$ \ \mathsf{StateShare}(dsk_{A,i}, st, A, D_{A,i}, t, z, c)$$

  It takes as input a secret authenticator, $dsk_{A,i} \in \mathcal{S}_k$, secret state, $st \in \mathcal{ST}$, and (optionally) a user identifier, $A \in \mathcal{U}_{id}$, device identifier, $D_{A,i} \in \mathcal{D}_{id}$, 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, $dsk_{A,i} \in \mathcal{S}_k$, 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 $c = \top$.

A DOGM protocol is defined in terms of a number of data structures, which we represents as sets of the following types. Namely, (a) user identifiers, $\mathcal{U}_{id}$, (b) device identifiers, $\mathcal{D}_{id}$, (c) group identifiers, $\mathcal{G}_{id}$, (d) public authentication values, $\mathcal{P}_k$, (e) secret authentication values, $\mathcal{S}_k$, (f) session states, $\mathcal{ST}$, (g) plaintext messages, $\mathcal{M}$, and (h) ciphertext messages, $\mathcal{C}$.

We expect secret authentication values to have a particular structure, depending on whether they authenticate a user or a device. In other words, $\mathcal{S}_k = u\mathcal{S}_k \cup d\mathcal{S}_k$, where:

- $u\mathcal{S}_k = \mathbb{N} \times \{0,1\}^*$ with each element (γ, st) consisting of the user's current generation, $\gamma \in \mathbb{N}$, and some arbitrary state, $st \in \{0,1\}^*$.

[31] An alternative approach, taken by Matrix [3, 2], shares the key material for historical ciphertexts. This has the advantage of allowing clients to decrypt the original ciphertexts (while undermining some of the guarantees we would expect to receive from Matrix' key rotation mechanisms) [3].

- $d\mathcal{S}_k = \{0,1\}^*$ with each element consists of some arbitrary state $st \in \{0,1\}^*$.

We expect DOGM session state to have the following structure:

- $A \in \mathcal{U}_{id}$ – the user identifier for this party.
- $D_{A,i} \in \mathcal{D}_{id}$ – the device identifier for this party.
- $gid \in \mathcal{G}_{id}$ – the group identifier for this session.
- $\rho \in \{\mathtt{snd}, \mathtt{rcv}\}$ – the role of the party in the current session. Note that parties can be directed to act as either a snd or rcv in concurrent or subsequent sessions.
- $t \in \mathbb{N}$ – the current stage of the session.
- $z \in \mathbb{N}$ – the current message index of the current stage.
- $status \in \{\bot, \mathtt{active}, \mathtt{reject}\}$ – the status of the session, initialised by $\bot$.
- $CU[0], \ldots, CU[n_p] \in \mathcal{U}_{id}$ – the current set of intended communication partners, where $CU[0]$ is the sending user.
- $CD[0,0], \ldots, CD[0,n_d], \ldots CD[n_p,n_d] \in \mathcal{D}_{id}$ – the current set of devices associated with the communication partners $CU[0] \ldots CU[n_p]$, where $CD[0]$ is the sending device.
- $T[t,z] \in \mathcal{C}^* \cup \{\bot\}$ – the z-th message sent in the t-th stage sent or received by $\pi_s^{A,i}$. We use $|T[t]|$ as the shorthand for the first value z such that $T[t,z] = \bot$, or the number of messages accepted in the t-th stage.
- $st \in \{0,1\}^*$ – any additional state used by the session during protocol execution. We allow the members of st to be accessed directly as if they were direct members of π. In other words, if $\pi = (\ldots, st = (x,y))$, x and y can be accessed as $\pi.x$ and $\pi.y$ respectively.

In order to support device revocation, we additionally require that each session stores a 'Γ' field that maps from user identifier to their expected device list generation, $\Gamma = \mathsf{Map}\{A : \gamma\} \in \mathcal{U}_{id} \times \mathbb{N}$.

7.3 Security of Device-Oriented Group Messaging Protocols

The DOGM security experiment aims to determine the security guarantees of messages sent within groups, as the membership of groups and the make-up of a user's devices change over time, in the face various user and device secrets being leaked or compromised. This security experiment does not aim to provide assurances towards the consistent view of group membership among devices, nor does it aim to determine *additive or subtractive closeness* over group membership [60].

Intuitively, a message is considered authentic if it is correctly attributed to the user that sent it, it is linked to the logical group that it was sent to and the message contents cannot be modified by anyone other than the sending device. 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.

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. They may also corrupt device long-term secrets or session secrets during this period.

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 experiment, the adversary returns a guess as to whether the challenger was returning the first or second plaintexts during the experiment.

Since we allow the adversary to corrupt secrets, 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, CONF and AUTH, for confidentiality and authenticity, respectively.

State sharing can affect the security of the protocol in two key ways. First, it can allow malicious devices to inject or edit messages into a target device's message transcript, breaking the authentication of the protocol. Second, it can enable a malicious device to access the plaintext of (or distinguish between) challenge ciphertexts, breaking the confidentiality of the protocol. We must be careful to capture these properties in the security experiment without providing the adversary with functionality that undermines the security of the protocol.

Say, for example, that the adversary (a) submits a challenge to a session π, (b) orchestrate a state share to a different session $\pi_{\prime}$, before (c) compromising the second session. It is important that the challenger is able to determine that the compromise of $\pi_{\prime}$ gave the adversary access to a challenge ciphertext. Since the challenger stores a set of session states marked as challenge ciphertexts in $\mathbf{C}$, the state sharing oracle takes care to track the propagation of the challenge ciphertext by marking the recipient state in $\mathbf{C}$.

Security Experiment The security experiment, defined in detail by Figure 23, begins with $\mathcal{C}$ running $\mathsf{DOGM.Gen}(A)$ n_p times to generate a public key pair (pk_A, sk_A) for each party $A \in \{P_1, \ldots, P_{n_p}\}$ and delivers all public-keys pk_A to $\mathcal{A}$. $\mathcal{A}$ can now issue $\mathsf{CorruptUser}$ queries to compromise the user secrets. After, $\mathcal{C}$ randomly samples a bit $b \leftarrow\$ \{0,1\}$, sets a flag $win \leftarrow 0$, and interacts with $\mathcal{A}$ via the queries in $\mathcal{O}$ (except $\mathsf{CorruptUser}$).

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

We now describe how the adversary in the DOGM security experiment may interact with the challenger; in turn, describing how an attacker may interact with sessions of the WhatsApp protocol. The adversary interacts with $\mathcal{C}$ via the queries within $\mathcal{O}$ (each of which is defined in Figure 23). The adversary is in complete control of the communication network – able to modify, inject, delete or delay messages. It is through the interface of these queries that this control is defined.

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 forward secrecy (FS).

In the first stage of the experiment, the adversary is given access to a single oracle.[32]

- The *user corruption* oracle, $\mathsf{CorruptUser}$, gives the adversary access to the secret authenticator of the user $A[A]$. Providing the user exists, i.e. $0 \leq A < n_p - 1$, the challenger returns sk_A to the adversary.

$$\mathsf{CorruptUser}(A) \rightarrow \{sk_A, \perp\}$$

[32] The experiment has a pre-determined maximum number of users, all of which are initialised by the challenger before handing control over to the adversary.

```
$\mathsf{IND\text{-}CCA}^{\mathsf{DOGM}}_{n_p,n_d,n_i,n_s,n_m}(\mathcal{A})$

// Initialise challenge bit, $b$, win flag, $win$, challenges, $\mathbf{C}$, state sharing log, $\mathbf{Sh}$, device map, $\mathbf{D}$, & device session counter, $\mathbf{S}$.
$b \leftarrow\$ \{0,1\};\ win \leftarrow 0;\ \mathbf{C} \leftarrow 0;\ \mathbf{Sh} \leftarrow 0;\ \mathbf{D} \leftarrow \mathsf{Map}\{\cdot : \varnothing\};\ \mathbf{S} \leftarrow \mathsf{Map}\{\cdot : 0\}$
// Initialise $n_p$ user identities.
$\mathbf{for}\ 0 \le A < n_p :\quad pk_A, sk_A \leftarrow\$ \mathsf{DOGM.Gen}(A)$
// Allow the adversary to statically compromise a subset of users.
$ast \leftarrow \mathcal{A}^{\mathsf{CorruptUser}}(\texttt{exp-init}, [pk_0, pk_1, pk_2, \ldots, pk_{n_p-1}])$
// Next, give control back to the adversary who may now orchestrate a number of DOGM sessions, and manage user devices.
$b' \leftarrow \mathcal{A}^{\mathcal{O}\setminus\{\mathsf{CorruptUser}\}}(\texttt{exp-main}, ast)$
// Did adversary break authentication, by causing an oracle to set the $win$ flag, or confidentiality, by guessing challenge bit?
$\mathbf{if}\ win \lor (b = b' \land \mathsf{DOGM.CONF}) : \mathbf{return}\ 1$
$\mathbf{else} : \mathbf{return}\ 0$
```

```
$\mathsf{Create}(A, i)$   // Create new device and register with user

$\mathbf{assert}\ i \notin \mathbf{D}[A] \land \alpha^B_j \neq (\texttt{revoke}, \gamma)$
$dpk_{A,i}, dsk_{A,i}, pk_A, sk_A, c$
    $\leftarrow\$ \mathsf{DOGM.Reg}(A, i, sk_A, pk_A)$
// Pre-populate public user authenticators.
$dsk_{A,i}.\mathfrak{U} \leftarrow \mathsf{Map}\{B : pk_B\ \mathbf{for}\ 0 \le B < n_p\}$
$\mathbf{D}[A] \leftarrow\cup \{i\};\ \mathbf{S}[A,i] \leftarrow 0$
$\mathbf{return}\ dpk_{A,i}, c$
```

```
$\mathsf{Init}(A, i, \rho, gid)$   // New unidirectional session

$s \leftarrow \mathbf{S}[A,i] + 1;\ \mathbf{S}[A,i] \leftarrow s$
$dsk_{A,i}, \pi^s_{A,i} \leftarrow \mathsf{DOGM.Init}(A, i, \rho, dsk_{A,i}, gid)$
$\mathbf{return}\ s$
```

```
$\mathsf{AddMember}(A, i, s, B, j, c)$   // Add member

$dsk_{A,i}, \pi^s_{A,i}, c' \leftarrow\$ \mathsf{DOGM.Add}(dsk_{A,i}, \pi^s_{A,i}, B, j, c)$
$\mathbf{return}\ c'$
```

```
$\mathsf{RemoveMember}(A, i, s, B, j, c)$   // Remove member

$dsk_{A,i}, \pi^s_{A,i}, c' \leftarrow \mathsf{DOGM.Remove}(dsk_{A,i}, \pi^s_{A,i}, B, j, c)$
$\mathbf{return}\ c'$
```

```
$\mathsf{Revoke}(A, i)$

$pk_A, sk_A, dpk_{A,i}, dsk_{A,i}, c$
    $\leftarrow \mathsf{DOGM.Rev}(A, i, pk_A, sk_A, dpk_{A,i}, dsk_{A,i})$
$\alpha^B_j \leftarrow (\texttt{revoke}, sk_A.\gamma)$
$\mathbf{return}\ c$
```

```
$^*\mathsf{Revoked?}(A, i, s, B, j, \gamma') :=$
  $(\alpha^B_j = (\texttt{revoke}, \gamma))\ \mathbf{st}\ \gamma' \ge \gamma)$
  $\land\ (B \notin corr\text{-}user)$
```

```
$\mathsf{Encrypt}(A, i, s, m_0, m_1)$   // Encrypt message for session

$\mathbf{if}\ |m_0| \neq |m_1| : \mathbf{return}\ \bot$
$dsk_{A,i}, \pi^s_{A,i}, c \leftarrow\$ \mathsf{DOGM.Encrypt}(dsk_{A,i}, \pi^s_{A,i}, m_b)$
$\mathbf{if}\ c = \bot : \mathbf{return}\ \bot$
// Has message been encrypted for a known revoked device?
$\mathbf{if}\ \exists\ (B, j) \in \pi^{A,i}_s.CD\ \mathbf{st}\ ^*\mathsf{Revoked?}(A, i, s, B, j, \pi^{A,i}_s.\Gamma[B]) :$
    $win \leftarrow 1$
$\mathbf{if}\ m_0 \neq m_1 : \mathbf{C} \leftarrow\cup (c, A, i, s, \pi^{A,i}_s.t, \pi^{A,i}_s.z)$
$\mathbf{return}\ c$
```

```
$\mathsf{Decrypt}(A, i, s, c)$   // Receive message for session

$dsk_{A,i}, \pi^s_{A,i}, m \leftarrow \mathsf{DOGM.Decrypt}(dsk_{A,i}, \pi^s_{A,i}, c)$
$B, j \leftarrow \pi^{A,i}_s.CD[0];\ t, z, \cdot \leftarrow \pi^s_{A,i}.T|-1|;\ \gamma \leftarrow \pi^{A,i}_s.\Gamma[B]$
$\mathbf{if}\ m = \bot :$   // non-application message or failure
    $\mathbf{return}\ \bot$
// Sent by known revoked device?
$revoked \leftarrow {}^*\mathsf{Revoked?}(A, i, s, B, j, \gamma)$
$replay \leftarrow (c \in \pi^s_{A,i}.T)$
$forgery \leftarrow {}^*\mathsf{Forgery?}(A, i, s, B, j, c)$
$win \leftarrow revoked \lor replay \lor (forgery \land$
    $\mathsf{DOGM.AUTH}[A, i, s, \gamma, B, j, t, z])$
$\mathbf{if}\ (c, \cdot)\ \mathbf{in}\ \mathbf{C} : \mathbf{return}\ \bot$
$\mathbf{return}\ m$
```

```
$\mathsf{StateShare}(A, i, s, t, z, B, j, c)$   // Orchestrate state share

$dsk_{A,i}, \pi^s_{A,i}, c'$
    $\leftarrow\$ \mathsf{DOGM.StateShare}(dsk_{A,i}, \pi^s_{A,i}, B, j, t, z, c)$
// State sharing with known revoked device?
$\mathbf{if}\ c' \neq \bot \land {}^*\mathsf{Revoked?}(A, i, s, B, j, \pi^{A,i}_s.\Gamma[B]) : win \leftarrow 1$
$\mathbf{if}\ c' = \top :$   // $\pi^{A,i}_s$ accepted state share
    $injection \leftarrow \nexists\ (\cdot, c, B, j, \cdot, t, z, A, i, \cdot)\ \mathbf{in}\ \mathbf{Sh}$
    $win \leftarrow (injection \land$
        $\mathsf{DOGM.AUTH}[A, i, s, \pi^{A,i}_s.\Gamma[B], \pi^{A,i}_s.CD[0], t, z])$
$\mathbf{Sh} \leftarrow\cup (c, c', A, i, s, t, z, B, j, \pi^{A,i}_s.\Gamma[B])$
$\mathbf{return}\ c'$
```

```
$\mathsf{CorruptUser}(A)$

$corr\text{-}user \leftarrow\cup A;$
$\mathbf{return}\ sk_A$
```

```
$\mathsf{CorruptDevice}(A, i)$

$corr\text{-}dev \leftarrow\cup (A, i)$
$\mathbf{return}\ dsk_{A,i}$
```

```
$\mathsf{Compromise}(A, i, s)$

$corr\text{-}sess \leftarrow\cup (A, i, s, \pi^{A,i}_s.t, \pi^{A,i}_s.z)$
$\mathbf{return}\ \pi^s_{A,i}$
```

```
$^*\mathsf{Forgery?}(A, i, s, B, j, c) :=$
  $\nexists (r, t, z)\ \mathbf{st}\ c \in \pi^r_{B,j}.T[t, z]$   // Does there not exist an honest session that sent a matching
  $\land (\pi^s_{A,i}.\rho = \texttt{rcv} \land \pi^r_{B,j}.\rho = \texttt{snd})$   // ciphertext, is itself a sending session for which
  $\land (B = \pi^s_{A,i}.CU[0] \land (B, j) = \pi^s_{A,i}.CD[0])$   // recipient marked as its sending session,
  $\land (A \in \pi^r_{B,j}.CU \land (A, i) \in \pi^r_{B,j}.CD)$   // and recipient is one of the intended recipients?
```

Fig. 23: Security experiment defining the security of DOGM protocols with revocation. We highlight changes from the original DOGM security experiment.

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, Create, allows the adversary to trigger the creation of new devices.

$$\mathsf{Create}(A, i) \xrightarrow{\$} \{dpk_{A,i}, \perp\}$$

- The *session initialisation* oracle, Init, allows the adversary to trigger the initialisation of a new session for the device $D_{A,i}$ representing the user A.

$$\mathsf{Init}(A, i, \rho, gid) \xrightarrow{\$} \{(s, gid), (\perp)\}$$

- The *member addition* oracle, AddMember, allows the adversary to direct a session, $\pi_s^{A,i}$, to add a device, $D_{B,j}$, representing the party B, to its group messaging session.

$$\mathsf{AddMember}(A, i, s, B, j, c) \xrightarrow{\$} \{c, \perp\}$$

- $\mathsf{AddMember}(A, i, s, B, j, c) \mapsto \{c', \perp\}$: allows $\mathcal{A}$ to direct session $\pi_s^{A,i}$ to add a new device $D_{B,j}$ owned by party B to their group messaging session.
- The *member removal* oracle, RemoveMember, allows the adversary to direct a session, $\pi_s^{A,i}$, to remove a device, $D_{B,j}$, representing the party B, from its group messaging session.

$$\mathsf{RemoveMember}(A, i, s, B, j, c) \xrightarrow{\$} \{c, \perp\})$$

- The *device revocation* oracle, Revoke, allows the adversary to direct party A to revoke device (A, i). This, in turn, triggers the challenger to execute the revocation algorithm, Rev, on behalf of party A and return the resulting ciphertext, if output, to the adversary.

$$\mathsf{Revoke}(A, i) \mapsto \{c, \perp\}$$

- The *encryption* oracle, 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.

$$\mathsf{Encrypt}(A, i, s, m_0, m_1) \xrightarrow{\$} \{c, \perp\}$$

- The *decryption* oracle, Decrypt, allows the adversary to direct a session $\pi_s^{A,i}$ to process a message and receive its output.

$$\mathsf{Decrypt}(A, i, s, c) \rightarrow \{m', \perp\}$$

- The *state sharing* oracle, 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.

$$\mathsf{StateShare}(A, i, s, B, j, t, z, c) \xrightarrow{\$} \{c', \bot\}$$

- The *device corruption* oracle, CorruptDevice, gives the adversary access to the current value of the secret device authenticator value, $dsk_{A,i}$. $\mathcal{C}$ returns $dsk_{A,i}$ to $\mathcal{A}$.

$$\mathsf{CorruptDevice}(A, i) \rightarrow \{dsk_{A,i}, \bot\}$$

- The *user corruption* oracle, CorruptUser, gives the adversary access to the current session state of the session $\pi_s^{A,i}$.

$$\mathsf{Compromise}(A, i, s) \rightarrow \{\pi_s^{A,i}, \bot\}$$

The DOGM security experiment is parameterised by $\Lambda_{\mathsf{DOGM}} = (n_p, n_d, n_i, n_s, n_m)$ where

- n_p is the number of cryptographic identities for users in the experiment,
- n_d is the maximum number of devices (each identified by a cryptographic identity) that may be associated with each user in the experiment,
- n_i is the maximum number of DOGM protocol sessions the adversary may initiate (for each device),
- n_s is the maximum number of messaging stages allowed in each protocol session, and
- n_m is the maximum number of messages that may be sent in each stage.

We use $\pi_s^{A,i}$ both as an identifier for the s-th DOGM session executed by A's device $D_{A,i}$ and to identify the collection of per-session variables that it maintains.

7.4 Expressing WhatsApp as a DOGM Protocol

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 31. *WA-DOGM is a device-oriented group messaging protocol with revocation that instantiates the DOGM with revocation formalism with algorithms (Gen, Reg, Rev, Init, Add, Remove, Encrypt, Decrypt, StateShare) in Figures 24 to 26.*

```
$\mathsf{Gen}(ipk_p \overset{is}{=} \varnothing)$   // New primary device
1 :  $isk_p, ipk_p, orb \leftarrow\!\!\$\ \mathsf{WA\text{-}PO.Setup}()$
2 :  $sk_A \leftarrow \mathsf{Obj}(isk_p : isk_p,\ orb : orb,\ \gamma : orb.\gamma)$
3 :  $pk_A \leftarrow \mathsf{Obj}(ipk_p : ipk_p,\ orb : orb,\ \gamma : orb.\gamma)$
4 :  return $pk_A,\ sk_A$
```

```
$\mathsf{Reg}(ipk_p, ipk_c \overset{is}{=} \varnothing, sk_A, pk_A)$   // Register companion device
1 :  $pst, (xpk, spk), epks \leftarrow\!\!\$\ \mathsf{WA\text{-}PAIR.Init}(1^\lambda)$
2 :  $\sigma_{xpk} \leftarrow \mathsf{XEd.Sign}(pst.isk, \mathtt{FICTION} \parallel xpk)$
3 :  $\sigma_{spk} \leftarrow \mathsf{XEd.Sign}(pst.isk, \mathtt{0x05} \parallel spk)$
4 :  $skb \leftarrow \mathsf{Obj}(\mathsf{SKB}, pst.ipk, spk, \sigma_{spk}, epks, xpk, \sigma_{xpk})$
5 :  $sk_A.orb \leftarrow \mathsf{WA\text{-}PO.Attract}(sk_A.isk_p, pst.ipk_c, sk_A.orb)$
6 :  $sk_A.orb \leftarrow \mathsf{WA\text{-}PO.Attract}(pst.isk_c, pk_A.ipk_p, sk_A.orb)$
7 :  $sk_A.\gamma \leftarrow sk_A.orb.\gamma$
8 :  $pk_A.orb \leftarrow sk_A.orb$
9 :  $pk_A.\gamma \leftarrow sk_A.\gamma$
10 : $dpk_{A,i} \leftarrow \mathsf{Obj}($
11 :     $ipk_p : pk_A.ipk_p,\ ipk : pst.ipk,$
12 :     $orb : sk_A.orb,\ skb : skb,\ \gamma : orb.\gamma)$
13 : $dsk_{A,i} \leftarrow \mathsf{Obj}($
14 :     $ipk_p : pk_A.ipk_p,\ isk : pst.isk,\ ipk : pst.ipk,\ pst : pst,$
15 :     $\mathcal{PO} : \mathsf{Map}\{ipk_p : orb\},\ \mathcal{SKB} : \mathsf{Map}\{pst.ipk : skb\},$
16 :     $\gamma : orb.\gamma)$
17 : return $dpk_{A,i},\ dsk_{A,i},\ pk_A,\ sk_A,\ (dpk_{A,i})$
```

```
$\mathsf{Reg}(ipk_p, ipk_c \overset{is}{=} ipk_p, sk_A, pk_A)$   // Register primary device
1 :  $pst, (xpk, spk), epks \leftarrow\!\!\$\ \mathsf{WA\text{-}PAIR.Init}(1^\lambda)$
2 :  $\sigma_{xpk} \leftarrow \mathsf{XEd.Sign}(sk_A.isk_p, \mathtt{FICTION} \parallel xpk)$
3 :  $\sigma_{spk} \leftarrow \mathsf{XEd.Sign}(sk_A.isk_p, \mathtt{0x05} \parallel spk)$
4 :  $skb \leftarrow \mathsf{Obj}(\mathsf{SKB}, pk_A.ipk_p, spk, \sigma_{spk}, epks, xpk, \sigma_{xpk})$
5 :  $dpk_{A,i} \leftarrow \mathsf{Obj}($
6 :      $ipk_p : pk_A.ipk_p,\ ipk : pk_A.ipk_p,$
7 :      $orb : pk_A.orb,\ skb : skb,\ \gamma : orb.\gamma)$
8 :  $dsk_{A,i} \leftarrow \mathsf{Obj}($
9 :      $ipk_p : pk_A.ipk_p,\ isk : sk_A.isk_p,\ ipk : pk_A.ipk_p,$
10 :     $\mathcal{PO} : \mathsf{Map}\{ipk_p : orb\},\ \mathcal{SKB} : \mathsf{Map}\{ipk_p : skb\},$
11 :     $pst : pst,\ \gamma : orb.\gamma)$
12 : return $dpk_{A,i},\ dsk_{A,i},\ pk_A,\ sk_A,\ (dpk_{A,i})$
```

```
$\mathsf{Rev}(ipk_p, ipk_c, pk_A, sk_A, dpk_{A,i}, dsk_{A,i})$   // $ipk_p$ revokes $ipk_c$
1 :  $sk_A.orb \leftarrow \mathsf{WA\text{-}PO.Repel}(sk_A.isk, dpk_{A,i}.ipk_c, sk_A.orb)$
2 :  $sk_A.\gamma \leftarrow sk_A.orb.\gamma$
3 :  $pk_A.orb \leftarrow sk_A.orb$
4 :  $pk_A.\gamma \leftarrow sk_A.\gamma$
5 :  return $pk_A, sk_A, dpk_{A,i}, dsk_{A,i}, (pk_A)$
```

Fig. 24: Device management in WhatsApp expressed within the DOGM with revocation formalism.

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 keys used as the device identifier has not been generated yet). An exception to this is when registering the 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 $\mathcal{SKB}$. We store these in *isk* merely for the convenience to be available whenever we need them. Note that, modulo cryptography controls, the adversary can fill these at will by sending the appropriate ciphertexts of type PO or SKB, cf. Decrypt.
- 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. In each user public authenticator, we store 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 3. As discussed in Sections 4 and 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 3 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 session 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 3, which described the behaviour of a WhatsApp client, managing any number of simultaneous messaging sessions. 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 handled by the DOGM challenger on behalf of the protocol, the identifier no longer serves a functional purpose.

- Add and Remove 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 3. Instead we offload this responsibility to the challenger in the DOGM security experiment. Thus, these algorithms lean heavily on the SK and UNI primitives.

- Encrypt and Decrypt model sending and receiving messages using an outbound (or inbound) session (respectively).

  Note that we require minor changes to the SK algorithms described in Figure 13. We therefore replace the use of PAIR with WA-PAIR and UNI with WA-RSS. For the latter change, since the WA-RSS formalism does not capture the notion of stage or message index, we must lift the initialisation and maintenance of the stage identifier, *usid*, and message index, z, to

```
$\mathsf{Init}(ipk_p,\ ipk,\ \rho \overset{is}{=} \mathtt{snd},\ dsk_{A,i},\ gid)$
$skst \leftarrow \mathsf{SK.Init}(\mathtt{snd}, ipk, \emptyset)$
$\pi \leftarrow \mathsf{Obj}(gid : gid,\ A : ipk_p,\ D_{A,i} : ipk,\ \rho : \mathtt{snd},\ status : \mathtt{active},$
    $t : skst.t,\ z : skst.z,\ CU : [ipk_p],\ CD : [(ipk_p, ipk)],$
    $T : \mathsf{Map}\{\},\ \Gamma : \mathsf{Map}\{ipk_p : dsk_{A,i}.\gamma\},\ ipk_p : dsk_{A,i}.ipk_p,$
    $ipk : dsk_{A,i}.ipk,\ skst : skst,\ hist : [\,])$
$\mathbf{return}\ dsk_{A,i}, \pi$
```

```
$\mathsf{Add}(dsk_{A,i},\ \pi \overset{is}{=} \mathsf{Obj}(\rho : \mathtt{snd}), ipk_{p*},\ ipk_{c*},\ \varnothing)$
$\mathbf{assert}\ ipk_{c*} \in \{ipk_{p*}\}$
    $\vee\ ipk_{c*} \in \mathsf{WA\text{-}PO.Orbit}(ipk_{p*}, \pi.\Gamma[ipk_{p*}], isk.\mathcal{PO}[ipk_{p*}])$
$skb_c^* \leftarrow isk.\mathcal{SKB}[ipk_{c*}]$
$\mathbf{assert}\ \mathsf{XEd.Verify}(skb_c^*.ipk, \mathtt{0x05} \,\|\, skb_c^*.spk, skb_c^*.\sigma_{spk})$
$\mathbf{assert}\ \mathsf{XEd.Verify}(skb_c^*.ipk, \mathtt{FICTION} \,\|\, skb_c^*.xpk, skb_c^*.\sigma_{xpk})$
$\pi.skst \leftarrow \mathsf{SK.Add}(\pi.skst, ipk_{c*})$
$\pi.CU \leftarrow_{app} \{ipk_{p*}\};\ \pi.CD \leftarrow_{app} \{(ipk_{p*}, ipk_{c*})\}$
$\pi.t \leftarrow \pi.skst.t;\ \pi.z \leftarrow \pi.skst.z$
$\mathbf{return}\ dsk_{A,i},\ \pi,\ \perp$
```

```
$\mathsf{Remove}(dsk_{A,i},\ \pi \overset{is}{=} \mathsf{Obj}(\rho : \mathtt{snd}),\ ipk_{p*},\ ipk_{c*},\ \varnothing)$
$\pi.CD \leftarrow\setminus \{(ipk_{p*}, ipk_{c*})\}$
$\mathbf{if}\ \nexists\ (ipk_{p*}, \cdot\,) \in \pi.CD :$
    $\pi.CU \leftarrow\setminus \{ipk_{p*}\}$
$\pi.skst \leftarrow \mathsf{SK.Rem}(\pi.skst, ipk_{c*})$
$\pi.t \leftarrow \pi.skst.t$
$\pi.z \leftarrow \pi.skst.z$
$\mathbf{return}\ dsk_{A,i},\ \pi,\ \varnothing$
```

```
$\mathsf{Encrypt}(dsk_{A,i},\ \pi \overset{is}{=} \mathsf{Obj}(\rho : \mathtt{snd}),\ m)$
$meta \leftarrow \mathsf{ICDC.Generate}($
    $\pi.ipk_p, \pi.\Gamma, \pi.skst.mem, dsk_{A,i}.\mathcal{PO})$
$\pi.skst, \pi.pst, (c_P, c_U) \leftarrow\$\ \mathsf{SK.Enc}($
    $\pi.skst, \pi.pst, dsk_{A,i}.\mathcal{SKB}, meta, m)$
$\pi.t \leftarrow \pi.skst.t;\ \pi.z \leftarrow \pi.skst.z$
$\pi.T[\pi.t, \pi.z] \leftarrow_{app} (c_P, c_U)$
$\pi.hist \leftarrow_{app} (\pi.ipk_p, \pi.ipk, \pi.t, \pi.z, m)$
$\mathbf{return}\ dsk_{A,i},\ \pi,\ (c_P, c_U)$
```

```
$^*\mathsf{ProcessDL}(dsk_{A,i},\ \pi,\ orb^*)$
$ipks_{\checkmark} \leftarrow \{ipk_{p*}\} \cup \mathsf{WA\text{-}PO.Orbit}(orb^*.ipk_p, \pi.\Gamma[orb^*.ipk_p], orb^*)$
$\mathbf{if}\ ipks_{\checkmark} \neq \perp :$
    $\mathbf{assert}\ orb^*.ipk_p = dsk_{A,i}.\mathfrak{U}[orb^*.ipk_p].ipk_p$
    $\pi.\mathcal{PO}[orb^*.ipk_p] \leftarrow orb^*$
    $\pi.\Gamma[orb^*.ipk_p] \leftarrow \mathsf{max}(orb^*.\gamma, \pi.\Gamma[orb^*.ipk_p])$
$\mathbf{if}\ \pi.\rho = \mathtt{snd} :$
    $\mathbf{for}\ (ipk_p^*, ipk_c^*) \in \pi.CD$
            $\mathbf{st}\ ipk_p^* = orb^*.ipk_p \wedge ipk_{c*} \notin ipks_{\checkmark} :$
        $dsk_{A,i}, \pi \leftarrow \mathsf{Remove}(dsk_{A,i}, \pi, ipk_p^*, ipk_c^*)$
$\mathbf{return}\ dsk_{A,i},\ \pi$
```

```
$\mathsf{Init}(ipk_p,\ ipk,\ \rho \overset{is}{=} \mathtt{rcv},\ dsk_{A,i},\ gid)$
$\pi \leftarrow \mathsf{Obj}(gid : gid,\ A : ipk_p,\ D_{A,i} : ipk,\ \rho : \mathtt{rcv},\ status : \mathtt{active},$
    $t : \varnothing,\ z : \varnothing,\ CU : [\varnothing, ipk_p],\ CD : [\varnothing, (ipk_p, ipk)],$
    $T : \mathsf{Map}\{\},\ \Gamma : \mathsf{Map}\{ipk_p : dsk_{A,i}.\gamma\},\ ipk_p : dsk_{A,i}.ipk_p,$
    $ipk : dsk_{A,i}.ipk,\ skst : skst,\ hist : [\,], usid : \mathsf{Map}\{\})$
$\mathbf{return}\ dsk_{A,i}, \pi$
```

```
$\mathsf{Add}(dsk_{A,i},\ \pi \overset{is}{=} \mathsf{Obj}(\rho : \mathtt{rcv}), ipk_{p*},\ ipk_{c*},\ \varnothing)$
$\mathbf{assert}\ \pi.CU[0] = \pi.CD[0] = \varnothing$
$\mathbf{assert}\ ipk_{c*} \in \{ipk_{p*}\}$
    $\vee\ ipk_{c*} \in \mathsf{WA\text{-}PO.Orbit}(ipk_{p*}, \pi.\Gamma[ipk_{p*}], dsk_{A,i}.\mathcal{PO}[ipk_{p*}])$
$skb_c^* \leftarrow dsk_{A,i}.\mathcal{SKB}[ipk_{c*}]$
$\mathbf{assert}\ \mathsf{XEd.Verify}(skb_c^*.ipk, \mathtt{0x05} \,\|\, skb_c^*.spk, skb_c^*.\sigma_{spk})$
$\mathbf{assert}\ \mathsf{XEd.Verify}(skb_c^*.ipk, \mathtt{FICTION} \,\|\, skb_c^*.xpk, skb_c^*.\sigma_{xpk})$
$\pi.skst \leftarrow \mathsf{SK.Init}(\mathtt{rcv}, ipk_{c*}, \emptyset)$
$\pi.CU[0] \leftarrow ipk_{p*}$
$\pi.CD[0] \leftarrow (ipk_{p*}, ipk_{c*})$
$\mathbf{return}\ dsk_{A,i},\ \pi,\ \varnothing$
```

```
$\mathsf{Decrypt}(dsk_{A,i}, \pi, C \overset{is}{=} (c_P, c_U))$
// Process protocol ciphertext (if it is a recipient session)
$\mathbf{assert}\ \pi.\rho = \mathtt{rcv}$
$ipk_{p*}, ipk_{c*} \leftarrow \pi.CD[0]$
$skb_* \leftarrow dsk_{A,i}.\mathcal{SKB}[ipk_{c*}]$
$\mathbf{assert}\ \mathsf{XEd.Verify}(ipk_{c*}, \mathtt{0x05} \,\|\, skb_*.spk, skb_*.\sigma_{spk})$
$\mathbf{assert}\ \mathsf{XEd.Verify}(ipk_{c*}, \mathtt{FICTION} \,\|\, skb_*.xpk, skb_*.\sigma_{xpk})$
$\pi.skst, dsk_{A,i}.pst, meta, m \leftarrow \mathsf{SK.Dec}($
        $\pi.skst, dsk_{A,i}.pst, skb_*, c_P, c_U)$
$\mathbf{assert}\ m \neq \perp$
$\pi.\Gamma \leftarrow \mathsf{ICDC.Process}(\pi.ipk_p, \pi.\Gamma, ipk_{p*}, meta, dsk_{A,i}.\mathcal{PO})$
$\mathbf{assert}\ ipk_{c*} \in \{ipk_{p*}\}$
    $\vee\ ipk_{c*} \in \mathsf{WA\text{-}PO.Orbit}(ipk_{p*}, \pi.\Gamma[ipk_{p*}], dsk_{A,i}.\mathcal{PO}[ipk_{p*}])$
$t^\star, ust_{in}^\star \leftarrow {}^*\mathsf{SK.FindSession}(\pi.skst.usts_{in}, c_U.usid)$
$z^\star \leftarrow ust_{in}^\star.z$
$\mathbf{if}\ t^\star > \pi.t :$    // record new stage
    $\pi.t \leftarrow t^\star$
    $\pi.usid[t^\star] \leftarrow ust_{in}^\star.usid$    // assoc from stage to session
$\mathbf{if}\ t^\star = \pi.t :$    // move message index forward
    $\pi.z \leftarrow \mathsf{max}[\pi.z, z^\star]$
$\pi.T[t^\star, z^\star] \leftarrow (c_P, c_U)$
$\pi.hist \leftarrow_{app} (ipk_{p*}, ipk_{c*}, t^\star, z^\star, m)$
$\mathbf{return}\ dsk_{A,i},\ \pi,\ m$
```

```
$\mathsf{Decrypt}(dsk_{A,i}, \pi, c_S)$
// Process server-provided updates to public state
$\mathbf{if}\ c_S\ \wedge c_S.type = \mathsf{PO} :$
    $dsk_{A,i}, \pi \leftarrow {}^*\mathsf{ProcessDL}(dsk_{A,i}, c_S)$
$\mathbf{if}\ c_S\ \wedge c_S.type = \mathsf{SKB}\ \wedge c_S.ipk \neq dsk_{A,i}.ipk :$
    $dsk_{A,i}.\mathcal{SKB}[c_S.ipk] \leftarrow c_S$
$\mathbf{return}\ dsk_{A,i},\ \pi,\ \perp$
```

Fig. 25: Group messaging in WhatsApp expressed within the DOGM with revocation formalism.

```
$\mathsf{StateShare}(dsk_{A,i},\ \pi,\ ipk_{p*},\ ipk_{c*},\ t,\ z,\ c \overset{is}{=} \varnothing)$
$\textbf{assert}\ \pi.ipk_p = \pi.ipk = ipk_{p*}$   // Ensure this is primary device of requestee
$\textbf{assert}\ ipk_{c*} \in \{ipk_{p*}\}$
    $\vee\ ipk_{c*} \in \mathsf{WA\text{-}PO.Orbit}(ipk_{p*}, \pi.\Gamma[ipk_{p*}], dsk_{A,i}.\mathcal{PO}[ipk_{p*}])$
$hist' \leftarrow [\,(ipk'_p, ipk'_c, t', z', m') \in \pi.hist\ :\ t' = t \wedge z' \geq z\,]$
$dsk_{A,i}, pst, c_P, c_{hist}, \tau_{hist} \leftarrow \mathsf{HS.Share}(\texttt{primary}, dsk_{A,i}.pst, dsk_{A,i}.\mathcal{SKB}[ipk_{c*}], hist')$
$\textbf{return}\ dsk_{A,i},\ \pi,\ (c_P, c_{hist}, \tau_{hist})$

$\mathsf{StateShare}(dsk_{A,i},\ \pi,\ ipk_{p*},\ ipk_{c*},\ t,\ z,\ c \overset{is}{=} (c_P, c_{hist}, \tau_{hist}))$
$\textbf{assert}\ \pi.ipk_p = ipk_{p*} = ipk_{c*}$   // Ensure sharer is our primary device
$skb \leftarrow dsk_{A,i}.\mathcal{SKB}[\pi.ipk_p]$
$\textbf{assert}\ \mathsf{XEd.Verify}(\pi.ipk_p, \texttt{0x05} \,\|\, skb.spk, skb.\sigma_{spk})$   // Check signed pre-key signatures
$\textbf{assert}\ \mathsf{XEd.Verify}(\pi.ipk_p, \texttt{FICTION} \,\|\, skb.xpk, skb.\sigma_{xpk})$
$dsk_{A,i}.pst, hist' \leftarrow \mathsf{HS.Receive}(\texttt{companion}, dsk_{A,i}.pst, skb, c_P, c_{hist}, \tau_{hist})$
$\textbf{if}\ hist' = \bot :$
    $\textbf{return}\ dsk_{A,i},\ \pi,\ \bot$
$\pi.hist \leftarrow_{app} [\,(ipk'_p, ipk'_c, t', z', m')\ \textbf{in}\ hist'\ \textbf{st}\ (ipk'_p, ipk'_c) = \pi.CD[0] \wedge t' = t \wedge z' \geq z\,]$
$\textbf{return}\ dsk_{A,i},\ \pi,\ \top$
```

Fig. 26: State sharing in WhatsApp expressed within the DOGM with revocation formalism.

the SK algorithms. We must, additionally, ensure that these values are included as the additional data given to the WA-RSS.Signcrypt and WA-RSS.Unsigncrypt algorithms.[33]

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,

[33] Specifically, we ask that the SK algorithms initialise *usid* and z as is done in line 1 of UNI.Init in Figure 12. We ask that the SK algorithms increment the message index upon successful encryption and decryption. We replace line 8 from UNI.Enc and line 2 from UNI.Dec with the inclusion of $(usid, z)$ as the additional data in SK calls to WA-RSS.Signcrypt and WA-RSS.Unsigncrypt.

(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 $\perp$ to indicate that the share failed.

Remark 4. As discussed in Section 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: DOGM. Our description of WA-DOGM checks the signed pre-key signature, σ_{spk}, when WhatsApp's PAIR protocol would have done so. This can be seen in lines 5 and 6 of the ciphertext decryption case of Decrypt in Figure 25 and lines 3 and 4 of the receiving case of StateShare in Figure 26.

7.5 Security Analysis of WhatsApp in the DOGM model

We now analyse the security of the WA-DOGM protocol within this new 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 now detail 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) that restrict patterns of adversarial queries within the DOGM security experiment. 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. [34]

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. 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[35] and disallow authentication breaks where the partnered sending session was compromised. Putting this all together, when a recipient

[34] WhatsApp's whitepaper does not provide enough detail in its threat model [66, Defining End-to-End Encryption] to determine which cases should be categorised as expected or unexpected breaks.

[35] 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

session, $\pi_s^{A,i}$, accepts a message as originating from $\pi_r^{B,j}$, we check whether the compromise oracle, 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 $\pi_{A,i}^s$ 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 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 CorruptDevice call, and disallow both attacks.[36]

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 following authentication predicate.

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.

[36] We contrast this with prior analysis of Matrix [3] 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 results suggesting otherwise.

Definition 32. *An instance of the DOGM with revocation security experiment fulfils the* WA-DOGM.*AUTH* *authenticity predicate if, processing a* Decrypt(A, i, s, c) *query in order to decrypt a message for 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 secrets have been compromised, i.e. either CorruptUser(A) *or* CorruptDevice$(A, 0)$ *was issued at any point before the authentication break.*

2) The sending user's secrets have been compromised, i.e. either CorruptUser(B) *or* CorruptDevice$(B, 0)$ *was issued before the recipient received the inbound Sender Keys session for that stage.*

3) Either the sending or recipient device secrets have been compromised, i.e. either CorruptDevice(B, j) *or* CorruptDevice(A, i) *was issued before the recipient received the inbound Sender Keys session for that stage.*[37]

4) The session state of the sender was compromised at a point in time when the sending session $\pi_r^{B,j}$ *was in the partnered Sender Keys stage for the received ciphertext, i.e.* Compromise(B, j, r) *was previously issued at stage* $\pi_r^{B,j}.t = t'$ *such that* $\pi_s^{A,i}.usid[t] = \pi_r^{B,j}.usid[t']$*.*

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. It is, additionally, used implicitly when sharing history containing a challenge ciphertext.

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 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

[37] 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.

the resulting plaintext with another device.[38] 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.*[39]

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. Luckily WhatsApp restricts history sharing to occurring from the primary device to a companion device, minimising the impact of such attacks. 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 now codify these requirements through the confidentiality predicate.

Definition 33. *An instance of the DOGM with revocation security experiment fulfils the* WA-DOGM.*CONF* *confidentiality predicate if, for every challenge ciphertext in the security experiment, $c \in \mathbf{C}$, corresponding to an* $\mathsf{Encrypt}(A, i, s, m_0, m_1)$ *query for the session $\pi_s^{A,i}$, with intended recipients $CU := \pi_s^{A,i}.CU$ and $CD := \pi_s^{A,i}.CD$ at stage $t^\star := \pi_s^{A,i}.t$ and message index $z^\star := \pi_s^{A,i}.z$, none of the following conditions are true.*

1) *The sending user has been compromised, i.e. a query to* $\mathsf{CorruptUser}(A)$ *or* $\mathsf{CorruptDevice}(A, 0)$ *was issued, at any point in the experiment.*

2) *The sending device has been compromised, i.e. a query to* $\mathsf{CorruptDevice}(A, i)$ *was issued, before the inbound Sender Keys session for this stage was distributed.*[40]

3) *A recipient user has been compromised, i.e. a query to* $\mathsf{CorruptUser}(B)$ *or* $\mathsf{CorruptDevice}(B, 0)$ *was issued for a user $B \in CU$, at any point in the experiment.*

[38] 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.

[39] A more accurate approach would be to track when such state sharing events have actually occurred.

[40] 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.

4) Any device representing a recipient user has been compromised, i.e. a query to $\mathsf{CorruptDevice}(B, j)$ *was issued for a user* $B \in CU$ *and any device* $j \in [n_d]$ *at any point in the experiment.*

5) The sending session has been compromised, i.e. a query to $\mathsf{Compromise}(A, i, s)$ *was issued, within the same stage* $t^\star$ *and before the session has ratcheted forward past the challenge message index* $z^\star$*.*

6) A recipient session has been compromised, i.e. a query to $\mathsf{Compromise}(B, j, r)$ *was issued for a session* $\pi_r^{B,j}$ *representing a device* (B, j) *in the challenge recipient list* CD *at a stage* $\pi_r^{B,j}.t = t'$ *that is partnered to the challenge stage and it has not ratcheted forward past the challenge message index* $z^\star$*:* $\pi_s^{A,i}.usid[t^\star] = \pi_r^{B,j}.usid[t'] \ \wedge \ z' < z^\star$*.*

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 5 (IND-CCA security of WA-DOGM). *The* WA-DOGM *protocol specified in Definition 31 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* $\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{WA\text{-}DOGM}}(A)$ *can be bound by:*

$$
\begin{aligned}
&\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{WA\text{-}DOGM}}(n_p,\ n_d,\ n_i,\ n_s,\ n_m)\\
&\quad\le \Big[\quad /\!\!/ \textit{ Case 1: Authentication}\\
&\qquad n_p \cdot \mathsf{Adv}^{w\mathsf{PO}}_{\mathsf{WA\text{-}PO}}(\lambda,\ n_d - 1,\ 2\cdot n_d,\ 2\cdot n_d - 1)\ +\\
&\qquad n_p \cdot n_d \cdot \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda,\ 2)\ +\\
&\qquad \mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda,\ n_p\cdot n_d,\ 2\cdot n_e,\ n_p\cdot n_q)\ +\\
&\qquad n_p \cdot n_d \cdot n_i \cdot n_s \cdot \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{WA\text{-}RSS}}(\lambda,\ n_m)\ +\\
&\qquad n_p \cdot (n_d - 1) \cdot \big[\\
&\qquad\quad \mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda,\ n_p\cdot n_d,\ 2\cdot n_e,\ n_p\cdot n_q)\ +\\
&\qquad\quad 2\cdot n_e\cdot n_p\cdot n_q \cdot \big(\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF}}(\lambda,\ 1)\ + \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{HMAC}}(\lambda,\ 1)\big)\\
&\qquad \big]\\
&\quad \Big] + \Big[\quad /\!\!/ \textit{ Case 2: Confidentiality}\\
&\qquad n_p \cdot \mathsf{Adv}^{w\mathsf{PO}}_{\mathsf{WA\text{-}PO}}(\lambda,\ n_d - 1,\ 2\cdot n_d,\ 2\cdot n_d - 1)\ +\\
&\qquad n_p \cdot n_d \cdot \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, 2)\ +\\
&\qquad \mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda,\ n_p\cdot n_d,\ 2\cdot n_e,\ n_p\cdot n_q)\ +
\end{aligned}
$$

$$
\begin{aligned}
&(n_p \cdot n_d \cdot n_i \cdot n_s \cdot n_m) \cdot (n_p \cdot n_d \cdot n_i \cdot n_s) \cdot \Big[\\
&\quad \mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda,\ n_p \cdot n_d,\ 2 \cdot n_e,\ n_p \cdot n_q) + \\
&\quad \mathsf{Adv}^{\mathsf{PFS\text{-}OAE}}_{\mathsf{WA\text{-}RSS}}(\lambda,\ n_m) + \\
&\quad n_p \cdot (n_d - 1) \cdot 2 \cdot n_e \cdot n_p \cdot n_q \cdot \big(\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF}}(\lambda,\ 1) \ + \ \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{AES\text{-}CBC}}(\lambda,\ 1) \big) \\
&\ \Big] \\
\Big]
\end{aligned}
$$

This bound is negligible in the security parameter, under the usage parameters $\Lambda = (n_p, n_d, n_i, n_s, n_m)$, in the following conditions.

1) *WA-PO is a weakly secure public key orbit (as specified by Definition 19) with advantage $\mathsf{Adv}^{w\mathsf{PO}}_{\mathsf{WA\text{-}PO}}(\lambda, n_{ch}, n_\sigma, n_g)$. In Theorem 1, we derive an advantage term and prove it to be negligible in the security parameter λ for any PPT adversary, under certain assumptions.*

2) *WA-PAIR provides secure pairwise channels (as specified by Definition 23) with advantage $\mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda, n_d, n_i, n_m)$. In Theorem 2, we derive an advantage term and prove it to be negligible in λ for any PPT adversary, under certain assumptions.*

3) *UNI instantiates a PFS-OAE and SUF-CMA secure ratcheted symmetric signcryption scheme (as specified by Definitions 28 and 29) with advantages $\mathsf{Adv}^{\mathsf{PFS\text{-}OAE}}_{\mathsf{WA\text{-}RSS}}(\lambda, n_m)$ and $\mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{WA\text{-}RSS}}(\lambda, n_m)$ (respectively). In Theorems 3 and 4, we derive advantage terms and prove these to be negligible for any PPT adversary under certain assumptions.*

4) *AES-CBC instantiates an IND-CPA secure symmetric encryption scheme (as specified by Definition 9) 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.*

5) *HKDF instantiates a secure PRF (as specified by Definition 6) with advantage $\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF}}(\lambda, n_q)$, which we assume to be negligible in the security parameter λ for any PPT adversary.*

6) *HMAC instantiates an EUF-CMA secure MAC (as specified by Definition 3) with advantage $\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{HMAC}}(\lambda, n_q)$, which we assume to be negligible in the security parameter λ for any PPT adversary.*

7) *XEd instantiates a SUF-CMA secure digital signature scheme (as specified by Definition 15) 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.*

Note that WhatsApp (as deployed) is instantiated with the security parameter $\lambda = 256$.

Proof outline We separate the proof into two cases. In *Case 1*, we consider the event where the adversary wins the game by causing the *win* flag to be set, i.e.

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 $\mathsf{WA\text{-}PAIR}$ scheme. We then use the $\mathsf{PAIR\text{-}SEC}$ security of the $\mathsf{WA\text{-}PAIR}$ 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 $\mathsf{WA\text{-}RSS}$ scheme), or a state sharing ciphertext (to allow injecting messages through the history sharing scheme). We additionally demonstrate that the $\mathsf{WA\text{-}PAIR}$ leaks no information about the contents of its messages. We proceed to demonstrate that forging any $\mathsf{WA\text{-}RSS}$ message implies breaking the $\mathsf{EUF\text{-}CMA}$ security of the $\mathsf{WA\text{-}RSS}$ 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 $\mathsf{EUF\text{-}CMA}$ 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 $\mathsf{WA\text{-}PAIR}$ channel by the same arguments as in *Case 1*. Thanks to the confidentiality provided by the $\mathsf{PAIR\text{-}SEC}$ security of the $\mathsf{WA\text{-}PAIR}$ scheme, we replace the $\mathsf{WA\text{-}RSS}$ 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 $\mathsf{WA\text{-}RSS}$ 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 $\mathsf{PFS\text{-}OAE}$ security of the $\mathsf{WA\text{-}RSS}$ 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 the win flag to be set to true, 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 $\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{WA\text{-}DOGM}}(\mathcal{A}) \leq \mathsf{Adv}_{\mathsf{AUTH}} + \mathsf{Adv}_{\mathsf{CONF}}$ 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 the *win* flag to be set to true via the following sequence of games.

Game 0. This is the standard DOGM 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 $A \in [n_p]$. The event is triggered if any session, say $\pi_r^{B,j}$, has $\mathsf{WA\text{-}PO.Orbit}$ 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 Create query. Specifically, if there exists $ipk_c \in \mathcal{P}$ for which no query of the form $\mathsf{Create}(A, i) \mapsto dpk_{A,i}$ has been issued for some $i \in [n_d]$ such that $dpk_{A,i}.ipk = ipk_c$.

2) There exists an identity key ipk_c within $\mathcal{P}$ that has been revoked by A in the processing of an honest 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 $\mathsf{Revoke}(A, i) \mapsto dpk_{A,i}$ has been issued for some $i \in [n_d]$ such that $dpk_{A,i}.ipk = ipk_c$ and $\pi_r^{B,j}.\Gamma[A] \geq dpk_{A,i}.\gamma$.

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 $\mathcal{C}_{w\mathsf{PO}}$, Sign.

Whenever the adversary issues a $\mathsf{CorruptUser}(A)$ or $\mathsf{CorruptDevice}(A, 0)$ query, we utilise the $\mathcal{C}_{w\mathsf{PO}}$ challenger's 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 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 $\mathsf{WA\text{-}PO.Orbit}$ 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 $\mathsf{CorruptUser}(A)$ nor $\mathsf{CorruptDevice}(A, 0)$ 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_p users in the experiment and, in turn, each respective abort event, we therefore find:[41]

$$\mathsf{Adv}_{\mathsf{G0}} \leq \mathsf{Adv}_{\mathsf{G1}} + n_p \cdot \mathsf{Adv}^{w\mathsf{PO}}_{\mathsf{WA\text{-}PO}}(\lambda,\ n_d - 1,\ 2 \cdot n_d,\ 2 \cdot n_d - 1)$$

From this game onwards, it is not possible for the adversary to cause the *win* flag to be set to true by breaking revocation.

Game 2. We define a set of events, $\{event_{ipk}^{A,i} \ :\ \forall\ A \in [n_p],\ i \in [n_d]\}$, 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 $event_{ipk}^{A,i}$ 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 $\mathsf{CorruptDevice}(A, i)$ or $\mathsf{CorruptUser}(A)$ 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 $D \coloneqq n_p \cdot n_d$ 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

[41] 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).

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 $\mathsf{CorruptDevice}(A, i)$ or $\mathsf{CorruptUser}(A)$ 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:

$$\mathsf{Adv}_{\mathsf{G1}} \;\leq\; \mathsf{Adv}_{\mathsf{G2}} + n_p \cdot n_d \cdot \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, 2)$$

Game 3. In this game, we abort if any fresh[42] 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.

To do so, we introduce a reduction $\mathcal{B}$ against a challenger for the $\mathsf{PAIR\text{-}SEC}$ security of the $\mathsf{WA\text{-}PAIR}$ 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 $\mathsf{PAIR\text{-}SEC}$ 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 $\mathsf{Encrypt}(i, j, sid, m, m)$ 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 $\mathsf{CorruptDevice}(B', j')$ for some device (B', j'), our reduction uses its device identifier mapping to submit the appropriate $\mathsf{CorruptIdentity}$, $\mathsf{CorruptShared}$ or $\mathsf{CorruptSession}$ queries to $\mathcal{C}_{\mathsf{PAIR}}$, making sure to query $\mathsf{CorruptSession}$ for all of that device's sessions. Similarly, whenever the inner adversary calls $\mathsf{CorruptUser}(B')$ for some user B', our reduction uses its device identifier mapping to submit the appropriate $\mathsf{CorruptIdentity}$ query to $\mathcal{C}_{\mathsf{PAIR}}$. Note that the authentication predicate implies the $\mathsf{WA\text{-}PAIR}$ authentication predicate, which means that if $\pi_s^{A,i}$ is fresh at the time the forgery is accepted, then so too is any pairwise channel used by $\pi_s^{A,i}$. 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 $\mathcal{C}_{\mathsf{PAIR}}$ setting the *win* flag to true.

[42] We consider a session, $\pi_s^{A,i}$, *fresh* for the message at stage t and message index z if the authentication predicate is satisfied, i.e. $\mathsf{WA\text{-}DOGM.AUTH}(A, i, s, t, z, B, j) \mapsto \textbf{true}$ where $(B, j) \coloneqq \pi_s^{A,i}.CD[0]$ and $B \coloneqq \pi_s^{A,i}.CU[0]$. A session can be fresh for any message if the authentication predicate is satisfied no matter the values of (t, z).

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:

$$\mathsf{Adv}_{\mathsf{G2}} \leq \mathsf{Adv}_{\mathsf{G3}} + \mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda, n_p \cdot n_d, 2 \cdot n_e, n_p \cdot n_q)$$

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.[43]

Game 4. In this game, we abort if any fresh session, $\pi_s^{A,i}$, accepts an inbound WA-RSS session, st_r, sent under a pairwise channel from the designated sending device, $\pi_s^{A,i}.CD[0]$, *but* st_r was not honestly output by that device. 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 fresh session that causes *win* to be set to true 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**.

[43] 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.

Game 5.1. In this game, we guess the sending session, $\pi_r^{B,j}$, that generated the inbound $\mathsf{WA\text{-}RSS}$ session state. We additionally guess the stage, t', of the sending session when the relevant inbound $\mathsf{WA\text{-}RSS}$ 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.

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, $st_r^\dagger \neq st_r$, to decrypt c. Thus, we find:

$$\mathsf{Pr}[abort_{uni\text{-}forge}] \;\leq\; n_p \cdot n_d \cdot n_i \cdot n_s \cdot \mathsf{Adv}_{\mathsf{G5.1}}$$

Game 5.2. In this game, we abort if $\pi_s^{A,i}$ sets *win* to true upon acceptance of a $\mathsf{WA\text{-}RSS}$ 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 $\mathsf{WA\text{-}RSS}$ 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 $\mathsf{Send}(m)$ 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 a fresh session $\pi_s^{A,i}$ wishes to decrypt, $\mathcal{B}$ simply queries $\mathsf{Receive}(c)$, with the given ciphertext c, to $\mathcal{C}_{\mathsf{SUF\text{-}CMA}}$. Note that, since we abort only when a *fresh* session $\pi_s^{A,i}$ accepts a malicious $\mathsf{WA\text{-}RSS}$ message, we do not need to handle the adversary's Compromise queries for this particular $\mathsf{WA\text{-}RSS}$ state. It follows that, whenever the abort event triggers, we have submitted a forged $\mathsf{WA\text{-}RSS}$ ciphertext c to the $\mathcal{C}_{\mathsf{SUF\text{-}CMA}}$ challenger when submitting the decryption query, $\mathsf{Receive}(c)$, 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 $\mathsf{SUF\text{-}CMA}$ security of $\mathsf{WA\text{-}RSS}$:

$$\mathsf{Adv}_{\mathsf{G5.1}} \;\leq\; \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{WA\text{-}RSS}}(\lambda, n_m)$$

We note here that the adversary can no longer win by causing a device to decrypt forged $\mathsf{WA\text{-}RSS}$ ciphertexts.

We now turn to considering the authenticity of history sharing, and can be sure that we are in case (b) whereby the first fresh 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.[44]

[44] An alternative approach could be to utilise the already determined authentication of the pairwise channel ciphertexts alongside the hash $h \coloneqq \mathsf{SHA256}(c' \parallel \tau)$ included within them.

Game 6. In this game, we introduce an abort event that is triggered if $\pi_s^{A,i}$ has $\mathsf{DM.Dec}$ 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 $\pi_s^{A,i}$ is fresh, inspecting the implementation of $\mathsf{StateShare}$ 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, $\pi_s^{A,i}$, causes *win* to be set. We make this choice from up to n_p users and up to $n_d - 1$ of their companion devices. Thus:

$$\mathsf{Adv}_{\mathsf{G6}} \ \leq \ n_p \cdot (n_d - 1) \cdot \mathsf{Adv}_{\mathsf{G7}}$$

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, $\tilde{m}$, 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 $\mathsf{PAIR\text{-}SEC}$ security of the $\mathsf{WA\text{-}PAIR}$ 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 $\mathsf{PAIR\text{-}SEC}$ 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, $\tilde{m}$, 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 $\tilde{m}$. Observe that the authentication predicate for $\mathsf{WA\text{-}DOGM}$ implies the confidentiality predicate for $\mathsf{WA\text{-}PAIR}$ 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 $\tilde{m}$ instead and we are in **Game 8**. Thus, any adversary that can distinguish our change can be turned into a successful adversary against the $\mathsf{PAIR\text{-}SEC}$ security of $\mathsf{WA\text{-}PAIR}$ and we find:

$$\mathsf{Adv}_{\mathsf{G7}} \ \leq \ \mathsf{Adv}_{\mathsf{G8}} + \mathsf{Adv}_{\mathsf{WA\text{-}PAIR}}^{\mathsf{PAIR\text{-}SEC}}(\lambda, n_p \cdot n_d, 2 \cdot n_e, n_p \cdot n_q)$$

Game 9. In this game we guess at the beginning of the experiment which history sharing ciphertext $(c_P, c_{hist}, \tau_{hist})$ it is whose processing causes *win* to be set, and abort if this guess is incorrect.

Observe that the number of history sharing ciphertexts that a single device may receive is limited by the number of pairwise messages they are able to exchange with their primary device. We make this guess from a pool of up to $H \coloneqq 2 \cdot n_e \cdot n_p \cdot n_q$ pairwise messages that may be exchanged between the two devices. We implement our guess by picking an index between 0 and $H - 1$ (inclusive) uniformly at random and counting the relevant history sharing queries. Our guess is the h-th such query, and we abort if our guess is incorrect. Thus:

$$\mathsf{Adv}_{\mathsf{G8}} \leq 2 \cdot n_e \cdot n_p \cdot n_q \cdot \mathsf{Adv}_{\mathsf{G9}}$$

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 $\mathsf{StateShare}$ query to $(A, 0)$, instead of computing k honestly $\mathcal{B}$ instead queries the $\mathcal{C}_{\mathsf{PRF}}$ challenger's 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, and thus:

$$\mathsf{Adv}_{\mathsf{G9}} \leq \mathsf{Adv}_{\mathsf{G10}} + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF}}(\lambda, 1)$$

Game 11. In this game, we introduce an abort event that triggers if the adversary forges our guessed history-sharing ciphertext to $\pi^s_{A,i}$.

Specifically, we introduce a reduction $\mathcal{B}$ that interacts with an $\mathsf{EUF\text{-}CMA}$ 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 $\mathsf{EUF\text{-}CMA}$ security. $\mathcal{B}$ returns the history-sharing 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 $\mathsf{EUF\text{-}CMA}$ security of HMAC and thus:

$$\mathsf{Adv}_{\mathsf{G10}} \leq \mathsf{Adv}_{\mathsf{G11}} + \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{HMAC}}(\lambda, 1)$$

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, since $\pi^{A,i}_s$ is fresh at stage t and message index z, every partnered session belonging to its primary device, $(A, 0)$, is also fresh. Specifically, we must show that $\mathsf{WA\text{-}DOGM.AUTH}(A, 0, p, t^\star, z, B, j) = \mathbf{true}$ for all $p \in [n_i]$ for which there exists a $t^\star$ such that $\pi^p_{A,0}.usid[t^\star] = \pi^s_{A,i}.usid[t]$. Recalling the authentication predicate, we see that $\mathsf{WA\text{-}DOGM.AUTH}(A, i, s, t, z, \gamma, B, j) = \mathbf{true}$ implies that $\mathsf{WA\text{-}DOGM.AUTH}(A, 0, p, t^\star, z, B, j) = \mathbf{true}$ for all partnered sessions belonging to device $(A, 0)$. Thanks to the changes in **Game 5**, 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 via the following sequence of games.

Game 0. This is the standard DOGM 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 $A \in [n_p]$. The event is triggered if any session, say $\pi^{B,j}_r$, has WA-PO.Orbit calculate an orbit $\mathcal{P}$ for the user A (which may be itself) at generation $\pi^{B,j}_r.\Gamma[A]$, 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 Create query. Specifically, if there exists $ipk_c \in \mathcal{P}$ for which no query of the form $\mathsf{Create}(A, i) \mapsto dpk_{A,i}$ has been issued for some $i \in [n_d]$ such that $dpk_{A,i}.ipk = ipk_c$.
2) There exists an identity key ipk_c within $\mathcal{P}$ that has been revoked by A in the processing of an honest 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 $\mathsf{Revoke}(A, i) \mapsto dpk_{A,i}$ has been issued for some $i \in [n_d]$ such that $dpk_{A,i}.ipk = ipk_c$ and $\pi^{B,j}_r.\Gamma[A] \geq dpk_{A,i}.\gamma$.

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 $\mathsf{PO.Attract}$ when adding companion keys, $\mathsf{PO.Repel}$ when removing companion keys, and $\mathsf{PO.Refresh}$ 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 Sign limited signing oracle exposed by $\mathcal{C}_{w\mathsf{PO}}$.

Whenever the adversary issues a $\mathsf{CorruptUser}(A)$ or $\mathsf{CorruptDevice}(A, 0)$ query, we utilise the $\mathcal{C}_{w\mathsf{PO}}$ challenger's $\mathsf{Compromise}$ 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 Eject query. The reduction tracks the expected values of the "to" and "from" sets of the $w\mathsf{PO}$ 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 $\mathsf{WA\text{-}PO.Orbit}$ 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_p users in the experiment and, in turn, each respective abort event, we therefore find:

$$\mathsf{Adv}_{\mathsf{G0}} \;\leq\; \mathsf{Adv}_{\mathsf{G1}} + n_p \cdot \mathsf{Adv}^{w\mathsf{PO}}_{\mathsf{WA\text{-}PO}}(\lambda,\ n_d - 1,\ 2 \cdot n_d,\ 2 \cdot n_d - 1)$$

Game 2. We make an analogous change to **Game 2** in *Case 1*. We define a set of events, $\{event_{ipk}^{A,i}\ :\ \forall\ A \in [n_p],\ i \in [n_d]\}$, 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 $event_{ipk}^{A,i}$ 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 $\mathsf{CorruptDevice}(A, i)$ or $\mathsf{CorruptUser}(A)$ 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 $D \coloneqq n_p \cdot n_d$ 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 a call to $\mathsf{CorruptDevice}(A, i)$ or $\mathsf{CorruptUser}(A)$ 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:

$$\mathsf{Adv}_{\mathsf{G1}} \leq \mathsf{Adv}_{\mathsf{G2}} + n_p \cdot n_d \cdot \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{XEd}}(\lambda, 2)$$

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 $\mathsf{CorruptUser}(B)$, $\mathsf{CorruptDevice}(B, 0)$ and $\mathsf{CorruptDevice}(B, j)$ have previously been issued by the challenger.

To do so, we introduce a reduction $\mathcal{B}$ against a challenger for the $\mathsf{PAIR\text{-}SEC}$ security of the $\mathsf{WA\text{-}PAIR}$ 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 $\mathsf{PAIR\text{-}SEC}$ 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 $\mathsf{Encrypt}(i, j, sid, m, m)$ 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 $\mathsf{CorruptDevice}(B', j')$ for some device (B', j'), our reduction uses its device identifier mapping to submit the appropriate $\mathsf{CorruptIdentity}$, $\mathsf{CorruptShared}$ or $\mathsf{CorruptSession}$ queries to $\mathcal{C}_{\mathsf{PAIR}}$, making sure to query $\mathsf{CorruptSession}$ for all of that device's sessions. Similarly, whenever the inner adversary calls $\mathsf{CorruptUser}(B')$ for some user B', our reduction uses its device identifier mapping to submit the appropriate $\mathsf{CorruptIdentity}$ query to $\mathcal{C}_{\mathsf{PAIR}}$.

Note that, since we only trigger the abort event when none of $\mathsf{CorruptUser}(B)$, $\mathsf{CorruptDevice}(B, 0)$ and $\mathsf{CorruptDevice}(B, j)$ for the claimed sending device (B, j) have previously been issued by the challenger in the emulated experiment, the

$\mathsf{WA\text{-}PAIR}$ 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 $\mathcal{C}_{\mathsf{PAIR}}$ setting the *win* flag to true.

It follows that anytime the inner adversary triggers the abort event, then it can be turned into an authentication break against the $\mathsf{PAIR\text{-}SEC}$ security of $\mathsf{WA\text{-}PAIR}$ and thus:

$$\mathsf{Adv}_{\mathsf{G2}} \leq \mathsf{Adv}_{\mathsf{G3}} + \mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda, n_p \cdot n_d, 2 \cdot n_e, n_p \cdot n_q)$$

The changes introduced in this game ensure that all pairwise channel messages exchanged between uncorrupted devices were honestly generated.

Game 4. We consider a series of N hybrids where $N := n_p \cdot n_d \cdot n_i \cdot n_s \cdot n_m$; one for each possible challenge encryption that may be triggered in the game. For the h-hybrid, we look to bound the advantage of the adversary in determining the challenge bit through the h-th challenge encryption. We find, by a hybrid argument:

$$\mathsf{Adv}_{\mathsf{G3}} \leq n_p \cdot n_d \cdot n_i \cdot n_s \cdot n_m \cdot \mathsf{Adv}_{\mathsf{G4}}$$

In what follows, we bound the advantage of an adversary in a single such hybrid, and do so through the following sequence of games.

Game 5. In this game, we guess the session $\pi_s^{A,i}$ and stage t of the h-th challenge encryption, such that the adversary issues a $\mathsf{Encrypt}(A, i, s, m_0, m_1)$ 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-th challenge encryption query is not made to $\pi_s^{A,i}$ within stage t. It follows that,

$$\mathsf{Adv}_{\mathsf{G4}} \leq n_p \cdot n_d \cdot n_i \cdot n_s \cdot \mathsf{Adv}_{\mathsf{G5}}$$

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 will assume that it may be applied to our guessed challenge query). Letting c be our challenge ciphertext, such that $\mathbf{C} = \{ (c, A, i, s, t, z) \}$, 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 z, the communicating partners, CU and CD, or the plaintexts, m_0 and m_1, ahead of the challenge.

Game 6. 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 $\mathsf{WA\text{-}RSS}$ 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, $\tilde{m}$, 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 $\mathsf{WA\text{-}RSS}$ 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.[45]

Consider the following reduction, $\mathcal{B}$, which we construct against a challenger for the $\mathsf{PAIR\text{-}SEC}$ security of the $\mathsf{WA\text{-}PAIR}$ 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 $\mathsf{PAIR\text{-}SEC}$ 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 $\mathsf{WA\text{-}RSS}$ 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, $\tilde{m}$, 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 $\tilde{m}$. Observe that the confidentiality predicate for $\mathsf{WA\text{-}DOGM}$ implies the confidentiality predicate for $\mathsf{WA\text{-}PAIR}$.

When the bit b sampled by $\mathcal{C}_{\mathsf{PAIR}}$ is 0, then the output ciphertext encrypts m honestly and we are in **Game 5**. However, if the bit b sampled by $\mathcal{C}_{\mathsf{PAIR}}$ is 1, then the output ciphertext encrypts $\tilde{m}$ instead and we are in **Game 6**. Thus, any adversary that can distinguish our change can be turned into a successful adversary against the $\mathsf{PAIR\text{-}SEC}$ security of $\mathsf{WA\text{-}PAIR}$ and we find:

$$\mathsf{Adv}_{\mathsf{G5}} \;\leq\; \mathsf{Adv}_{\mathsf{G6}} + \mathsf{Adv}^{\mathsf{PAIR\text{-}SEC}}_{\mathsf{WA\text{-}PAIR}}(\lambda, n_p \cdot n_d, 2 \cdot n_e, n_p \cdot n_q)$$

Game 7. 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 $\mathsf{WA\text{-}RSS}$ session state, the challenger has full knowledge of the set CU at the point in time these replacements need to be made.

[45] See that our implementation of history sharing in $\mathsf{WA\text{-}DOGM}$ shares all plaintexts in the requested stage with a message index greater than or equal to the requested message index.

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 $\mathsf{StateShare}(B, 0, p, B, j, t', z')$ query from a session $\pi^{p}_{B,0}$ for which $B \in CU$ and $\pi^{p}_{B,0}.usid[t'] = \pi^{A,i}_{s}.usid[t]$ 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 PRF oracle with r and replaces k with the output, which we denote $\tilde{k}$.

By **Game 6**, 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 $\tilde{k}$ is identically distributed to k and we are in **Game 6**. However, if the bit b sampled by $\mathcal{C}_{\mathsf{PRF}}$ is 0, then $\tilde{k} \leftarrow_{\$} \{0,1\}^{896}$ instead and we are in **Game 7**.

Thus, any adversary that can distinguish our change can be turned into a successful adversary against the PRF security of HKDF. 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:

$$\mathsf{Adv}_{\mathsf{G6}} \;\leq\; \mathsf{Adv}_{\mathsf{G7}} + n_p \cdot (n_d - 1) \cdot 2 \cdot n_e \cdot n_p \cdot n_q \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF}}(\lambda, 1)$$

Game 8. In this game, we demonstrate that no history sharing ciphertext encrypting m_b leaks anything about the challenge plaintext, by the $\mathsf{IND\text{-}CPA}$ security of $\mathsf{AES\text{-}CBC}$. We do so by replacing the ciphertext encrypting the history sharing attachment with a random bit string of the same length. As in **Game 7**, 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 $\mathsf{IND\text{-}CPA}$ challenger $\mathcal{C}_{\mathsf{IND\text{-}CPA}}$: when **Game 8** would normally encrypt the plaintext m_b using aek (truncated from k') using a call to $\mathsf{AES\text{-}CBC}$, 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 7**, 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 7**. 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 8**.

Any adversary that can distinguish our change can be turned into a successful adversary against the $\mathsf{IND\text{-}CPA}$ security of $\mathsf{AES\text{-}CBC}$ and thus:

$$\mathsf{Adv}_{\mathsf{G7}} \;\leq\; \mathsf{Adv}_{\mathsf{G8}} + n_p \cdot (n_d - 1) \cdot 2 \cdot n_e \cdot n_p \cdot n_q \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{AES\text{-}CBC}}(\lambda, 1)$$

Game 9. In this game, we show that if the adversary, $\mathcal{A}$, can distinguish between the encryption of m_0 and m_1 by $\pi^{A,i}_{s}$ then they can be turned into a successful adversary against the $\mathsf{PFS\text{-}OAE}$ security of the $\mathsf{WA\text{-}RSS}$ scheme.

Specifically, we define a reduction $\mathcal{B}$ that acts identically to **Game 9**. However, when the test session $\pi_s^{A,i}$ initialises the $\mathsf{WA\text{-}RSS}$ state at the beginning of stage T, $\mathcal{B}$ instead initialises a $\mathsf{PFS\text{-}OAE}$ challenger $\mathcal{C}_{\mathsf{PFS\text{-}OAE}}$ that generates a receiving state st_r for them. Since the receiving state is never sent in the pairwise channel (by **Game 6**) then this replacement cannot be detected by the adversary. Whenever $\pi_s^{A,i}$ needs to encrypt using the $\mathsf{WA\text{-}RSS}$ sending state st_s, $\mathcal{B}$ instead queries $\mathcal{C}_{\mathsf{PFS\text{-}OAE}}$ with $\mathsf{Send}(m_0, m_1)$. If, after the challenge encryption query was made to $\mathcal{B}$, $\mathcal{A}$ calls $\mathsf{Compromise}(A, i, s)$, $\mathcal{B}$ queries $\mathcal{C}_{\mathsf{PFS\text{-}OAE}}$ with $\mathsf{Corrupt}$ 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 $\mathcal{C}_{\mathsf{PFS\text{-}OAE}}$.

The success of $\mathcal{A}$ is bound by the success of $\mathcal{B}$, and thus we find:

$$\mathsf{Adv}_{\mathsf{G8}} \leq \mathsf{Adv}_{\mathsf{WA\text{-}RSS}}^{\mathsf{PFS\text{-}OAE}}(\lambda, n_m)$$

This completes our analysis of *Case 2*.

We combine the two cases above to arrive at the upper bound in the theorem statement. Observe that the above bound is a polynomial function of the experiment parameters, $\Lambda_{\mathsf{DOGM}} = (n_p, n_d, n_i, n_s, n_m)$, 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. □

8 Interpretation & Discussion

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.

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.[46] 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

[46] 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.

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 by sending 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.[47] 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.

Adversarially-controlled group management. As mentioned above and in prior works, WhatsApp's design and implementation are insecure in allowing 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,[48] 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.

Unclear client enforcement of group membership. As discussed in Section 3.3, we were not able to find evidence that clients delete the inbound Sender Keys sessions associated with past members of the group (when they are removed). If it is the case that such sessions are not removed (and the removal is not enforced through some alternate means), it follows that the removal of a participant from the group would not prevent them from sending messages in that group.

[47] We note that the inclusion of ICDC information within group messaging ciphertexts would improve the speed with which communicating partners will detect a device revocation.

[48] Ghost users were a 2018 GHQ proposal to circumvent end-to-end encryption [47].

Lack of domain separation. WhatsApp (like Signal) utilises the same key pair for the long-term device keys used for key exchange and signatures.[49] Following prior work, we do not model this lack of domain separation and stress that the security implications are unknown.

Lack of post-compromise security. Our analysis in Section 5 follows [28, 30] 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 are created.[50] 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 [25].[51] Since there is little practical difference between compromise of a device's identity key and pairwise session state, our analysis in Section 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.[52]

Echoing the discussion in [3], we note that, while this contrasts with PCS guarantees in the cryptographic literature, this is not unusual. 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

[49] We note that WhatsApp adds another domain to the mix in comparison to Signal.

[50] The PAIR.AUTH and 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.

[51] In the terminology of [27], this setting does not provide *PCS via state* but does provide *PCS via weak compromise*, if all compromised sessions have healed or expired.

[52] 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 [34].

settings [1], 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.

References

1. Albrecht, M.R., Blasco, J., Jensen, R.B., Mareková, L.: Collective information security in large-scale urban protests: the case of hong kong. In: Bailey, M., Greenstadt, R. (eds.) USENIX Security 2021. pp. 3363–3380. USENIX Association (Aug 2021)
2. Albrecht, M.R., Celi, S., Dowling, B., Jones, D.: Practically-exploitable cryptographic vulnerabilities in Matrix. In: Ristenpart, T., Traynor, P. (eds.) 44th IEEE Symposium on Security and Privacy (2023). https://doi.org/10.1109/SP46215.2023.10351027
3. Albrecht, M.R., Dowling, B., Jones, D.: Device-oriented group messaging: A formal cryptographic analysis of matrix' core. In: 2024 IEEE Symposium on Security and Privacy. pp. 2666–1685. IEEE Computer Society Press (May 2024). https://doi.org/10.1109/SP54263.2024.00075
4. Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 129–158. Springer, Cham (May 2019). https://doi.org/10.1007/978-3-030-17653-2_5
5. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Security analysis and improvements for the IETF MLS standard for group messaging. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part I. LNCS, vol. 12170, pp. 248–277. Springer, Cham (Aug 2020). https://doi.org/10.1007/978-3-030-56784-2_9
6. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Modular design of secure group messaging protocols and the security of MLS. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021. pp. 1463–1483. ACM Press (Nov 2021). https://doi.org/10.1145/3460120.3484820
7. Backendal, M., Bellare, M., Günther, F., Scarlata, M.: When messages are keys: Is HMAC a dual-PRF? In: Handschuh, H., Lysyanskaya, A. (eds.) CRYPTO 2023, Part III. LNCS, vol. 14083, pp. 661–693. Springer, Cham (Aug 2023). https://doi.org/10.1007/978-3-031-38548-3_22
8. Balbás, D., Collins, D., Gajland, P.: WhatsUpp with sender keys? Analysis, improvements and security proofs. In: Guo, J., Steinfeld, R. (eds.) ASIACRYPT 2023, Part V. LNCS, vol. 14442, pp. 307–341. Springer, Singapore (Dec 2023). https://doi.org/10.1007/978-981-99-8733-7_10
9. Balbás, D., Collins, D., Gajland, P.: Analysis and improvements of the sender keys protocol for group messaging (2022). https://doi.org/10.22429/Euc2022.028, https://arxiv.org/abs/2301.07045
10. Barooti, K., Collins, D., Colombo, S., Huguenin-Dumittan, L., Vaudenay, S.: On active attack detection in messaging with immediate decryption. In: Handschuh, H., Lysyanskaya, A. (eds.) CRYPTO 2023, Part IV. LNCS, vol. 14084, pp. 362–395. Springer, Cham (Aug 2023). https://doi.org/10.1007/978-3-031-38551-3_12
11. Bellare, M.: New proofs for NMAC and HMAC: Security without collision-resistance. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 602–619. Springer, Berlin, Heidelberg (Aug 2006). https://doi.org/10.1007/11818175_36

12. Bellare, M., Desai, A., Jokipii, E., Rogaway, P.: A concrete security treatment of symmetric encryption. In: 38th FOCS. pp. 394–403. IEEE Computer Society Press (Oct 1997). https://doi.org/10.1109/SFCS.1997.646128
13. Bellare, M., Micciancio, D.: A new paradigm for collision-free hashing: Incrementality at reduced cost. In: Fumy, W. (ed.) EUROCRYPT'97. LNCS, vol. 1233, pp. 163–192. Springer, Berlin, Heidelberg (May 1997). https://doi.org/10.1007/3-540-69053-0_13
14. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. Journal of Cryptology **21**(4), 469–491 (Oct 2008). https://doi.org/10.1007/s00145-008-9026-x
15. Bellare, M., Rogaway, P.: Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient cryptography. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 317–330. Springer, Berlin, Heidelberg (Dec 2000). https://doi.org/10.1007/3-540-44448-3_24
16. Bernstein, D.J.: Curve25519: New Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer, Berlin, Heidelberg (Apr 2006). https://doi.org/10.1007/11745853_14
17. Bienstock, A., Fairoze, J., Garg, S., Mukherjee, P., Raghuraman, S.: A more complete analysis of the Signal double ratchet algorithm. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part I. LNCS, vol. 13507, pp. 784–813. Springer, Cham (Aug 2022). https://doi.org/10.1007/978-3-031-15802-5_27
18. Blazy, O., Boureanu, I., Lafourcade, P., Onete, C., Robert, L.: How fast do you heal? A taxonomy for post-compromise security in secure-channel establishment. In: Calandrino, J.A., Troncoso, C. (eds.) USENIX Security 2023. pp. 5917–5934. USENIX Association (Aug 2023)
19. Blum, J., Booth, S., Chen, B., Gal, O., Krohn, M., Len, J., Lyons, K., Marcedone, A., Maxim, M., Mou, M.E., Namavari, A., OConnor, J., Rien, S., Steele, M., Green, M., Kissner, L., Stamos, A.: Zoom cryptography whitepaper (nov 2023), https://github.com/zoom/zoom-e2e-whitepaper/, https://github.com/zoom/zoom-e2e-whitepaper/
20. Boneh, D., Partap, A., Rotem, L.: Accountable threshold signatures with proactive refresh. Cryptology ePrint Archive, Report 2022/1656 (2022), https://eprint.iacr.org/2022/1656
21. Caforio, A., Durak, F.B., Vaudenay, S.: Beyond security and efficiency: On-demand ratcheting with security awareness. In: Garay, J. (ed.) PKC 2021, Part II. LNCS, vol. 12711, pp. 649–677. Springer, Cham (May 2021). https://doi.org/10.1007/978-3-030-75248-4_23
22. Camenisch, J., Lysyanskaya, A.: Efficient revocation of anonymous group membership. Cryptology ePrint Archive, Report 2001/113 (2001), https://eprint.iacr.org/2001/113
23. Campion, S., Devigne, J., Duguey, C., Fouque, P.A.: Multi-device for Signal. In: Conti, M., Zhou, J., Casalicchio, E., Spognardi, A. (eds.) ACNS 20International Conference on Applied Cryptography and Network Security, Part II. LNCS, vol. 12147, pp. 167–187. Springer, Cham (Oct 2020). https://doi.org/10.1007/978-3-030-57878-7_9
24. Chase, M., Deshpande, A., Ghosh, E., Malvai, H.: SEEMless: Secure end-to-end encrypted messaging with less trust. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 1639–1656. ACM Press (Nov 2019). https://doi.org/10.1145/3319535.3363202
25. Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the Signal messaging protocol. Journal of Cryptology **33**(4), 1914–1983 (Oct 2020). https://doi.org/10.1007/s00145-020-09360-1

26. Cohn-Gordon, K., Cremers, C., Garratt, L., Millican, J., Milner, K.: On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 1802–1819. ACM Press (Oct 2018). https://doi.org/10.1145/3243734.3243747
27. Cohn-Gordon, K., Cremers, C.J.F., Garratt, L.: On post-compromise security. In: Hicks, M., Köpf, B. (eds.) CSF 2016 Computer Security Foundations Symposium. pp. 164–178. IEEE Computer Society Press (2016). https://doi.org/10.1109/CSF.2016.19
28. Cremers, C., Fairoze, J., Kiesl, B., Naska, A.: Clone detection in secure messaging: Improving post-compromise security in practice. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 1481–1495. ACM Press (Nov 2020). https://doi.org/10.1145/3372297.3423354
29. Cremers, C., Hale, B., Kohbrok, K.: The complexities of healing in secure group messaging: Why cross-group effects matter. In: Bailey, M., Greenstadt, R. (eds.) USENIX Security 2021. pp. 1847–1864. USENIX Association (Aug 2021)
30. Cremers, C., Jacomme, C., Naska, A.: Formal analysis of session-handling in secure messaging: Lifting security from sessions to conversations. In: Calandrino, J.A., Troncoso, C. (eds.) USENIX Security 2023. pp. 1235–1252. USENIX Association (Aug 2023)
31. Davies, G.T., Faller, S.H., Gellert, K., Handirk, T., Hesse, J., Horváth, M., Jager, T.: Security analysis of the WhatsApp end-to-end encrypted backup protocol. In: Handschuh, H., Lysyanskaya, A. (eds.) CRYPTO 2023, Part IV. LNCS, vol. 14084, pp. 330–361. Springer, Cham (Aug 2023). https://doi.org/10.1007/978-3-031-38551-3_11
32. Dimeo, A., Gohla, F., Goßen, D., Lockenvitz, N.: SoK: Multi-device secure instant messaging. Cryptology ePrint Archive, Report 2021/498 (2021), https://eprint.iacr.org/2021/498
33. Dowling, B., Fischlin, M., Günther, F., Stebila, D.: A cryptographic analysis of the TLS 1.3 handshake protocol. Journal of Cryptology **34**(4), 37 (Oct 2021). https://doi.org/10.1007/s00145-021-09384-1
34. Dowling, B., Günther, F., Poirrier, A.: Continuous authentication in secure messaging. In: European Symposium on Research in Computer Security. pp. 361–381. Springer (2022)
35. Hanzlik, L., Loss, J., Wagner, B.: Token meets wallet: Formalizing privacy and revocation for FIDO2. Cryptology ePrint Archive, Report 2022/084 (2022), https://eprint.iacr.org/2022/084
36. Jaeger, J., Kumar, A., Stepanovs, I.: Symmetric signcryption and E2EE group messaging in keybase. In: Joye, M., Leander, G. (eds.) EUROCRYPT 2024, Part III. LNCS, vol. 14653, pp. 283–312. Springer, Cham (May 2024). https://doi.org/10.1007/978-3-031-58734-4_10
37. Katz, J., Lindell, Y.: Introduction to Modern Cryptography, Second Edition. CRC Press (Nov 2014)
38. Keybase: Meet your sigchain (and everyone else's) (2022), https://book.keybase.io/docs/server
39. Kocher, P.C.: On certificate revocation and validation. In: Hirschfeld, R. (ed.) FC'98. LNCS, vol. 1465, pp. 172–177. Springer, Berlin, Heidelberg (Feb 1998). https://doi.org/10.1007/bfb0055481
40. Krawczyk, H., Bellare, M., Canetti, R.: RFC 2104: HMAC: Keyed-hashing for message authentication (Feb 1997), https://datatracker.ietf.org/doc/html/rfc2104

41. Krawczyk, H., Eronen, P.: RFC 5869: HMAC-based Extract-and-Expand Key Derivation Function (HKDF) (2010), https://datatracker.ietf.org/doc/html/rfc5869
42. Krawczyk, H.: Cryptographic extraction and key derivation: The HKDF scheme. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 631–648. Springer, Berlin, Heidelberg (Aug 2010). https://doi.org/10.1007/978-3-642-14623-7_34
43. Krawczyk, H., Bellare, M., Canetti, R.: HMAC: Keyed-hashing for message authentication. IETF Internet Request for Comments 2104 (Feb 1997)
44. Lawlor, S., Lewi, K.: Deploying key transparency at WhatsApp (Apr 2023), https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/
45. Lawlor, S., Lewi, K.: Deploying key transparency at WhatsApp (Apr 2023), https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/ (archived on 2025-03-25)
46. Len, J., Chase, M., Ghosh, E., Jost, D., Kesavan, B., Marcedone, A.: ELEKTRA: Efficient lightweight multi-dEvice key TRAnsparency. In: Meng, W., Jensen, C.D., Cremers, C., Kirda, E. (eds.) ACM CCS 2023. pp. 2915–2929. ACM Press (Nov 2023). https://doi.org/10.1145/3576915.3623161
47. Levy, I., Robinson, C.: Principles for a more informed exceptional access debate. Lawfare (Nov 2018), https://web.archive.org/web/20230427102948/https://www.lawfareblog.com/principles-more-informed-exceptional-access-debate
48. Lewi, K., Kim, W., Maykov, I., Weis, S.: Securing update propagation with homomorphic hashing. Cryptology ePrint Archive, Report 2019/227 (2019), https://eprint.iacr.org/2019/227
49. Malvai, H., Kokoris-Kogias, L., Sonnino, A., Ghosh, E., Oztürk, E., Lewi, K., Lawlor, S.F.: Parakeet: Practical key transparency for end-to-end encrypted messaging. In: NDSS 2023. The Internet Society (Feb 2023)
50. Marlinspike, M.: Private Group Messaging (May 2014), https://signal.org/blog/private-groups/
51. Marlinspike, M.: The Double Ratchet Algorithm (Nov 2016), https://signal.org/docs/specifications/doubleratchet/, revision 1
52. Marlinspike, M.: The X3DH Key Agreement Protocol (Nov 2016), https://signal.org/docs/specifications/x3dh/, revision 1
53. Marlinspike, M.: The Sesame Algorithm: Session Management for Asynchronous Message Encryption (Apr 2017), https://signal.org/docs/specifications/sesame/, revision 2
54. Melara, M.S., Blankstein, A., Bonneau, J., Felten, E.W., Freedman, M.J.: CONIKS: Bringing key transparency to end users. In: Jung, J., Holz, T. (eds.) USENIX Security 2015. pp. 383–398. USENIX Association (Aug 2015)
55. Meta Platforms, Inc: akd – An implementation of an auditable key directory, https://github.com/facebook/akd (archived on 2025-01-16)
56. Okamoto, T., Pointcheval, D.: The gap-problems: A new class of problems for the security of cryptographic schemes. In: Kim, K. (ed.) PKC 2001. LNCS, vol. 1992, pp. 104–118. Springer, Berlin, Heidelberg (Feb 2001). https://doi.org/10.1007/3-540-44586-2_8
57. Perrin, T.: The XEdDSA and VXEdDSA signature schemes (Oct 2016), https://signal.org/docs/specifications/xeddsa/, revision 1
58. Poettering, B., Rösler, P., Schwenk, J., Stebila, D.: SoK: Game-based security models for group key exchange. In: Paterson, K.G. (ed.) CT-RSA 2021. LNCS, vol. 12704, pp. 148–176. Springer, Cham (May 2021). https://doi.org/10.1007/978-3-030-75539-3_7

59. Rogaway, P.: Authenticated-encryption with associated-data. In: Atluri, V. (ed.) ACM CCS 2002. pp. 98–107. ACM Press (Nov 2002). https://doi.org/10.1145/586110.586125
60. Rösler, P., Mainka, C., Schwenk, J.: More is less: On the end-to-end security of group chats in signal, WhatsApp, and threema. In: 2018 IEEE European Symposium on Security and Privacy. pp. 415–429. IEEE Computer Society Press (Apr 2018). https://doi.org/10.1109/EuroSP.2018.00036
61. Saint-Andre, P.: RFC 6120: Extensible messaging and presence protocol (XMPP): Core (2011), https://datatracker.ietf.org/doc/html/rfc6120
62. Saint-Andre, P.: RFC 6120: Extensible messaging and presence protocol (XMPP): Instant messaging and presence (2011), https://datatracker.ietf.org/doc/html/rfc6121
63. Secure hash standard. National Institute of Standards and Technology, NIST FIPS PUB 180-2, U.S. Department of Commerce (Aug 2002)
64. Tzeng, W.G., Tzeng, Z.J.: Robust forward-secure signature schemes with proactive security. In: Kim, K. (ed.) PKC 2001. LNCS, vol. 1992, pp. 264–276. Springer, Berlin, Heidelberg (Feb 2001). https://doi.org/10.1007/3-540-44586-2_19
65. Vaudenay, S.: Secure communications over insecure channels based on short authenticated strings. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 309–326. Springer, Berlin, Heidelberg (Aug 2005). https://doi.org/10.1007/11535218_19
66. WhatsApp LLC: WhatsApp Encryption Overview (Jan 2023), https://whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf, version 6
67. WhatsApp LLC: WhatsApp Key Transparency Overview (Aug 2023), https://www.whatsapp.com/security/WhatsApp-Key-Transparency-Whitepaper.pdf
68. WhatsApp LLC: WhatsApp Key Transparency Overview (Aug 2023), https://www.whatsapp.com/security/WhatsApp-Key-Transparency-Whitepaper.pdf
69. WhatsApp LLC: WhatsApp Encryption Overview (Aug 2024), https://whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf, version 8

A Supporting Material for Proof of Theorem 2

A.1 Signal's Freshness Predicate

We give a translation of the two-party Signal protocol's freshness predicate, as derived in [25], in the syntax of PAIR channels and our PAIR-SEC security experiment.

$$\underline{\mathsf{SIGNAL.FRESH}(i,j,sid,z)} \coloneqq \mathsf{SIGNAL.CLEAN}(type,i,j,sid,z) \wedge \mathsf{SIGNAL.VALID}(i,j,sid,z)$$

$$\underline{\mathsf{SIGNAL.VALID}(i,j,sid,z)} \coloneqq$$
$$^{*}\mathsf{PAIR.FindSession}(sk_i.ssts[ipk_j],sid)[0].status[z] = \mathtt{accept} \wedge \mathsf{SIGNAL.CLEAN}(\mathtt{state},i,j,sid,z)$$

$$\underline{\mathsf{SIGNAL.CLEAN}(\mathtt{triple},i,j,sid,z \overset{is}{=} [0])} \coloneqq$$
$$\mathsf{SIGNAL.CLEAN}(\mathtt{LM},i,j,sid) \vee \mathsf{SIGNAL.CLEAN}(\mathtt{EL},i,j) \vee \mathsf{SIGNAL.CLEAN}(\mathtt{EM},i,j)$$

$$\underline{\mathsf{SIGNAL.CLEAN}(\mathtt{triple+DHE},i,j,sid,z \overset{is}{=} [0])} \coloneqq$$
$$\mathsf{SIGNAL.CLEAN}(\mathtt{triple},i,j,sid,z) \wedge \mathsf{SIGNAL.CLEAN}(\mathtt{EE},i,j,sid,0,0)$$

$$\underline{\mathsf{SIGNAL.CLEAN}(\mathtt{asym\text{-}ri},i,j,sid,z \overset{is}{=} [\textbf{asym-ri}:1])} \coloneqq$$
$$\mathsf{SIGNAL.CLEAN}(\mathtt{EE},i,j,0,[\textbf{asym-ri}:1])$$
$$\vee (\mathsf{SIGNAL.CLEAN}(\mathtt{state},i,j,sid,z) \wedge \mathsf{SIGNAL.CLEAN}(\mathtt{state},i,j,sid,[0]))$$

$\underline{\mathsf{SIGNAL.CLEAN}(\texttt{asym-ri}, i, j, sid, z \overset{is}{=} [\textbf{asym-ri}: x \geq 2])}$ $:=$
  $\mathsf{SIGNAL.CLEAN}(\texttt{EE}, i, j, [\textbf{asym-ri}: x], [\textbf{asym-ir}: x-1]) \vee$
  $(\mathsf{SIGNAL.CLEAN}(\texttt{state}, i, j, sid, z) \ \wedge \mathsf{SIGNAL.CLEAN}(\texttt{asym-ir}, i, j, sid, [\textbf{asym-ir}: x-1]))$

$\underline{\mathsf{SIGNAL.CLEAN}(\texttt{asym-ir}, i, j, sid, z \overset{is}{=} [\textbf{asym-ir}: x])}$ $:=$ $\mathsf{SIGNAL.CLEAN}(\texttt{EE}, i, j, [\textbf{asym-ir}: x], [\textbf{asym-ri}: x])$
  $\vee (\mathsf{SIGNAL.CLEAN}(\texttt{state}, i, j, sid, z) \ \wedge \mathsf{SIGNAL.CLEAN}(\texttt{asym-ri}, i, j, sid, [\textbf{asym-ri}: x]))$

$\underline{\mathsf{SIGNAL.CLEAN}(\texttt{sym-ir}, i, j, sid, z \overset{is}{=} [\textbf{sym-ir}: 0, 1])}$ $:=$
$\mathsf{SIGNAL.CLEAN}(\texttt{state}, i, j, sid, z) \ \wedge \mathsf{SIGNAL.CLEAN}(\,\cdot\,, i, j, sid, [0])$

$\underline{\mathsf{SIGNAL.CLEAN}(\texttt{sym-ir}, i, j, sid, z \overset{is}{=} [\textbf{sym-ir}: x \geq 1, y = 1])}$ $:=$
$\mathsf{SIGNAL.CLEAN}(\texttt{state}, i, j, sid, z) \ \wedge \mathsf{SIGNAL.CLEAN}(\texttt{asym-ir}, i, j, sid, [\textbf{asym-ir}: x])$

$\underline{\mathsf{SIGNAL.CLEAN}(\texttt{sym-ir}, i, j, sid, z \overset{is}{=} [\textbf{sym-ir}: x \geq 0, y \geq 2])}$ $:=$
  $\mathsf{SIGNAL.CLEAN}(\texttt{state}, i, j, sid, z) \ \wedge \mathsf{SIGNAL.CLEAN}(\texttt{sym-ir}, i, j, sid, [\textbf{sym-ir}: x, y-1])$

$\underline{\mathsf{SIGNAL.CLEAN}(\texttt{sym-ri}, i, j, sid, z \overset{is}{=} [\textbf{sym-ri}: x \geq 1, y = 1])}$ $:=$
  $\mathsf{SIGNAL.CLEAN}(\texttt{state}, i, j, sid, z) \ \wedge \mathsf{SIGNAL.CLEAN}(\texttt{asym-ri}, i, j, sid, [\textbf{asym-ri}: x])$

$\underline{\mathsf{SIGNAL.CLEAN}(\texttt{sym-ri}, i, j, sid, z \overset{is}{=} [\textbf{sym-ri}: x \geq 0, y \geq 2])}$ $:=$
  $\mathsf{SIGNAL.CLEAN}(\texttt{state}, i, j, sid, z) \ \wedge \mathsf{SIGNAL.CLEAN}(\texttt{sym-ri}, i, j, sid, [\textbf{sym-ri}: x, y-1])$

$\mathsf{SIGNAL.CLEAN}(\texttt{LM}, i, j, sid)$

```
$\cdot, sst \leftarrow {}^*\mathsf{PAIR.FindSession}(sk_i.ssts[ipk_j], sid)$
if $sst.role = \texttt{init}$: return $(\texttt{corr-ident}, i, \cdot) \notin \mathbf{L} \ \wedge (\texttt{corr-share}, j, \cdot) \notin \mathbf{L}$
elseif $sst.role = \texttt{resp}$: return $(\texttt{corr-share}, i, \cdot) \notin \mathbf{L} \ \wedge (\texttt{corr-ident}, j, \cdot) \notin \mathbf{L}$
```

$\mathsf{SIGNAL.CLEAN}(\texttt{EL}, i, j, sid, z \overset{is}{=} [0])$

```
$\cdot, sst \leftarrow {}^*\mathsf{PAIR.FindSession}(sk_i.ssts[ipk_j], sid)$
if $sst.role = \texttt{init}$: return $(\texttt{corr-sess}, i, j, sid, [0], \cdot) \notin \mathbf{L} \ \wedge (\texttt{corr-ident}, i, \cdot) \notin \mathbf{L}$
elseif $sst.role = \texttt{resp}$: return $\mathsf{SIGNAL.CLEAN}(\texttt{peerE}, i, j, sid, [0]) \ \wedge (\texttt{corr-ident}, i, \cdot) \notin \mathbf{L}$
```

$\mathsf{SIGNAL.CLEAN}(\texttt{EM}, i, j, sid, z \overset{is}{=} [0])$

```
$\cdot, sst \leftarrow {}^*\mathsf{PAIR.FindSession}(sk_i.ssts[ipk_j], sid)$
if $sst.role = \texttt{init}$: return $(\texttt{corr-sess}, i, j, sid, [0], \cdot) \notin \mathbf{L} \ \wedge (\texttt{corr-share}, i, \cdot) \notin \mathbf{L}$
elseif $sst.role = \texttt{resp}$: return $\mathsf{SIGNAL.CLEAN}(\texttt{peerE}, i, j, sid, [0]) \ \wedge (\texttt{corr-share}, i, \cdot) \notin \mathbf{L}$
```

$\mathsf{SIGNAL.CLEAN}(\texttt{EE}, i, j, sid, z, z')$

```
$\cdot, sst \leftarrow {}^*\mathsf{PAIR.FindSession}(sk_i.ssts[ipk_j], sid)$
if $sst.role = \texttt{init}$: return $(\texttt{corr-session}, i, j, sid, z, \cdot) \notin \mathbf{L} \ \wedge \mathsf{SIGNAL.CLEAN}(\texttt{peerE}, i, j, sid, z')$
elseif $sst.role = \texttt{resp}$: return $\mathsf{SIGNAL.CLEAN}(\texttt{peerE}, i, j, sid, z) \ \wedge (\texttt{corr-session}, i, j, sid, z', \cdot) \notin \mathbf{L}$
```

$\mathsf{SIGNAL.CLEAN}(\texttt{peerE}, i, j, sid, z \overset{is}{=} [t : x, y])$

```
$\cdot, sst \leftarrow {}^*\mathsf{PAIR.FindSession}(sk_j.ssts[ipk_i], sid)$
if $sst = (\varnothing, \varnothing)$: return false
elseif $x = 0 \ \wedge sst.role = \texttt{init}$: return $(\texttt{dec}, j, i, sid, \cdot)$ precedes $(\texttt{corr-share}, j, \cdot)$ in $\mathbf{L}$
else : return $(\texttt{corr-session}, j, i, sid, z, \cdot) \notin \mathbf{L}$
    $\wedge \exists \ (\texttt{enc}, j, i, sid, z, (c_{kex}, \cdot), \cdot) \in \mathbf{L},\ (\texttt{dec}, i, j, sid, z, (c_{kex'}, \cdot), \cdot) \in \mathbf{L} \ : \ c_{kex} = c_{kex'}$
```

$\underline{\mathsf{SIGNAL.CLEAN}(\texttt{state}, i, j, sid, z)}$ $:=$ $(\texttt{corr-session}, i, j, sid, z, \cdot) \notin \mathbf{L} \ \wedge (\texttt{corr-session}, j, i, sid, z, \cdot) \notin \mathbf{L}$

A.2 Security Reductions

```
$\mathcal{B}^{\mathsf{Send,Test,Rev}^*}_{\mathsf{SIGNAL},\mathcal{A}}(\mathit{pubinfo})$

$b \leftarrow_\$ \{0,1\}$;   $\mathit{win} \leftarrow 0$;   $\mathbf{L} \leftarrow [\,]$
$\mathit{ssts} \leftarrow \mathsf{Map}\{\}$;   $\mathbf{MK} \leftarrow \mathsf{Map}\{\}$;   $\mathit{used\text{-}epks} \leftarrow \mathsf{Map}\{\}$
$(\mathit{ipk}_0, \ldots, \mathit{ipk}_{n_p-1}),\ (\mathit{spks}_0, \ldots, \mathit{spks}_{n_p-1}),\ (\mathit{epks}_0, \ldots, \mathit{epks}_{n_p-1}) \leftarrow \mathit{pubinfo}$
$b' \leftarrow \mathcal{A}^{\mathsf{Enc,Dec,Corrupt}^*}(\{(\mathit{ipk}_0, \mathit{spks}_0), \ldots, (\mathit{ipk}_{n_p-1}, \mathit{spks}_{n_p-1})\},\ \{\mathit{epks}_0, \ldots, \mathit{epks}_{n_p-1}\})$
return $b = b' \ \wedge \mathsf{PAIR.CONF}(\mathbf{L})$
```

```
$\mathsf{Enc}(i, j, \mathit{info}, \mathit{sid}, m_0, m_1)$

assert $\mathsf{len}(m_0) = \mathsf{len}(m_1)$
if $\mathit{sid} = \varnothing$: $\mathit{sid} \leftarrow \mathit{info}[0]$
$s \leftarrow \mathit{ssts}[i, j, \mathit{sid}]$
if $s = \varnothing$:
    $s \leftarrow \mathsf{len}(\mathit{ssts}[i, \cdot])$
    $\mathsf{Send}(i, s, (\mathit{ipk}_j, 0, \mathtt{init}))$
    $c_{kex} \leftarrow \mathsf{Send}(i, s, \mathit{sid})$
    $\mathit{ssts}[i, j, \mathit{sid}] \leftarrow s$
else:
    $c_{kex} \leftarrow \mathsf{Send}(i, s, \varnothing)$
if $(i, j, \mathit{sid}, \mathit{stage}) \notin \mathbf{MK}$:
    $\mathbf{MK}[i, j, \mathit{sid}, \mathit{stage}] \leftarrow \mathsf{Test}(i, s, \mathit{stage})$
$\mathit{mk} \leftarrow \mathbf{MK}[i, j, \mathit{sid}, \mathit{stage}]$
assert $\mathit{mk} \neq \perp$
$c_{msg} \leftarrow \mathsf{WA\text{-}AEAD.Enc}(\mathit{mk}, c_{kex}, m_b)$
$c \leftarrow (c_{kex}, c_{msg})$
$\mathbf{L} \leftarrow_{app} (\mathtt{enc}, i, j, \mathit{sid}, m_0, m_1, c)$
return $\mathit{sid}, c$
```

```
$\mathsf{Dec}(i, j, \mathit{info}, \mathit{sid}, c)$

$c_{kex}, c_{msg} \leftarrow c$
if $\mathit{sid} = \varnothing$: $\mathit{sid} \leftarrow c_{kex}.\mathit{epk}_{resp}$
$s \leftarrow \mathit{ssts}[i, j, \mathit{sid}]$
if $s = \varnothing$:
    assert $\mathit{sid}$ in $\mathit{epks}_i \setminus \mathit{used\text{-}epks}[i]$
    $\mathit{used\text{-}epks}[i] \leftarrow_\cup \{\mathit{sid}\}$
    $s \leftarrow \mathsf{len}(\mathit{ssts}[i, \cdot])$
    $\mathit{ssts}[i, j, \mathit{sid}] \leftarrow s$
    $\mathsf{Send}(i, s, (\mathit{ipk}_j, 0, \mathit{sid}, \mathtt{resp}))$
    $\mathsf{Send}(i, s, c_{kex})$
else:
    $\mathsf{Send}(i, s, c_{kex})$
if $(j, i, \mathit{sid}, \mathit{stage}) \notin \mathbf{MK}$:
    $\mathbf{MK}[j, i, \mathit{sid}, \mathit{stage}] \leftarrow \mathsf{Test}(i, s, \mathit{stage})$
$\mathit{mk} \leftarrow \mathbf{MK}[j, i, \mathit{sid}, \mathit{stage}]$
assert $\mathit{mk} \neq \perp$
$m \leftarrow \mathsf{WA\text{-}AEAD.Dec}(\mathit{mk}, c_{kex}, c_{msg})$
if $m = \perp$: return $\perp$
$\mathit{replay} \leftarrow (\mathtt{dec}, i, j, \mathit{sid}, c, \cdot) \in \mathbf{L}$
$\mathit{forgery} \leftarrow (\mathtt{enc}, j, i, \mathit{sid}, \cdot, \cdot, c) \notin \mathbf{L}$
$\mathit{win} \leftarrow \mathit{replay} \ \vee (\mathit{forgery} \ \wedge \mathsf{PAIR.AUTH}(\mathbf{L}))$
$\mathbf{L} \leftarrow_{app} (\mathtt{dec}, i, j, \mathit{sid}, c, m)$
if $c \ \in^* \mathsf{Challenges}(\mathbf{L})$: return $\perp$
return $\mathit{sid}, m$
```

```
$\mathsf{CorruptIdentity}(i)$

assert $0 \leq i < n_p$
$\mathit{corr} \leftarrow \mathsf{RevLongTermKey}(i)$
$\mathbf{L} \leftarrow_{app} (\mathtt{corr\text{-}ident}, i, \mathit{corr})$
return $\mathit{corr}$
```

```
$\mathsf{CorruptShared}(i)$

assert $0 \leq i < n_p$
$\mathit{esks} \leftarrow [\mathsf{RevEphemKey}(i, e)$
    for $(e, \mathit{epk})$ in $\mathit{epks}_i$
    if $\mathit{epk} \notin \mathit{used\text{-}epks}]$
$\mathit{ssk} \leftarrow \mathsf{RevMedTermKey}(i, 0)$
$\mathit{corr} \leftarrow \mathit{ssk}, \mathit{esks}$
$\mathbf{L} \leftarrow_{app} (\mathtt{corr\text{-}share}, i, \mathit{corr})$
return $\mathit{corr}$
```

```
$\mathsf{CorruptSession}(i, j, \mathit{sid})$

assert $0 \leq i, j < n_p$
$s \leftarrow \mathit{ssts}[i, j, \mathit{sid}]$
$\mathit{sst} \leftarrow \mathsf{RevState}(i, s, s.\mathit{stage})$
$\mathit{rchsk} \leftarrow \mathsf{RevRand}(i, s, s.\mathit{stage})$
$\mathit{corr} \leftarrow (\mathit{sst}, \mathit{rchsk})$
$\mathbf{L} \leftarrow_{app} (\mathtt{corr\text{-}sess}, i, j, \mathit{sid}, \mathit{corr})$
return $\mathit{corr}$
```

Fig. 27: $\mathcal{B}^{\mathsf{Send,Test,Rev}^*}_{\mathsf{SIGNAL},\mathcal{A}}(\mathit{pubinfo})$

```
$\mathcal{B}^{\mathsf{EUF\text{-}CMA,Enc}}_{\mathsf{WA\text{-}AEAD},\mathcal{A}}()$

$\mathit{forged\text{-}msg} \leftarrow \varnothing$;   $g_i \leftarrow\$ \{0 \dots n_p - 1\}$;   $g_j \leftarrow\$ \{0 \dots n_p - 1\}$;   $g_s \leftarrow\$ \{0 \dots n_i - 1\}$;   $g_z \leftarrow\$ \{0 \dots n_m - 1\}$
$\mathbf{S} \leftarrow \mathsf{Map}\{\}$;   $\mathbf{Z} \leftarrow \mathsf{Map}\{\cdot : 0\}$;   $\mathit{win} \leftarrow 0$;   $\mathbf{L} \leftarrow []$;   $\mathit{ssts} \leftarrow \mathsf{Map}\{\}$;   $\mathbf{MK} \leftarrow \mathsf{Map}\{\}$
for $i = 0, 1, 2, \dots, n_p - 1$:
    $\mathit{ipk}_i, \mathit{isk}_i \leftarrow\$ \mathsf{SIGNAL.KeyGen}()$;   $\mathit{spk}_i, \mathit{ssk}_i \leftarrow\$ \mathsf{SIGNAL.MedTermKeyGen}()$
    for $e = 0, 1, 2, \dots, n_e - 1$:  $\mathit{epk}_e, \mathit{esk}_e \leftarrow\$ \mathsf{SIGNAL.EphemKeyGen}()$
    $\mathit{esks}_i \leftarrow \{\mathit{esk}_e : 0 \le e < n_e\}$;   $\mathit{epks}_i \leftarrow \{\mathit{epk}_e : 0 \le e < n_e\}$
$b' \leftarrow \mathcal{A}^{\mathsf{Enc,Dec,Corrupt}^*}(\{(\mathit{ipk}_0, \mathit{spk}_0), (\mathit{ipk}_1, \mathit{spk}_1), \dots, (\mathit{ipk}_{n_p-1}, \mathit{spk}_{n_p-1})\}, \{\mathit{epks}_0, \mathit{epks}_1, \dots, \mathit{epks}_{n_p-1}\})$
return $\mathit{forged\text{-}msg}$
```

```
$\mathsf{Enc}(i, j, \mathit{info}, \mathit{sid}, m_0, m_1)$

assert $m_0 = m_1$
if $\mathit{sid} = \varnothing$:
    $\mathit{epk}_j \leftarrow \mathit{info}[0]$;   $\mathit{sid} \leftarrow \mathit{epk}_j$
$\mathit{sst} \leftarrow \mathit{ssts}[i, j, \mathit{sid}]$
if $\mathit{sst} = \varnothing$:
    $\mathit{sst}, \cdot \leftarrow\$ \mathsf{SIGNAL.Activate}(\mathit{isk}_i, \mathit{ssk}_i, \mathtt{init}, \mathit{ipk}_j)$
    $\mathit{sst}, c_{kex} \leftarrow\$ \mathsf{SIGNAL.Run}(\mathit{isk}_i, \mathit{ssk}_i, \mathit{sst}, (\mathit{spk}_j, \mathit{sid}))$
    $\mathbf{S}[i, j, \mathit{sid}] \leftarrow \mathsf{len}(\mathbf{S}[i, j, \cdot])$
else:
    $\mathit{sst}, c_{kex} \leftarrow\$ \mathsf{SIGNAL.Run}(\mathit{isk}_i, \mathit{ssk}_i, \mathit{sst}, \varnothing)$
assert $\mathit{sst.status}[\mathit{sst.stage}] = \mathtt{accept}$
if $(i, j, \mathit{sid}, \mathit{sst.stage}) \notin \mathbf{MK}$:
    $\mathbf{MK}[i, j, \mathit{sid}, \mathit{sst.stage}] \leftarrow\$ \mathcal{K}$
$\mathit{mk} \leftarrow \mathbf{MK}[i, j, \mathit{sid}, \mathit{sst.stage}]$
$\mathbf{Z}[i, j, \mathit{sid}] \leftarrow \mathbf{Z}[i, j, \mathit{sid}] + 1$
if $(i, j, \mathbf{S}[i, j, \mathit{sid}], \mathbf{Z}[i, j, \mathit{sid}]) = (g_i, g_j, g_s, g_z)$:
    $c_{msg} \leftarrow \mathsf{AEAD.Enc}(c_{kex}, m_0)$
else:
    $c_{msg} \leftarrow \mathsf{WA\text{-}AEAD.Enc}(\mathit{mk}, c_{kex}, m_0)$
$c \leftarrow (c_{kex}, c_{msg})$
$\mathit{ssts}[i, j, \mathit{sid}] \leftarrow \mathit{sst}$
$\mathbf{L} \leftarrow_{app} (\mathtt{enc}, i, j, \mathit{sid}, m_0, m_1, c)$
return $\mathit{sid}, c$
```

```
$\mathsf{Dec}(i, j, \mathit{info}, \mathit{sid}, c)$

$c_{kex}, c_{msg} \leftarrow c$
if $\mathit{sid} = \varnothing$:  $\mathit{epk}_i \leftarrow c_{kex}.\mathit{epk}_{resp}$;   $\mathit{sid} \leftarrow \mathit{epk}_i$
$\mathit{sst} \leftarrow \mathit{ssts}[i, j, \mathit{sid}]$
if $\mathit{sst} = \varnothing$:
    $[\mathit{esk}] \leftarrow [\mathit{esk}'$ in $\mathit{esks}_i$ if $\mathit{epk}_i = \mathsf{PK}(\mathit{esk}')]$
    $\mathit{sst}, \cdot \leftarrow\$ \mathsf{SIGNAL.Activate}(\mathit{isk}_i, \mathit{ssk}_i, \mathtt{resp}, \mathit{ipk}_j, \mathit{esk})$
    $\mathit{sst}, \cdot \leftarrow\$ \mathsf{SIGNAL.Run}(\mathit{isk}_i, \mathit{ssk}_i, \mathit{sst}, c_{kex})$
    $\mathit{esks} \leftarrow [\mathit{esk}'$ in $\mathit{esks}$ if $\mathit{esk} \neq \mathit{esk}']$
    $\mathbf{S}[i, j, \mathit{sid}] \leftarrow \mathsf{len}(\mathbf{S}[i, j, \cdot])$
else:  $\mathit{sst}, \cdot \leftarrow\$ \mathsf{SIGNAL.Run}(\mathit{isk}_i, \mathit{ssk}_i, \mathit{sst}, c_{kex})$
assert $\mathit{sst.status}[\mathit{sst.stage}] = \mathtt{accept}$
if $(j, i, \mathit{sid}, \mathit{sst.stage}) \notin \mathbf{MK}$:  $\mathbf{MK}[j, i, \mathit{sid}, \mathit{sst.stage}] \leftarrow\$ \mathcal{K}$
$\mathit{mk} \leftarrow \mathbf{MK}[j, i, \mathit{sid}, \mathit{sst.stage}]$
$\mathbf{Z}[i, j, \mathit{sid}] \leftarrow \mathbf{Z}[i, j, \mathit{sid}] + 1$
if $(j, i, \mathbf{S}[j, i, \mathit{sid}], \mathbf{Z}[i, j, \mathit{sid}]) = (g_i, g_j, g_s, g_z)$:
    $m \leftarrow \mathsf{AEAD.Dec}(c_{kex}, c_{msg})$
else:  $m \leftarrow \mathsf{WA\text{-}AEAD.Dec}(\mathit{mk}, c_{kex}, c_{msg})$
if $m = \perp$:  return $\perp$
$\mathit{ssts}[i, j, \mathit{sid}] \leftarrow \mathit{sst}$
$\mathit{replay} \leftarrow (\mathtt{dec}, i, j, \mathit{sid}, c, \cdot) \in \mathbf{L}$
$\mathit{forgery} \leftarrow (\mathtt{enc}, j, i, \mathit{sid}, \cdot, \cdot, c) \notin \mathbf{L}$
$\mathit{guess} \leftarrow (j, i, \mathbf{S}[j, i, \mathit{sid}], \mathbf{Z}[i, j, \mathit{sid}]) = (g_i, g_j, g_s, g_z)$
if $\mathit{replay} \lor (\mathit{forgery} \land \mathsf{PAIR.AUTH}(\mathbf{L}))$:
    if $\mathit{guess}$:  $\mathit{forged\text{-}msg} \leftarrow c$ else:  abort
$\mathbf{L} \leftarrow_{app} (\mathtt{dec}, i, j, \mathit{sid}, c, m)$
return $\mathit{sid}, m$
```

```
$\mathsf{CorruptIdentity}(i)$

assert $0 \le i < n_p$
$\mathit{corr} \leftarrow \mathit{isk}_i$
$\mathbf{L} \leftarrow_{app} (\mathtt{corr\text{-}ident}, i, \mathit{corr})$
return $\mathit{corr}$
```

```
$\mathsf{CorruptShared}(i)$

assert $0 \le i < n_p$
$\mathit{corr} \leftarrow (\mathit{ssk}_i, \mathit{esks}_i)$
$\mathbf{L} \leftarrow_{app} (\mathtt{corr\text{-}share}, i, \mathit{corr})$
return $\mathit{corr}$
```

```
$\mathsf{CorruptSession}(i, j, \mathit{sid})$

assert $0 \le i, j < n_p$
$\mathit{corr} \leftarrow \mathit{ssts}[i, j, \mathit{sid}]$
$\mathbf{L} \leftarrow_{app} (\mathtt{corr\text{-}sess}, i, j, \mathit{sid}, \mathit{corr})$
return $\mathit{corr}$
```

Fig. 28: $\mathcal{B}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{WA\text{-}AEAD},\mathcal{A}}$

```
$\mathcal{B}^{\mathsf{IND\$\text{-}CPA,Enc}}_{\mathsf{WA\text{-}AEAD},\mathcal{A}}()$
$b \leftarrow\!\$\ \{0,1\}$;   $\mathbf{L} \leftarrow []$;   $ssts \leftarrow \mathsf{Map}\{\}$;   $\mathbf{MK} \leftarrow \mathsf{Map}\{\}$
for $i = 0, 1, 2, \ldots, n_p - 1$:
  $ipk_i, isk_i \leftarrow\!\$\ \mathsf{SIGNAL.KeyGen}()$;   $spk_i, ssk_i \leftarrow\!\$\ \mathsf{SIGNAL.MedTermKeyGen}()$
  for $e = 0, 1, 2, \ldots, n_e - 1$:  $epk_e, esk_e \leftarrow\!\$\ \mathsf{SIGNAL.EphemKeyGen}()$
  $esks_i \leftarrow \{esk_e : 0 \le e < n_e\}$;   $epks_i \leftarrow \{epk_e : 0 \le e < n_e\}$
$b' \leftarrow \mathcal{A}^{\mathsf{Enc,Dec,Corrupt^*}}(\{(ipk_0, spk_0), (ipk_1, spk_1), \ldots, (ipk_{n_p-1}, spk_{n_p-1})\}, \{epks_0, epks_1, \ldots, epks_{n_p-1}\})$
return $(b = b' \wedge \mathsf{PAIR.CONF}(\mathbf{L}))$
```

```
$\mathsf{Enc}(i, j, info, sid, m_0, m_1)$
assert $\mathsf{len}(m_0) = \mathsf{len}(m_1)$
if $sid = \varnothing$:
  $epk_j \leftarrow info[0]$;   $sid \leftarrow epk_j$
$sst \leftarrow ssts[i, j sid]$
if $sst = \varnothing$:
  $sst, \cdot \leftarrow\!\$\ \mathsf{SIGNAL.Activate}(isk_i, ssk_i, \mathtt{init}, ipk_j)$
  $sst, c_{kex} \leftarrow\!\$\ \mathsf{SIGNAL.Run}(isk_i, ssk_i, sst, (spk_j, sid))$
else:
  $sst, c_{kex} \leftarrow\!\$\ \mathsf{SIGNAL.Run}(isk_i, ssk_i, sst, \varnothing)$
assert $sst.status[sst.stage] = \mathtt{accept}$
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 $m_0 \neq m_1 \ \wedge |^*\mathsf{Challenges}(\mathbf{L})| = x$:
  $c_{msg} \leftarrow \mathsf{AEAD.Enc}(c_{kex}, m_b)$
elseif $m_0 \neq m_1 \ \wedge |^*\mathsf{Challenges}(\mathbf{L})| > n_d^2 \cdot n_i \cdot n_m$:
  abort
else:
  $c_{msg} \leftarrow \mathsf{WA\text{-}AEAD.Enc}(mk, c_{kex}, m_b)$
$c \leftarrow (c_{kex}, c_{msg})$
$ssts[i, j, sid] \leftarrow sst$
$\mathbf{L} \leftarrow_{app} (\mathtt{enc}, i, j, sid, m_0, m_1, c)$
return $sid, c$
```

```
$\mathsf{Dec}(i, j, info, sid, c)$
$c_{kex}, c_{msg} \leftarrow c$
if $sid = \varnothing$:
  $epk_i \leftarrow c_{kex}.epk_{resp}$;   $sid \leftarrow epk_i$
$sst \leftarrow ssts[i, j, sid]$
if $sst = \varnothing$:
  $[esk] \leftarrow [esk'$ in $esks_i$ if $epk_i = \mathsf{PK}(esk')]$
  $sst, \cdot \leftarrow\!\$\ \mathsf{SIGNAL.Activate}(isk_i, ssk_i, \mathtt{resp}, ipk_j, esk)$
  $sst, \cdot \leftarrow\!\$\ \mathsf{SIGNAL.Run}(isk_i, ssk_i, sst, c_{kex})$
  $esks \leftarrow [esk'$ in $esks$ if $esk \neq esk']$
else:
  $sst, \cdot \leftarrow\!\$\ \mathsf{SIGNAL.Run}(isk_i, ssk_i, sst, c_{kex})$
assert $sst.status[sst.stage] = \mathtt{accept}$
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 $m = \bot$:  return $\bot$
$ssts[i, j, sid] \leftarrow sst$
$\mathbf{L} \leftarrow_{app} (\mathtt{dec}, i, j, sid, c, m)$
if $c \in {}^*\mathsf{Challenges}(\mathbf{L})$:  return $\bot$
return $sid, m$
```

```
$\mathsf{CorruptIdentity}(i)$
assert $0 \le i < n_p$
$corr \leftarrow isk_i$
$\mathbf{L} \leftarrow_{app} (\mathtt{corr\text{-}ident}, i, corr)$
return $corr$
```

```
$\mathsf{CorruptShared}(i)$
assert $0 \le i < n_p$
$corr \leftarrow (ssk_i, esks_i)$
$\mathbf{L} \leftarrow_{app} (\mathtt{corr\text{-}share}, i, corr)$
return $corr$
```

```
$\mathsf{CorruptSession}(i, j, sid)$
assert $0 \le i, j < n_p$
$corr \leftarrow ssts[i, j, sid]$
$\mathbf{L} \leftarrow_{app} (\mathtt{corr\text{-}sess}, i, j, sid, corr)$
return $corr$
```

Fig. 29: $\mathcal{B}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{WA\text{-}AEAD},\mathcal{A}}$