An extended abstract [TRB+25a] of this article appears in the proceedings of ACM CCS 2025. The full version [TRB+25b] of this article is available in the IACR eprint archive. The source code of the corresponding implementation as well as a performance evaluation are available as a Zenodo record [Sch25].

Generic Anonymity Wrapper for Messaging Protocols

Lea Thiemt
FAU Erlangen-Nürnberg
Erlangen, Germany
lea.thiemt@fau.de

Paul Rösler
FAU Erlangen-Nürnberg
Erlangen, Germany
paul.roesler@fau.de

Alexander Bienstock*
J.P. Morgan AI Research and J.P. Morgan AlgoCRYPT CoE
New York, NY, USA
alex.bienstock@jpmchase.com

Rolfe Schmidt
Signal Messenger
Mountain View, CA, USA
rolfe@signal.org

Yevgeniy Dodis
New York University
New York, NY, USA
dodis@cs.nyu.edu

Abstract

Modern messengers use advanced end-to-end encryption protocols to protect message content even if user secrets are ever temporarily exposed. Yet, encryption alone does not prevent user tracking, as protocols often attach metadata, such as sequence numbers, public keys, or even plain user identifiers. This metadata reveals the social network as well as communication patterns between users. Existing protocols that hide metadata in Signal (i.e., Sealed Sender), for MLS-like constructions (Hashimoto et al., CCS 2022), or in mesh networks (Bienstock et al., CCS 2023) are relatively inefficient or specially tailored for only particular settings. Moreover, all existing practical solutions reveal crucial metadata upon exposures of user secrets.

In this work, we introduce a formal definition of Anonymity Wrappers (AW) that generically hide metadata of underlying two-party and group messaging protocols. Our definition captures forward and post-compromise anonymity as well as authenticity in the presence of temporary state exposures. Inspired by prior wrapper designs, the idea of our provably secure AW construction is to use shared keys of the underlying wrapped (group) messaging protocols to derive and continuously update symmetric keys for hiding metadata. Beyond hiding metadata on the wire, we also avoid and hide structural metadata in users' local states for stronger anonymity upon their exposure.

We implement our construction, evaluate its performance, and provide a detailed comparison with Signal's current approach based on Sealed Sender: Our construction reduces the wire size of small 1:1 messages from 441 bytes to 114 bytes. For a group of 100 members, it reduces the wire size of outgoing group messages from 7240 bytes to 155 bytes. We see similar improvements in computation time for encryption and decryption, but these improvements come with substantial storage costs for receivers. For this reason, we develop extensions with a Bloom filter for compressing the receiver storage. Based on this, Signal considers deploying our solution.

1 Introduction

Most modern messengers are based on advanced encryption protocols that continuously update the used key material. Due to these updates, exposing the local secrets of a user does not affect the confidentiality of past or future encrypted messages, which is called *Forward Secrecy* (FS) and *Post-Compromise Security* (PCS), respectively.

Privacy and Anonymity. While these encryption protocols provide strong confidentiality for the communicated payload, metadata about the communicating users remains unprotected. In particular, without further measures, central messaging providers learn the social network and the communication patterns between their users. A range of techniques has been proposed to protect user anonymity in messaging, targeting different layers of the communication stack. Network-level approaches, such as onion routing [RSG98, DMS04] and mix networks [Abe99, SW06], obfuscate communication paths and patterns in transit. Similarly, server-side techniques like Private Information Retrieval [CKGS98, SJ17] and Oblivious Message Retrieval [LT22] prevent servers from learning which messages users send or receive. While these mechanisms are essential for achieving full anonymity in practice, they are orthogonal to our work. Concretely, our approach does not hide metadata which a routing service attaches to the ciphertext. Instead, our approach operates at the protocol level and falls in line with the approaches we discuss in the following. Notably, our protocols can be efficiently instantiated by standard building blocks.

Sealed Sender. With Signal's anonymization layer, called *Sealed Sender*, a sender S public-key-encrypts each messaging ciphertext c' with the receiver's long-term key pk_R. While the Double Ratchet ciphertext c' contains visible metadata like a sequence number or repeating sender public keys, the Sealed Sender ciphertext $c^* = \mathrm{enc}(pk_R, c')$ looks random. For efficient delivery via the central messaging server, the sender attaches the receiver's identifier to the anonymized ciphertext: $c = (c^*, R)$; the actual Sealed Sender protocol is slightly more complex and, additionally, offers sender authentication. Overall, this anonymizing public-key encryption layer hides all explicit metadata towards observers on the wire, except the receiver identity, which serves as routing information. The receiver can simply decrypt all incoming ciphertext with their long-term secret key sk_R. Recent analyses of Sealed Sender [MKA+21, BH23] demonstrate that, despite hiding explicit metadata, the interaction between communication partners can reveal their identities; this issue arises on the routing level and, as mentioned above, is outside the scope of our work. However, the approach of Sealed Sender has three further crucial drawbacks: (a) when sending messages to a group $G = \{R_i\}_{i\in[n]}$, the sender encrypts the compact group messaging ciphertext individually to every other group member $R_i, i \in [n]$, which induces a linear computation and communication overhead; (b) if a receiver is ever corrupted, their exposed

*Alexander Bienstock did not contribute to the implementation for this paper.

long-term secret key sk_R reveals the sender identity of every ciphertext sent to them; (c) it is unclear how the Diffie-Hellman-based design of Sealed Sender can be made post-quantum secure.

Double Ratchet Extension. To protect hidden metadata even after exposures of user secrets, the specification of Signal's Double Ratchet (DR) [PM16a] proposes an alternative concept to Sealed Sender, called *Header Encryption*. With this, users regularly derive fresh additional keys k_i from the key updates of the DR, which are used to encrypt metadata of the messaging ciphertexts via $c_i^* = \text{enc}(k_i, c_i')$. However, it remains unclear from this concept specification how receivers identify the right key for decrypting ciphertexts c_i^* with encrypted metadata. (Note that receivers have multiple ongoing DR session in parallel—at least one with each sender—and possibly many header encryption keys in each session; if the header-encrypted ciphertext contains no visible metadata about its session or its sender, identifying the right key may require expensive trial decryptions.) Therefore, Signal's Header Encryption is not suited for practice.

Metadata in MLS. The MLS standard [BBR+23] for secure group messaging proposes a concept similar to Header Encryption, called *Private Messages*. There, all metadata of messaging ciphertexts c_i', except a session identifier for the group chat and the current epoch therein, are encrypted with dedicated keys: $c_i = (id_G, ep_G, \text{enc}(k_{id_G,ep_G}, c_i'))$. Using the revealed group identifier and epoch number (id_G, ep_G), receivers can efficiently find the right key k_{id_G,ep_G} in their local state for decrypting the remaining metadata. The ongoing MIMI standardization initiative extends MLS for *interoperable* messaging; the MIMIMI [KK25] draft, which extends MIMI for hiding metadata, proposes techniques that are similar to MLS's Private Messages. So far, Signal's Sealed Sender, MLS's Private Messages, and MIMIMI have not been formally analyzed.

Private Messages in MLS only hide metadata for messaging ciphertexts. The metadata of the remaining MLS traffic is not protected by this, which includes cryptographic key updates and group membership operations. Hashimoto et al. [HKP22] propose a specialized extension for this. Their extension is designed to facilitate key updates in tree-based Continuous Group Key Agreement (CGKA) protocols without leaking the structure of the group or the initiator of such updates.

Further Academic Work. From a rather theoretical perspective, Dowling et al. [DHRR22] develop a strongly anonymous protocol that continuously establishes fresh keys in a unidirectional two-party session; for this, they extend prior non-anonymous concepts of unidirectional messaging [BSJ+17, PR18b, PR18a, BRV20, RSS23, CR25]. While this hides metadata on the wire as well as in the users' local states, the novel pairing-based Lamport signature scheme in their protocol is too inefficient for real-world deployment.

Finally, as a byproduct of their messaging protocol for mesh networks, Bienstock et al. [BRT23] propose a simple anonymizing wrapper protocol for the DR. Similar to Header Encryption, senders regularly derive fresh keys k_i from the DR to anonymize messaging ciphertexts c_i' that contain metadata. In addition to each derived key k_i, a random looking tag t_i is derived, which is attached to the final anonymized ciphertext $c_i = (\text{enc}(k_i, c_i'), t_i)$ before sending. The receiver pre-computes multiple derived keys and tags (k_j, t_j) in advance such that, when receiving an anonymized ciphertext $c_j = (c_j^*, t_j)$, they can take the attached tag t_j to find the matching de-anonymization key k_j for decryption. Our protocol is inspired by this. Yet, the construction by Bienstock et al. [BRT23] has notable drawbacks: (a) since it does not offer sender authentication, it is not directly generalizable to group messaging; (b) the size of the lists of pre-computed keys and tags in the receiver state can be too large for deployment in practice; (c) although the authors aimed to reduce metadata in the local user states, their construction leaks communication patterns due to stored sequence numbers of (not yet) received ciphertexts.

In conclusion, there exist many specially tailored techniques for hiding metadata of certain messaging protocols. An overview thereof can be found in Table 1. All practically relevant techniques reveal some metadata upon user corruptions and, more importantly, there has been no attempt to formalize a cryptographic abstraction that covers the idea of a generic Anonymity Wrapper.

Cryptographic Primitive. We introduce the concept of an Anonymity Wrapper (AW) as a cryptographic primitive, which generically hides metadata from existing messaging protocols. This primitive has four algorithms: AW.up_S, which initializes or updates the sender state, AW.up_R, which does the same for the receiver state, AW.enc_S, which wraps ciphertexts of underlying messaging protocols to hide metadata and anonymize their users, and AW.dec_R, which de-anonymizes wrapped ciphertexts.

Anonymity under State Exposures. We require that AW protocols offer Forward Anonymity (FA), which means that a wrapped ciphertext c must look random even if the local states of sender or receivers are exposed *after* c was sent and received, respectively. Furthermore, AW must offer Post-Compromise Anonymity (PCA), which means that the update algorithms AW.up_S and AW.up_R refresh the user states such that the wrapped ciphertext c looks random even if the user states were exposed *before* such an update. We formalize these two requirements with *game-based* definitions in Section 3.1.

We complement these definitions in Section 5.1 focusing on the metadata leakage of the sender and receiver state to meaningfully capture a broader sense of anonymity under state exposures. For this, consider an example protocol with random looking ciphertexts, which fulfills the two above requirements. However, this protocol stores sequence numbers and timestamps for all (not yet) received ciphertexts in the local sender and receiver states, which does not contradict these two requirements. Yet, upon state exposure, the sequence numbers reveal the users' communication patterns, which intuitively undermines our goal. For this reason, our *UC-based* definition in Section 5.1 specifies which metadata the user states of an ideal AW protocol should (not) leak. This definition is slightly more complex than our compact game-based definition.

We note that, when wrapped around an underlying messaging protocol, only our wrapper avoids and hides metadata in its own state. However, when our wrapper entirely replaces an underlying messaging protocol (see below), also the state metadata is fully protected.

	Group protocol	FA Snd	FA Rcv	PCA Snd	PCA Rcv	Snd auth.	$\|c\|$ in groups	State obf.	PQ ready	Deployed in
Double Ratchet (DR) [PM16a]	✗	✗	✗	✗	✗			✗	✗	WA, Messenger
Post-Quantum DRs [Ste24, DJK+25]	✗	✗	✗	✗	✗			✗	✓	iMessage
DR & Header Enc. [PM16a]	✗	✓	✓	✓	✓			✗	(✓)	Impractical
DR & Sender Keys (SK) [RMS18, BCG23]	✓	✗	✗	✗	✗	✓	1	✗	(✓)	WA, Messenger
DR & Sealed Sender (SS) [Lun18]	✗	✓✓	✗	✓✓	✗			✗	✗	Signal
DR & SK & SS	✓	✓✓	✗	✓✓	✗	✓	n	✗	✗	Signal
MLS [BBR+23]	✓	(✓)	(✓)	(✓)	(✓)	✓	1	✗	✓	Wire
MLS Metadata Hiding [HKP22]	✓	(✓)	(✓)	(✓)	(✓)	✓	1	✗	✓	
MIMIMI [KK25]	✓	(✓)	(✓)	(✓)	(✓)	✓	1	✗	✓	
Strongly Anon. RKE [DHRR22]	✗	✓✓	✓✓	✓✓	✗			✓	✗	
ASMesh [BRT23]	✗	✓✓	✓✓	✓	✓			✗	✓	
Anonymity Wrapper (AW)	✓	✓✓	✓✓	✓	✓	✗	1	✓	✓	
Authenticated AW (AAW)	✓	✓✓	✓✓	✓	✓	✓	1	✓	✓	

Table 1: Comparison of messaging protocols for two-party and group chats in composition with methods that hide metadata on the wire and from local user states. ✓/✗: A protocol has/does not have the property. We distinguish Forward Anonymity (FA) and Post-Compromise Anonymity (PCA) for sender and receiver state exposure. Here, ✓✓ means the protocol has immediate FA/PCA on exposure, ✓ indicates that a key update is required to restore it, and (✓) indicates that the given FA/PCA only partially fulfills our strong anonymity definition. For group chat protocols, we list whether senders are authenticated in the presence of malicious receivers and we provide the asymptotic communication overhead. Besides our protocols, only one other hides all relevant metadata in user states. There exist post-quantum secure variants/replacements for all protocols except Sealed Sender and [DHRR22] and we list the real-world messengers that use these protocols. Post-quantum readiness of compositions with the DR is marked with (✓) to denote that post-quantum variants of the DR must be used.

Composition with Messaging Protocols. The strength of our approach is that an AW protocol can simply be added on top of existing messaging protocols. Yet, for initializing an AW session and for updating the user states, algorithms $\mathsf{AW.up_S}$ and $\mathsf{AW.up_R}$ take fresh symmetric keys that are shared between sender and the receivers from some external source. This source can be an initial authenticated key exchange protocol (like X3DH [PM16b] or Noise [Per18, DRS20]) or the wrapped messaging protocol itself; for this, generic composition has been formally studied in [FG14, BFWW11, BSJ+17].

Application to Double Ratchet. For example, when applying our wrapper approach to the Signal two-party Double Ratchet (DR) algorithm, sender and receiver regularly derive additional keys from the DR to feed them into the AW update algorithms. As a result, FS and PCS of the DR are leveraged for FA and PCA of the AW protocol wrapped around it. Thereby, our protocol is directly applicable to the two-party chats in WhatsApp, Signal, Facebook Messenger, iMessage, etc. An evaluation of the resulting performance is given in Section 4.2, where we also conduct a comparison with the Signal Sealed Sender wrapping protocol. By reducing the ciphertext overhead by a factor of 4 and the en- and decryption time by a factor of 14, we demonstrate that our results do not only offer strong security and privacy guarantees but also that the efficiency is far better than with Sealed Sender.

Replacing Sender Keys. As another example, our AW protocol can be wrapped around Signal's Sender Keys [BCG23], which is the group messaging protocol in WhatsApp, Signal, Facebook's Messenger, Matrix, etc. We observe that our notion of anonymity already includes the confidentiality guarantees of Sender Keys, and we add sender authenticity (see below) to AW, which yields an *Authenticated* Anonymity Wrapper (AAW). Overall, AAW offers slightly stronger security guarantees than Sender Keys and it hides metadata with FA and PCA. Thus, one can simply replace Sender Keys in the listed messengers with our AAW protocol to gain far stronger privacy for the users. Our performance evaluation in Section 4.2 demonstrates the practicality of our approach and, in comparison with Signal's composition of Sender Keys and Sealed Sender, our AAW protocol is substantially more efficient: recall that the main drawback of Sealed Sender is the *linear* communication overhead because every compact group message is individually public-key-encrypted to each group member. In contrast, our wrapper only adds a *small constant* communication overhead and it is based on symmetric encryption, such that confidentiality and anonymity can be achieved even against quantum adversaries.

Roadmap. Our main contribution is formalizing the concept of Anonymity Wrappers (AW) for hiding metadata of generically wrapped two-party and group messaging protocols. We introduce game-based definitions of anonymity in Section 3.1 and of symmetric and sender authenticity in Section 3.2, where the latter yields Authenticated AW (AAW). Next, we present our AW and AAW constructions in Section 4 and we sketch the proofs that are provided in the appendix. With the implementation of our construction [Sch25], we conduct a performance evaluation in Section 4.2. By measuring computation and communication costs, we demonstrate the practical deployability of our constructions, and we compare these costs with the Signal Sealed Sender metadata hiding protocol. Concretely, our construction substantially outperforms Signal's current approach in all dimensions except the receiver state size, which is a few hundred kilobytes per session. Motivated by this, we develop

optimizations for the receiver state in Section 6. In particular, we propose a Bloom-filter-based compression that decreases the state size to a few dozen kilobytes in total. Finally, with the UC-based definition in Section 5.1, we propose a formalization that complements our game-based definitions for meaningfully capturing that local user states should hide metadata. We currently collaborate with Signal for implementing and, possibly, deploying AW and AAW in their messenger.

Anonymity Guarantees in Practice. To clarify the impact of our results, we discuss anonymity under state corruptions in a broader sense: when a user's cryptographic key material is exposed, a possible scenario is that their whole device was compromised, including the user's address book. One could argue that this reveals the user's social network and thus makes FA and PCA redundant. Yet, these anonymity guarantees can still substantially contribute to the user's anonymity. That is, users can have an enormous address book and communicate with very few of those people and contacts can be deleted either intentionally or when a user gets a new device. This means that prior to the exposure and after a key update, an adversary thus does not learn whether a user was or is actually in contact with a specific user in their address book. The anonymity guarantees become even more meaningful when an app implements features to cryptographically hide contacts and thus does not reveal them on exposure. For instance, opening a conversation requires entering a PIN hardened with a system like Signal's SecureValueRecovery protocol [Mes25b].

Limitations. As mentioned above, our work is only effective with further, complementary measures for comprehensive anonymity and it provokes future research in multiple directions. This includes anonymous contact discovery and communication initialization, routing that leverages the guarantees of our constructions for fully (i.e., not only sender-)anonymous messaging, anonymous session synchronization in multi-device settings, or plausible anonymity in the broader messaging app environment, which includes hiding contacts in the address book. On a rather technical level, our proofs in the random oracle model could be lifted to the standard model; in this work, we use the random oracle model to simplify the proofs for better comprehensibility and compactness. Finally, for broader applicability, we develop (A)AW as wrapper protocols that need external input of symmetric keys from (continuous) key exchange protocols like the DR; for avoiding redundancy, it remains an interesting open problem to modify messaging protocols like the DR directly for offering anonymity.

1.1 Technical Overview

Initialization and State Updates. The protocol execution between the sender and one or more receivers begins with executing algorithms $\mathsf{AW.up_S}$ and $\mathsf{AW.up_R}$, respectively, which take a shared symmetric input key k_0 to establish an internal shared symmetric chain key $ck_{0,0}$. Whenever sender and receivers re-run the update algorithms for PCA on another input key k_i, a new epoch i begins in which chain key $ck_{i,0}$ is derived from the mix of k_i and an internal secret derived from the prior epoch's initial chain key $ck_{i-1,0}$.

Anonymizing Ciphertexts. To anonymize some input message m—which can be a plaintext message or a ciphertext with metadata from the underlying messaging protocol—, the sender takes current chain key $ck_{i,j-1}$ to derive a new chain key $ck_{i,j}$, a tag $t_{i,j}$, and a message key $k_{i,j}$ that encrypts m. Using authenticated encryption, the anonymized ciphertext $c_{i,j} = (\mathrm{enc}(k_{i,j}, m), t_{i,j})$ looks pseudorandom and, therefore, hides all metadata on the wire. We note that methods for anonymously routing and forwarding anonymous ciphertexts from sender to receiver are outside the scope of this work; by just attaching (and, thereby, leaking) the receiver identity to the anonymous ciphertext, efficient delivery is facilitated.

Key Computation. While the sender always only has the single current chain key in their state, the receiver pre-computes multiple key-tag pairs $(k_{i,j}, t_{i,j})$ in advance to enable immediate decryption of ciphertexts that arrive out of order. Since anonymized ciphertexts look random—even to the receiver—, the pre-computed tags identify the right key for de-anonymizing incoming ciphertexts. For this reason, the pre-computed key-tag pairs for all sessions of the receiver are stored in a single consolidated receiver state.

Receiver State. Conceptually, the so far described AW construction is similar to the simple design by Bienstock et al. [BRT23]. However, a weakness of the construction in [BRT23] is that, along with every pre-computed key-tag pair, the receiver stores the corresponding epoch number (i.e., the number of updates so far) and the sequence number in that epoch; this is done to delete unnecessarily pre-computed key-tag pairs during epoch progressions. Upon adversarial state exposure, the revealed epoch and sequence numbers leak which ciphertexts have (not) been received so far.

To avoid this, our construction maintains three constant sized lists of pre-computed key-tag pairs: one list for *future* ciphertexts in the *current* epoch, one list for *older* not yet received ciphertext, and, potentially, one list for future ciphertexts in the *next* epoch. If needed, the list for older ciphertexts is filled with indistinguishable dummy entries. Furthermore, when advancing to the next epoch and, thereby, cleaning key-tag pairs, we use the first unnecessarily pre-computed tag as an implicit pointer for deletion instead of explicit epoch or sequence numbers. Using this slightly advanced referencing scheme, which we elaborate on in Section 5, our construction does not leak which ciphertexts have (not) been received so far.

Sender Authentication. Due to authenticated encryption, our anonymized ciphertexts are authenticated *symmetrically*. For securely anonymizing *group* messaging protocols with potentially malicious receivers, we add sender authentication: the sender can generate a signature key pair (sk, vk) with every update and the receiver obtains the corresponding verification key vk upon their update. When anonymizing a message, the sender signs the anonymized ciphertext. To hide the resulting signature σ on the wire, the sender derives an additional key k_{otp} from the current chain key, which one-time-pad-encrypts σ: $c = (c^*, t, \mathrm{sign}(sk, (c^*, t)) \oplus k_{otp})$. To ensure that the relation between pre-computed key-tag pairs in the receiver state remains hidden, the receiver does not store vk in plain; instead, it stores a randomized commitment of vk with every key-tag pair: $(k_{i,j}, t_{i,j}, \mathrm{comm}(r_{i,j}, vk_i))$. To verify signatures on receipt, the sender includes vk in every message that will be

anonymized; the receiver then verifies the signature and checks if the commitment opens to vk.

Recent literature [JKS24, JK25] formalized fine-grained attacker models against confidentiality and authenticity—but not anonymity—for the *stateless* primitive Symmetric Signcryption. The construction idea behind this is that authenticated encryption with a long-term symmetric key is enriched with and bound to signature-based authentication with a long-term asymmetric key. In essence, our *stateful* Authenticated AW construction sketched above follows a similar design approach but adds continuous key updates, which yields confidentiality, authenticity, as well as anonymity with FS, FA, PCS, and PCA. A detailed discussion of the differences between AAW and Symmetric Signcryption is provided in Appendix A.2.

Performance. We implement our construction and evaluate its performance in Section 4.2 by comparing it to Signal's current protocol.

For two-party messaging, our protocol would be wrapped around the Double Ratchet, whereas Signal currently anonymizes Double Ratchet ciphertexts with Sealed Sender. Besides the strengthened anonymity under state exposures, our protocol enhances the communication overhead from 441 bytes to 114 bytes per ciphertext, and it reduces the en- and decryption time from about 60 µs to 4 µs.

In groups, the performance gain is even more evident: Signal currently encrypts group messages using Sender Keys; the resulting compact ciphertext is wrapped with Sealed Sender to each group member individually. This yields a total packet size of $440 + 68 \cdot n$ bytes, where n is the group size, and an encryption time of 1.1 **m**s for a group of size $n = 100$. Our AAW protocol can replace this entirely, which yields a constant ciphertext size of 155 bytes and and a constant encryption time of 21 **µ**s; decryption time is halved from 61 µs to 32 µs.

The main drawback of our construction is the size of pre-computed key-tag pairs in the receiver state: for AW, the receiver stores 120 KB globally plus 288 KB per session; for AAW, the latter increases to 480 KB per session.

Receiver State Compression. The simplest reduction of space can be achieved by not storing dummy tag-key pairs in the list for old not yet received ciphertexts, which only leaks minimal additional metadata.

For the two remaining lists that contain pre-computed tag-key pairs for future ciphertexts, we develop a Bloom-filter-based enhancement in Section 6. Concretely, instead of storing many tag-key pairs for the future explicitly, we only store very few of them for the standard case in which only few ciphertexts are dropped or re-ordered during delivery. For the bad case in which many ciphertexts are dropped or re-ordered, the receiver pre-computes the tags and only stores them implicitly in session-wise Bloom filters. Upon receipt of a bad-case ciphertext for which the tag-key pair is not stored explicitly, the receiver searches for the ciphertext's tag in all sessions' Bloom filters; when this is found, the full tag-key pair is computed explicitly. While this approach only increases the computation time of the receiver minimally, it reduces the storage size by a factor 14, which yields a construction with practical performance in all dimensions.

2 Preliminaries

We use $[n]$, where n is an element of the natural numbers $\mathbb{N}$, as shorthand for the set $\{1, 2, ..., n\}$, and $(x_i)_{i\in[n]}$ as shorthand for the sequence $(x_1, x_2, ..., x_n)$. We use $x \leftarrow y$ to assign the value y to the variable x. We also use this notation to assign multiple values in one line, e.g., $(x_i)_{i\in[n]} \leftarrow 0^n$ denotes $x_1 \leftarrow 0, ..., x_n \leftarrow 0$. If Y is a set, we write $x \leftarrow_{\$} Y$ to denote that the value x is randomly sampled from Y. For a function f with (potentially empty) input $(a_i)_{i\in[n],n\in\mathbb{N}}$, we write $(x_j)_{j\in[m],m\in\mathbb{N}} \leftarrow_{(\$)} f((a_i)_{i\in[n]})$ to indicate that f is a deterministic (or probabilistic) function which outputs values $(x_j)_{j\in[m],m\in\mathbb{N}}$. Similarly, we define a function f through $f : V \rightarrow_{(\$)} W$, denoting f takes as input arguments from V and outputs (probabilisticly) a value in W. To initialize variable x, set V, and dictionary D with empty values, we write $x, V, D[\cdot] \leftarrow \perp$[3]. The Cartesian product of two sets V, W is denoted as $V \times Y$, and we use $X \xleftarrow{\cup} V$ as shorthand for $X \leftarrow X \cup V$. We denote bitwise XOR of two variables by $x \oplus y$.

In security definitions and proofs, we denote the adversary as $\mathcal{A}, \mathcal{B}, \mathcal{C}$, etc. Writing $(x_j)_{j\in[m]} \leftarrow_{(\$)} \mathcal{A}((a_i)_{i\in[n]})$ means that $\mathcal{A}$ is executed on some (potentially empty) input $(a_i)_{i\in[n]}$ and (probabilisticly) outputs some value(s) $(x_j)_{j\in[m]}$. In security definitions and constructions, we use the keyword "Require *condition*" to indicate that if the *condition* evaluates to false, we immediately rewind all changes which happened during the function execution and return $\perp$. When using a hash function H with input a which returns multiple values $(x_1, ..., x_n, y_1, ..., y_m) \leftarrow \mathrm{H}((a_i)_{i\in[n]})$, for $n, m \in \mathbb{N}$, where some values $y_1, ..., y_m$ are not required for further computation, we implicitly discard these values and write $(x_1, ..., x_n) \leftarrow \mathrm{H}((a_i)_{i\in[n]})$.

Our constructions use an Authenticated Encryption with Associated Data (AEAD) scheme AE = (AE.gen, AE.enc, AE.dec) with pseudo-random (IND$) and strongly unforgeable (SUF-CMA) ciphertexts, a signature scheme S = (S.gen, S.sgn, S.vfy) with strong unforgeability (SUF-CMA), and a commitment scheme C = (C.com, C.vfy) that provides binding and hiding. We use the data structures linked list (LL) and hash table (HT) with standard interfaces. Furthermore, we use a Counting Bloom Filter (CBF), which is a compact and efficient representation of a modifiable set with membership testing: elements can be added to and removed from the set and the membership of elements in the set can be tested with a configurable low false positive rate. We provide the full security definitions for these standard building blocks in Appendix B.

3 Anonymity Wrapper

The concept of anonymity wrappers was introduced informally by Bienstock et al. [BRT23] and formally, but specially tailored to MLS-like group messaging protocols, by Hashimoto et al. [HKP22]. We take a step back and propose a generic notion that captures anonymous wrapping of two-party and group messaging and, as demonstrated by our construction in Section 4, allows for highly efficient instantiations. More concretely, we consider an anonymity wrapper AW to be connected to and wrapped around arbitrary underlying two-party or group messaging protocols such that the wrapped messaging traffic sent over the network—and seen by messaging providers—reveals no metadata. Thereby, we consider efficient (anonymous) routing of messaging traffic out of scope:

senders can either attach the receiver identity to the anonymized traffic or employ more advanced routing techniques that hide this information.

Syntax. An anonymity wrapper is a tuple of algorithms $\mathrm{AW} = (\mathrm{AW.up_S}, \mathrm{AW.up_R}, \mathrm{AW.enc_S}, \mathrm{AW.dec_R})$ for some sender state space $\mathcal{ST}_S$, update key space $\mathcal{K}$, verification key space $\mathcal{VK}$, receiver state space $\mathcal{ST}_R$, session identity space $\mathcal{ID}$, message space $\mathcal{M}$, and ciphertext space C:

- $\mathrm{AW.up_S} : (\{\perp\} \cup \mathcal{ST}_S) \times \mathcal{K} \to_\$ \mathcal{ST}_S \times (\{\diamond\} \cup \mathcal{VK})$ Initializes (with first input $\perp$) or updates a sender state with some update key $k \in \mathcal{K}$; for asymmetric sender authentication, some verification key $vk \in \mathcal{VK}$ can be output and shared with receivers.
- $\mathrm{AW.up_R} : (\{\perp\} \cup \mathcal{ST}_R) \times \mathcal{ID} \times \mathcal{K} \times (\{\diamond\} \cup \mathcal{VK}) \to \mathcal{ST}_R$ Initializes (with first input $\perp$) or updates the receiver state for a session locally identified by $id \in \mathcal{ID}$ with some update key k and, possibly, some verification key vk.
- $\mathrm{AW.enc_S} : \mathcal{ST}_S \times \mathcal{M} \to_\$ \mathcal{ST}_S \times C$ Anonymizes a payload message $m \in \mathcal{M}$.
- $\mathrm{AW.dec_R} : \mathcal{ST}_R \times C \to \mathcal{ST}_R \times \mathcal{ID} \times \mathcal{M}$ De-anonymizes a ciphertext $c \in C$ by extracting message m in session id.

Consolidated Receiver State. Each sender uses an individual state $st_S \in \mathcal{ST}_S$ for every session in which they communicate as a *sender*, where the number of receivers per session is unbounded. Whenever the sender wants to send in a session via $\mathrm{AW.enc_S}$, the respective sender state is selected and used by the higher level application. Receivers, in contrast, cannot know in advance to which session a received ciphertext belongs, since ciphertexts hide this information; thus, receivers cannot select the matching individual receiver state *before* processing a ciphertext with $\mathrm{AW.dec_R}$. For this reason, receivers store the information for all the sessions in which they receive in a single, consolidated receiver state $st_R \in \mathcal{ST}_R$. This means that, in our single-device setting, every user has one sender state for each of their sessions and a single consolidated receiver state for all of their sessions.

Session Identifier. To update the key material of a session, the sender can just take the corresponding sender state st_S together with an update key k_i via $(st_S, vk) \leftarrow_\$ \mathrm{AW.up_S}(st_S, k_i)$. In contrast, since the receiver maintains a consolidated state st_R for all its receiver sessions, updates of a particular session in that state need to be addressed with identifier id via $st_R \leftarrow \mathrm{AW.up_R}(st_R, id, k_i, vk)$. This identifier is only used and specified locally by each receiver individually. Therefore, different receivers of the same session can use different identifiers for that session. For authentication, algorithm $\mathrm{AW.dec_R}$ outputs the identifier of the session in which a ciphertext was received.

Update Keys. Sender and receiver(s) of a session use the same initial key k_0 to initiate a session and, thereafter, keys k_i to continuously update their session state. Each fresh key starts a new *epoch* in the session, which allows for forward and post-compromise anonymity. The source of the initial key and of update keys can be some external key exchange or the underlying messaging protocol itself. For example, when the AW wraps traffic of a two-party Double Ratchet session, sender and receiver can derive symmetric keys k_i from internal DR root-key updates that they then feed into the updates of the AW session. In contrast, for anonymizing group chats that are based on Signal's Sender Keys, our AW protocol can essentially replace Sender Keys, which avoids redundancy and strengthens security. Thereby, the update keys are always randomly sampled and distributed from the sender to each receiver via their existing pairwise channels (see [BCG23]).

Verification Keys. In a multi-receiver session for a group chat, it can be desirable that authenticity of sent ciphertexts is maintained even if some of the session's receivers act maliciously. For this, we let the sender's update algorithm possibly output asymmetric verification keys; these keys belong to signing keys that are internally held in the sender's updated state. During receiver updates, these verification keys can be included in the receiver state such that, upon decryption, the authenticity of incoming ciphertexts can be verified. We call this variant *Authenticated* AW (AAW). (In stateless communication, similar concepts, yet without anonymity, have been studied under the term Symmetric Signcryption [JKS24, JK25]; we elaborate on this in Appendix A.2.) For single-receiver sessions, this feature can be undesirable; thus, senders and receivers output and input symbol $\diamond$ in this standard AW case instead.

Out-of-order Receipt. Modern messaging protocols like the Double Ratchet Algorithm [PM16a] for two-party chats or Signal's Sender Key Mechanism [BCG23] for group chats offer so-called *Immediate Decryption*. This means that, even if ciphertexts are received out of order, their encrypted payload messages can be decrypted *immediately*. For practicality, both mentioned real-world protocols limit the out-of-order tolerance with parameters `past` and `fut`[1] such that at most `past` ciphertexts older than the newest decrypted one can be decrypted, and at most $\mathtt{fut} - 1$ omitted ciphertexts after the newest decrypted one are tolerated for successful decryption of a newer one. We follow the same approach and parameterize our definitions with these two constants: for correctness, we require that ciphertexts in the current receivable window that is defined by $(\mathtt{past}, \mathtt{fut})$ must be decryptable, and for security we require that corruptions of receiver secrets do not affect the anonymity of ciphertext outside the current receivable window. Thus, protocols with different levels of tolerance for unreliable delivery can be analyzed with our parameterizable definitions.

Correctness. Intuitively, an (Authenticated) Anonymity Wrapper is correct if the decrypted message $(st_R, id, \boldsymbol{m}) \leftarrow \mathrm{AW.dec_R}(st_R, c)$ of a received ciphertext c equals the message encrypted in that sent ciphertext $c \leftarrow_\$ \mathrm{AW.enc_S}(st_S, \boldsymbol{m})$ and the receiver's local identifier id belongs to the matching session as long as: (a) out-of-order delivery did not exceed the tolerance parameters $(\mathtt{past}, \mathtt{fut})$ and (b) sender and receiver continuously updated their states compatibly with equal input keys k_i and honestly forwarded verification keys vk_i via $(st_S, vk_i) \leftarrow_\$ \mathrm{AW.up_S}(st_S, k_i)$ and $st_R \leftarrow \mathrm{AW.up_R}(st_R, id, k_i, vk_i)$, respectively.

Formally, this is captured by game CORR in Appendix A.1, Figure 4 for which we require that $\Pr[\mathrm{CORR}_{\mathrm{AW},\mathtt{past},\mathtt{fut}}(\mathcal{A}) = 1] = 0$ for all adversaries $\mathcal{A}$.

[1] $\mathtt{past} = 2,000$ and $\mathtt{fut} = 25,000$ in lines 8 and 9 in https://github.com/signalapp/libsignal/blob/525e8bce0c84576014268ab5b3982612b765f598/rust/protocol/src/consts.rs

3.1 Game-based Anonymity

To model anonymity, we define game $\text{IND}^b_{\text{AW,past,fut}}$ with challenge bit $b \in \{0,1\}$ in Figure 1 that is parameterized with the receivable window via (past, fut) and proceeds in two phases: Initially, adversary $\mathcal{A}$ specifies the number of receivers $n \in \mathbb{N}$ in the *challenge session* with the single sender as well as the local identifiers id_i^* with which each receiver $i \in [n]$ refers to that joint challenge session (see line 00). The adversary acts as the senders for all remaining non-challenge sessions with each receiver. Since all fresh input update keys are sampled uniformly at random, a standard hybrid argument proves that this single-sender, single-session game generalizes to multi-user, multi-session security; for this, note that (A)AW does not use or rely on any long-term key material that would be shared between sessions. Moreover, existing literature [BFWW11, FG14, BSJ+17] covers generic composition with (continuous) key exchange protocols that provide the fresh, independent update keys.

After this challenge session initialization step, $\mathcal{A}$ is invoked with its ephemeral state ω to continue with querying the following oracles:

- $\text{Up}_S(k)$: updates the sender state with key k or, if input $k = \bot$, with a randomly sampled key (line 12); if $k \neq \bot$ and this is either the initial epoch or the previous epoch was marked *trivially de-anonymized* (see below), this new epoch is marked *trivially de-anonymized*, too (lines 10-11).
  Variables. ep_S: sender's current epoch; s: send operations in that epoch; X: de-anonymized send operations due to exposures; K: sender's update keys; VK: sender's verification keys.
- $\text{Up}_R(i, id, k, vk)$: updates session id in the state of receiver i with verification key vk and key k or, for empty input $k = \bot$ in *challenge session* $id = id_i^*$, key k is synchronized with the sender by taking their update key from this epoch (line 27); if the keys of the sender and receiver i are not synchronized, i is marked *out of sync* (lines 25-26).
  Variables. n: number of possible receivers in challenge session; id_i^*: receiver i's local identifier for challenge session; ep_{R_i}: receiver i's current epoch in challenge session; oos_i: epoch in which receiver i went out of sync in challenge session.
- $\text{Enc}(m, ch)$: anonymizes message m and outputs resulting real ciphertext c or, if $ch = 1$ and $b = 1$, a random challenge ciphertext $c^* \leftarrow_\$ C$ instead (lines 33-34).
  Variables. RC: real ciphertexts; CC: output challenge ciphertexts (real or random); CH: challenges.
- $\text{Dec}(i, c)$: de-anonymizes ciphertext c and outputs identifier id and message m; if c was a random challenge ciphertext output by Enc, the underlying hidden real ciphertext is de-anonymized instead (line 38). If c is considered *receivable* based on parameters (past, fut), it is marked received (lines 40-41). Variable. R_i: ciphertexts received by i.
- Expose_S: outputs sender state st_S and marks all subsequent ciphertexts in the current epoch *trivially de-anonymized*.
- $\text{Expose}_R(i)$: outputs state st_{Ri} of receiver i and marks all ciphertexts in the current or prior epochs *trivially de-anonymized* that were both not yet received by i and sent before i was marked *out of sync*.

The adversary terminates with a guess b' and wins if $b = b'$, unless a challenge query to oracle $\text{Enc}(m, ch = 1)$ was marked *trivially de-anonymized*. Based on this, we define the advantage of adversary $\mathcal{A}$ as $\text{Adv}^{\text{ind}}_{\text{AW}}(\mathcal{A}) = |\Pr[\text{IND}^0_{\text{AW}}(\mathcal{A}) = 1] - \Pr[\text{IND}^1_{\text{AW}}(\mathcal{A}) = 1]|$.

Game $\text{IND}^b_{\text{AW,past,fut}}(\mathcal{A})$
00 $(\omega, n, (id_i^*)_{i\in[n]}) \leftarrow_\$ \mathcal{A}()$
01 $(ep_S, (ep_{R_i})_{i\in[n]}) \leftarrow 0^{1+n}$
02 $(s, (oos_i)_{i\in[n]}) \leftarrow \infty^{1+n}$
03 $(K[\cdot], VK[\cdot], RC[\cdot], CC[\cdot]) \leftarrow \bot^4$
04 $((R_i)_{i\in[n]}, X, CH) \leftarrow \emptyset^{2+n}$
05 $(st_S, (st_{Ri})_{i\in[n]}) \leftarrow \bot^{1+n}$
06 $b' \leftarrow_\$ \mathcal{A}(\omega)$
07 · Require $CH \cap X = \emptyset$
08 Stop with b'

Oracle $\text{Up}_S(k)$
09 Require $k \in \mathcal{K} \cup \{\bot\}$
10 · If $k \neq \bot \wedge (ep_S{=}0 \vee (ep_S, s) \in X)$:
11 · $X \xleftarrow{\cup} \{(ep_S + 1, s') \mid s' \in \mathbb{N}\}$
12 If $k = \bot$: $k \leftarrow_\$ \mathcal{K}$
13 $ep_S \leftarrow ep_S + 1$; $s \leftarrow 0$
14 $(st_S, vk) \leftarrow_\$ \text{AW.up}_S(st_S, k)$
15 $K[ep_S] \leftarrow k$; $VK[ep_S] \leftarrow vk$
16 Return vk

Oracle Expose_S
17 · $X \xleftarrow{\cup} \{(ep_S, s') \mid s' > s\}$
18 Return st_S

Oracle $\text{Expose}_R(i)$
19 · $X \xleftarrow{\cup} \{(ep', s') \notin R_i \mid ep' \leq \min(ep_{R_i}, oos_i - 1)\}$
20 Return st_{Ri}

Oracle $\text{Up}_R(i, id, k, vk)$
21 Require $i \in [n] \wedge k \in \mathcal{K} \cup \{\bot\}$
22 If $id = id_i^* \wedge ep_{R_i} < oos_i$:
23 Require $K[ep_{R_i} + 1] \neq \bot$
24 $ep_{R_i} \leftarrow ep_{R_i} + 1$
25 · If $k \notin \{K[ep_{R_i}], \bot\}$ $\vee\ vk \neq VK[ep_{R_i}] \neq \diamond$:
26 · $oos_i \leftarrow \min(ep_{R_i}, oos_i)$
27 Else if $ep_{R_i} < oos_i$: $k \leftarrow K[ep_{R_i}]$
28 $st_{Ri} \leftarrow \text{AW.up}_R(st_{Ri}, id, k, vk)$
29 Return

Oracle $\text{Enc}(m, ch)$
30 Require $s < \infty$
31 $(st_S, c) \leftarrow_\$ \text{AW.enc}_S(st_S, m)$
32 $s \leftarrow s + 1$; $RC[ep_S, s] \leftarrow c$; $c^* \leftarrow c$
33 · If ch:
34 · If $b = 1$: $c^* \leftarrow_\$ C$
35 · $CH \xleftarrow{\cup} \{(ep_S, s)\}$
36 $CC[c^*] \leftarrow c$
37 Return c^*

Oracle $\text{Dec}(i, c)$
38 If $CC[c] \neq \bot$: $c \leftarrow CC[c]$
39 $(st_{Ri}, id, m) \leftarrow \text{AW.dec}_R(st_{Ri}, c)$
40 · If $\exists (ep', s') \in Rcvbl(R_i, RC, ep_{R_i}, \text{past}, \text{fut}) : RC[ep', s'] = c \wedge ep_{R_i} < oos_i$:
41 · $R_i \xleftarrow{\cup} \{(ep', s')\}$
42 Return (id, m)

Figure 1: Multi-receiver games IND for AW scheme AW. Helper Procedure *Rcvbl* is provided in Figure 4. Teal code is only relevant for AAW with sender authentication.

Intuitive Security Requirements. The game described above intuitively captures *Forward Anonymity* and *Post-Compromise Anonymity*: every ciphertext c must look random, unless the sender state was exposed in the current epoch before c was sent or a receiver state in the current (or later) epoch was exposed before c was received. Ciphertexts sent and received before an exposure or in earlier or later epochs are required to remain anonymous. Furthermore, we require that differing update keys between sender and receiver drive the parties out of sync such that subsequent receiver exposures become harmless.

Limitations of Definition. While the above definition requires that ciphertexts on the wire look random, it does not require that metadata in the local sender and receiver states are avoided or hidden. Concretely, an example construction with pseudo-random ciphertexts would be secure with respect to the game in Figure 1 even if a receiver stores a list of sequence numbers and timestamps for all their received ciphertexts in each of their sessions in their local state. Thus, an adversary who exposes this state learns the receiver's social network as well as the communication patterns therein. Since we aim for anonymity under state exposures, we develop a complementary, slightly more complex UC-based definition in Section 5 that captures that metadata is also avoided or hidden

in sender and receiver states. Our construction in Section 4 takes both definitions into account.

We see it as an advantage that we have two complementary definitions for a broad sense of anonymity. While we do not see a clean approach for extending our game-based definition to capture metadata hiding in the user state[2], our special UC-based definition is already very sophisticated and complex. Thus, the above compact game captures the core requirements of AW protocols and our UC-based definition can be viewed as a meaningful extension.

3.2 Game-based Authenticity

Besides anonymity, we require that an Anonymity Wrapper (AW) offers authenticity. The basic requirement is that decryption only succeeds for honestly sent ciphertexts, unless a state exposure of *either of the session members* allows for a trivial ciphertext forgery. With stronger sender authenticity for an *Authenticated* Anonymity Wrapper (AAW), we require that decryption only succeeds for honestly sent ciphertexts, even if any of the session's receiver states were ever exposed; thus, with this second notion, only state exposures of *the sender* allow for trivial ciphertext forgeries.

More concretely, we define game $\text{AUTH}_{\text{AW},\texttt{past},\texttt{fut}}$ that, again, proceeds in two phases: initially, adversary $\mathcal{A}$ specifies the number of receivers $n \in \mathbb{N}$ in the *challenge session* with the single sender as well as the local identifiers id_i^* with which each receiver $i \in [n]$ refers to that joint challenge session. After this initial step, $\mathcal{A}$ can query the following oracles; we use the terms *receivable* as well as *symmetrically* and *asymmetrically insecure* resp. *ineffective*, which we will elaborate on below:

- $\text{Up}_S(k)$: updates the sender state with key k or, if input $k = \bot$, with a randomly sampled key; if $k \neq \bot$, this update is marked *symmetrically ineffective*.
- $\text{Up}_R(i, id, k, vk)$: updates session id in the state of receiver i with verification key vk and key k or, for empty input $k = \bot$ in *challenge session* $id = id_i^*$, key k is synchronized with the sender by taking their update key from this epoch; if the keys of sender and receiver i are not synchronized, i is marked *out of sync*; moreover, for AAW, this epoch is marked *asymmetrically insecure* if the input vk was not honestly forwarded from the corresponding sender update.
- $\text{Enc}(m)$: anonymizes message m and outputs resulting ciphertext c.
- $\text{Dec}(i, c)$: de-anonymizes ciphertext c and outputs resulting identifier id and message m; if c is successfully decrypted in the challenge session id_i^*, but c is not considered *receivable* therein (because it was either received already or never honestly sent) while all receivable ciphertexts are marked *symmetrically secure* or, for AAW, *asymmetrically secure*, the adversary wins due to this non-trivial ciphertext forgery of c.
- Expose_S: outputs sender state st_S and marks all subsequent ciphertexts in the current epoch *symmetrically insecure*; moreover, for AAW, this epoch is marked *asymmetrically insecure*.
- $\text{Expose}_R(i)$: outputs state st_{Ri} of receiver i and marks all ciphertexts in the current or prior epochs *symmetrically insecure* that were not yet received by i.

Receivable and Synchronicity. Ciphertexts are marked *receivable* if they are covered by the out-of-order tolerance window, which is specified by parameters (past, fut). A ciphertext that is not considered receivable should not be decrypted successfully, unless some (not yet received) receivable ciphertexts are marked *insecure*; in the latter case, a ciphertext forgery is trivial.

Symmetrically Insecure and Ineffective. A ciphertext is marked *symmetrically insecure* if it is receivable by the receiver state when the latter is exposed, or if it is sent after the sender was exposed in the same epoch. Symmetric insecurity is inherited to a subsequent epoch if the next sender update takes a key that was chosen by the adversary; such an update is called *symmetrically ineffective*.

Asymmetrically Insecure. With sender authentication in AAW, epochs are marked *asymmetrically insecure* if the sender was exposed therein or if the update of the respective receiver was conducted with an adversarially chosen verification key.

Based on winning the game by querying oracle Dec with a non-trivial ciphertext forgery, we define the advantage of adversary $\mathcal{A}$ as $\text{Adv}^{\text{auth}}_{\text{AW}}(\mathcal{A}) = \Pr[\text{AUTH}_{\text{AW}}(\mathcal{A}) = 1]$. For space reasons, the pseudo-code specification of game AUTH_{AW} is deferred to Appendix A.2, Figure 5.

4 Construction

The goal of the construction is to create anonymous ciphertexts and reduce metadata in sender and receiver states. For the ciphertext processing, we adapt the construction from Bienstock et al. [BRT23] from two-party communication in mesh networks to generic group chats. As the state structure in [BRT23] and other existing protocols, such as [PM16a], leak information about past communication, we present a state structure which improves state anonymity. In the following, we first describe the contruction's message and ciphertext processing and sketch out how our construction satisfies the anonymity and authenticity definitions (the full proofs are in Appendix C). Then, we present the state management. We show a formal definition of state anonymity in Section 5.1. The full pseudocode of our construction is in Figure 2 and an illustration explaining receiver state operations is in Figure 3.

Message Flow and Ciphertext Processing. The general idea is that sender and receiver(s) initialize their states with a shared key k to derive message keys and tags. After a sender encrypts a message, they attach the tag and send it to the (one or more) receivers who find the corresponding decryption key in their states through the tag. Since all parties now share a symmetric key, each receiver could thus impersonate the sender. To prevent this, the AAW construction allows the sender to additionally generate a signing key pair (sk, vk) and distribute vk to all receivers.

In more detail, the protocol is as follows. When Alice wants to send a message to the group, she first updates her state with a shared key k to initiate a new epoch. In case Alice wants to sign her ciphertexts, she generates a new signing key pair (sk, vk) (line 04).

[2] In their Appendix B.2, Bienstock et al. [BRT23] provide a convincing explanation for this.

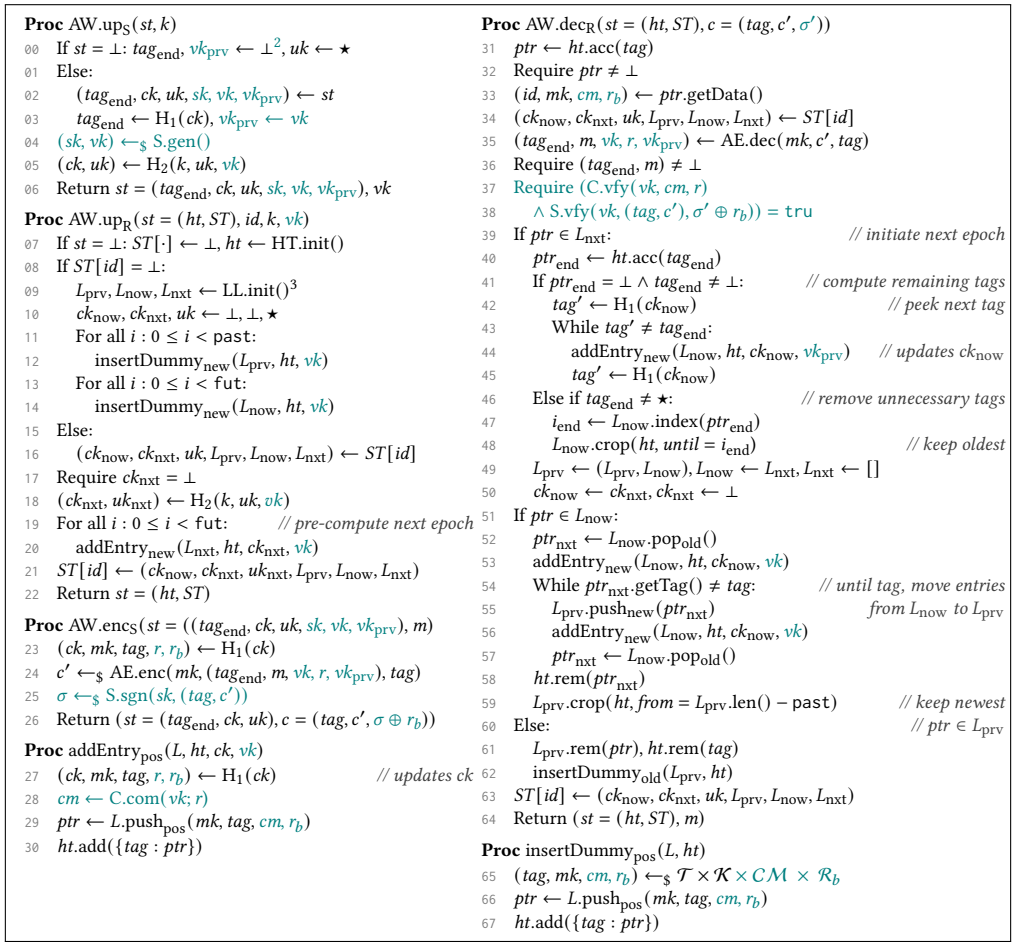

Figure 2: Construction for AW scheme AW. Teal code is only relevant for AAW with sender authentication.

She uses the input k, the update key uk from her state, and potentially the verification key vk, as input to a KDF to obtain the initial chain key ck and the update key for the next epoch (line 05). Similarly, the receiver(s) update their state with k, and potentially vk, and derive the epoch's initial chain key ck and the next epoch's update key uk (line 18). They use ck to pre-compute `fut` tags and key material for processing future ciphertexts sent by Alice (lines 19 and 20). Note that using the secret update key uk as input to the KDF ensures that, even if an adversary chooses the input key k for an update to a new epoch, they do not control the full key material in that new epoch, unless they also know uk from the prior epoch. Moreover, uk ensures that it does not go unnoticed when the receiver state is updated with different key material than the sender: After this point, the receiver will not be able to correctly decrypt any ciphertexts from the sender since the receiver key material diverges irrecoverably from the sender's as soon as the state update uses a dishonest k (or vk).

To send a message, Alice uses ck to derive message key mk (line 23) which she uses for encrypting the message into a ciphertext (line 24), and corresponding tag, which she attaches to the ciphertext. If she signs the ciphertext, she additionally derives a blinding value r_b to encrypt the signature σ (line 26), such that an observer cannot use the plain signature to identify the sender of that ciphertext. Since we only want to hide the signature, our construction uses an efficient one-time-pad-like approach which works with arbitrary signature schemes. Once Bob receives the ciphertext, he uses the tag to look up the corresponding mk (and blinding value for the signature) (lines 31 to 34). We note that Bob is able to immediately decrypt out-of-order ciphertexts since he pre-computes the next `fut` keys and retains up to `past` skipped keys.

Forward and Post-Compromise Anonymity. Our construction offers immediate forward security since both sender and receivers use a key for en-, respectively decrypting, one message and delete it after use. Moreover, the keys do not hold any information about previous keys. Since the security of the states may be restored through an update with fresh key material, our construction offers post-compromise security. Additionally, even if the adversary determines the key given to the update function, the derived chain key is not compromised since the KDF takes as second input the secret update key uk.

Theorem (Informal) 4.1. *Taking a secure AEAD scheme* AE *and random oracles* H_1, H_2*, the AW construction in Figure 2 fulfills the anonymity definition* $\text{IND}_{\text{AW,past,fut}}$ *from Figure 1.*

Proof Sketch for Ciphertext Anonymity. Informally, a ciphertext produced by a sender is anonymous because the tag looks random and the anonymity of the encrypted message can be reduced to the IND$ security of the AEAD scheme. Remaining advantages for an adversary to break anonymity can occur if there are collisions in the KDF, which we model as a random oracle, that are negligible events when the tag and key space is large enough. Moreover, if the sender attaches a signature in plain, an observer who has the per-epoch verification key could check the signature to determine which epoch a ciphertext belongs to. Since we blind the signature with a random value, this functions as a one-time-pad and the signature thus looks anonymous. We show the full proof in Appendix C.1.

Theorem (Informal) 4.2. *Taking secure AEAD scheme* AE*, commitment scheme* C*, signature scheme* S*, and random oracles* H_1, H_2*, the AW construction in Figure 2 fulfills the authenticity definition* $\text{AUTH}_{\text{AW,past,fut}}$ *from Figure 5.*

Proof Sketch for Ciphertext Authenticity. Ciphertexts from the construction are authentic since only the communicating parties know the shared key and the AEAD scheme offers ciphertext unforgeability. Additionally, the AAW construction provides authenticity even when a receiver is exposed who holds the symmetric key material. In this case, the sender's signing key remains secret and the authenticity guarantees can be reduced to the unforgeability of the signature scheme. The full proof for AUTH_{AW} is in Appendix C.2.

State Management. As described in Section 3, the sender Alice knows which session she is communicating with and, for that session, she needs to hold only key material for the next encryption in the state. On the other hand, the receiver Bob does not know to which session an incoming ciphertext belongs and thus has a consolidated state over all sessions. This state holds pre-computed tags and keys for immediate decryption of out-of-order ciphertexts. In case Bob pre-computed unnecessary keys which will never be used by Alice, he should be able to delete these to avoid the memory from growing too much. A straight-forward solution would be to index each key with a message number (e.g., as in [PM16a, BRT23]) and delete key material based on the index. However, such a state structure leaks which ciphertexts were (not) received since the respective index would (not) be missing from the state. In our construction, we thus refrain from using explicit indices and instead use ordered lists to store only the order in which keys were derived. When a receiver uses key material, they delete the entry from the list, and, since tags and keys look random, this construction does not leak which entries are missing, i.e., which ciphertexts were (not) received.

We could use one consolidated ordered list and, after a state update, append the new keys to the existing list. However, this approach is impractical: Once the receiver appended keys from the new epoch, the list alone would not suffice to let the receiver know which keys belong to the previous epoch and can be deleted. To solve this without storing any additional data, we create three separate lists such that the receiver is able to handle pre-computation

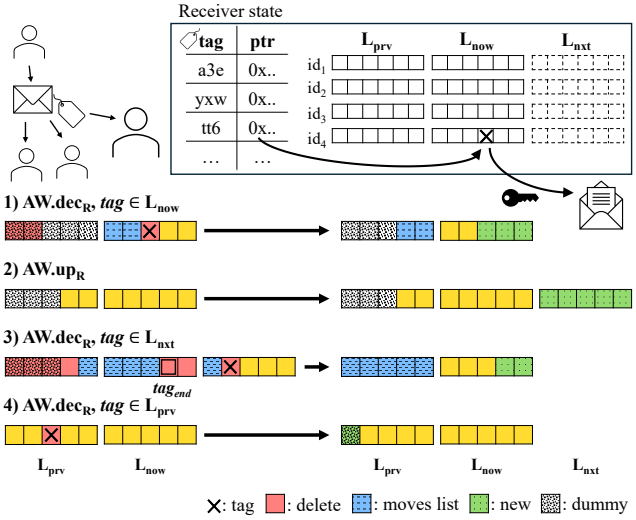

Figure 3: Illustration of the receiver state of our construction from Figure 2. A detailed description is in Section 4.

and deletion of keys. First, the receiver stores one list (L_{now}) which holds the `fut` pre-computed for the current epoch. When a ciphertext arrives (Figure 3, (1)) and the receiver uses a pre-computed key mk_i (line 51), the key mk_{i+1}, which is next in the list (line 54), moves to the beginning of the list, and we pre-compute the necessary keys such that the list has again `fut` keys (lines 52 to 58). If keys from L_{now} are skipped, the receiver keeps them in state to allow for delayed ciphertexts (line 55). For that, the keys move (in order) to the list L_{prv}, potentially evicting older keys (line 59), as L_{prv} has the capacity to store `past` entries. When the receiver updates their state with fresh key material, they fill a new list L_{nxt} with `fut` pre-computed keys for the next epoch (lines 19 and 20; Figure 3, (2)). Then, when the sender sends a ciphertext in this new epoch (i.e., the receiver's list L_{nxt}; Figure 3, (3)), they tell the receiver the tag after the last key they used for the previous epoch (tag_{end}, line 03). The receiver learns that any list entries after (including) tag_{end} will not be needed and they delete the excess entries (lines 46 to 48) or pre-computes necessary tags until tag_{end} (lines 41 to 45). To finalize the state update, the receiver first moves all keys which are older than just used key to L_{prv} and then moves the `fut` pre-computed keys for the new epoch from L_{nxt} to L_{now}.

Additionally, to reduce metadata leakage through list sizes, we enforce constant size for the lists of the receiver. Specifically, L_{now} and (potentially) L_{nxt} always hold `fut` pre-computed entries. To keep the list L_{prv}, which has key material of delayed ciphertexts, of constant size `past`, we use dummies: When initializing the receiver state, we fill L_{prv} with `past` dummies (lines 11 and 12) and L_{now} with `fut` dummies (lines 13 and 14; cf. Figure 3, (1)), and whenever we delete an entry from L_{prv} after decrypting a ciphertext, we fill it up with a dummy (line 62; Figure 3, (4)).

Through the lists, we eliminate explicit connections between a stored key and the epoch it was derived in. However, when using AAW, the receiver needs a verification key vk to verify the signature. Storing vk linked to each message key would reveal which epoch a message key belongs to. It is also not an option to only

send *vk* to the receiver because then, a corruption of the message key would allow an adversary to send a dishonest *vk*. Thus, the receiver instead hides *vk* by pre-computing and storing commitments to *vk* (line 28), where the commitment scheme can be efficiently instantiated by a hash function. That is, for a state update, the receiver obtains *vk* and additionally derives random openings *r* from the chain key which they use to create commitments to the verification key. The receiver then discards the openings and the verification key. The sender derives the same *r* from the chain key and sends (*vk*,*r*) to the receiver (line 24) who then checks if the commitment matches (*vk*,*r*) (line 37). Then, the receiver immediately uses the obtained *vk* to pre-compute the commitments for upcoming decryptions (lines 44, 53 and 56).

AW for Dynamic Groups. Our construction readily extends to dynamic groups. That is, to add a user to the group, the sender takes their current key material and sends it to the user (we provide the construction of this function in the Appendix, Figure 12). The added user then cannot read any prior communication since they have only the keys for future communication. To remove a user from a group, the senders perform a key update without sending the fresh key material to the removed user.

4.1 Generic Wrapper vs. Direct Replacement

AW requires external fresh key material to use as initial chain keys. These keys can be agreed upon by an underlying messaging protocol itself or by some external protocol, enabling AW to operate in two distinct modes: as a generic wrapper around existing protocols or as a direct messaging protocol.

Direct Replacement. When AW update keys are distributed via external mechanisms, AW can directly encrypt plaintext messages to directly provide a secure messaging protocol. For example, AAW update keys can be distributed over a pre-existing secure channel, just as the Sender Key protocol distributes the sender's key material via secure two-party channels. Group members can then use AAW to directly encrypt and decrypt messages yielding a protocol with security properties similar to Sender Key plus FA and PCA.

Generic Wrapper. Many existing messaging protocols already perform Continuous Key Agreement that can serve as a source of AW update keys. For instance, when combining AW and DR, one can extend DR's internal KDF—which derives a new root key and chain key in each epoch from the current root key and a Diffie-Hellman key—to also output an update key for AW. Similarly, post-quantum secure ratchet protocols [Ste24, DJK+25] can produce AW update keys with both DH-based and post-quantum security guarantees. This approach requires updating the interface of secure messaging protocols to optionally return sending and receiving update keys, but the modification is straightforward. Since AW and AAW are generic constructions, their state updates do not need to have the same frequency as the key updates of the underlying wrapped messaging protocol. Given that AW updates are more expensive than regular message encryption, one can develop policies to use only selected update keys and amortize the computational costs over multiple messages. In the particular composition with DR, we acknowledge that AW and the symmetric chain in DR have similar structures, but we expect the improvement from removing redundancy to be relatively modest. This optimization, along with determining optimal update frequencies, presents interesting directions for future work and deployment in applications like Signal.

4.2 Performance

To evaluate the concrete performance of AW and AAW, we implemented both as generic wrappers in Rust as an addition to Signal's official libsignal_protocol crate [Mes25a] and make the implementation available at [Sch25]. We evaluated AW wrapping both Double Ratchet and Sender Keys messages and evaluated AAW wrapping plaintext messages directly, where the update keys input to AW and AAW, respectively, were sampled uniformly at random. A full integration with DR or other messaging protocols, as described in Section 4.1, remains a task for deployment in practice.

AW and AAW were instantiated using 16 byte tags, using HKDF [KE10] for random oracles, and using AES-256-GCM [MV04] for authenticated encryption. For AAW, commitments were 32 bytes of HKDF output and EdDSA [JL17] with the curve Ed25519 [LHT16] was used for signing.

All measurements of Double Ratchet, Sender Keys, and Sealed Sender were performed using Signal's production implementation in libsignal, and timing data was collected using libsignal's benchmarking framework. Experiments were performed on a 32 core Intel® Core™ i9-14900K processor.

Communication Cost. Table 2 shows the size of encrypted messages for Signal's deployed implementations of group and 1:1 messaging alongside AW and AAW based alternatives. AW yields significantly smaller messages for all group sizes and since Signal's Sealed Sender scales linearly in the size of the group, the difference for large groups is dramatic.

		Message Size (bytes)
Group	SK	109
	AW(SK)	157
	AAW	155
	SS(SK)	440 + 68 · group_size
1:1	DR	66
	AW(DR)	114
	SS(DR)	441

Table 2: Size of group and 1:1 messages sent using Signal's deployed techniques—Sender Keys, Sealed Sender, and Double Ratchet—compared with alternatives using AW **and** AAW**.** AW(SK) **and** AW(DR) **denote** AW **wrapping Sender Keys and Double Ratchet messages respectively.** SS(SK) **and** SS(DR) **denote Sealed Sender wrapping Sender Keys and Double Ratchet messages respectively.**

Computation Cost. Table 3 shows the mean time to encrypt and decrypt messages in microseconds for Signal's deployed implementations of group and 1:1 messaging alongside AW and AAW based alternatives. Since Sealed Sender requires a separate encrypted envelope for each recipient, encryption costs grow significantly with the number of recipients. Encryption and decryption are not the

only computation costs for AW and AAW, they must also process updates. The cost of these updates is amortized over the length of an epoch, so a tradeoff between amortized update cost and PCS/PCA is possible. With our implementation, AW had a mean time of 1.5245 ms to update, and AAW had a mean time of 2.7591 ms to update.

	Group Size	2	5	10	100	1000
Group	SK	20.098 (31.201)				
	AW(SK)	21.796 (32.502)				
	AAW	21.079 (32.185)				
	SS(SK)	131.12 (61.286)	283.58 (61.286)	343.41 (61.286)	1,116.4 (61.286)	6,052.5 (61.286)
1:1	DR	2.1979 (2.7178)				
	AW(DR)	3.7412 (4.2402)				
	SS(DR)	61.171 (52.750)				

Table 3: Mean encryption (decryption) times, measured in microseconds, for group and 1:1 messages sent using Signal's deployed techniques—Sender Keys, Sealed Sender, and Double Ratchet—compared with alternatives using AW **and** AAW**.** AW(SK) **and** AW(DR) **denote** AW **wrapping Sender Keys and Double Ratchet messages respectively.** SS(SK) **and** SS(DR) **denote Sealed Sender wrapping Sender Keys and Double Ratchet messages respectively.**

Storage Cost. AW's data structures add directly to the storage costs of existing Double Ratchet and Sender Key sessions. Using `fut` = `past` = 2000, AW adds 96 bytes to each session for sending, 288,096 bytes to each session for the recipient keys and lists (96 bytes for keys and 48 bytes per list entry), and 120,000 bytes per recipient for a compact representation of the tag-indexed hash table (20 bytes per entry). AAW adds 192 bytes to each session for sending 480,096 bytes to each session for the recipient keys and lists (96 bytes for keys and 80 bytes per list entry), and 120,000 bytes per recipient for a compact representation of the tag-indexed hash table (20 bytes per entry). This is a significant cost considering that current double ratchet sessions, even with Signal's upcoming post-quantum additions, are typically less than 5 KB in size.

5 Reducing State Metadata

We designed the state structure of our AW construction such that metadata leakage on state exposures is reduced. Particularly, our construction prevents an adversary from learning about past communication. To model the exact information which an adversary can learn about the state, we present a formal definition for capturing state metadata in this section. Game-based approaches appear unsuitable for defining this: since user states are highly construction-dependent, it is unclear how a game could cleanly model that certain metadata in exposed states can be leaked while other metadata must be hidden from adversaries. A natural alternative would be employing a construction-dependent simulator. For this reason, we use a UC definition [Can01] in which our ideal functionality $\mathcal{F}_{\text{AW}}$ explicitly describes which information a state exposure reveals.[3]

[3]To avoid the *commitment problem* caused by a corruption of a receiver after a secure ciphertext is seen by the adversary, we use a technique of, e.g., [AJM22] and the references therein, that considers a weakened variant of UC-security which only guarantees UC properties for a restricted set of admissible environments that do not trigger the commitment problem.

$\mathcal{F}_{\text{AW}}$ inherits the same protocol syntax as introduced in Section 3. In the following, we first describe the ideal functionality $\mathcal{F}_{\text{AW}}$ in Section 5.1 and argue how our construction realizes $\mathcal{F}_{\text{AW}}$. Lastly, we discuss state metadata leakage of the underlying protocol and possible extensions of AW to protect state anonymity of the underlying protocol.

5.1 Ideal Functionality

The ideal functionality, $\mathcal{F}_{\text{AW}}$, is formally defined in Appendix D in Figures 13 and 14. There we also provide a detailed explanation of $\mathcal{F}_{\text{AW}}$. Here, we provide a high-level overview of the functionality and our design choices for it. This functionality captures multiple senders $S_{i\in[n]}$ and multiple receivers $R_{j\in[m]}$, where each sender S_i communicates with a fixed subset of receivers in a given session.

The functionality allows senders and receivers to update their states as well as send and receive messages, respectively. The functionality keeps track of the abstract components of sender and receiver states during these operations such as current epoch, current chain key stored, current message keys stored by receivers, current dummy keys stored by receivers, etc. This is similar to the way in which the anonymity game of Figure 1 keeps track of the current epoch and index within that epoch for the sender, as well as the current epoch of the receivers and for which (epoch, index) pairs they have already received messages. We model receiving in $\mathcal{F}_{\text{AW}}$ by giving the receiver all messages stored in ciphertexts on the delivery server and intended for them which have not yet been received. Once a ciphertext stored on the delivery server has been received by all intended receivers, it is deleted.

Importantly, $\mathcal{F}_{\text{AW}}$ does not send any information to the adversary when a message is sent or received. Indeed, in order to analyze the anonymity that AW can *cryptographically* achieve, we choose not to model any consequences of *network* observation that can lead to de-anonymization. For example, an all-powerful network adversary that can see from where ciphertexts originate and to where they are sent can already obtain a lot of de-anonymizing information, and necessitates *network routing* anonymization techinques in addition to *cryptographic* anonymization techniques. Instead, we assume idealized channels that reveal nothing about what is sent/received or by/to whom, to focus on the cryptographic aspects.

We do, however, allow the adversary to expose the delivery server, which leaks ciphertexts that have not yet been delivered to all recipients. This models a delivery server complying with a subpoena, a breach of the delivery server, etc. Hence, any ciphertext that the adversary sees through such an exposure should look completely random and anonymous, except for those for which it has obtained the corresponding key material through a party exposure (see below). Now we explain the other effects of adversarial exposure of any sender or receiver. Such exposures leak the abstract chain keys stored by senders, and thus any chain/message key which can be derived by them, as well as the chain key, stored message keys, and dummy keys stored by receivers. Thus, any ciphertext that the adversary sees through an exposure of the delivery server is compromised if the corresponding key is ever obtained through a party exposure. Through subsequent leaks of parties in the same session, the adversary can also learn how newly exposed

chain keys can be derived from previously exposed chain keys. Furthermore, upon exposures of the delivery server, the adversary can also learn how any previously exposed chain keys can be used to derive the message keys for those ciphertexts. However, it is important to observe that for a given exposed message key, if no chain key that can be used to derive it is ever exposed, then it should look completely random to the adversary and indistinguishable from dummy keys in receiver states.

Finally, the adversary can take some actions that correspond to network drops and malicious behavior of the delivery server. For instance, the adversary can instruct $\mathcal{F}_{\text{AW}}$ to randomly drop some proportion of ciphertexts that are still on the delivery server. Additionally, the adversary can instruct $\mathcal{F}_{\text{AW}}$ to drop some ciphertexts that it saw during an exposure of the delivery server, as well as change the underlying message of such a ciphertext, if it has the necessary key material to do so.

5.2 Our AW Construction

Informally, our AW construction realizes $\mathcal{F}_{\text{AW}}$ as follows. The sender only stores key material needed immediately for the next encryption which does not reveal information about any previous communication; a key exposure only compromises future communication in the epoch. Moreover, tag_{end} is derived only after the final encryption in an epoch, and thus never attached to a ciphertext. It can only be used to determine the length of the previous epoch if a chain key in the previous epoch was exposed. The receiver deletes tag_{end} from their state (if it was pre-computed) upon initiation of a new epoch and an adversary thus cannot use tag_{end} to determine epoch ranges in L_{prv}.

Further, the receiver state hides information about (past and ongoing) communication as follows. Entries in the lists and hash table are message keys, tags, and random blinding values, which are independent of each other and cannot be used to infer previous or future values. Additionally, in AAW, the receiver does not store verification keys in plaintext but commitments which hides the keys such that an adversary cannot associate entries to epochs. Only when a chain key is exposed, the adversary can derive all upcoming entries and identify entries from that epoch and potential gaps in the sequence of list entries. Otherwise, the adversary can neither tell which entries in L_{prv} belong to which epoch, nor how many entries were not yet received since the list has constant size and, if necessary, is filled with dummies that are indistinguishable from real entries. From L_{now} (and L_{nxt}), the adversary only learns which tags belong to the current epoch (and next epoch) and are expected to be received.

5.3 State Anonymity of Underlying Protocol

The AW uses key material established through an underlying protocol to anonymize that protocol's ciphertexts. While the state of the AW is carefully designed to reduce metadata, the underlying protocol may have an arbitrary state structure. Thus, fully compromising a party can still reveal the information which AW hides because of the underlying protocol's state. Consider, for instance, Signal's Double Ratchet: The pre-computed keys in the receiver state are indexed with sequence numbers which AW explicitly omits. To avoid metadata leakage through the underlying protocol, we can use AW to also anonymize the state. The construction idea is that sender and receiver share a symmetric key which the receiver uses to encrypt their underlying protocol's state. After the receiver encrypted the state, they discard the key. The sender includes the key in each ciphertext such that the receiver can decrypt their state temporarily, encrypt it again, discard the key, and so on. If the adversary exposes the receiver, the receiver state remains anonymous. However, the adversary still obtains the message key from AW to decrypt the next ciphertext and thereby obtain the state key. To avoid these weak anonymity guarantees, the parties could make use of the potential state granularity of the underlying protocol and use different keys to encrypt different parts of the state. If there is a high granularity, the parties could derive a state encryption key for every derived message key and link the tags to their respective memory section. Such a construction would intuitively inherit FA and PCA guarantees from AW.

Finally, we note again that using (A)AW as the messaging protocol itself (e.g., when replacing Sender Keys with AAW), our state anonymity guarantees hold directly.

6 Trade-Offs and Optimizations

While the computation and communication overhead of AW clearly outperforms the Signal Sealed Sender protocol, the size of the receiver state in AW is worse. In the following, we present possible optimization strategies. It is important to note that the parameters influencing these optimizations depend on the specific application. Relevant factors include, e.g., the available memory, the degree of privacy concerns, and the likelihood of receiving ciphertexts out of order. Therefore, we only present general concepts and leave the task of identifying optimal configurations and evaluating concrete instantiations to the respective use case.

6.1 Efficient Data Structures

We begin to reduce the memory of the lists L_{now} and L_{nxt}. These lists contain `fut` pre-computed keys, tags, and potentially commitments to verification keys to allow for immediate decryption. In the case that delays and drops (especially of many consecutive) ciphertexts are rare, a large part of the pre-computed values is usually not needed for receiving the next ciphertext. Decreasing `fut` is still not an option in order to maintain functionality with immediate decryption even in the anticipated worst case.

Instead, we propose that the receiver stores full values, including entries in the hash table, only for the next x keys which are most likely to be received. That is, if the last decryption used the key mk_i (with an entry in L_{now} or L_{nxt}), we assume that it is most likely that the next received ciphertext corresponds to a key $mk_j, i < j \leq i+x$. For the remaining $\texttt{fut}-x$ values, the receiver does not store the full entry but only information to decide whether an incoming ciphertext is receivable, i.e., has key material within this range. This can be achieved, e.g., through a Counting Bloom Filter (CBF) (cf. Appendix B.4) as follows. When initializing their state, the receiver first pre-computes the initial x values, as described in the AW construction (Section 4), and then stores the resulting chain key ck_x for the corresponding session. Then, they continue deriving the upcoming $\texttt{fut} - x$ tags and add (only) the tags to the CBF, also storing the resulting chain key $ck_{\texttt{fut}}$.

When a ciphertext arrives, the receiver first looks up the tag in the hash table which spans all sessions. If they find the tag in the hash table, they proceed as in the AW construction. Otherwise, the tag is not in hash table, in which case the receiver goes through the CBFs of all sessions and checks whether the tag is in one of them. If a CBF from some session contains the tag, the receiver loads the corresponding chain key ck_x and derives entries until arriving at the entry which corresponds to the tag (until at most $\texttt{fut} - x$ steps). Note that, if the match was a false positive, this would result in one wasted iteration over the $\texttt{fut} - x$ tags. Otherwise, the receiver arrived at the corresponding chain key ck_i, uses the matching mk_i to decrypt the ciphertext, and moves all skipped entries to the list L_{prv}. If the decryption was successful, the receiver then updates their state, starting from the chain key ck_i which corresponds to the received tag, as follows: (1) they re-fill the fully pre-computed values such that they store full entries for the entries which correspond to the next x chain keys $ck_j, i < j \leq i + x$, (2) they delete all tags for chain keys $ck_j, j < i + x$ from the CBF, (3) they add all tags for chain keys $ck_j, i + x < j \leq i + \texttt{fut}$ to the CBF.

Verification Key Commitments in the CBF. One problem arises in the CBF approach when implementing AAW which concerns the commitments to the sender's verification key. Recall that the receiver stores commitments to the sender's verification key such that a state exposure does not reveal which epoch a key belongs to. When a ciphertext arrives, the receiver uses the opening, which is included in the message, to decrypt the verification key and check the signature. Only the decrypted verification key allows the receiver to precompute new commitments such that the signature verification of the following $\texttt{fut}$ entries is possible. Now, looking at the new construction, when a ciphertext arrives whose tag is in the CBF, the tag is the only information that the receiver stores for this incoming ciphertext. That means, the receiver neither has the corresponding commitment to the verification key, nor can it recover the verification key from entries which are not in the Bloom filter, as the receiver does not store the corresponding opening (but only obtains it through a message from the sender).

We note that hiding the epoch for precomputed keys and tags is only meaningful in the list of previously skipped keys L_{prv}. All precomputed information in the lists L_{now} and L_{prv} necessarily belongs to two consecutive epochs, respectively. For this reason, we can store the verification keys for the current and next epoch in plaintext without using the commitment approach. Only for entries in the list L_{prv}, we hide the verifications keys via commitments to conceal the entries' epochs. For computing the commitments for L_{prv}, we propose the following adaptation.

Instead of storing commitments of verification keys for the current (and next) epoch, the receiver stores the verification keys in plaintext together with the openings for the commitments. More specifically, the first x entries of lists L_{now} and L_{nxt} in the receiver state store the openings r. That means, whenever the receiver derives an opening, i.e., when computing the values for the x entries in the hash table, the receiver does not immediately compute the commitment, but instead stores the opening. Then, when moving an entry from L_{now} to the list of older keys L_{prv}, the receiver uses the stored opening to compute the commitment to the corresponding epoch's verification key. When the receiver decrypts a ciphertext from a new epoch $i + 1$, they move the necessary entries from L_{now} to L_{prv}, thereby computing the remaining commitments. Then, they discard the verification key for the epoch i.

Impact on Provable Security. Using a CBF for the updated state and storing the verification key for the current and next epoch in plaintext is provably secure and falls within our anonymity models (both for Section 3 and Section 5.1). First, the CBF in this construction functions exactly like the hashtable from the original state. The CBF is thus simply a more compact representation used to decide whether an incoming tag is receivable. Concretely, the adversary can only interact with the CBF by submitting decryption queries. Per our construction, the decryption only returns messages if the tag was a true positive. In all other cases, the receiver does not have the key material, the decryption fails, and the state remains unchanged. Thus, the CBF is not affected by adversarial queries (e.g., as studied in [FPUV22, MFS23]).

Second, storing the verification key in plaintext for the current (and next) list does not give the adversary any more information on state exposure as before. That is because the verification key alone does not reveal any information about the sender's identity. Moreover, the adversary, by the structure of the receiver state, already knows that all entries in the current list L_{now} (and future list L_{nxt}) belong to the same epoch. Lastly, knowing the verification key only allows the adversary to deanonymize ciphertexts (by checking the signatures) if they also know the decryption key for the signature. Otherwise, the signature is encrypted and the verification key is of no use for the adversary. Additionally, observe that the ciphertexts whose verification key is leaked on exposure are exactly a subset of the ciphertext whose decryption keys are leaked. That means, the exposed verification key does not enable the adversary to deanonymize any (prior or future) ciphertexts beyond what can already be deanonymized with the decryption keys alone.

Estimated Savings. Depending on the chosen parameters for the CBF, this approach can significantly reduce the storage while adding relatively small computational overhead as operations on the CBF are mainly inexpensive hash operations. We support this claim by sketching out an exemplary performance calculation in the following.

Storage Costs. The following calculation applies to both L_{now} and L_{nxt}, assuming they have the same requirements. Assume we want to offer immediate decryption for the next $\texttt{fut} = 2000$ ciphertexts, and we fully pre-compute $x = 100$ of them, which means that our CBF stores $n = 1900$ tags. We allow for false positive rate of $p = 0.01$. Let n_s be the number of sessions in which the receiver is. Using the equations from [Blo70], we calculate the optimal number of entries m and the optimal number of hash functions k as

$$m = -\frac{n \cdot \ln p}{(\ln 2)^2} \approx 18,200 \qquad \text{and} \qquad k = \lceil \frac{m}{n} \cdot \ln 2 \rceil = 7.$$

If all of the 1900 tags map to the same index, a counter would need to be able to store $\lceil \log_2 1900 \rceil = 11$ bits. However, it is very unlikely that each tag maps to the same entry, and thus, we choose to go down to the practical size of 8 bits.[4] The overall storage

[4] To be fully certain that overflows are avoided, one could choose to use 11-bit counters, or the next practical chunk size of 16-bit counters, which would increase the resulting storage by 2.

cost for the CBF is then $8 \cdot 18,200$ bits = 18,200 bytes for each list (L_{now} and L_{prv}) times the number of sessions. With an efficient instantiation of CBF (e.g., [BMP+06]), we can reduce this space by a factor of 2, resulting in 18,200 bytes per session in addition to storing two lists of 100 full values (which include list and hash table entries). To estimate the storage cost of the 100 full values for both lists, we use the values reported in Section 4.2. For AW, we calculate $2 \cdot (96 + 100 \cdot (48 + 20)) = 13,792$ bytes; and for AAW $2 \cdot (96 + 100 \cdot (80 + 20)) = 20,192$ bytes. This results in an overall storage of $18,200 + 13,792 = 31,992$ bytes for AW and $18,200 + 20,192 = 38,392$ bytes for AAW. Additionally, for AAW, we additionally store two 32-byte verification keys; storing the openings for the commitments instead of the commitments themselves does not change the occupied storage. Comparing this with the original construction, the new approach results in around 7.8% for AW and 6.4% for AAW of the required storage size.

Computational Costs. Each full entry (in the hash table and a list) is computed once—either for immediate use or for storage in L_{now} or L_{prv}. The CBF adds one insertion and one deletion per tag, totaling $2k$ hash operations. In the (assumed rare) case of many dropped or re-ordered ciphertexts, i.e., when a tag is missing from the hash table, the receiver performs a CBF lookup: this involves k hash operations and checking k entries per CBF. In the worst case, all session CBFs must be checked, yielding $O(k \cdot n_s)$ lookups, matching the cost of a false positive. Thus, the total overhead per tag is $2k+1$ hashes and up to $O(k \cdot n_s)$ lookups. As both hashes and lookups are cheap, and CBF lookups are rare, the added overhead is low, making a CBF-based approach a practical alternative to the original design.

Optimizations. The construction can be improved using efficient CBF variants [RKK14, PRM16]. If out-of-order delivery is infrequent, reducing x and storing more items in the CBF can lower storage needs. Accepting a higher false positive rate can also save space. Similarly, reducing counter size from 8 to 4 bits halves storage if overflows are unlikely. Additionally, going through the CBFs by relevance of the sessions would decrease the time it takes to find a match. Enhancing parameters depends on specific use cases and requires insight into network and application behavior, making this an interesting optimization problem for tailored deployments.

6.2 Omitting Dummy Keys

The receiver adds dummy values to their state during initialization or when using keys from L_{prv}. This keeps L_{prv} at a constant size and prevents leakage of past communication patterns. When ciphertexts arrive mostly in order, few keys are skipped, so most entries in L_{prv} are dummies and not functionally needed. Omitting these dummies can save up to `past` entries, which, based on our evaluation in Section 4.2, reduces storage by up to a third—around 136,000 bytes for AW and 200,000 bytes for AAW per session.

Impact on Provable Security. Since the dummy keys are never used for encryption, this change does impact the game-based anonymity guarantees from Section 3. Regarding the model for state anonymity (Section 5.1), there is only a minor change to the anonymity guarantees: Removing the dummies allows the adversary to learn the number of not received old ciphertexts but nothing else about the past communication. Therefore, this (informal) trade-off for anonymity can be a reasonable optimization option.

Acknowledgments

We would like to thank Benedikt Auerbach and Guillermo Pascual-Perez for initial and ongoing discussions about related questions. We are also thankful for the feedback from the anonymous reviewers.

Lea Thiemt was supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Research and Training Group 2475 “Cybercrime and Forensic Computing” (grant number 393541319/GRK2475/2-2024).

References

[Abe99] Masayuki Abe. Mix-networks on permutation networks. In Kwok-Yan Lam, Eiji Okamoto, and Chaoping Xing, editors, *Advances in Cryptology - ASIACRYPT'99*, pages 258–273, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[AJM22] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 34–68. Springer, Cham, August 2022.

[BBR+23] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023.

[BCG23] David Balbás, Daniel Collins, and Phillip Gajland. WhatsUpp with sender keys? Analysis, improvements and security proofs. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part V*, volume 14442 of *LNCS*, pages 307–341. Springer, Singapore, December 2023.

[BFWW11] Christina Brzuska, Marc Fischlin, Bogdan Warinschi, and Stephen C. Williams. Composability of Bellare-Rogaway key exchange protocols. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011*, pages 51–62. ACM Press, October 2011.

[BH23] Eric Brigham and Nicholas Hopper. Poster: No safety in numbers: traffic analysis of sealed-sender groups in signal. *CoRR*, abs/2305.09799, 2023.

[BHMS16] Colin Boyd, Britta Hale, Stig Frode Mjølsnes, and Douglas Stebila. From stateless to stateful: Generic authentication and authenticated encryption constructions with application to TLS. In Kazue Sako, editor, *CT-RSA 2016*, volume 9610 of *LNCS*, pages 55–71. Springer, Cham, February / March 2016.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[BMP+06] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In Yossi Azar and Thomas Erlebach, editors, *Algorithms – ESA 2006*, pages 684–695, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[BRT23] Alexander Bienstock, Paul Rösler, and Yi Tang. ASMesh: Anonymous and secure messaging in mesh networks using stronger, anonymous double ratchet. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 1–15. ACM Press, November 2023.

[BRV20] Fatih Balli, Paul Rösler, and Serge Vaudenay. Determining the core primitive for optimally secure ratcheting. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 621–650. Springer, Cham, December 2020.

[BSJ+17] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 619–650. Springer, Cham, August 2017.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, November 1998.

[CR25] Daniel Collins and Paul Rösler. GURKE: Group unidirectional ratcheted key exchange. In *Advances in Cryptology - CRYPTO 2025*, 2025.

[DHRR22] Benjamin Dowling, Eduard Hauck, Doreen Riepel, and Paul Rösler. Strongly anonymous ratcheted key exchange. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part III*, volume 13793 of *LNCS*, pages 119–150. Springer, Cham, December 2022.

[DJK+25] Yevgeniy Dodis, Daniel Jost, Shuichi Katsumata, Thomas Prest, and Rolfe Schmidt. Triple ratchet: A bandwidth efficient hybrid-secure signal protocol. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology - EUROCRYPT 2025*, 2025.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In Matt Blaze, editor, *USENIX Security 2004*, pages 303–320. USENIX Association, August 2004.

[DRS20] Benjamin Dowling, Paul Rösler, and Jörg Schwenk. Flexible authenticated and confidential channel establishment (fACCE): Analyzing the noise protocol framework. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part I*, volume 12110 of *LNCS*, pages 341–373. Springer, Cham, May 2020.

[FG14] Marc Fischlin and Felix Günther. Multi-stage key exchange and the case of Google's QUIC protocol. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 1193–1204. ACM Press, November 2014.

[FGJ24] Marc Fischlin, Felix Günther, and Christian Janson. Robust channels: Handling unreliable networks in the record layers of QUIC and DTLS 1.3. *Journal of Cryptology*, 37(2):9, April 2024.

[FPUV22] Mia Filic, Kenneth G. Paterson, Anupama Unnikrishnan, and Fernando Virdia. Adversarial correctness and privacy for probabilistic data structures. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 1037–1050. ACM Press, November 2022.

[HKP22] Keitaro Hashimoto, Shuichi Katsumata, and Thomas Prest. How to hide MetaData in MLS-like secure group messaging: Simple, modular, and post-quantum. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 1399–1412. ACM Press, November 2022.

[JK25] Joseph Jaeger and Akshaya Kumar. Analyzing group chat encryption in MLS, session, signal, and matrix. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology - EUROCRYPT 2025*, 2025.

[JKS24] Joseph Jaeger, Akshaya Kumar, and Igors Stepanovs. Symmetric signcryption and E2EE group messaging in keybase. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part III*, volume 14653 of *LNCS*, pages 283–312. Springer, Cham, May 2024.

[JL17] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017.

[KE10] Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010.

[KK25] Raphael Robert Konrad Kohbrok. Mimi metadata minimalization (mimimi). https://www.ietf.org/id/draft-kohbrok-mimi-metadata-minimalization-02.html, 04 2025.

[KPB03] Tadayoshi Kohno, Adriana Palacio, and John Black. Building secure cryptographic transforms, or how to encrypt and MAC. Cryptology ePrint Archive, Report 2003/177, 2003.

[LHT16] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic Curves for Security. RFC 7748, January 2016.

[LT22] Zeyu Liu and Eran Tromer. Oblivious message retrieval. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 753–783. Springer, Cham, August 2022.

[Lun18] Joshua Lund. Technology preview: Sealed sender for signal. https://signal.org/blog/sealed-sender/, 10 2018.

[Mes25a] Signal Messenger. libsignal. https://github.com/signalapp/libsignal/, 2025.

[Mes25b] Signal Messenger. Securevaluerecovery2. https://github.com/signalapp/SecureValueRecovery2, 2025.

[MFS23] Sam A. Markelon, Mia Filic, and Thomas Shrimpton. Compact frequency estimators in adversarial environments. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 3254–3268. ACM Press, November 2023.

[MKA+21] Ian Martiny, Gabriel Kaptchuk, Adam J. Aviv, Daniel S. Roche, and Eric Wustrow. Improving Signal's sealed sender. In *NDSS 2021*. The Internet Society, February 2021.

[MV04] David A. McGrew and John Viega. The galois/counter mode of operation (gcm). https://csrc.nist.rip/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf, 1 2004.

[Per18] Trevor Perrin. The noise protocol framework. https://noiseprotocol.org/noise.pdf, 07 2018.

[PM16a] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf, 11 2016.

[PM16b] Trevor Perrin and Moxie Marlinspike. The x3dh key agreement protocol. https://signal.org/docs/specifications/x3dh/x3dh.pdf, 11 2016.

[PR18a] Bertram Poettering and Paul Rösler. Asynchronous ratcheted key exchange. Cryptology ePrint Archive, Report 2018/296, 2018.

[PR18b] Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 3–32. Springer, Cham, August 2018.

[PRM16] Salvatore Pontarelli, Pedro Reviriego, and Juan Antonio Maestro. Improving counting bloom filter performance with fingerprints. *Information Processing Letters*, 116(4):304–309, 2016.

[RKK14] Ori Rottenstreich, Yossi Kanizo, and Isaac Keslassy. The variable-increment counting bloom filter. *IEEE/ACM Transactions on Networking*, 22(4):1092–1105, 2014.

[RMS18] Paul Rösler, Christian Mainka, and Jörg Schwenk. More is less: On the end-to-end security of group chats in signal, whatsapp, and threema. In *2018 IEEE EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 415–429. IEEE, 2018.

[RSG98] M.G. Reed, P.F. Syverson, and D.M. Goldschlag. Anonymous connections and onion routing. volume 16, pages 482–494, 1998.

[RSS23] Paul Rösler, Daniel Slamanig, and Christoph Striecks. Unique-path identity based encryption with applications to strongly secure messaging. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 3–34. Springer, Cham, April 2023.

[RZ18] Phillip Rogaway and Yusi Zhang. Simplifying game-based definitions - indistinguishability up to correctness and its application to stateful AE. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 3–32. Springer, Cham, August 2018.

[Sch25] Rolfe Schmidt. Artifact – generic anonymity wrapper for messaging protocols. https://zenodo.org/records/16929590, 2025.

[SJ17] Hua Sun and Syed Ali Jafar. The capacity of private information retrieval. volume 63, pages 4075–4088, 2017.

[Ste24] Douglas Stebila. Security analysis of the imessage PQ3 protocol. https://security.apple.com/assets/files/Security_analysis_of_the_iMessage_PQ3_protocol_Stebila.pdf, 2024.

[SW06] Vitaly Shmatikov and Ming-Hsiu Wang. Timing analysis in low-latency mix networks: Attacks and defenses. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *Computer Security – ESORICS 2006*, pages 18–33, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[TRB+25a] Lea Thiemt, Paul Rösler, Alexander Bienstock, Rolfe Schmidt, and Yevgeniy Dodis. Generic anonymity wrapper for messaging protocols. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS 2025*. ACM, 2025.

[TRB+25b] Lea Thiemt, Paul Rösler, Alexander Bienstock, Rolfe Schmidt, and Yevgeniy Dodis. Generic anonymity wrapper for messaging protocols. Cryptology ePrint Archive, Paper 2025, 2025.

A Supplementary AW Definitions

A.1 Correctness

In Figure 4, we formally define correctness via game CORR in which adversary $\mathcal{A}$ can query oracles Up_S, Up_R, Enc, and Dec to execute the corresponding algorithms AW.up_S, AW.up_R, AW.enc_S, and AW.dec_R. As long as oracles Up_S and Up_R are queried with matching input and output keys k and vk in the challenge session, respectively, it is required that algorithm AW.dec_R outputs *correct* messages m and session identifiers id (see line 36). For this, helper procedure *Rcvbl* computes the set of ciphertexts that are expected to be receivable based on the out of order tolerance parameters (`past`, `fut`). Similar concepts of unreliable delivery in secure channel protocols are covered in [KPB03, BHMS16, RZ18, FGJ24].

A.2 Game-Based Authenticity

Figure 5 captures the game for defining authenticity that is described in Section 3.2. Helper procedure *Expsd* traces symmetrically and asymmetrically insecure ciphertexts.

Encrypt-then-Sign Binding. Our construction of Authenticated AW from Section 4 first anonymizes payload with authenticated encryption and, then, directly signs the resulting ciphertext without, additionally, signing metadata that is already contained in the encrypted payload. Since recent literature on Symmetric Signcryption [JKS24, JK25] may suggest that further binding measures between authenticated encryption and signature is necessary, we

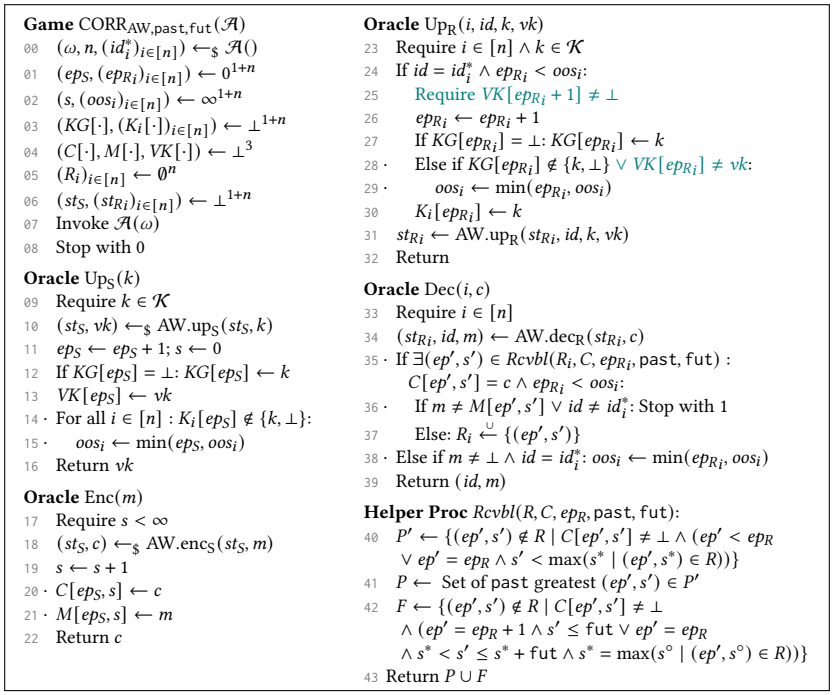

Figure 4: Game CORR **for AW scheme** AW**.**

elaborate on the differences between AAW and Symmetric Signcryption.

Symmetric Signcryption [JKS24, JK25] is modeled as a standalone stateless primitive that offers confidentiality with long-term symmetric keys as well as authenticity with long-term asymmetric keys. For this reason, the security definition of Symmetric Signcryption models that symmetric encryption keys and asymmetric signing keys can both be re-used for multiple messages and across different communication settings; this motivates that, in the security definitions of [JKS24, JK25], encryption keys and signing keys can be corrupted fully independently.

In contrast, whenever the exposure of the sender state in our AAW setting reveals the signing key, corresponding future encryption keys of the same epoch are leaked as well. As a result, our definitions from figures 1 and 5 consider confidentiality of payload hopelessly lost once asymmetric authenticity for that same payload is not achievable anymore. This relation of dependent corruption renders additional binding measures between authenticated encryption and signature in our construction from Figure 2 redundant.

Another difference towards [JKS24, JK25] is that, in AAW, there is only a single sender in each session and that sender only exists for the corresponding group of receivers; this sender is authenticated with its chain of signing keys. The user behind that sender may only become relevant on the higher level application and on the lower level (continuous) key exchange that produces fresh, symmetric input update keys for (A)AW. Thus, the sender, its chain of signing keys, its group of receivers, and the session collapse to the same concept in (A)AW, which obviates the need for binding this information explicitly. Furthermore, our construction is based on probabilistic authenticated encryption and (A)AW has no associated data input. For these reasons, parallel authentication and additional binding measures for encryption nonces or externally input associated data strings are not necessary in our setting.

Nevertheless, we want to emphasize that our (A)AW security definitions require, and our (A)AW constructions securely offer strong authentication. The mentioned techniques developed for Symmetric Signcrytion only become necessary due to the sophisticated corruption model that is motivated by parallel use of the long-term symmetric and asymmetric key material, which does not apply to our setting.

B Cryptographic Primitives

B.1 Authenticated Encryption with Associated Data

The AEAD scheme provides the following three algorithms (AE.gen, AE.enc, AE.dec):

- AE.gen: $() \rightarrow_\$ \mathcal{K}$ Generates a symmetric key $k \in \mathcal{K}$.
- AE.enc: $\mathcal{K} \times \mathcal{M} \times \mathcal{AD} \rightarrow_\$ \mathcal{C}$ Takes a symmetric key k, a message $m \in \mathcal{M}$, and an associated data string $ad \in \mathcal{AD}$ and encrypts the message such that the resulting ciphertext $c \in \mathcal{C}$ and associated data are authenticated.
- AE.dec: $\mathcal{K} \times \mathcal{C} \times \mathcal{AD} \rightarrow \mathcal{M} \cup \{\perp\}$ Takes a symmetric key k, a ciphertext c, and an associated data string ad, and decrypts the message if ciphertext and associated data string are authentic.

Security: Indistinguishability of Ciphertexts from Random under Chosen-Plaintext Attacks. The advantage of an adversary $\mathcal{A}$ against

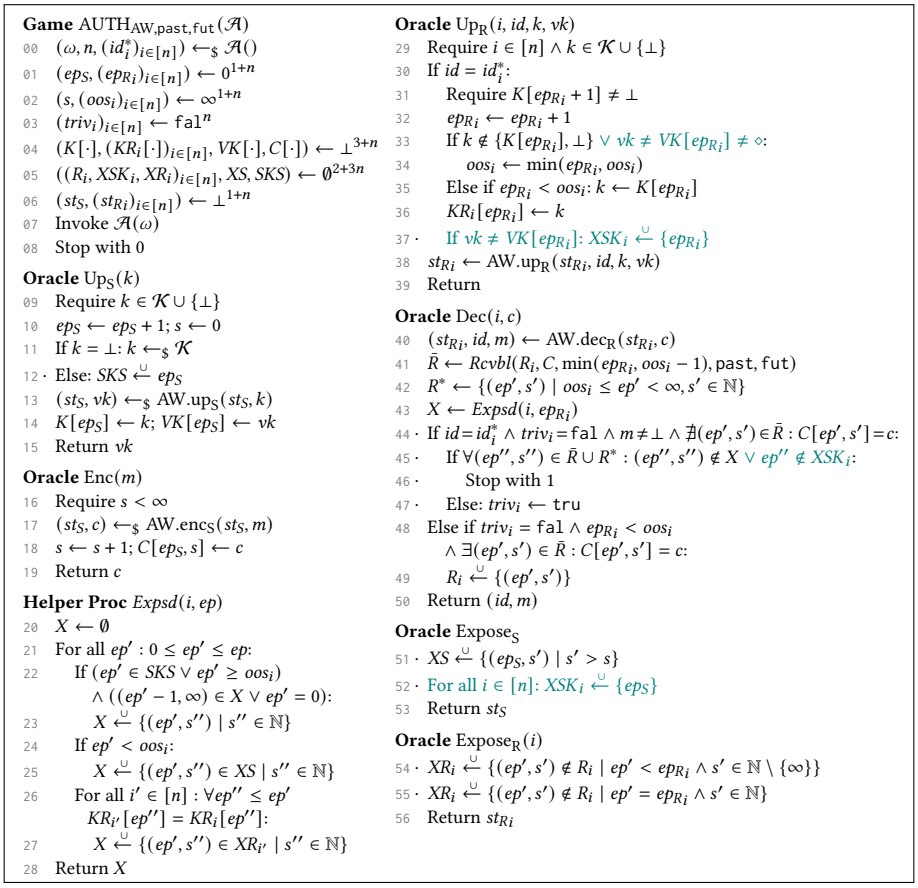

Figure 5: Multi-receiver games AUTH **for AW scheme** AW**. Helper Procedure** *Rcvbl* **is provided in Figure 4.**

the multi-instance authenticated encryption scheme with associated data AE in game IND\$ from Figure 6 is defined as:

$$\text{Adv}^{\text{ind\$}}_{\text{AE}}(\mathcal{A}) \coloneqq \left|\Pr[\text{IND\$}^0_{\text{AE}}(\mathcal{A}) = 1] - \Pr[\text{IND\$}^1_{\text{AE}}(\mathcal{A}) = 1]\right|.$$

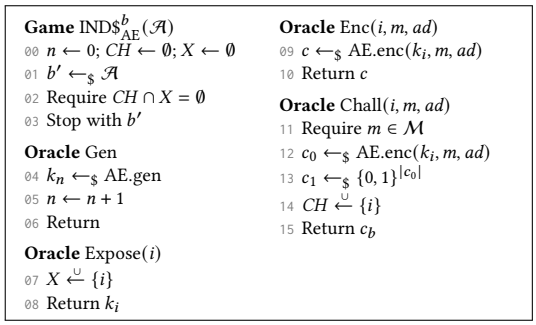

Figure 6: Games IND\$ **for authenticated encryption scheme with associated data** AE**.**

Security: Strong Existential Unforgeability under Chosen-Message Attacks. The advantage of an adversary $\mathcal{A}$ against the multi-instance authenticated encryption scheme with associated data AE in game SUF-CMA from Figure 7 is defined as:

$$\text{Adv}^{\text{suf-cma}}_{\text{AE}}(\mathcal{A}) \coloneqq \Pr[\text{SUF-CMA}_{\text{AE}}(\mathcal{A}) = 1].$$

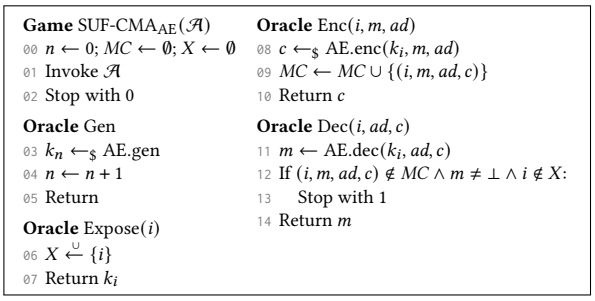

Figure 7: Game SUF-CMA for authenticated encryption scheme with associated data AE**.**

B.2 Signature Scheme S

Syntax. The signature scheme is a tuple of algorithms S = (S.gen, S.sgn, S.vfy) for some message space $\mathcal{M}$, signing key space $\mathcal{SK}$, verification key space $\mathcal{VK}$, and signature space $\mathcal{S}$:

- S.gen : $() \to_\$ (\mathcal{SK}, \mathcal{VK})$ Generates a pair of signing key sk and verification key vk.
- S.sgn : $\mathcal{SK} \times \mathcal{M} \to_\$ \mathcal{S}$ Creates a signature $\sigma \in \mathcal{S}$ for m using sk.
- S.vfy : $\mathcal{SK} \times \mathcal{M} \times \mathcal{S} \to \{\texttt{tru}, \texttt{fal}\}$ Verifies whether σ is a valid signature for m using sk.

Game SUF-CMA$_S(\mathcal{A})$
00 $n \leftarrow 0$; $MS \leftarrow \emptyset$; $CR \leftarrow \emptyset$
01 $(i, m, \sigma) \leftarrow_\$ \mathcal{A}$
02 $b \leftarrow$ S.vfy(vk_i, m, σ)
03 If $(i, m, \sigma) \notin MS \wedge i \notin CR \wedge b = \texttt{tru}$:
04 Stop with 1
05 Stop with 0

Oracle Corrupt(i)
06 $CR \xleftarrow{\cup} \{i\}$
07 Return sk_i

Oracle Gen
08 $(sk_n, vk_n) \leftarrow_\$$ S.gen
09 $n \leftarrow n + 1$
10 Return vk_{n-1}

Oracle Sign(i, m)
11 $\sigma \leftarrow_\$$ S.sgn(sk_i, m)
12 $MS \xleftarrow{\cup} \{(i, m, \sigma)\}$
13 Return σ

Figure 8: Game SUF-CMA **for signature scheme** S.

Security: Strong Existential Unforgeability. We define the advantage of an adversary $\mathcal{A}$ against the multi-instance signature scheme S in game SUF-CMA from Figure 8 as:

$$\mathrm{Adv}^{\text{suf-cma}}_{\mathrm{S}}(\mathcal{A}) \coloneqq \Pr[\text{SUF-CMA}_{\mathrm{S}}(\mathcal{A}) = 1].$$

B.3 Commitment scheme C

Syntax. The commitment scheme is a tuple of algorithms C = (C.com, C.vfy) for some message space $\mathcal{M}$, randomness space $\mathcal{R}$, and commitment space $\mathcal{CM}$:

- C.com : $\mathcal{M} \times \mathcal{R} \to \mathcal{CM}$ Creates a commitment $cm \in \mathcal{CM}$ to message $m \in \mathcal{M}$ and opening $r \in \mathcal{R}$.
- C.vfy : $\mathcal{M} \times \mathcal{CM} \times \mathcal{R} \to \{\texttt{tru}, \texttt{fal}\}$ Verifies if cm is a valid commitment to m using opening r.

Security: Binding. We define the advantage of an adversary $\mathcal{A}$ against commitment scheme C in game BIND from Figure 9 as:

$$\mathrm{Adv}^{\text{bind}}_{\mathrm{C}}(\mathcal{A}) \coloneqq \Pr[\text{BIND}^0_{\mathrm{C}}(\mathcal{A}) = 1].$$

Game BIND$_C(\mathcal{A})$
00 $(m, r, m', r') \leftarrow_\$ \mathcal{A}$
01 If $(m, r) \neq (m', r')$
$\wedge\ cm = \text{C.com}(m, r) = \text{C.com}(m', r')$
$\wedge\ \text{C.vfy}(m, cm, r) = \texttt{tru}$:
02 Stop with 1
03 Stop with 0

Figure 9: Game BIND **for commitment scheme** C.

B.4 Counting Bloom Filter

A Counting Bloom Filter (CBF) is a data structure that supports efficient insertions, deletions, and membership queries using an array of integer counters. To add an element, we hash it k independent hash functions, each producing an index into the counter array. We then increment the counters at these k positions by 1. Similarly, we remove an element from the CBF by hashing it and decreasing the counter at the resulting indices by 1. To test membership, we verify that all corresponding counters are non-zero. It can occur that the CBF yields false positives. This false positive probability p is taken into account when choosing the size and number of hash functions for the CBF. The CBF does not report false negatives as long as counters do not underflow.

B.5 Linked List

A linked list is a data structure with the following interface.

- LL.init() $\to L$: Initializes a new empty linked list.
- LL.push$_{\text{new}}$/push$_{\text{old}}(data)$: Appends an item containing $data$ to the list as the newest/oldest entry.
- LL.pop$_{\text{new}}$/pop$_{\text{old}}() \to ptr$: Removes the oldest/newest item from the list and returns a pointer to it.
- LL.len() $\to n$: Returns the length of the list.
- LL.rem(ptr): Removes the item which ptr points to from the list.
- LL.crop($ht, from = i$): Crops the list such that all items starting from index i until the highest index remain in the list. Removes deleted items from the hash table ht.
- LL.crop($ht, until = i$): Crops the list such that all items starting from index 0 until index i remain in the list. Removes deleted items from the hash table ht.

B.6 Hash Table

A hash table is a data structure with the following interface.

- HT.init() $\to ht$: Initializes a new empty hash table.
- HT.add($\{key : value\}$): Adds an entry to the hash table indexed with key storing $value$.
- HT.acc(key) $\to value$: Returns $value$ stored under index key.
- HT.rem(key): Removes the entry indexed by key from the hash table.

C Proofs

C.1 Game-based Anonymity

Theorem C.1. *Let* AE *be an AEAD scheme and* $\mathrm{H}_1, \mathrm{H}_2$ *random oracles. Let* $\mathcal{A}$ *be an adversary against the* IND *security of the* AW *scheme with parameters* fut, past. *Let* $\mathcal{B}$ *be an adversary against the* SUF-CMA *security of the AEAD scheme, and let* C *be an adversary against the* IND\$ *security of the AEAD scheme. Let* $\mathcal{CK}$, $\mathcal{UK}$, *and* $\mathcal{T}$ *denote the chain key space, update key space, and tag space, respectively. We show that our* AW *construction is secure with respect to* $\text{IND}_{\text{AW,fut,past}}$ *such that*

$$\mathrm{Adv}^{\text{IND}}_{\text{AW,fut,past}}[\mathcal{A}] \leq \frac{(q_{\text{enc}} \cdot (1 + \texttt{fut}) + q_{\mathrm{H}_1})^2}{|\mathcal{CK}|} + \frac{(q_{\mathrm{H}_2} + q_{\text{up}_S})^2}{|\mathcal{UK}|}$$

$$+ q_{\mathrm{H}_1} \cdot \left(\frac{q_{\text{enc}}}{|\mathcal{CK}|} + \frac{q_{\text{up}_S}}{\min(|\mathcal{CK}|, |\mathcal{UK}|)}\right) + \frac{(q_{\text{enc}} \cdot (1 + \texttt{fut}) + q_{\mathrm{H}_1})^2}{|\mathcal{T}|}$$

$$+ \mathrm{Adv}^{\text{SUF-CMA}}_{\text{AE}}[\mathcal{B}] + \mathrm{Adv}^{\text{IND\$}}_{\text{AE}}[C],$$

where $q_{\text{enc}}, q_{\text{dec}}, q_{\text{up}_S}$ *are the number of queries to oracles* Enc, Dec, Up_S, *and* $q_{\mathrm{H}_1}, q_{\mathrm{H}_2}$ *the number of queries to the random oracles* $\mathrm{H}_1, \mathrm{H}_2$, *respectively.*

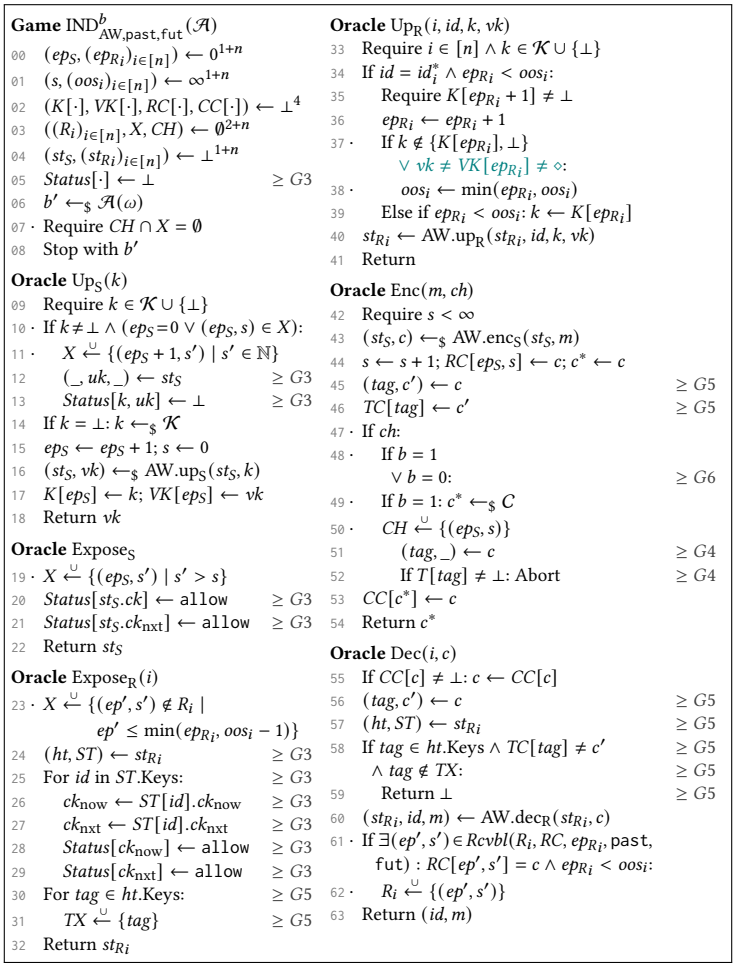

Figure 10: Game hops multi-receiver games IND **for AW scheme** AW**. The annotations** $\geq Gi$ **indicate that the line is valid from game** i **and in all following games. Teal code is only relevant for AAW with sender authentication.**

We create a sequence of games from G0 to G6 given in Figure 10. The random oracles H_1 and H_2 in Figure 11 both offer an internal interface for the game and an interface to the adversary.

Game 0. We start with the IND game for AW (Figure 1).

$$\mathrm{Adv}^{\mathrm{ind}}_{\mathrm{AW,fut,past}}[\mathcal{A}] = \mathrm{Adv}^{\mathrm{G0}}_{\mathcal{A}}$$

Game 1. We first introduce lazy sampling for the random oracle. That means, the random oracle randomly generates a return value whenever it is queried with a new input value. If this input is queried again, the random oracle remains consistent and returns the initially sampled value. This only changes the time of computation and thus an adversary cannot distinguish between Game 0 and Game 1.

$$\mathrm{Adv}^{\mathrm{G0}}_{\mathcal{A}} = \mathrm{Adv}^{\mathrm{G1}}_{\mathcal{A}}$$

Game 2. In this game, we show that the probability of chain key and update key collisions in the random oracle outputs is negligible. If the random oracle samples a used chain key (update key) twice, a loop occurs. This lets the adversary win if they, e.g., exposed the first usage of the chain key, and then the chain key comes up in another epoch and is used to obtain the encryption key for a challenge. The number of chain keys which are used as input to the random oracle is bounded by q_{enc} encryption queries plus at most $q_{\mathrm{enc}} \cdot \texttt{fut}$ queries for pre-computed values, and q_{H_1} adversarial random oracle queries. Now assume the same update key is sampled twice, once in epoch $i-2$, and once in epoch i. The adversary could expose at an earlier epoch $i-2$ to obtain the update key. Epoch i is then considered unexposed, even if the adversary controls the input chain key. Knowing the update key and the chain key, the adversary can now query the random oracle to obtain the encryption keys. The number of update keys is bounded by $q_{\mathrm{up_S}}$ update queries and q_{H_2} adversarial random oracle queries. With the birthday bound, we obtain the following advantage to distinguish between Game 1 and Game 2.

$$\mathrm{Adv}^{\mathrm{G1}}_{\mathcal{A}} \leq \mathrm{Adv}^{\mathrm{G2}}_{\mathcal{A}} + \frac{(q_{\mathrm{enc}} \cdot (1 + \texttt{fut}) + q_{\mathrm{H}_1})^2}{|\mathcal{CK}|} + \frac{(q_{\mathrm{H}_2} + q_{\mathrm{up_S}})^2}{|\mathcal{UK}|}$$

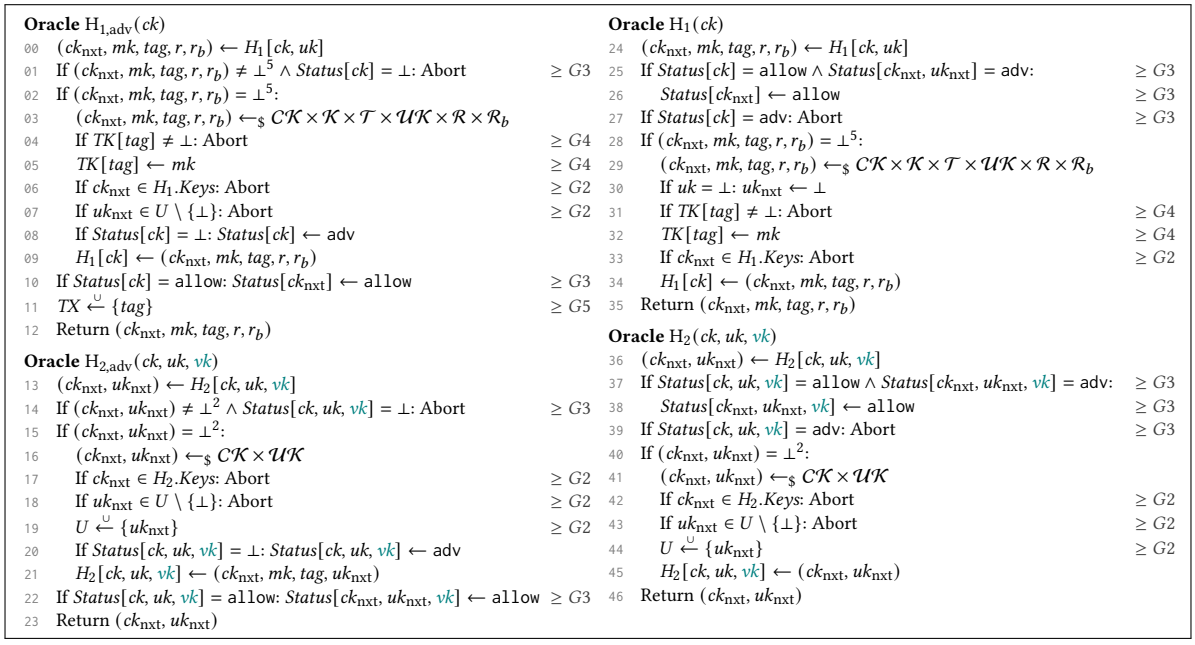

Figure 11: External and internal random oracles for the proof in Appendix C.1. The annotations $\geq Gi$ indicate that the line is valid from game i and in all following games. Teal code is only relevant for AAW with sender authentication.

Game 3. With this game, we show that the probability with which the adversary correctly guesses a secret chain key (and update key), and can thus query the random oracle for a decryption key, is negligible. We implement this by aborting in the random oracle whenever the adversary queries a secret value which the game used before, or when the game arrives at a secret value which was first queried by the adversary. For this, we set permissions for each random oracle input through the map *Status*. When the adversary is the first to query a value, we mark it with adv. Internally generated (secret) values keep the empty flag ($\perp$). The exception for when an adversary can query an internally used value is when they exposed the sender or receiver state to obtain a chain key ck or set the chain key ck in $\mathrm{up_S}$ *while knowing the update key* uk. For this, we introduce the flag allow which marks that the adversary may query the respective input. We mark an entry with allow when the sender is exposed (Lines 20 and 21), the receiver is exposed (Lines 26 to 29), or the adversary set the initial chain key in a sender update and knows the next update key (Lines 12 and 13). Moreover, after the adversary queried an internally used but "allow'ed" entry, they may query the next entry (but they may not skip an entry). The random oracle captures this by propagating the flag allow to the next entry.

The adversary successfully distinguishes Game 2 and Game 3 if they query the random oracle at an internally used chain key ck or if they query an entry (ck, uk) used for a state update. In the first case, the probability of guessing one of the chain keys from q_{enc} encryptions is $\frac{q_{\mathrm{enc}}}{|\mathcal{CK}|}$. The probability of guessing the update query pair (ck, uk) from $q_{\mathrm{up_S}}$ update queries is $\frac{q_{\mathrm{up_S}}}{|\mathcal{CK}|+|\mathcal{UK}|}$. If we assume that the adversary knows one of the keys, we arrive at the upper bound $\frac{q_{\mathrm{up_S}}}{\min(|\mathcal{CK}|,|\mathcal{UK}|)}$. An adversary which makes q_{H_1} queries to the random oracles thus has the following advantage to distinguish between Game 2 and Game 3.

$$\mathrm{Adv}^{\mathrm{G2}}_{\mathcal{A}} \leq \mathrm{Adv}^{\mathrm{G3}}_{\mathcal{A}} + q_{\mathrm{H}_1} \cdot \left(\frac{q_{\mathrm{enc}}}{|\mathcal{CK}|} + \frac{q_{\mathrm{up_S}}}{\min(|\mathcal{CK}|, |\mathcal{UK}|)}\right)$$

Observation for AAW. AAW uses a random value r_b derived from the chain key to blind the signature. If the adversary knows r_b they could use the verification key returned from $\mathrm{AW.up_S}$ to verify the signature by which they could determine whether the ciphertext is a random string or a real encryption. The advantage of such an adversary is negligible because of the following. The adversary only knows this value if a (prior) chain key is exposed in which case anonymity is lost anyway. Otherwise, they can only learn this value by guessing the corresponding chain key (covered in Game 3) or correctly guessing the r_b which functions as a one time pad and is thus bounded by the number of encryptions q_{enc} over the size of the space of random values $\mathcal{R}_b$.

Game 4. We now show that the probability of tag collisions is negligible. We bound the probability of tag collisions so that we can unambiguously address SUF-CMA challengers in the following game. The random oracle samples a tag whenever a new input is queried. This happens in at most q_{enc} encryption queries plus fut pre-computed values, and q_{H_1} adversarial queries to the random oracle. Tag collisions are thus limited by the birthday bound in the tag space $\mathcal{T}$ and the advantage to distinguish between games is as follows.

$$\mathrm{Adv}^{\mathrm{G3}}_{\mathcal{A}} \leq \mathrm{Adv}^{\mathrm{G4}}_{\mathcal{A}} + \frac{(q_{\mathrm{enc}} \cdot (1 + \mathtt{fut}) + q_{\mathrm{H}_1})^2}{|\mathcal{T}|}$$

Game 5. In Game 5, we reduce the capabilities of the active adversary to a passive adversary by enforcing that the adversary cannot forge a valid ciphertext. We modify the game such that decryption fails (i.e., the return value is $\perp$) when the decryption oracle receives a ciphertext which is a non-trivial forgery, i.e., for which the corresponding key was not leaked. We track the tags whose keys are exposed through the set TX by adding tags on receiver exposure and adversarial random oracle queries. To forbid non-trivial forgeries, we first observe that the adversary needs to submit a tag that is in the receiver state to trigger the decryption. If the tag is in the receiver state and the submitted ciphertext is not the one produced by the game, we simulate a failed decryption by returning $\perp$. When a receiver is out-of-sync, there must have been some input key which let the receiver state diverge from the sender state. From this point on, the collision-free update key which is sampled by the random oracle and cannot be determined by the receiver enforces that an out-of-sync receiver cannot get back in-sync with the receiver. Thus, if the game produces a tag using the sender state, this tag will not appear in the receiver state after it becomes out-of-sync and the decryption will fail. We note that the adversary can also submit a forgery for the pre-computed tags in a receiver state. For these forgeries, the adversary additionally has to correctly guess a randomly sampled tag which would reduce their win advantage by a factor of $\frac{\mathtt{fut}}{|\mathcal{T}|}$. The following reduction thus has an upper bound.

We now reduce this change to the SUF-CMA security of the underlying AEAD scheme. We use a multi-instance SUF-CMA game where each encryption and decryption are forwarded to a SUF-CMA challenger which is indexed with the respective tag. If a distinguisher can detect the change between Game 4 and Game 5, that means they can create a valid ciphertext forgery and thus break the SUF-CMA security of the underlying AEAD scheme. Note that whenever we address a SUF-CMA challenger in the reduction, we can do this unambiguously through the tag since we ruled out tag collisions in Game 4. This also means that multiple receivers which store the same tag communicate with the same SUF-CMA challenger.

- Enc: Forward the message and associated data to the encryption oracle of a new challenger. Use the *tag* derived from the chain key to associate the index of the SUF-CMA challenger with the tag. Use the challenger's encryption as ciphertext and the derived *tag* as return value.
- Dec: Look up the *tag* attached to the ciphertext in the receiver's hash table. If the tag is in the state, disregard the message key in the state and instead forward the decryption to the SUF-CMA challenger whose index corresponds to *tag* to obtain the decrypted message m. Otherwise, there is no key material for the submitted ciphertext in the receiver state and the decryption fails, i.e., $m \leftarrow \perp$. Send m to the distinguisher.
- Random Oracle: For all entries which the adversary queries, take the *tag* which the random oracle generates, corrupt the challenger whose index corresponds to *tag* to obtain mk and program mk into the random oracle.
- Expose_R: Corrupt all SUF-CMA challengers whose index corresponds to a tag in the receiver state to obtain the keys mk. Embed all mk in the respective positions in the receiver state and also in the random oracle. Note that the adversary exposes only one receiver state at a time. Thus, while the exposed key material may be in other receiver states as well, the adversary only ever sees the other receiver states when they explicitly expose them. We can therefore lazily program the other receiver states on exposure.
- Expose_S: As the sender state exposure only reveals chain keys and not message keys, we do not need to corrupt any SUF-CMA challenger here. All corruptions on adversarial queries are handled through the random oracle.

The advantage of a distinguisher to detect the change between Game 4 and Game 5 is bounded by the advantage of an adversary $\mathcal{B}$ which plays the multi-instance SUF-CMA game against the AEAD scheme.

$$\text{Adv}^{\text{G4}}_{\mathcal{A}} \leq \text{Adv}^{\text{G5}}_{\mathcal{A}} + \text{Adv}^{\text{SUF-CMA}}_{\text{AE}}[\mathcal{B}]$$

Game 6. In this game hop, we replace the real challenge ciphertext with a random ciphertext to arrive in the random world where an adversary has no advantage of winning the IND game. Note that a receiver, after they become out-of-sync, will never have key material for a challenge ciphertext in their state, as we prevented key collisions and thus the update key enforces that sender and receiver diverge. We reduce detecting this change to a multi-instance AEAD-IND$ game, which means that a successful distinguisher between Game 5 and Game 6 can break the IND$ of the underlying AEAD scheme. The reduction is as follows.

- Enc: On a challenge encryption, derive the tag through the random oracle and forward the message to the encryption interface of a new AEAD challenger along with the tag as associated data. Save the message and tag. Proceed as before on non-challenge encryptions.
- Dec: When receiving a decryption query, check if the tag corresponds to a prior challenge encryption. If yes, find the corresponding message and send it to the distinguisher. Otherwise, the decryption oracle remains unchanged.
- The other oracles are not affected and provide the usual interfaces to the distinguisher. That is because we do not allow state exposures or key updates which reveal key material for challenge ciphertexts.

The advantage of a distinguisher to detect the change between Game 5 and Game 6 is bounded by the advantage of an adversary C which plays the multi-instance IND$ game against the AEAD scheme.

$$\text{Adv}^{\text{G5}}_{\mathcal{A}} \leq \text{Adv}^{\text{G6}}_{\mathcal{A}} + \text{Adv}^{\text{IND\$}}_{\text{AE}}[C]$$

In the end, we arrive at the game where the adversary always obtains random ciphertexts and $\mathcal{A}$'s best strategy is to guess the bit b, i.e., $\text{Adv}^{\text{G6}}_{\mathcal{A}} = 0$.

C.2 Game-based Authenticity

Game AUTH in Figure 5 captures that it should be hard for an adversary to submit a non-trivial forgery. In the following, we first show that AW satisfies the basic AUTH definition. Then, we show

that AAW satisfies the extended definition AUTH_{AAW} which also captures exposures of the signing key.

Theorem C.2. *Let* AE *be an AEAD scheme and* H_1, H_2 *random oracles. Let* $\mathcal{A}$ *be an adversary against the* AUTH *security of the* AW *scheme with parameters* fut, past. *Let* $\mathcal{B}$ *be an adversary against the* SUF-CMA *security of the AEAD scheme. Let* $\mathcal{CK}$, $\mathcal{UK}$, *and* $\mathcal{T}$ *denote the chain key space, update key space, and tag space, respectively. We show that our* AW *construction is secure with respect to* $\text{AUTH}_{\text{AW,fut,past}}$ *such that*

$$\text{Adv}^{\text{IND}}_{\text{AW,fut,past}}[\mathcal{A}] \leq \frac{(q_{\text{enc}} \cdot (1 + \text{fut}) + q_{\text{H}_1})^2}{|\mathcal{CK}|} + \frac{(q_{\text{H}_2} + q_{\text{up}_\text{S}})^2}{|\mathcal{UK}|}$$
$$+ q_{\text{H}_1} \cdot \left(\frac{q_{\text{enc}}}{|\mathcal{CK}|} + \frac{q_{\text{up}_\text{S}}}{\min(|\mathcal{CK}|, |\mathcal{UK}|)}\right) + \frac{(q_{\text{enc}} \cdot (1 + \text{fut}) + q_{\text{H}_1})^2}{|\mathcal{T}|}$$
$$+ \text{Adv}^{\text{SUF-CMA}}_{\text{AE}}[\mathcal{B}],$$

where $q_{\text{enc}}, q_{\text{dec}}, q_{\text{up}_\text{S}}$ *are the number of queries to oracles* Enc, Dec, Up_S, *and* $q_{\text{H}_1}, q_{\text{H}_2}$ *the number of queries to the random oracles* H_1, H_2, *respectively.*

Proof for AW. Let AUTH'_{AW} be the game for AW where line 45 is always false and thus line 47 always applies, such that the adversary never wins the game. To detect this change, we prove that a distinguisher would need to create a non-trivial ciphertext forgery, i.e., a forgery where the corresponding key has not been exposed. First, we observe that an adversary could create such a forgery if they learn the encryption key through other means than exposure. That is, like in the IND proof, if the adversary correctly guesses the input to the random oracle or if there are any key or tag collisions in the random oracle. We can bound this probability like Game 1-4 in the proof for IND. Now, the distinguisher can only detect the change if they successfully hand in a forgery against the underlying AEAD scheme. To show that a distinguisher between AUTH and AUTH′ would break the SUF-CMA security of the underlying AEAD scheme, we use the reduction from Game 5 in the IND game. We note that, in the IND proof, we argued that once a receiver becomes out-of-sync, their future key material will never be in-sync again with the sender. Thus, submitting a forgery against an out-of-sync receiver is irrelevant since we do not enforce any IND security guarantees for the ciphertexts which are sent with out-of-sync key material. In contrast, the AUTH definition enforces that ciphertexts also cannot be forged against receivers which are out-of-sync with the sender. Note that we enforce that if two receivers diverge with the same keys, exposing one of them also exposes the other one, thus making a forgery against both of them trivial. Overall, the advantage of an adversary against AUTH_{AW} corresponds to the advantage of a distinguisher between AUTH_{AW} and AUTH'_{AW} which is bounded by the accumulated advantage of the distinguishers from Game 1 to Game 5 in the IND proof.

Theorem C.3. *Let* AE *be an AEAD scheme and* H_1, H_2 *random oracles. Let* $\mathcal{A}$ *be an adversary against the* AUTH *security of the* AAW *scheme with parameters* fut, past. *Let* $\mathcal{B}$ *be an adversary against the* AUTH *security of the* AW *scheme with parameters* fut, past. *Let* $\mathcal{C}$ *be an adversary against the* BIND *security of the Commitment scheme. Let* $\mathcal{D}$ *be an adversary against the* SUF-CMA *security of the Signature scheme. Let* $\mathcal{CK}$, $\mathcal{UK}$, *and* $\mathcal{T}$ *denote the chain key space, update key space, and tag space, respectively. We show that our* AW *construction is secure with respect to* $\text{AUTH}_{\text{AW,fut,past}}$ *such that*

$$\text{Adv}^{\text{AUTH}}_{\text{AAW,fut,past}}[\mathcal{A}] \leq \text{Adv}^{\text{AUTH}}_{\text{AW,fut,past}}[\mathcal{B}]$$
$$+ \text{Adv}^{\text{bind}}_{\text{C}}[\mathcal{C}] + \text{Adv}^{\text{suf-cma}}_{\text{S}}[\mathcal{D}].$$

Proof for AAW. Now, we show that AAW satisfies the extended AUTH_{AAW} definition. The extended definition says that it is hard for an adversary to forge a ciphertext even if either the AEAD key or the signing key are compromised. To bound the advantage of an adversary to win AUTH_{AAW}, we start with an adversary $\mathcal{A}$ playing the AUTH_{AAW} game. Then, we modify AUTH_{AAW} in two hops such that finally, the adversary has no winning advantage, i.e., Line 45 is always false. We make the following case distinction. If both AEAD key and signing key are compromised, no security guarantees hold. If neither key is compromised, the adversary is bounded by the smaller advantage of both breaking security of the AEAD scheme (see above) or breaking the singature and commiment scheme (as described in the following). If for a ciphertext c the signing key is compromised and the AEAD key is not (i.e., the sender was compromised after sending c), the proof for AAW corresponds to the AUTH proof for AW. In case the AEAD key is compromised and the signing key is not (i.e., the receiver was compromised but not the sender which holds the signing key; and the verification key for the receiver was not set by the adversary), we can reduce the detection of this change to breaking the security of the underlying commitment and signature scheme. Note that we only need to consider the case where the adversary submits a non-trivial forgery against the verification key which corresponds to the sender's (non-exposed) signing key because of the following observation. The receiver initially stores commitments for the verification key vk given via AW.up_R. If vk was chosen by the adversary in AW.up_R, then there are no security guarantees for the signature. Otherwise, vk given through AW.up_R corresponds to the sender's signing key. Since the receiver does not store vk in the state but only holds it in memory while running AW.up_R and AW.dec_R to pre-compute commitments. If the adversary sent some vk' which got accepted by the receiver, then the receiver would use this vk' to pre-compute commitments. But for this event to occur, the adversary would need to create a non-trivial forgery against vk in the first place (or trivial forgery) through which they already win the AUTH game (or looses the ability to win).

First, we prove that if the adversary attempts to submit a signature forgery, it has to be a forgery against the verification key for which the receiver stores the commitment. Let $\text{AUTH}'_{\text{AAW}}$ be the game where we abort if the adversary sends a (vk', r') which is not the same (vk, r) used for the commitment. To implement this game hop, we create a mapping from tags to pairs of (vk, r). When the receiver pre-computes tags and the openings r of commitments for a given verification key vk, we store for each tag the corresponding pair (vk, r). When the receiver now successfully decrypts a ciphertext and obtains (vk', r'), we look up the stored (vk, r) and abort if $(vk', r') \neq (vk, r)$. A distinguisher can only differentiate between AUTH and $\text{AUTH}'_{\text{AAW}}$ if they find a $(vk', r') \neq (vk, r)$ which breaks the binding security BIND_C of the commitment scheme.

The advantage of a distinguisher to differentiate between AUTH_{AAW} and $\text{AUTH}'_{\text{AAW}}$ is bounded by the advantage of an adversary C

against the binding security BIND_{C} of the commitment scheme as follows.

$$\text{Adv}^{\text{AUTH}}_{\mathcal{A},\text{AAW}} \leq \text{Adv}^{\text{AUTH}'}_{\mathcal{A},\text{AAW}} + \text{Adv}^{\text{bind}}_{\text{C}}[C]$$

Second, we prove that the adversary cannot forge a valid signature for the verification key vk or otherwise they break the underlying commitment scheme. We modify $\text{AUTH}'_{\text{AAW}}$ such that the condition in Line 45 (Figure 5) is always false and call this game $\text{AUTH}''_{\text{AAW}}$. Since we enforced that the adversary has to submit the forgery against the honest verification key, a successful distinguisher between $\text{AUTH}'_{\text{AAW}}$ and $\text{AUTH}''_{\text{AAW}}$ breaks the SUF-CMA security of the underlying signature scheme. The reduction for a distinguisher between AUTH′ and AUTH′′ plays against a multi-instance SUF-CMA challenger as follows:

- Up_{S}: Obtain the verification key vk provided by a new SUF-CMA challenger, embed vk in the sender state, and return vk.
- Enc: Forward the encrypted ciphertext and tag to the signing oracle of the current SUF-CMA challenger. Attach the challenger's signature to the returned ciphertext.
- Expose_{S}: Corrupt the current SUF-CMA challenger to obtain the signing key sk. Embed sk in the sender state.

The advantage of a distinguisher to differentiate between AUTH_{AAW} and $\text{AUTH}''_{\text{AAW}}$ is bounded by the advantage of an adversary $\mathcal{D}$ against the SUF-CMA security of the signature scheme as follows.

$$\text{Adv}^{\text{AUTH}'}_{\mathcal{A},\text{AAW}} \leq \text{Adv}^{\text{AUTH}''}_{\mathcal{A},\text{AAW}} + \text{Adv}^{\text{suf-cma}}_{\text{S}}[\mathcal{D}]$$

Through Game AUTH′′ we prohibited the adversary from winning and thus $\text{Adv}^{\text{AUTH}''}_{\mathcal{A},\text{AAW}} = 0$.

C.3 Adding a Group Member

Figure 12 shows the functions for adding a user to an existing group. The sender extracts a screenshot of its current key material through AW.add_{S} which the new user uses to initialize their state with $\text{AW.join}_{\text{R}}$.

Proc $\text{AW.add}_{\text{S}}(st = (tag_{\text{end}}, ck, uk, sk, vk, vk_{\text{prv}}))$
00 Return $keys = (ck, uk, vk)$

Proc $\text{AW.join}_{\text{R}}(st, keys = (ck, uk, vk), id)$
01 If $st = \bot$: $ST[\cdot] \leftarrow \bot$, $ht \leftarrow \text{HT.init}()$
02 $L_{\text{prv}}, L_{\text{now}}, L_{\text{nxt}} \leftarrow \text{LL.init}()^3$
03 For all $i : 0 \leq i <$ `past`:
04 $\quad \text{insertDummy}_{\text{new}}(L_{\text{prv}}, ht)$
05 $ck_{\text{nxt}} \leftarrow \bot$
06 $(ck_{\text{now}}, uk_{\text{nxt}}) \leftarrow (ck, uk)$
07 For all $i : 0 \leq i <$ `fut`:
08 $\quad \text{addEntry}_{\text{new}}(L_{\text{now}}, ht, ck_{\text{now}}, vk)$
09 $ST[id] \leftarrow (ck_{\text{now}}, ck_{\text{nxt}}, uk_{\text{nxt}}, L_{\text{prv}}, L_{\text{now}}, L_{\text{nxt}})$
10 Return $st = (ht, st)$

Figure 12: Functions for adding a group member in AW scheme AW. **Teal code is only relevant for AAW with sender authentication. The functions** addEntry **and** insertDummy **are shown in Figure 2.**

D Ideal Functionality for State Anonymity

The ideal functionality, $\mathcal{F}_{\text{AW}}$, is formally defined in Figures 13 and 14. This functionality captures multiple senders $S_{i\in[n]}$ and multiple receivers $R_{j\in[m]}$, where each sender S_i communicates with a fixed subset of receivers, denoted by $G[i] = g_i \subseteq [m]$.

First, we describe the update commands (in Figure 14). When a sender S_i wishes to update their state, they send to $\mathcal{F}_{\text{AW}}$ command **UpS**. As a result, $\mathcal{F}_{\text{AW}}$, stores the last encryption key identifier for the current epoch, ep (i.e., the index of the last message sent in this epoch). It then updates S_i's epoch to $ep + 1$ and registers S_i's current chain key identifier to reflect the new epoch, via dictionary `Ck`: $\texttt{Ck}[S_i] \leftarrow (ep, 0)$. On the receiver side, when receiver R_j wishes to update their state corresponding to Sender S_i, they send to $\mathcal{F}_{\text{AW}}$ command $\textbf{UpR}(i)$. If this is the first time $\textbf{UpR}(i)$ has been received from R_j, $\mathcal{F}_{\text{AW}}$ adds `past` + `fut` dummy key identifiers to the receiver's state (Note that these dummy keys are synchronized for each receiver in g_i for this session, via dictionary $\texttt{Du}_p$). $\mathcal{F}_{\text{AW}}$ also registers that the receiver has obtained key material for the new epoch in dictionary `Up` and registers that the first `fut` encryption key identifiers for the new epoch as well as the `fut`-th chain key identifier for the new epoch are in the receiver's state.

We note that to facilitate updates, we implicitly assume that any protocol instantiating $\mathcal{F}_{\text{AW}}$ will need to use some internal mechanism to sample fresh keys k_{ep}, where for each sender S_i, they and all receivers $R_j \in g_i$ must be able to obtain the same key k_{ep} for each epoch ep. This can be facilitated, e.g., by using a sub-functionality $\mathcal{F}_{\text{key}}$ for each sender S_i and associated g_i, which keeps a separate epoch counter for S_i and each $R_j \in g_i$, and delivers key k_{ep} to those parties who have already received the keys for epochs $ep' < ep$, when requested. As written earlier, we consider the actual protocol for sampling and delivering these keys to parties out of scope for this work.

Now we explain how messages are sent (Figure 13). Sender S_i uses command **Send**, to send a message m. The functionality $\mathcal{F}_{\text{AW}}$ first retrieves S_i's current chain key identifier that it has stored, and assigns it to the message via dictionary `M`, then increments S_i's current chain key identifier. Note that we do not send any information to the simulator $\mathcal{S}$ when S_i sends a message. Indeed, in order to analyze the anonymity that AW can *cyptographically* achieve, we choose not to model any consequences of *network* observation that can lead to de-anonymization, and instead assume idealized channels that reveal nothing about what is sent/received or by/to whom. Note that we still allow the adversary to leak ciphertexts that are still stored on the delivery server via command **ExposeDB** (see below).

Next, we describe how messages are received (in Figure 13). A receiver R_j uses command **Receive** to download from the delivery server all yet-unreceived messages from all sessions $g_i \ni R_j$ of which it is a part. First, $\mathcal{F}_{\text{AW}}$ marks each such message as delivered to R_j. If R_j does not have the key identifier stored for any of these messages, which can occur if, e.g., the message is too old or too many messages have been skipped, the message will not be delivered to R_j. Then, $\mathcal{F}_{\text{AW}}$ checks to which epoch each message belongs. If the given message belongs to the epoch following the one that R_j is currently in for this session, $\mathcal{F}_{\text{AW}}$ advances the receiver to the next epoch, registers the first `fut` encryption key identifiers for the new

epoch as well as the `fut`-th chain key identifier for the new epoch in the receiver's state, and finally removes the oldest key identifiers (including dummies) from the receiver state until only the newest `past` + `fut` identifiers remain. Else if the message corresponds to one of the newest `fut` keys stored for this epoch, the next `fut` encryption key identifiers are registered in the receiver's state, the chain key identifier is updated, and the oldest key identifiers from the receiver state are removed until only the newest `past` + `fut` or `past` + 2 · `fut` identifiers remain (depending on if the receiver has key material for the next epoch or not, respectively). Otherwise, the message corresponds to one of the oldest `past` keys. In all cases, the message's key identifier is removed from the receiver state. In the last case, it is replaced by a dummy identifier. Finally, $\mathcal{F}_{\text{AW}}$ deletes the messages from the delivery server which have now been delivered to all intended receivers and then delivers the above processed messages to R_j. Note again: since we want to model *cryptographic* anonymity, we do not send any information to $\mathcal{S}$ upon this delivery.

Now we discuss adversarial actions which can be taken by $\mathcal{S}$ (in Figure 14). First, command **DropRandom**(x) instructs $\mathcal{F}_{\text{AW}}$ to randomly drop x% of the messages which have been sent by some S_i but not yet received by all $R_j \in g_i$. We include this command to model random network drops. Next, command **ExposeDB** is used by $\mathcal{S}$ to expose the ciphertexts that are still stored on the delivery server. This command of course reveals to $\mathcal{S}$ all such ciphertexts and marks them as exposed, but also, if a chain key was previously exposed for the epoch in which such a ciphertext was created, $\mathcal{F}_{\text{AW}}$ gives information about how the ciphertext's encryption key was derived from that chain key. This matches the logic that the real world adversary would be able to trial decrypt this ciphertext with all encryption keys that can be derived from the chain keys it posseses. Additionally, $\mathcal{S}$ can instruct $\mathcal{F}_{\text{AW}}$ to drop some messages that it received through such an **ExposeDB** command by using command **Drop**.

Now, we discuss state exposures. The simulator $\mathcal{S}$ uses command **ExposeR**(j) to expose receiver R_j. Functionality $\mathcal{F}_{\text{AW}}$ first colects all key identifiers marked as stored in R_j's state. Importantly, if any of those keys correspond to a ciphertext that was leaked by an **ExposeDB** command when the key had not previously been leaked, $\mathcal{F}_{\text{AW}}$ aborts the command (in order to avoid the committment problem, see, e.g., [AJM22] and the references theirein). Then $\mathcal{F}_{\text{AW}}$ marks those key identifiers as exposed, and if any older chain key from the epoch in which a key was created was exposed, $\mathcal{F}_{\text{AW}}$ collects information for how that key was derived from that older chain key. Also, $\mathcal{F}_{\text{AW}}$ marks all keys which can be derived from the current chain key as exposed. Finally, $\mathcal{F}_{\text{AW}}$ adds to the above stored key identifiers any dummies stored in R_j's state and sends all the collected information to $\mathcal{S}$. Similarly, $\mathcal{S}$ can use command **ExposeS**(i) to expose sender S_i. Functionality $\mathcal{F}_{\text{AW}}$ does much of the same as above; collecting derivation information from any chain keys leaked in the past to the one currently held and marking those encryption key identifiers that can be derived from the current chain key as exposed. However, $\mathcal{F}_{\text{AW}}$ also marks S_i's signing key of the current epoch as exposed before leaking to $\mathcal{S}$ the current chain key identifier, the collected deriviation information, and the key identifier of the last key from S_i's previous epoch.

Finally, $\mathcal{S}$ can use command **Inject** to change the message contents of any ciphertext (i) for which $\mathcal{S}$ has the corresponding encryption key though **ExposeR** or **ExposeS**, (ii) for which $\mathcal{S}$ has the corresponding signing key through the latter, and (iii) which has not already been delivered to all receivers.

E Artifact Appendix

E.1 Abstract

This artifact appendix describes how to install and test the programs used to to implement and evaluate protocols in this paper, and how to run tests to verify the specific claims.

E.2 Description and Requirements

E.2.1 Security, privacy, and ethical concerns. None.

E.2.2 How to access. The artifact [Sch25] is packaged in an archive accessible at https://zenodo.org/records/16929590.

E.2.3 Hardware dependencies. None.

E.2.4 Software dependencies. This artifact requires the following software dependencies:

- `Rust 1.88.0`: On Unix-like systems, the rust toolchain can be installed with `rustup` using the command:

  ```
  curl --proto '=https' --tlsv1.2 \\
    -sSf https://sh.rustup.rs | sh
  ```

  To install the specific version, you can run:

  ```
  rustup install 1.88.0
  rustup default 1.88.0
  ```

  but be aware that this will override the default version of Rust on your system, affecting other Rust projects. The tests are likely to work with earlier versions of Rust, but have not been tested.

E.2.5 Benchmarks. None.

E.3 Set-up

With the Rust toolchain installed, the artifact can be built by opening a shell in the root directory of the project and running the following:

```
cd simple-aw-artifact && cargo build --release
```

This will compile the Rust code and produce an executable in the `simple-aw-artifact/target/release` directory.

E.3.1 Installation. Once compiled, no further installation is needed.

E.3.2 Basic test. To run all unit tests and verify the implementation:

```
cargo test
```

To run specific test suites:

```
# Test cryptographic primitives
cargo test -p signal-crypto

# Test protocol implementations, including Double
# Ratchet, Sender Keys, Sealed Sender, AW, and AAW
```

On Ideal Functionality $\mathcal{F}_{\text{AW}}$ Initialization Init(n, m, G):
Initializes $\mathcal{F}_{\text{AW}}$ with n senders $S_{i\in[n]}$ and m receivers $R_{j\in[m]}$; $G[i] = g_i \subseteq [m]$ specifies that sender S_i communicates with receivers $R_{j\in g_i}$. Note that a sender id i now describes a session which includes the sender and its receivers. $\mathcal{F}_{\text{AW}}$ interacts with senders, receivers and simulator $\mathcal{S}$, and stores variables:

- $\texttt{Ep}[S_i]$ / $\texttt{Ep}[R_j][i] \leftarrow -1$: current epoch of sender S_i / receiver R_j in session i
- $\texttt{Up}[R_j][i] \leftarrow -1$: receiver R_j's next epoch after a state update in session i
- $\texttt{K}[i][ep][s]$: map of keys, where each key is uniquely described through session i, epoch ep, and index s; this describes both encryption keys and chain keys. The key in a sender state is always a chain key. In a receiver state, the newest key in the current epoch and potentially the upcoming epoch are chain keys, all others are encryption keys. *A key (i, ep, s) is newer than a key (i, ep', s') if* $ep > ep' \vee ep = ep' \wedge s > s'$. K also stores the following attributes for all i, ep, s.
  - $\texttt{K}[i][ep][s].\texttt{rkid} \leftarrow_{\$} \mathcal{KID}$: random unique key identifier which is leaked to $\mathcal{S}$
  - $\texttt{K}[i][ep][s].\texttt{X} \leftarrow \texttt{fal}$: indicates whether the key is exposed
  - $\texttt{K}[i][ep][s].\texttt{stR}_j \leftarrow \texttt{fal}$: indicates whether receiver R_j has the key in its state
- $\texttt{M}[\texttt{mid}]$: map of messages, where
  - $\texttt{M}[\texttt{mid}].\texttt{m} \leftarrow \perp$: message m
  - $\texttt{M}[\texttt{mid}].\texttt{key} \leftarrow \perp^3$: key described through (i, ep, s) used to encrypt the message
  - $\texttt{M}[\texttt{mid}].\texttt{hasC} \leftarrow \texttt{fal}$: indicates whether the simulator has simulated a ciphertext without knowing the associated key
  - $\texttt{M}[\texttt{mid}].\texttt{p}_{R_j} \leftarrow \texttt{fal}$: indicates whether the message has been processed by R_j
  - $\texttt{M}[\texttt{mid}].\texttt{X} \leftarrow \texttt{fal}$: indicates whether the mid has been exposed to $\mathcal{S}$
- $\texttt{Ck}[S_i] \leftarrow \perp^2$ / $\texttt{Ck}[R_j][i] \leftarrow \emptyset$: current chain key/keys specified by (ep, s) of sender S_i / receiver R_j in session i // *receiver may have multiple keys in its state after an update*
- $\texttt{NE}[i][ep] \leftarrow \perp$: index s of newest exposed chain key in session i and epoch ep
- $\texttt{Lk}[i][ep] \leftarrow \perp$: index s of sender i's last encryption key from epoch ep
- $\texttt{XSK}[i][ep] \leftarrow \texttt{fal}$: indicates whether sender i's signing key for epoch ep was exposed
- $\texttt{Du}[R_j][i] \leftarrow []$: list of dummy key identifiers rkid for receiver R_j in session i. When dummy keys are treated as part of the receiver state, the dummy keys are considered the oldest key material, i.e., the keys which are discarded first.

- **On Send(mid, m) from S_i:**
  - skip if $\texttt{M}[\texttt{mid}] \neq \perp$
  - obtain current sender key $(i, ep, s) \leftarrow \texttt{Ck}[S_i]$ and assign it to the message $\texttt{M}[\texttt{mid}].\texttt{key} \leftarrow (i, ep, s), \texttt{M}[\texttt{mid}].\texttt{m} \leftarrow \texttt{m}$
  - update sender's chain key: $\texttt{Ck}[S_i] \leftarrow (ep, s + 1)$
- **On Receive(R_j) from R_j:**
  - create set of messages $Ms \leftarrow \emptyset$ to be delivered to R_j. Process all messages $\texttt{mid} \in \texttt{M}$ which are sent to a group where R_j is a member, i.e., the message's associated key $(i, ep, s) \leftarrow \texttt{M}[\texttt{mid}].\texttt{key}$ is from a session $i : j \in g_i$.
    - first, indicate that the message was processed by R_j: $\texttt{M}[\texttt{mid}].\texttt{p}_{R_j} \leftarrow \texttt{tru}$
    - skip to next message if no key material for mid is in the receiver state ($\texttt{K}[i][ep][s].\texttt{stR}_j = \texttt{fal}$)
    - if the receiver state was recently updated ($\texttt{Up}[R_j][i] = \texttt{Ep}[R_j][i] + 1$) and the message is from the new epoch ($ep = \texttt{Up}[R_j][i]$), advance receiver state to next epoch: $\texttt{Ep}[R_j][i] \leftarrow \texttt{Ep}[R_j][i] + 1$. Then, remove the message's key from the receiver state ($\texttt{K}[i][ep][s].\texttt{stR}_j \leftarrow \texttt{fal}$) and include the next fut keys in the receiver state ($\forall s < s^* \leq s + \texttt{fut} : \texttt{K}[i][ep][s^*] \leftarrow \texttt{tru}$). Update the receiver's set of chain keys ($\texttt{Ck}[R_j][i] \leftarrow \{(ep, s + \texttt{fut})\}$). Remove the oldest keys from the receiver state and the dummies $\texttt{Du}[i][R_j]$, until only the newest past + fut keys remain.
    - else if the message's key is in the newest fut keys ($(ep', s') \leftarrow \texttt{Ck}[R_j][i] \wedge ep = ep' \wedge s' - \texttt{fut} \leq s < s'$), remove the message's key from the receiver state ($\texttt{K}[id][ep][i].\texttt{stR}_j \leftarrow \texttt{fal}$) and include the next fut keys in the receiver state ($\forall i < i' \leq i + \texttt{fut} : \texttt{K}[i][ep][s] \leftarrow \texttt{tru}$). Remove the receiver's current chain key in ep and add the new key ($\texttt{Ck}[R_j][i] \leftarrow \texttt{Ck}[R_j][i] \setminus \{ep', s'\} \cup \{ep, s + \texttt{fut}\}$). If R_j does not have key material for the next epoch ($|\texttt{Ck}[i][R_j]| = 1$), then remove the oldest keys from the receiver state and the dummies $\texttt{Du}[i][R_j]$, until only the newest past + fut keys remain. Otherwise, keep the next epoch's pre-computed keys, i.e., remove until the newest past + fut + fut remain.
    - otherwise, the key is in the oldest past keys. Then, remove the message's key from the receiver state ($\texttt{K}[i][ep][s].\texttt{stR}_j \leftarrow \texttt{fal}$) and add an entry to the dummy list: $\texttt{Du}[i][R_j] \leftarrow_{\$} \mathcal{KID}$
    - add message to set Ms: $Ms \xleftarrow{\cup} \{\texttt{mid}\}$
  - delete messages mid from M which are now processed by all intended receivers: $\forall i \in [n] :$ if $\forall j \in g_i : \texttt{M}[\texttt{mid}].\texttt{p}_{R_j} = \texttt{tru}$, set $\texttt{M} \leftarrow \texttt{M} \setminus \texttt{M}[\texttt{mid}]$
  - send Ms to R_j

Figure 13: Ideal functionality $\mathcal{F}_{\text{AW}}$ for AW scheme AW. Teal code is only relevant for AAW with sender authentication.

```
cargo test -p libsignal-protocol

# Test core utilities
cargo test -p libsignal-core
```

E.4 Evaluation workflow

Benchmark programs and tests are provided to reproduce all implementation dependent claims in the paper.

E.4.1 Major Claims. The following claims are made in the paper, and are supported by the experiments in this artifact:

(C1): AW and AAW have small communication cost overhead when compared with Signal's current Sealed Sender implementation. Specifically, the paper reports the following measured message sizes for plaintexts less than 16 bytes in length:
- Sender Keys: 109 bytes
- AW-wrapped Sender Keys: 157 bytes
- AAW messages (using AAW as a standalone group messaging protocol): 155 bytes
- Sealed Sender-wrapped Sender Keys: $440 + 68N$ bytes where N is the group size.
- Double Ratchet: 66 bytes
- AW-wrapped Double Ratchet: 114 bytes
- Sealed Sender-wrapped Double Ratchet: 441 bytes

(C2): The computational time overhead for AW and AAW encryption and decryption are significantly smaller than the corresponding overhead for Sealed Sender. In the paper we report the following numbers which came from tests on a 32-core Intel Core i9-14900K processor.
- Sender Keys: 20.098μs (encrypt), 31.201μs (decrypt)
- AW-wrapped Sender Keys: 21.796μs (encrypt), 32.502μs (decrypt)
- AAW messages (using AAW as a standalone group messaging protocol): 21.079μs (encrypt), 31.185μs (decrypt)
- Sealed Sender-wrapped Sender Keys depends on group size, N:
  - $N = 2$: 131.12μs (encrypt), 61.286μs (decrypt)
  - $N = 5$: 283.58μs (encrypt), 61.286μs (decrypt)
  - $N = 10$: 343.41μs (encrypt), 61.286μs (decrypt)
  - $N = 100$: 1, 116.4μs (encrypt), 61.286μs (decrypt)
  - $N = 1000$: 6, 052.5μs (encrypt), 61.286μs (decrypt)
- Double Ratchet: 2.1979μs (encrypt), 2.7178μs (decrypt)
- AW-wrapped Double Ratchet: 3.7412μs (encrypt), 4.2402μs (decrypt)
- Sealed Sender-wrapped Double Ratchet: 61.171μs (encrypt), 52.750μs (decrypt)

(C3): The computational cost of AW and AAW updates is significant compared to their encryption and decryption times. Specifically, on the same Intel Core i9 test device AW was measured to have a mean update time of 1.5245ms and AAW was measured to have a mean update time of 2.7591ms.

E.4.2 Experiments. This section provides instructions for reproducing results that support the claims of the paper.

(E1): *[Message Sizes] [1 human-minute + 1 compute-minute]: produce serialized messages for each protocol and measure its size.*

Execution Execute the command

```
cargo test aw -- --nocapture
```

Results To assess claim C1, review the message sizes for the various protocol messages printed in the output and compare them to those reported in the paper

(E2): *[Computation time] [1 human-minute + 5 compute-minutes]: benchmark the computation time required for encryption and decryption in all protocols, and updates for* AW *and* AAW.

Execution Execute the command

```
cargo bench --bench sealed_sender
```

Results A number of benchmark tests will run and statistics will be reported after each test. To assess claim C2 look at the times reported for the following tests:
- Sender Keys baseline: `sk/encrypt` and `sk/decrypt`.
- AW wrapping Sender Keys: `aw-sk/encrypt` and `aw-sk/ decrypt`.
- AAW as standalone protocol: `aaw/encrypt` and `aaw/ decrypt`.
- Sealed Sender wrapping Sender Keys: `v2/encrypt/multirecipient/<N>` reports encryption time for groups of size N, `v2/decrypt` reports decryption time.
- Double Ratchet baseline: `dr/encrypt` and `dr/decrypt`.
- AW wrapping Double Ratchet: `aw-dr/encrypt` and `aw-dr/decrypt`.
- Sealed Sender wrapping Double Ratchet: `v1/encrypt` and `v1/decrypt`.

To asses claim C3, look at benchmark results for `aw/update` and `aaw/update`.

Note that the precise times reported by these benchmarks depend on the platform used for testing. The claims in the paper are about the relative performance of the evaluated protocols, and those differences should be stable on different platforms even though the absolute numbers will vary.

E.5 Notes on Reusability

This implementation was built as an experimental integration into https://github.com/signalapp/libsignal, Signal Messenger's core library. As libsignal is a special purpose library and is not designed for general use, the implementations of AW and AAW are not easy to reuse directly as packaged here. With that said, the files aw.rs and aaw.rs have minimal dependencies and should be simple for a developer to incorporate into a new project. The implementations of the benchmarks and tests give direct examples of how the code for both protocols can be used.

E.6 Contribution Note

Alex Bienstock did not contribute to the implementation of this artifact.

- **On UpR(i) from R_j**:
  - skip if the receiver did not move to the next epoch after the last state update: $\mathtt{Ep}[R_j][i] + 1 = \mathtt{Up}[R_j][i]$
  - indicate that the receiver has key material for an epoch update: $ep = \mathtt{Up}[R_j][i] \leftarrow \mathtt{Up}[R_j][i] + 1$
  - if this is the first time key material is added to the receiver's state, add $\mathtt{past} + \mathtt{fut}$ dummies to the receiver state: For k in $\mathtt{past} + \mathtt{fut}$: $\mathtt{Du}[i][R_j] \leftarrow_\$ \mathcal{KID}$
  - include first $\mathtt{fut}$ keys of new epoch in receiver state: $\forall s' : 0 \le s' < \mathtt{fut}: \mathtt{K}[i][ep][s'].\mathtt{stR}_j \leftarrow \mathtt{tru}$
  - include new chain key in receiver state: $\mathtt{Ck}[R_j][i] \xleftarrow{\cup} \{K[i][ep][\mathtt{fut}]\}$
- **On UpS from S_i**:
  - get current chain key $(ep, s) \leftarrow \mathtt{Ck}[i]$ and set last encryption key for this epoch $\mathtt{Lk}[i][ep] \leftarrow s$
  - update sender epoch: $\mathtt{Ep}[S_i] \leftarrow ep + 1$
  - update key in sender state for new epoch $ep \leftarrow \mathtt{Ep}[S_i]$: $\mathtt{Ck}[S_i] \leftarrow (ep, 0)$
- **On ExposeR(j) from $\mathcal{S}$**: Collect set of keys $Ks \leftarrow \emptyset$ and derivation info $DI[] \leftarrow []$ in each session i:
  - collect ordered list of randomized key identifiers in R_j's state: $K \leftarrow [\mathtt{K}[i][ep][s].\mathtt{rkid} : \forall ep, s, \mathtt{K}[i][ep][s].\mathtt{stR}_j = \mathtt{tru}]$, add list to the keys set: $Ks \xleftarrow{\cup} \{K\}$
    For each of these keys $(i, ep, s) : \mathtt{K}[i][ep][s].\mathtt{rkid} \in K$, set $\mathtt{K}[i][ep][s].\mathtt{X} \leftarrow \mathtt{tru}$, then collect derivation info $DI[\mathtt{rkid}]$ as follows:
    - *abort* if this would reveal a key for a previously key-less simulated ciphertext: $\exists \mathtt{mid} : \mathtt{M}[\mathtt{mid}].\mathtt{key} = (\mathtt{rkid}, ep) \wedge \mathtt{M}[\mathtt{mid}].\mathtt{hasC} = \mathtt{tru}$
    - if a prior chain key (i, ep, s') from epoch ep was exposed ($\bot \neq s' \leftarrow \mathtt{NE}[i][ep] \wedge s' < s$), collect derivation info from (i, ep, s') to (i, ep, s): $DI[\mathtt{rkid}] \leftarrow [\mathtt{K}[i][ep][s^*].\mathtt{rkid} : s' \le s^* < s]$. Then update the newest exposed key for this session and epoch: $\mathtt{NE}[i][ep] \leftarrow s$
  - set all keys which can be derived from the current chain key(s) (i, ep, s) as exposed: $\forall s \le s' : \mathtt{K}[i][ep][s'].\mathtt{X} \leftarrow \mathtt{tru}$
  - prepend list of dummy identifiers $\mathtt{Du}[i][R_j]$ to K: $K \leftarrow \mathtt{Du}[i][R_j] + K$

  Send (Ks, DI) to $\mathcal{S}$
- **On ExposeS(i) from $\mathcal{S}$**:
  - get the current chain key $(ep, s) \leftarrow \mathtt{Ck}[S_i]$ and its identifier $\mathtt{rkid} \leftarrow \mathtt{K}[i][ep][s].\mathtt{rkid}$
  - initialize map for key derivation info $DI[] \leftarrow []$
  - if in the current epoch, a prior chain key (i, ep, s') was exposed ($\bot \neq s' \leftarrow \mathtt{NE}[i][ep] \wedge s' < s$), collect derivation info from (i, ep, s') to (i, ep, s): $DI[\mathtt{rkid}] \leftarrow [\mathtt{K}[i][ep][s^*].\mathtt{rkid} : s' \le s^* < s]$. Then update the highest exposed chain key for this session and epoch: $\mathtt{NE}[i][ep] \leftarrow s$
  - set all keys which can be derived from the current chain key (i, ep, s) as exposed: $\forall s \le s' : \mathtt{K}[i][ep][s'].\mathtt{X} \leftarrow \mathtt{tru}$
  - set the sender's signing key for this epoch as exposed: $\mathtt{XSK}[i][ep] \leftarrow \mathtt{tru}$
  - find the key identifier of the last key from the previous epoch: $\mathtt{rkid}_{\mathrm{prv}} \leftarrow \mathtt{K}[i][ep-1][s_{\mathrm{prv}}]$ where $s_{\mathrm{prv}} \leftarrow \mathtt{Lk}[i][ep-1]$
  - send $(\mathtt{rkid}, DI, \mathtt{rkid}_{\mathrm{prv}})$ to $\mathcal{S}$

- **On DropRandom(x) from $\mathcal{S}$**
  - sample a random subset of messages $\mathtt{M}' \subseteq \mathtt{M}$ which contains x% of the set $\mathtt{M}$ and remove these messages from $\mathtt{M}$: $\mathtt{M} \leftarrow \mathtt{M} \setminus \mathtt{M}'$

- **On ExposeDB from $\mathcal{S}$**:
  - process all messages $\mathtt{mid} \in \mathtt{M}$ to build a set $Ms \leftarrow \emptyset$ and key derivation info $DI[\cdot] \leftarrow []$ as follows:
    - $(i, ep, s) \leftarrow \mathtt{M}[\mathtt{mid}].\mathtt{key}$; $\mathtt{m} \leftarrow \mathtt{M}[\mathtt{mid}].\mathtt{m}$; $\mathtt{m}' \leftarrow \bot$; $\mathtt{rkid}' \leftarrow \bot$
    - if $\mathtt{K}[i][ep][s].\mathtt{X} = \mathtt{tru}$:
      - $\mathtt{m}' \leftarrow \mathtt{m}$; $\mathtt{rkid}' \leftarrow \mathtt{K}[i][ep][s].\mathtt{rkid}$
      - if a prior chain key (i, ep, s') from epoch ep was exposed ($\bot \neq s' \leftarrow \mathtt{NE}[i][ep] \wedge s' < s$), collect derivation info from (i, ep, s') to (i, ep, s): $DI[\mathtt{rkid}'] \leftarrow [\mathtt{K}[i][ep][s^*].\mathtt{rkid} : s' \le s^* < s]$. Then update the newest exposed exposed chain key for this session and epoch: $\mathtt{NE}[i][ep] \leftarrow s$
    - otherwise, mark that there now exists a ciphertext for $\mathtt{mid}$: $\mathtt{M}[\mathtt{mid}].\mathtt{hasC} \leftarrow \mathtt{tru}$
    - mark that $\mathcal{S}$ knows $\mathtt{mid}$: $\mathtt{M}[\mathtt{mid}].\mathtt{X} \leftarrow \mathtt{tru}$
    - $Ms \xleftarrow{\cup} \{(\mathtt{mid}, |\mathtt{m}|, \mathtt{m}', \mathtt{rkid}')\}$
  - send M, DI to $\mathcal{S}$
- **On Drop(`mid`) from $\mathcal{S}$**:
  - skip if the $\mathtt{mid}$ has not been shown to $\mathcal{S}$ ($\mathtt{M}[\mathtt{mid}].\mathtt{X} = \mathtt{fal}$)
  - remove $\mathtt{mid}$ from the messages: $\mathtt{M} \leftarrow \mathtt{M} \setminus \mathtt{M}[\mathtt{mid}]$
- **On Inject(`rkid`, `m`) from $\mathcal{S}$**:
  - retrieve key information ($(i, ep, s) : \mathtt{K}[i][ep][s].\mathtt{rkid} = \mathtt{rkid}$) and corresponding message ($\mathtt{mid} : \mathtt{M}[\mathtt{mid}].\mathtt{key} = (i, ep, s)$)
  - skip if $\mathtt{K}[i][ep][s].\mathtt{X} = \mathtt{fal} \vee \mathtt{XSK}[i][ep] = \mathtt{fal} \vee \forall j : \mathtt{M}[\mathtt{mid}].\mathrm{p}_{R_j} = \mathtt{tru}$
  - otherwise, replace the corresponding message $\mathtt{M}[\mathtt{mid}].\mathtt{m} = \mathtt{m}$

Figure 14: Ideal functionality $\mathcal{F}_{\mathrm{AW}}$ for the anonymizing wrapper continued. Teal code is only relevant for AAW with sender authentication.